

Projektbericht: Agile Entwicklung von Computerspielen

Teammitglieder:

Youssef Daoudi El Boukhrissi (4277022)

Snehann Sritharan (4262727)

Jan Kirschbaum (4262909)

16. Juli 2025

Inhaltsverzeichnis

1. Einleitung	4
1.1. Ziel des Praktikums	4
1.2. Überblick über den Report	4
2. Spielkonzept, Use Case, Anforderungen und UML	5
2.1. Spielkonzept und Use-Case	5
2.2. Anforderungsanalyse	6
2.3. UML-Klassendiagramm	7
3. Implementierung	8
3.1. Verwendete Engine - GoDot	8
3.2. Technische Umsetzung der Anforderungen	8
3.2.1. Spieler:	9
3.2.2. Gegner:	9
3.2.3. Checkpoint:	9
3.2.4. Speichern und Laden:	9
3.3. Ergebnisse	10
4. Entwicklungsprozess	11
4.1. Initialer Prozess	11
4.1.1. Ideen sammeln	12
4.1.2. Tasks erstellen	12
4.1.3. Anforderungen definieren	12
4.1.4. Unit-Testing	12
4.1.5. Entwicklung von Prototypen	13
4.1.6. Code-Implementierung	13
4.1.7. Documentation	13
4.1.8. Codereview	14
4.2. Anpassung des Prozesses	15
4.2.1. Absprache von den Ideen im Team	16
4.2.2. Konkretisierung der Anforderungsanalyse	16
4.2.3. Task Reihenfolge	16
4.2.4. Zeitkomplexitätseinschätzung	17
4.2.5. Peer-Review	17
4.3. Test-Driven Development	18
4.4. Prototyping	18
4.5. SCRUM	19
4.6. Lessons Learned	19
5. Fazit und Ausblick	20
A. Anhang	21
A.1. Use-Case-Diagramm	21
A.2. Anforderungen	21
A.2.1. Finale Version	21
A.2.2. Erste Version	34

A.3.	UML-Klassendiagramm	35
A.3.1.	Finale Version	35
A.3.2.	Erste Version	36
A.4.	Tests	36
A.4.1.	Unit-Tests GDMUT	36
A.4.2.	Unit-Tests GdUnit4	39
A.4.3.	Excel-Tests	68
A.5.	Doxygen mit Quellcode	78

1. Einleitung

1.1. Ziel des Praktikums

„Goal of the Internship is to learn agile development processes similar to SCRUM while developing a game and optimizing the own process in order to achieve a 0-error-level.“

Dieses Ziel bildete die Grundlage unseres Praktikumsprojekts. Dabei stand nicht nur das fertige Spiel im Vordergrund, sondern vor allem der gesamte Weg dorthin, von der ersten Idee bis zur finalen Umsetzung. Ziel war es, im Team ein vollständiges Spiel zu entwickeln und gleichzeitig agile Entwicklungsmethoden praxisnah kennenzulernen und Schritt für Schritt in den eigenen Arbeitsprozess zu integrieren. Für eine strukturierte und effektive Zusammenarbeit setzten wir SCRUM ein. Es ermöglichte uns eine iterative Herangehensweise, bei der durch regelmäßige Meetings, eine klare Aufgabenzuteilung und kontinuierliches Feedback der Entwicklungsprozess gezielt begleitet und stetig weiterentwickelt wurde.

Ein besonderer Fokus lag auf der praktischen Anwendung zentraler Prinzipien der Softwareentwicklung. Dazu gehörten unter anderem die Planung und Pflege von Anforderungen, die Erstellung von Prototypen, das Durchführen strukturierter Tests sowie die gemeinsame Reflexion über den Projektverlauf. Die Einführung von Test-Driven Development war für uns ein entscheidender Schritt, um die Qualität unseres Codes zu sichern und Probleme frühzeitig zu erkennen.

Dieser Bericht gibt einen detaillierten Einblick in unseren Entwicklungsprozess. Er dokumentiert nicht nur die technischen Fortschritte, sondern auch die Herausforderungen, Entscheidungen und Verbesserungen, die wir als Team im Laufe des Projekts vorgenommen haben. Dabei spiegelt er wider, wie wir gelernt haben, gemeinsam zu planen, umzusetzen und unsere Arbeitsweise Schritt für Schritt zu verbessern.

1.2. Überblick über den Report

Dieser Bericht behandelt die Entwicklung eines Spiels mit der Engine Godot. Er beginnt mit der Vorstellung des Spielkonzepts, der Use-Case-Definition, der Anforderungsanalyse und einem UML-Klassendiagramm.

Anschließend wird die technische Umsetzung erläutert – einschließlich der verwendeten Engine, der Implementierung der Spiellogik für Spieler, Gegner, Checkpoints, Speicher- und Ladefunktionen sowie der Darstellung von Ergebnissen.

Darauf folgt eine ausführliche Beschreibung des Entwicklungsprozesses: Beginnend mit der Ideensammlung, der Erstellung von Konzepten, Diagrammen und Prototypen, über Implementierung, Testing und Prozessrevision bis hin zur Anpassung des Vorgehens mit Methoden wie SCRUM, Test-Driven Development, Peer-Review und Lessons Learned.

Im Fazit werden Projekterfahrungen reflektiert und mögliche Weiterentwicklungen skizziert. Der Anhang enthält ergänzendes Material wie Use-Case-Diagramme, Anforderungen (erste und finale Version), UML-Diagramme, Tests (Unit- und Excel-Tests), Quellcode und Dokumentation via Doxygen.

2. Spielkonzept, Use Case, Anforderungen und UML

2.1. Spielkonzept und Use-Case

Das Spiel ist ein 2D-Platformer mit Jump-'n'-Run-Elementen. Der Spieler muss die Hindernisse und Gegner auf allen Maps überwinden, um das Spiel durchzuspielen. Ein vielfältiges Spielerlebnis ist gegeben durch verschiedene Spielmodi, wie Precision Platformer, Chase Mode oder Boss Fight.

Das Bewegungsrepertoire des Spielers umfasst die Standardbewegungen – Laufen nach links und rechts sowie Springen – sowie besondere Fähigkeiten wie Dashen und Doppelspringen. Zum Besiegen der Gegner kann der Spieler angreifen, sich heilen und Attacken blockieren. Beim Eliminieren des Gegners wird der Spieler mit Sünden, der Währung des Spiels, entlohnt. Um die Schwierigkeit des Spiels zu gewährleisten, wird der Spieler durch ein Staminasystem, bei Nutzung der besonderen Fähigkeiten, limitiert.

Der Anfang wird mit einem Tutorialraum eingeführt, indem der Spieler die Grundlagen des Spiels vermittelt bekommt. Darauf folgend wird der Spieler durch eine komplexe Architektur der Karte herausgefordert, welches mit einem Bossraum endet. Das Spiel endet wenn der Boss besiegt wird.

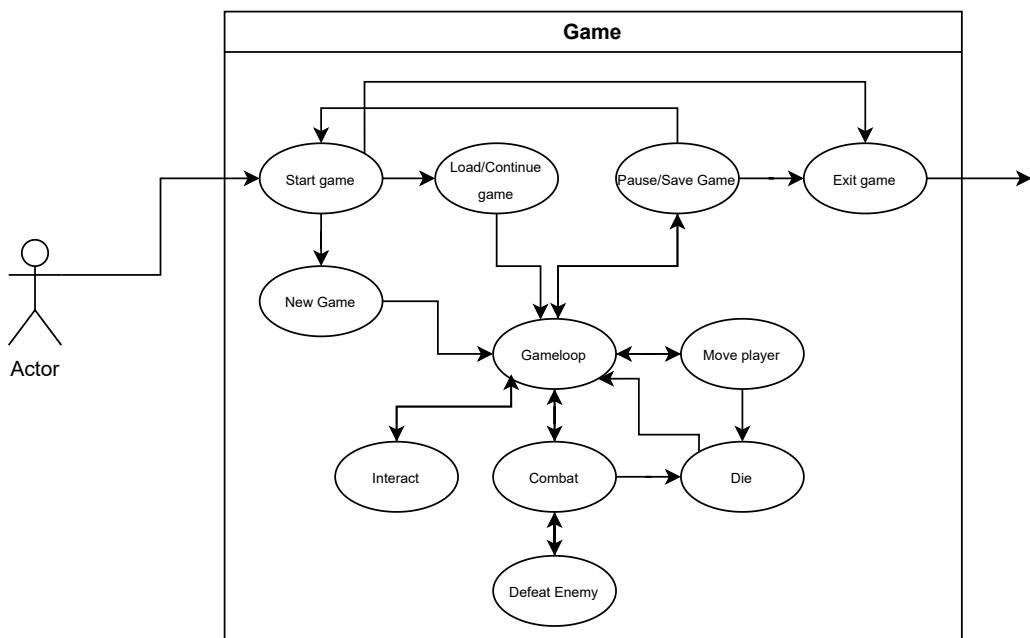


Abbildung 1: Finales Use-Case-Diagramm

Für das Spielkonzept haben wir ein Use-Case-Diagramm erstellt, um die wichtigsten Anforderungen und Funktionen des Spiels übersichtlich darzustellen. Dabei gibt es einen Hauptakteur – den Spieler –, der verschiedene Aktionen ausführen kann. Zu den grundlegenden Funktionen gehören das Starten, Speichern, Laden, Pausieren und Beenden des Spiels. Im eigentlichen Spiel kann der Spieler seine Figur bewegen, die Spielwelt erkunden und mit der Umgebung interagieren. Außerdem kann er gegen Gegner kämpfen. Dabei

besteht das Risiko, im Kampf oder durch andere Gefahren im Spiel zu sterben. Diese Möglichkeiten bilden die zentralen Spielmechaniken und sorgen für ein abwechslungsreiches Spielerlebnis.

2.2. Anforderungsanalyse

Die Anforderungsanalyse^{A.2} bildet die Grundlage für eine strukturierte Umsetzung des Spiels. Sie gliedert sich in funktionale Anforderungen, die testbare Eigenschaften definieren, und nicht funktionale Anforderungen, die Qualitätsmerkmale wie Performance und Benutzererlebnis festlegen. Auf Basis des Use-Case-Diagramms wurden zentrale Benutzer-System-Interaktionen identifiziert und daraus die relevanten Anforderungen abgeleitet.

2.2.1 Spieler

2.2.1.1 Movement - Der Spieler muss sich wie folgt bewegen können:

1. Der Spieler muss sich mit 100px/s nach rechts bewegen können, solange [Taste „D“] gedrückt wird
2. Der Spieler muss sich mit 100px/s nach links bewegen können, solange [Taste „A“] gedrückt wird
3. Der Spieler muss initial mit -300px/s nach oben springen können, wenn [Taste „W“] einmalig gedrückt wird

Abbildung 2: Ausschnitt der funktionalen Anforderungen

Die Bewegungen des Spielers wurden im Rahmen der Anforderungsanalyse genau festgelegt. Zum Beispiel muss sich der Spieler mit 100px/s nach rechts bewegen können, solange die Taste „D“ gedrückt wird. Die Bewegung nach links erfolgt genauso über die Taste „A“. Ein Sprung wird ausgelöst, wenn die Taste „W“ einmalig gedrückt wird – dabei bewegt sich der Spieler mit -300px/s nach oben.

Dieser Abschnitt zeigt nur einen kleinen Teil der vollständigen Analyse. Alle anderen Spielfunktionen, wie z.B. Angriff, Interaktion oder Benutzeroberfläche, wurden auf ähnliche Weise beschrieben und analysiert. So entsteht eine klare Grundlage für die spätere Umsetzung im Spiel. Alle Anforderungen wurden dabei nummeriert, thematisch gegliedert und quantitativ und spezifisch formuliert, z.B. durch konkrete Angaben zu Tasteneingaben, Bewegungsgeschwindigkeiten oder Positionen. Dies sorgt für eine klare und überprüfbare Grundlage für die spätere Umsetzung im Spiel.

2.3. UML-Klassendiagramm

Die Gestaltung des UML-Diagramms^{A.3} haben wir mithilfe der Requirements und dem Use-Case-Diagramm durchgeführt. Aufgrund des Klassendiagramms konnten wir eine Struktur für die Implementierung des Spiels visualisieren. Dies sorgte für eine reibungslose Umsetzung in die Game-Engine.

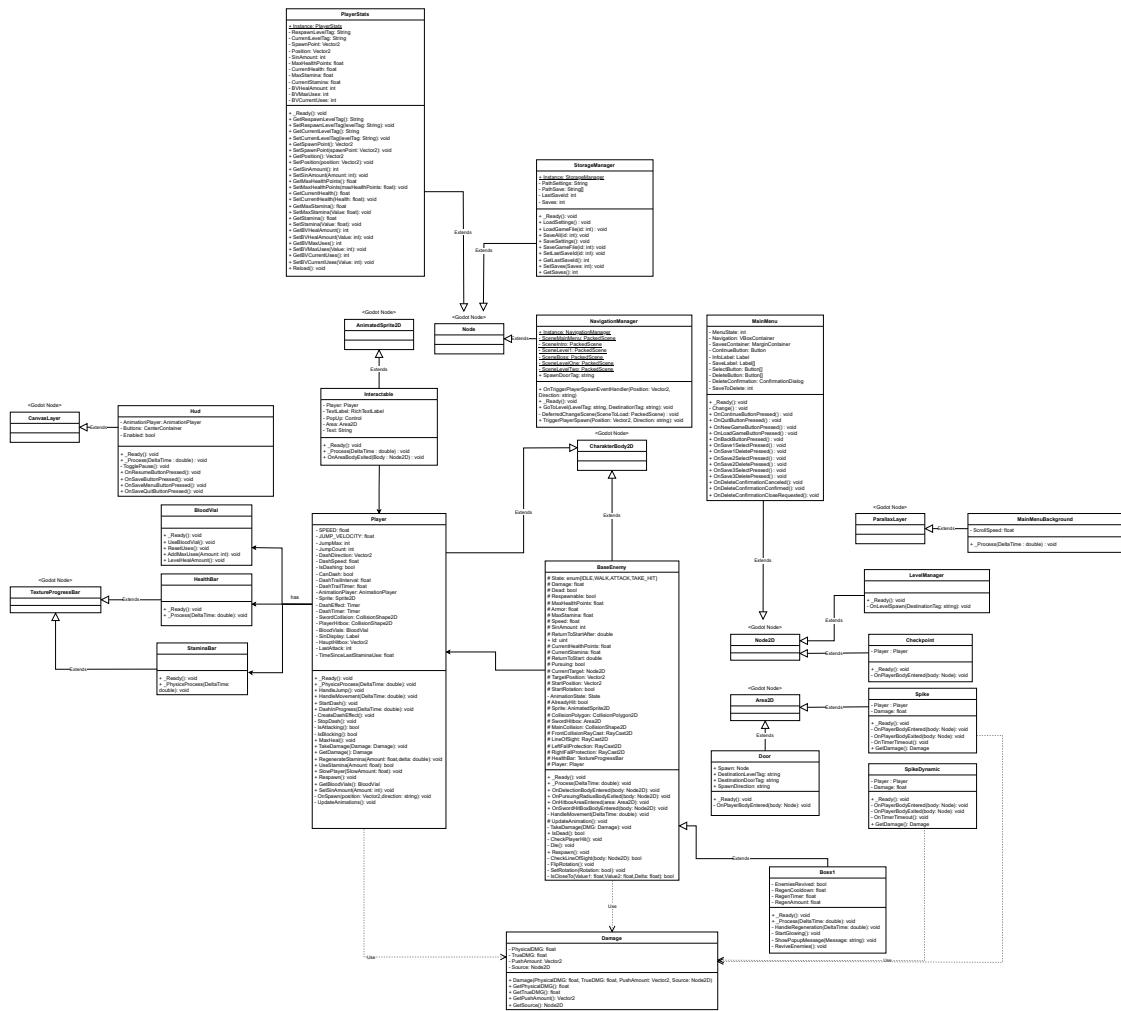


Abbildung 3: Finales UML-Klassendiagramm

Zuerst haben wir die Hauptelemente des Spiels als Klassen abgebildet. Diese sind Player und BaseEnemy, die im Zentrum des Diagramms zu finden sind. Alle anderen Klassen sorgen für eine Komplementierung des Spielerlebnisses. Zu allen Klassen haben wir uns dazu die Attribute, Funktionen und Assoziationen überlegt und konzipiert.

Die Klassen die von Node2D erben, bekommen automatisch bestimmte Attribute, wie zum Beispiel die Position übergeben. Alle anderen Nodes, wie Area2D oder CharacterBody2D, sind Unterklassen davon, die zusätzlich spezifischere Eigenschaften haben, wie Kollision.

Wir haben das Spiel gedanklich in fünf Bereiche unterteilt. Angefangen mit dem Spieler, der Attribute und Funktionen, die für die Bewegung, Angriffe und besondere Eigenschaften zuständig sind, besitzt. Außerdem hat der Spieler Assoziationen zu den Klassen

StaminaBar, HealthBar und BloodVial, die zu dem Bereich der graphischen Oberfläche gehören. Ebenfalls gehört die Hud und das MainMenu zu diesem. Der BaseEnemy ist die Klasse für den normalen Gegner, aber auch gleichzeitig für den ersten Boss verantwortlich. Diese Klasse hat die Funktion den Spieler zu erkennen, zu verfolgen und anzugreifen. Alle dieser Elemente befinden sich in der Spielwelt. Diese besteht neben der Tilemap¹ aus Spikes, die dem Spieler Schaden zufügen können, Türen, die für den Raumübergang benutzt werden und Checkpoints, welche für den Wiedereinstieg des Spielers sorgen. Ebenfalls wird der Spieler beim Passieren des Checkpoints geheilt. Der Levelübergang wird mithilfe vom NavigationManager und LevelManager verwaltet. Der letzte Bereich ist für die Speicherung und das Lagern der Attribute entscheidend, die durch den StorageManager und PlayerStats umgesetzt.

3. Implementierung

3.1. Verwendete Engine - GoDot

GoDot ist eine Open Source Engine für die 2D und 3D Spielentwicklung. Für das Prototyping setzten wir zunächst auf GDScript, da es sich schnell und unkompliziert einsetzen lässt. Im weiteren Verlauf entschieden wir uns für C#, um von der höheren Flexibilität und der gesteigerten Performance in der finalen Umsetzung zu profitieren. GoDot basiert auf Nodes, wie die im oben genannten UML-Klassendiagramm.

Nodes sind die grundlegenden Bausteine einer Szene - sie sind zuständig für die Funktionalität von Sprites², Kollision und Skripten. Die Struktur der Szenen ist eine Baumtopologie. Die Nodes befinden sich in Eltern-/ Kindbeziehungen zueinander. Dadurch können komplexe Gebilde einfach und übersichtlich aufgebaut und modelliert werden. Zusätzlich haben spezifische Nodes auch oft eingebaute Funktionen, wodurch physikalischen Eigenschaften automatisch von der Engine implementiert werden.

Für das Testen^{A.4} unserer Implementierung haben wir das Plugin GdMUT verwendet, das uns das Erstellen und Ausführen von Unit-Tests innerhalb von GoDot ermöglicht. Um gezielt Funktionen testen zu können, war es notwendig, eine virtuelle Testumgebung nachzubilden, in der die entsprechenden Komponenten isoliert ausgeführt werden. Dies erleichterte es uns, die Logik unabhängig vom restlichen Spielverlauf zu überprüfen und Fehler frühzeitig zu erkennen.

3.2. Technische Umsetzung der Anforderungen

In diesem Abschnitt wird die konkrete Umsetzung der zentralen Komponenten beschrieben. Die Entwicklung erfolgte zunächst prototypisch in GDScript und wurde anschließend in C# übertragen. Platzhalter-Assets wurden verwendet, um Funktionalitäten zu testen und darzustellen.

¹Eine Tilemap ist ein Raster aus kleinen Grafik-Kacheln, das zur Darstellung von 2D-Spielwelten dient.

²Eine Sprite ist eine 2D-Textur

3.2.1. Spieler:

Die Programmierung für die Bewegung des Spielers erfolgt zuerst durch die Abfrage der Tasteneingabe, um die Richtung³ zu bestimmen. Dabei berechnet die Engine, mit der von uns gewählten Geschwindigkeit, die Position. Beim Springen wird die Geschwindigkeit in negative y-Richtung erhöht. Das *Dashen* wird ebenfalls mit der Richtung berechnet, aber in diesem Fall wird noch ein Dasheffekt erstellt, wobei der Spielersprite dupliziert wird und dabei die Transparenz der Klone zunimmt. Zusätzlich wird die Kollision mit der Umgebung derart geprüft, so dass es möglich ist durch Gegner zu dashen, aber nicht durch Wände. Während eines *Angriffs* wird beim Schwertschwung die Kollision geprüft, ob diese einen Gegner trifft. Daraufhin wird dem Gegner Schaden übermittelt.

3.2.2. Gegner:

Jeder Gegner verfügt über eine Lebensanzeige, die oberhalb seines Sprites dargestellt wird. Sinkt die Lebenspunktanzahl auf null, gilt der Gegner als besiegt. Zur Spielererkennung überprüft der Gegner kontinuierlich einen festgelegten Bereich vor sich und reagiert, sobald der Spieler diesen betritt. Daraufhin beginnt der Gegner, den Spieler zu verfolgen. Verlässt der Spieler jedoch den definierten Verfolgungsbereich, bricht der Gegner die Verfolgung ab und kehrt zu seinem Startpunkt zurück. Falls der Gegner an eine Klippe steht, erkennt er diese mithilfe der selbst implementierten Edgedetection und bleibt stehen. Wie der Spieler kann auch der Gegner angreifen, jedoch erfolgt sein Angriff unmittelbar, sobald sich der Spieler in seiner Schlagreichweite befindet. Der *Boss* ist eine Unterklasse des Gegners. Er besitzt dieselben Grundfunktionen, ist jedoch größer, hat mehr Lebenspunkte und zusätzliche Rüstung. Außerdem verfügt er über eine zweite Kampfphase, in der er gestorbene Gegner wiederbelebt und durch eine goldene Form noch mehr verstärkte Rüstung erhält.

3.2.3. Checkpoint:

Beim Betreten eines *Checkpoints* wird die aktuelle globale Position des Checkpoints als neue Spawnposition des Spielers gespeichert. Dadurch erscheint der Spieler nach einem Tod an diesem Punkt erneut im Spiel. Zusätzlich wird der Spieler beim Durchschreiten des Checkpoints vollständig geheilt, und seine Stamina-Leiste wird komplett aufgefüllt, um eine optimale Vorbereitung auf die folgenden Spielabschnitte zu gewährleisten.

3.2.4. Speichern und Laden:

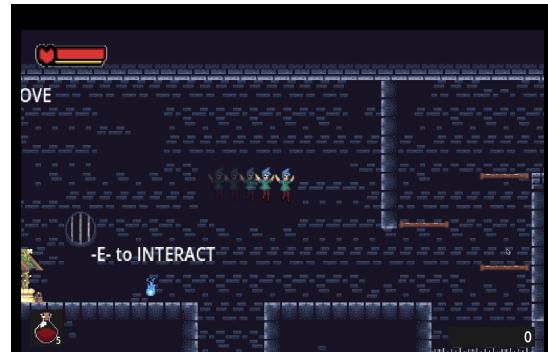
Für das Speichern und Wiederherstellen relevanter Spielvariablen beim Szenenwechsel oder Neuladen existiert eine zentrale Instanz der PlayerStats-Klasse, welche sämtliche benötigten Werte verwaltet. Um Spielstände auch außerhalb der aktuellen Spielsitzung zu sichern, werden die enthaltenen Informationen aus PlayerStats in jeweils eine separate Datei pro Spielstand geschrieben. Beim Laden eines Spielstands werden die gespeicherten Daten aus der entsprechenden Datei ausgelesen und in die PlayerStats-Instanz zurückgeführt. Das System unterstützt insgesamt bis zu drei gleichzeitig verwaltbare Spielstände.

³Ist ein 2D-Vektor der sich im Bereich von -1 und 1 befindet, wobei die y-Achse gespiegelt ist. Die Länge beträgt 1 oder 0.

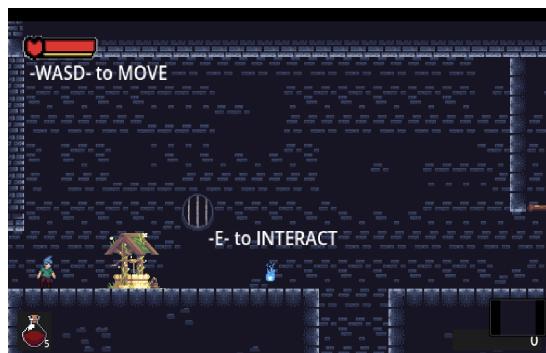
3.3. Ergebnisse



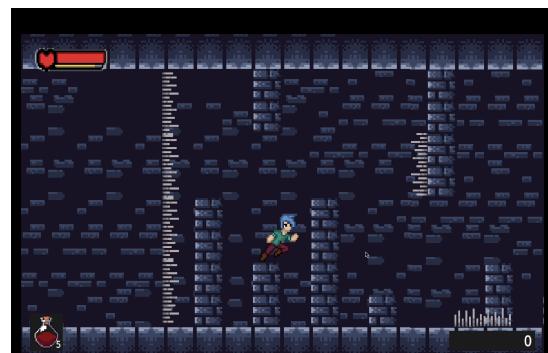
- MainMenu -



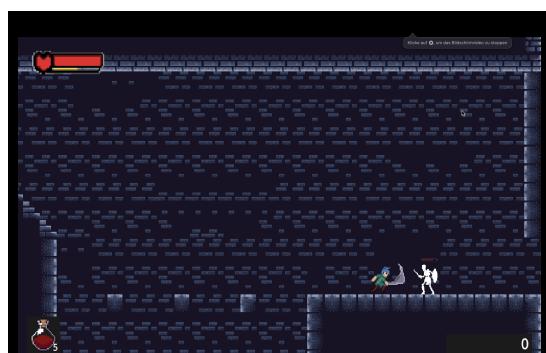
- Dash -



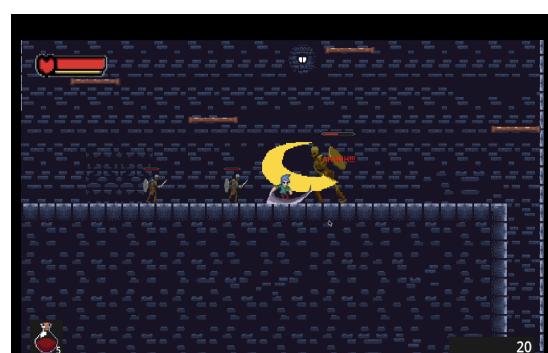
- Checkpoint -



- Jump'N'Run -



- Angriff auf ein Gegner -



- Boss zweite Phase -

Abbildung 4: Screenshots aus dem Spiel

4. Entwicklungsprozess

4.1. Initialer Prozess

In der folgenden Abbildung kann man unseren initialen Entwicklungsprozess sehen, welchen wir am Anfang des Projektes durchgeführt haben.

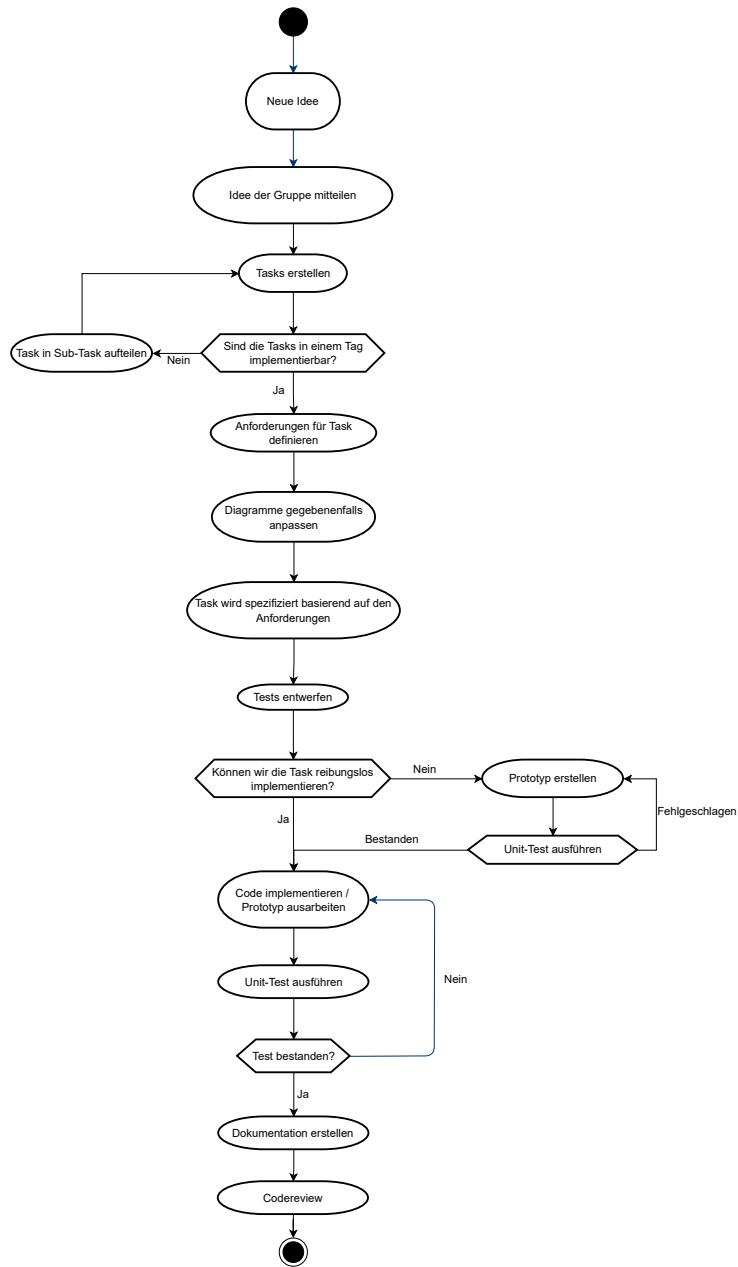


Abbildung 5: Initialer Prozess

4.1.1. Ideen sammeln

Der erste Schritt in unserem Entwicklungsprozess beginnt mit dem Sammeln und Mitteilen neuer Ideen. Sobald ein Teammitglied eine neue Idee hat, beispielsweise für ein Feature, eine Verbesserung oder einen technischen Lösungsansatz, wird diese in die Gruppe eingebracht.

Jedes Teammitglied sammelt seine Ideen eigenständig und stellt sie anschließend dem Team vor. Es findet in diesem Moment noch keine Diskussion über die Qualität, Umsetzbarkeit oder Priorität der Idee statt. Ziel dieses Schritts ist es, einen freien Raum für Kreativität zu schaffen, in dem alle Gedanken ungefiltert eingebracht werden können. So wird sichergestellt, dass auch unausgereifte oder unkonventionelle Ideen eine Chance bekommen, in einem späteren Schritt weiterentwickelt zu werden.

4.1.2. Tasks erstellen

Nach der Sammlung der neuen Idee wird diese in konkrete Aufgaben (Tasks) überführt. In diesem Schritt achten wir besonders darauf, dass die Aufgaben klar abgegrenzt, verständlich formuliert und zielgerichtet auf ein umsetzbares Ergebnis ausgerichtet sind. Ein wichtiger Bestandteil ist die Einschätzung, ob ein Task innerhalb eines Tages umsetzbar ist. Falls das nicht der Fall ist, wird der Task in kleinere Sub-Tasks unterteilt. Dadurch wird die Planung übersichtlicher, und der Fortschritt kann besser kontrolliert und koordiniert werden.

4.1.3. Anforderungen definieren

Nachdem die Tasks erstellt wurden, ging jedes Teammitglied dazu über, die Anforderungen eigenständig und basierend auf dem jeweiligen Task zu definieren. Dabei lag der Fokus auf einer quantitativen und präzisen Beschreibung, um eine klare Grundlage für die spätere Umsetzung und das Testen zu schaffen.

Die Anforderungen umfassten unter anderem konkrete Werte, wie z.B. Bewegungsgeschwindigkeiten von Spielfiguren, Kosten von Aktionen oder exakte Tastenbelegungen für bestimmte Funktionen. Dadurch wurde sichergestellt, dass die technische Umsetzung nicht auf vagen Annahmen beruht, sondern auf klar messbaren Vorgaben.

4.1.4. Unit-Testing

Zu Beginn unserer Testphase haben wir das Tool GDMUT verwendet, ein Plugin für die Godot Engine, das Unit-Tests über eine grafische Benutzeroberfläche ausführen kann. Allerdings stellten wir schnell fest, dass GDMUT in der Praxis zu eingeschränkt war, da sich damit keine komplexen Szenen simulieren oder realistische Testumgebungen abbilden ließen.

Aufgrund dieser Limitationen entschieden wir uns dazu, stattdessen GDUnit4 zu verwenden. GDUnit4 bietet eine robustere Grundlage für automatisierte Tests und lässt sich besser in bestehende Projekte integrieren. Damit konnten wir gezielte Funktionalitäten isoliert testen und unsere Testfälle strukturierter umsetzen.

Für Features, die nicht direkt durch automatisierte Tests prüfbar waren, z.B. visuelles Verhalten, Animationen, Benutzeroberfläche oder Kameraeffekte, haben wir zusätzlich Excel-basierte Testtabellen angelegt. Diese enthielten konkrete Kriterien, erwartete Ergebnisse und Prüfschritte für manuelle Tests, um auch diese Aspekte zuverlässig abzudecken.

Nach jeder Codeänderung wurde das Spiel außerdem manuell ausgeführt, um sicherzustellen, dass neue Anforderungen korrekt umgesetzt wurden und keine neuen Fehler entstanden. Im Falle eines fehlschlagenden Tests starteten wir eine gezielte Fehlersuche, um den Bug zu identifizieren und zu beheben.

4.1.5. Entwicklung von Prototypen

Im initialen Entwicklungsprozess legten wir großen Wert auf die Erstellung von Prototypen. Dabei haben wir jeden Task zunächst direkt in Form eines einfachen Prototyps umgesetzt, um möglichst schnell ein erstes Gefühl für die Funktionalität zu bekommen. Im Nachhinein stellten wir jedoch fest, dass es sinnvoller gewesen wäre, die einzelnen Tasks zunächst genauer zu analysieren, vor allem, um besser einschätzen zu können, ob ein Prototyp in dem jeweiligen Fall überhaupt notwendig ist. Die Prototypen selbst erstellten wir mithilfe von GDScript direkt in der GoDot Engine.

4.1.6. Code-Implementierung

Die Umsetzung des Codes erfolgte schrittweise und eng abgestimmt mit den zuvor definierten Anforderungen, Prototypen und Tests. Jeder Task wurde auf Basis der vorab erstellten Anforderungen und – sofern vorhanden – der entwickelten Prototypen implementiert. Dabei haben wir darauf geachtet, die Logik möglichst modular und testbar zu gestalten. Nach jeder abgeschlossenen Code-Implementierung führten wir sofort die zugehörigen Unit-Tests mit GDUnit4 aus. Ziel war es, frühzeitig festzustellen, ob die neue Funktionalität korrekt arbeitet und keine unerwarteten Nebeneffekte verursacht. Nur wenn alle relevanten Tests erfolgreich bestanden wurden, galt ein Task als abgeschlossen.

4.1.7. Documentation

Zur Dokumentation unseres Codes setzten wir Doxygen ein. Damit überarbeiteten wir unsere zuvor eher minimalistischen Kommentare und ergänzten sie mit strukturierten und detaillierten Beschreibungen. Jede Funktion und Klasse wurde mithilfe der Doxygen-Tags wie `brief`, `param` und `return` sorgfältig dokumentiert. Durch diese ausführliche Kommentierung wurde es für alle Teammitglieder deutlich einfacher, den Code der anderen nachzuvollziehen. Gleichzeitig war es auch für einen selbst hilfreich, um den eigenen Code schneller wieder zu verstehen, vor allem, wenn man bestimmte Funktionen nach langer Zeit nochmal anpassen oder erweitern musste. Ein weiterer Vorteil von Doxygen war die Möglichkeit, aus dem kommentierten Code automatisch eine übersichtliche HTML-Dokumentation zu generieren. Diese Webseite ermöglichte es uns, durch die gesamte Projektstruktur zu navigieren und Zusammenhänge zwischen Klassen, Methoden und Abhängigkeiten schnell zu erfassen.

4.1.8. Codereview

Am Ende des Prozesses überprüften wir, ob Änderungen an den Diagrammen beziehungsweise den Anforderungen vorgenommen werden mussten. Anschließend teilten wir die Änderungen des Projektes an alle Mitglieder mit. Diese haben gegebenenfalls ein Code-Review durchgeführt und Anmerkungen beigefügt.

4.2. Anpassung des Prozesses

Nach Feststellen von Problemen und Komplikationen während des Prozesses, entschieden wir uns, Änderungen beziehungsweise Verbesserungen vorzunehmen, um die Effizienz der Teamarbeit zu optimieren.

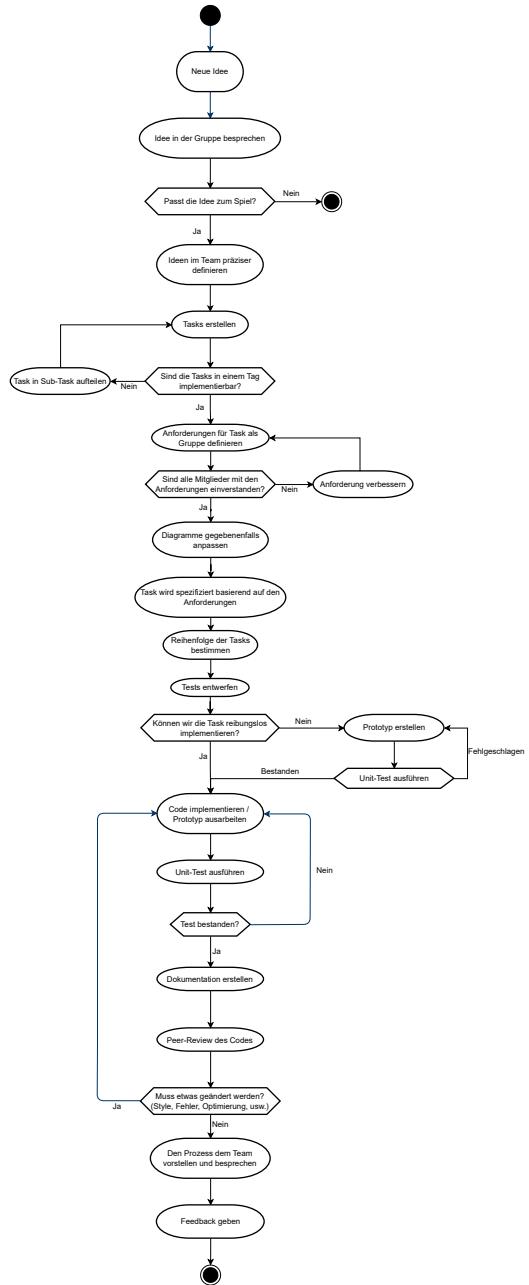


Abbildung 6: Prozess nach Verbesserungen

4.2.1. Absprache von den Ideen im Team

Problem:

Viele eingebrachten Ideen waren zu ambitioniert oder zu weit vom eigentlichen Spielkonzept entfernt. Dadurch kam es zu unnötigem Mehraufwand bei der Planung oder sogar zur Umsetzung von Ideen, die später verworfen wurden. Die Ressourcen des Teams wurden dadurch nicht effizient genutzt.

Prozessänderung:

Der Ideenfindungsprozess wurde gezielt angepasst. Bereits beim Einbringen einer neuen Idee wird nun explizit hinterfragt, ob die Idee zum Spiel passt. Diese einfache Kontrollfrage – „Passt die Idee zum Spiel?“ – dient als erstes Filterkriterium, um den Fokus auf sinnvolle und umsetzbare Vorschläge zu lenken. Erst wenn diese Frage eindeutig bejaht werden kann, wird die Idee weiterverfolgt und in mögliche Tasks überführt. Dadurch konnte der Anteil an überdimensionierten oder thematisch unpassenden Vorschlägen deutlich reduziert werden.

4.2.2. Konkretisierung der Anforderungsanalyse

Problem:

Obwohl die Anforderungen für einzelne Tasks quantitativ gut definiert waren, passten sie im Gesamtkontext des Spiels oft nicht gut zusammen. Vor allem die festgelegten Werte (z.B. Geschwindigkeiten, Abstände, Tastenbelegungen) führten im Zusammenspiel zu Inkonsistenzen im Spielerlebnis. Dieses Problem trat kontinuierlich bei nahezu jeder Implementierung auf und führte regelmäßig zu Nachbesserungen oder erneuten Abstimmungen.

Prozessänderung:

Die Anforderungsanalyse wurde konkretisiert, indem die Anforderungen nicht mehr einzeln, sondern gemeinsam im Team abgestimmt wurden. Dabei wurde besonders darauf geachtet, dass alle Werte im Kontext des gesamten Spiels zueinander passen. Durch diese teambasierte Definition wurde eine konsistenter Spielmechanik erreicht, da alle Beteiligten ein gemeinsames Verständnis für die Wirkung der Werte im Spiel entwickelt haben. Dies reduzierte Rückfragen, Anpassungsschleifen und Unstimmigkeiten während der Umsetzung deutlich.

4.2.3. Task Reihenfolge

Problem:

Während der Implementierung fiel häufig auf, dass viele Tasks stark voneinander abhängig waren, ohne dass diese Abhängigkeiten zuvor berücksichtigt wurden. Zum Beispiel konnte eine Funktion wie „Spieler erhält Schaden“ nicht getestet werden, da die dafür notwendigen Gegner- oder Spike-Objekte noch nicht implementiert waren. Dies führte zu Verzögerungen, unnötigem Leerlauf und einer ineffizienten Reihenfolge bei der Aufgabenbearbeitung.

Prozessänderung:

Um solche Abhängigkeitsprobleme zu vermeiden, wurde eine Task-Sequenzierung eingeführt. Vor Beginn einer neuen Arbeitsphase wurden alle offenen Tasks in eine sinnvolle Reihenfolge gebracht und nach Priorität geordnet. Dabei wurde explizit berücksichtigt,

welche Elemente technische oder logische Voraussetzungen für andere Tasks darstellen. Diese Anpassung sorgte für einen strukturierteren Arbeitsablauf, reduzierte Wartezeiten und stellte sicher, dass abhängige Features zuverlässig und rechtzeitig verfügbar waren.

4.2.4. Zeitkomplexitätseinschätzung

Problem:

Viele Tasks wurden zu Beginn ohne realistische Einschätzung der benötigten Zeit angenommen. Dadurch kam es oft zu unterschätzten Aufwänden, unerwartet langen Implementierungsphasen und Verzögerungen im Gesamtzeitplan. Die Aufgaben wirkten auf den ersten Blick einfach, entpuppten sich aber im Detail als deutlich komplexer, besonders bei der Einbindung von Engine-Funktionen oder bei umfangreicheren Tests.

Prozessänderung:

Um die Zeitplanung zu verbessern, wurde eine strukturierte Zeiteinschätzung vor Beginn jedes Tasks eingeführt. Dabei wurden Faktoren wie die Anzahl beteiligter Engine-Elemente, der Testaufwand und mögliche Abhängigkeiten berücksichtigt. Die geschätzte Zeit wurde dokumentiert und nach Abschluss mit dem tatsächlichen Aufwand verglichen. So entstand nach und nach ein realistischeres Gefühl für Komplexität und Planbarkeit, was zu besseren Entscheidungen bei der Taskvergabe und Sprintplanung führte.

4.2.5. Peer-Review

Problem:

Zu Beginn des Projekts wurde der Code oft ohne gemeinsames Feedback oder Qualitätskontrolle integriert. Dadurch entstanden Inkonsistenzen im Coding-Style, unnötig komplexe oder ineffiziente Lösungen sowie Fehler, die durch ein weiteres Paar Augen frühzeitig hätten erkannt werden können.

Prozessänderung:

Ein strukturierter Peer-Review-Prozess wurde eingeführt. Vor dem finalen Merge in das Hauptprojekt wurde jeder Codeabschnitt gemeinsam im Team besprochen. Dabei führten wir gezielt ein „Preview“ durch, in dem der Code gemeinsam betrachtet, analysiert und hinterfragt wurde. Der Fokus lag dabei auf folgenden Punkten:

- Einhaltung des Coding-Styles
- Lesbarkeit und Verständlichkeit des Codes
- Effizienz und Optimierungsmöglichkeiten
- Fehlermöglichkeiten und Randfälle

Am Ende jedes Reviews wurde konstruktives Feedback gegeben und ggf. Anpassungen vorgenommen, bevor der Code endgültig übernommen wurde.

4.3. Test-Driven Development

Ein zentraler Bestandteil unseres Entwicklungsprozesses war der Einsatz von Test-Driven Development (TDD). Dabei war es uns wichtig, Tests bereits vor der eigentlichen Implementierung des Codes zu schreiben. So konnten wir sicherstellen, dass jede Funktionalität eine konkrete Anforderung erfüllt und sich unser Code klar an definierten Zielen und messbaren Ergebnissen orientiert.

Zu Beginn fiel es uns jedoch schwer, eine klare Teststruktur zu etablieren, da viele Anforderungen zwar quantitativ vorhanden waren, aber im Kontext des gesamten Spiels noch nicht sauber abgestimmt waren. Dadurch war oft unklar, was genau getestet werden sollte. Einige Tests blieben dadurch zu allgemein, während andere übermäßig komplex wurden.

Zunächst setzten wir das GDMUT-Plugin ein, um Unit-Tests zu schreiben. Dabei zeigte sich allerdings schnell, dass GDMUT praktisch kaum geeignet war, da für jeden Test eine vollständige Spielumgebung simuliert werden musste. Es war notwendig, ganze Szenenbäume mit allen nötigen Nodes manuell zu erstellen, was enorm aufwändig war. Zusätzlich konnten die Tests nicht visuell nachvollzogen werden, was besonders bei Gameplay-relevanten Elementen wie Kollisionen oder Animationen zu Problemen führte.

Daher entschieden wir uns frühzeitig, GDMUT nicht weiter zu verwenden, und wechselten stattdessen zu GDUnit4. Dieses Tool ermöglichte eine strukturierte und automatisierte Testumgebung, die besser in unsere Arbeitsweise passte. Mit GDUnit4 konnten wir isolierte Funktionen direkt testen und erhielten eine klare Rückmeldung über Erfolg oder Fehlschläge der Tests. Die bessere Integration in die Godot-Umgebung und die effizientere Handhabung machten GDUnit4 zu einem zentralen Bestandteil unserer TDD-Strategie.

Für alle Funktionalitäten, die sich nicht sinnvoll automatisiert testen ließen, z.B. visuelles Verhalten, Übergänge, UI-Zustände oder dynamische Reaktionen, nutzten wir ergänzend ein Excel-basiertes Testprotokoll. Diese manuell gepflegten Tabellen dokumentierten die Anforderungen, erwartetes Verhalten und die tatsächlichen Ergebnisse für jede einzelne Testsituation. Dabei unterschieden wir gezielt zwischen:

- funktionalen Tests, z.B. Bewegungs- oder Kollisionsverhalten
- visuellen Tests, bei denen die Reaktion direkt im Spiel überprüft wurde

Durch die Kombination aus automatisierten GDUnit4-Tests und manuellen Excel-Tests konnten wir eine hohe Testabdeckung erreichen, sowohl auf Codeebene als auch im spielmechanischen und visuellen Bereich. Diese duale Teststrategie stellte sicher, dass neue Features zuverlässig funktionierten und gleichzeitig das Spielerlebnis konsistent blieb.

4.4. Prototyping

Anstatt für jede einzelne Funktion sofort einen Prototypen zu entwickeln, entschieden wir uns bewusst dafür, zunächst die jeweilige Task genauer zu analysieren. Ziel war es, einzuschätzen, ob ein Prototyp überhaupt notwendig ist oder ob die Funktionalität direkt in C# implementiert werden kann. In Fällen, in denen wir uns sicher waren, dass die Umsetzung ohne größere Schwierigkeiten möglich ist, begannen wir unmittelbar mit

der eigentlichen Implementierung im Produktivcode.

Wenn es jedoch Unklarheiten gab, erstellten wir einen Prototyp in GDScript. Dieser diente uns als schnelle und flexible Testumgebung, um Konzepte auszuprobieren und die technische Realisierbarkeit einzelner Funktionalitäten zu evaluieren. Den erstellten Prototyp haben wir iterativ weiterentwickelt, überarbeitet und optimiert, bis er unserem gewünschten Verhalten entsprach. Kam es dabei trotz mehrfacher Versuche zu Schwierigkeiten oder blieben bestimmte Aspekte unklar, haben wir uns im Team zu einem Meeting zusammengesetzt, um gemeinsam an einer Lösung zu arbeiten. Dieser Austausch war besonders hilfreich, um unterschiedliche Perspektiven zusammenzubringen und gemeinsam neue Lösungsansätze zu entwickeln. Sobald der Prototyp stabil und funktional überzeugend war, überführten wir den Code schrittweise von GDScript nach C#. Dabei achteten wir darauf, die Logik möglichst klar und sauber zu übernehmen.

4.5. SCRUM

Zu Beginn des Projekts haben wir dem SCRUM-Ansatz wenig geschätzt, obwohl er viele Elemente enthält, die unseren Arbeitsprozess deutlich verbessern konnten. Erst im Laufe der Entwicklung wurde uns bewusst, wie hilfreich bestimmte SCRUM-Aspekte tatsächlich sind. Besonders hilfreich war für uns das Konzept des Timeboxing, bei dem Themen im Meetings in klar begrenzten Zeitfenstern bearbeitet werden müssen. Dadurch konnten wir konzentrierter arbeiten und schneller zu Entscheidungen kommen.

Durch die Einführung von Sprints mit klar definierten Deadlines konnten wir unsere Motivation steigern, da der Fortschritt greifbarer wurde und wir strukturierter auf Zwischenziele hingearbeitet haben. Ein weiterer wichtiger Bestandteil waren unsere regelmäßigen Besprechungen, in denen wir nicht nur den aktuellen Stand austauschten, sondern auch gezielt Feedback zu einzelnen Aufgaben oder Codeabschnitten gaben. Dadurch entstand ein transparenterer Arbeitsprozess und Missverständnisse konnten frühzeitig geklärt werden.

Am Ende jedes Sprints hielten wir eine kurze Retrospektive ab, in der wir gemeinsam reflektierten, was gut funktioniert hat und wo es noch Verbesserungspotenzial gab. Diese Rückblicke haben maßgeblich dazu beigetragen, unseren Workflow kontinuierlich zu hinterfragen und sinnvoll weiterzuentwickeln.

4.6. Lessons Learned

Rückblickend konnten wir aus dem Projekt sehr viele wichtige Erkenntnisse mitnehmen, die wir in zukünftigen Projekten definitiv berücksichtigen werden. Einer der größten Lerneffekte war die Bedeutung Anforderungen klarer zu definieren. Zu Beginn war vieles vage formuliert, was uns beim Testen und Implementieren oft ausgebremst hat. Erst durch die gemeinsame Überarbeitung und Strukturierung der Anforderungen, wurde uns bewusst, wie viel einfacher die Umsetzung wird, wenn man ein sauberes Fundament hat. Auch unsere Einschätzung von Aufgaben und Zeitaufwand hat sich deutlich verbessert. Anfangs haben wir viele Features überschätzt oder falsch eingeschätzt, was zu Verzögerungen geführt hat. Durch Timeboxing, strukturierte Sprints und realistische Deadlines konnten wir im Laufe des Projektes unsere Planung schärfen und effizienter arbeiten.

SCRUM war dabei ein Meilenstein für unseren Arbeitsablauf. Was anfangs eher ignoriert wurde, hat sich mit zunehmender Projekterfahrung als extrem hilfreich herausgestellt. Vor allem, die regelmäßigen Meetings und die Retrospektiven haben für mehr Klarheit, Struktur und Motivation gesorgt.

Im Bereich Testing war der Einsatz von GDUnit4 ein voller Erfolg. Das Framework hat uns eine effiziente und zuverlässige Automatisierung von Tests ermöglicht und war aus unserem Entwicklungsprozess nicht mehr wegzudenken. Besonders hilfreich war, dass man den Code direkt implementieren und sofort testen konnte, ohne aufwändige Vorbereitungen oder komplexe Testumgebungen. Die Tests liefen automatisiert ab und lieferten schnell und klar Rückmeldung, ob die gewünschte Funktionalität korrekt umgesetzt wurde.

Peer-Reviews und direkte Rücksprache im Team haben dazu geführt, dass wir Missverständnisse früher erkannt und behoben haben. Außerdem war jeder immer auf dem neuesten Stand und konnte leichter in andere Bereiche einsteigen. Ein weiterer großer Fortschritt war unser Umgang mit Prototypen. Anfangs haben wir oft ohne Analyse einfach einen Prototyp erstellt. Später haben wir gelernt, zuerst zu prüfen, ob ein Prototyp wirklich nötig ist und wenn ja, diesen sauber zu entwickeln, zu testen und dann in C# zu übertragen.

Insgesamt haben wir nicht nur viele Erfahrungen in der Spieleentwicklung gesammelt, sondern auch gelernt, wie bedeutend gute Kommunikation, eine klare Struktur und ein agiler Prozess für erfolgreiche Teamarbeit sind. Dieses Projekt war für uns das erste echte Softwareprojekt, und genau deshalb war es so lehrreich.

5. Fazit und Ausblick

Für uns war es das erste gemeinsame Projekt an dem wir gearbeitet haben beziehungsweise für viele auch das erste Projekt überhaupt, deswegen war es zu erwarten, dass wir auf Probleme vor allem am Anfang stoßen. Zum Beispiel war das gemeinsame Arbeiten an einem Code eine komplett neue Erfahrung, die uns vor neue Herausforderungen gestellt hat. Dennoch konnten wir daraus viele Erfahrungen sammeln, die uns auch in unseren zukünftigen Projekten und Tätigkeiten zu effizienterer und produktiverer Arbeit bewegen. Wir können mit Stolz auf unsere Leistung zurückblicken, indem wir endlich unser theoretisches Wissen in einem realen Projekt anwenden konnten. Das Wichtigste, das wir dem Praktikum verdanken, ist, dass es das Interesse und die Begeisterung für technologische Entwicklung und Informatik generell neu in uns weckt oder wachsen lässt.

A. Anhang

A.1. Use-Case-Diagramm

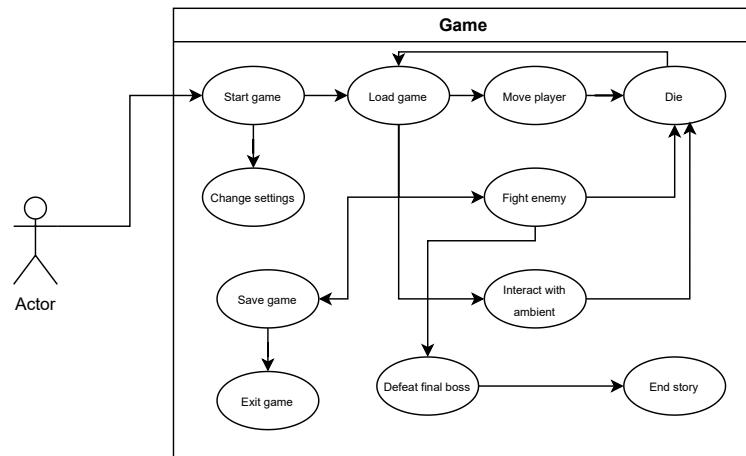


Abbildung 7: Erstes Use-Case-Diagramm

A.2. Anforderungen

A.2.1. Finale Version

Anforderungsanalyse

Inhaltsverzeichnis

1	Introduction	2
1.1	Purpose of the System	2
1.2	Scope of the System	2
1.3	Objectives and Success Criteria	2
1.4	Definitions, Acronyms, and Abbreviations	2
1.5	References	3
1.6	Overview	3
2	Proposed System	4
2.1	Overview	4
2.2	Functional Requirements	4
2.3	Nonfunctional Requirements	9
2.3.1	User interface and human factors	9
2.3.2	Documentation	9
2.3.3	Hardware considerations	9
2.3.4	Performance characteristics	9
2.4	Pseudo Requirements	10
2.5	System Models	10
2.5.1	Use Case Model	10
2.5.2	Class Diagrams	11
2.5.3	User-Interface	11
3	Glossary	12

1 Introduction

1.1 Purpose of the System

Das Ziel dieses Projekts ist die Entwicklung eines 2D-Action-Platformers, der dem Spieler ein unterhaltsames, herausforderndes und atmosphärisches Spielerlebnis bietet. Dabei steht das Zusammenspiel aus Bewegungsfreiheit, Kampfmechaniken und Erkundungselementen im Vordergrund.

1.2 Scope of the System

Das System umfasst alle Kernfunktionen eines Einzelspieler-Actionsspiels. Dazu gehören:

- Steuerung der Spielfigur mit Bewegungs-, Sprung- und Kampfmechaniken.
- Interaktion mit der Spielwelt durch Objekte, Gegner und Umgebungsbedingungen.
- Benutzeroberfläche zur Anzeige wichtiger Spielinformationen (Health, Stamina, Sünden, BloodVials).
- Verwaltung von Spielständen sowie grundlegende Konfigurationsmöglichkeiten.

1.3 Objectives and Success Criteria

Das Projekt gilt als erfolgreich, wenn folgende Punkte erreicht werden:

- Entwicklung einer stabilen, spielbaren Version mit vollständiger Implementierung der Bewegungs- und Kampfmechaniken.
- Integration von mindestens ein Spielabschnitten mit genau einem Endgegner.
- Übersichtliches und funktionales HUD zur Darstellung spielrelevanter Daten.
- Keine kritischen Fehler oder Abstürze bei regulärem Spielverlauf.
- Einhaltung der vorgegebenen Performance-Anforderungen.

1.4 Definitions, Acronyms, and Abbreviations

- TP – Trefferpunktzahl.
- DMG - Schaden.
- MAX_HEALTH - Maximale Trefferpunktzahl.
- MAX_STAMINA - Maximale Stamina.
- BVMaxUses - Maximale Anzahl an BloodVials.
- HEAL_AMOUNT - TP-Anzahl um die geheilt wird.
- HUD – Permanente Dateneinzeige für Spieler (HP, Stamina, Bloodvials, Sünden).
- FPS – Frames per Second.

- px - Pixel.
- Checkpoints - Ort, an dem der Spieler wiederbelebt wird.
- Spikes - Stacheln, die dem Spieler Schaden zufügen.
- Sünden – In-Game Währung, die durch Kämpfe gesammelt wird.
- Blood Vial – Heilitem, das Trefferpunkte wiederherstellt.
- Stamina – Ausdauer des Spielers.
- Fähigkeit - Eine Aktion, die bei Tastendruck ausgeführt wird.
- Dash - Schnelle Bewegungsfähigkeit des Spielers, um von einem Punkt zum anderen zu kommen.
- Achsen - x-Achse geht von links nach rechts, y-Achse geht von oben nach unten.
- Bewegungsrichtung - Ein 2D-Vektor, der normalisiert ist (Hat Betrag von 1) und die x und y Komponente können jeweils zwischen -1 und 1 liegen.
- Blickrichtung - Richtung in die der Spieler schaut, es ist entweder 0 oder 1 (Links oder Rechts).
- Velocity - Positionsänderung in x- und y-Richtung pro Sekunde.
- Detectionbereich - Bereich wo ein Gegner den Spieler erkennt.
- Armor - Rüstung des Gegners.
- Hitbox - Bereich für die Kollision eines Objektes

1.5 References

Im Rahmen der Entwicklung wurden folgende Technologien, Frameworks und Standards verwendet:

- **Godot Engine 4.4.1** – Open-Source-Spiel-Engine für 2D- und 3D-Spiele. Genutzt für die Implementierung der gesamten Spielmechanik, Grafik, Szenenstruktur und UI.
- **GDUnit4 5.1.0** – Framework für automatisierte Unit-Tests innerhalb von Godot-Projekten. Dient der Absicherung der Funktionalität auf Codeebene.
- **GDMUT** – Erweiterung/Toolset zur Erstellung und Verwaltung von Mock-Objekten und Testfällen in GDScript zur Unterstützung von GDUnit4.
- **Git** – Versionskontrollsystem zur Verwaltung des Projektfortschritts und der Zusammenarbeit im Team.

1.6 Overview

Dieses Dokument beschreibt alle funktionalen und nicht-funktionalen Anforderungen des Spiels. Dazu gehören Bewegungs-, Kampf- und Interaktionsmöglichkeiten, die Spielumgebung, Benutzeroberfläche, Performance und Entwicklungsstandards.

2 Proposed System

2.1 Overview

Alle Anforderungen sind nummeriert und enthalten eine quantitative Beschreibung.

2.2 Functional Requirements

2.2 Funktionale Anforderungen

2.2.1 Spieler

2.2.1.1 Movement - Der Spieler muss sich wie folgt bewegen können:

1. Der Spieler muss sich mit 100px/s nach rechts bewegen können, solange [Taste „D“] gedrückt wird
2. Der Spieler muss sich mit 100px/s nach links bewegen können, solange [Taste „A“] gedrückt wird
3. Der Spieler muss initial mit -300px/s nach oben springen können, wenn [Taste „W“] einmalig gedrückt wird
4. Der Spieler muss mit -300px/s doppelt springen können, wenn: [2x Taste „W“]
 - i) er sich in der Luft befindet
 - ii) er keine Fähigkeit währenddessen ausführt
 - iii) er mehr als 15 Stamina besitzt
5. Der Spieler muss mit 300px/s in die Bewegungsrichtung dashen können, wenn: [Taste „L-Shift“]
 - i) er mehr als 20 Stamina besitzt
 - ii) er nicht angreift
 - iii) die Bewegungsrichtung darf nicht (0,0) sein

2.2.1.2 Kampf - Der Spieler muss folgende kampfbezogene Aktionen ausführen können:

1. Der Spieler muss eine leichte Attacke in Blickrichtung durchführen können, wenn: [Taste „Linke Maustaste“] einmalig gedrückt wird
 - i) er mehr als 10 Stamina besitzt
2. Der Spieler muss eine schwere Attacke in Blickrichtung durchführen können, wenn: [Taste „Rechte Maustaste“] einmalig gedrückt wird
 - i) er mehr als 25 Stamina besitzt
3. Der Spieler muss einen Angriff blocken können, solange: [Taste „Leertaste“] gedrückt wird
4. Der Spieler muss sich mit einem Blood Vial heilen (+25TP) können [Taste „Q“]

2.2.1.3 Stamina – Der Spieler hat eine Ausdauer 0....MAX_STAMINA, diese regeneriert sich nach 3 Sekunden, wenn keine Stamina benötigt wurde. Folgende Aktionen reduzieren die Stamina:

1. Leichte Attacke [-10 Stamina]
2. Schwere Attacke [-25 Stamina]
3. Doppelsprung [-15 Stamina]
4. Dash [-20 Stamina]

5. Blocken stoppt Stamina Regeneration. Beim erfolgreichen Blocken wird von der Stamina der geblockte Damage (DMG) abgezogen. Falls es keine Stamina gibt, werden Lebenspunkte (TP) abgezogen.

2.2.1.4 Health – Der Spieler hat eine Trefferpunktanzahl (TP) von 0...MAX_HEALTH.

1. Der Spieler stirbt wenn TP <= 0
2. Der Spieler heilt sich wenn er ein Blood Vial trinkt [+25 TP]
3. Beim Checkpoint wird der Spieler komplett geheilt [TP = MAX_HEALTH]
4. Beim erleiden des Schadens verliert der Spieler den Schadenswert an TP [TP -= DMG]
5. Health regeneriert sich NICHT über Zeit

2.2.1.5 Leveling & Sünden – Der Spieler kann durch Gegner und Umgebung Sünden erlangen, mit diesen kann er folgende Attribute bei einem Checkpoint verbessern:

1. MAX_HEALTH + 25TP [100 Sünden]
2. MAX_STAMINA + 25 Stamina [100 Sünden]
3. Waffenangriff +5 DMG [100 Sünden]
4. Blood Vials: Heilkraft +5 TP [100 Sünden]

Glasscherben können verwendet werden, um ein extra Blood Vial zu erlangen

2.2.1.6 Tod – Der Spieler verliert alle Sünden beim Sterben.

2.2.1.7 Kollision – Der Spieler muss mit folgenden Objekten kollidieren können:

1. Boden und Wände
2. Gegner

2.2.2 HUD

2.2.2.1 Healthbar – Im Display muss die Healthbar oben links angezeigt werden. Diese muss die momentane TP des Spielers abbilden.

2.2.2.2 Staminabar – Im Display muss die Staminabar oben links unter der Healthbar angezeigt werden. Diese muss die momentane Stamina des Spielers abbilden.

2.2.2.3 Bloodvials – Im Display muss die Anzahl der Blood Vials unten links des HUDs angezeigt werden. Diese muss die momentane Anzahl der Blood Vials abbilden.

2.2.2.4 Sünden – Im Display muss die Anzahl der Sünden unten rechts des HUDs angezeigt werden. Diese muss die momentane Anzahl der Sünden abbilden.

2.2.3 Gegner & Boss

2.2.3.1 Movement:

1. Der Gegner muss den Spieler anfangen zu verfolgen, wenn dieser sich in seinem Detectionbereich (Kreis mit Radius von 70px) befindet.
2. Der Gegner muss aufhören den Spieler zu verfolgen, wenn dieser den Pursuingbereich (Kreis mit Radius von 179px) verlässt.
3. Jeder Gegner hat eine eigene Startposition [x,y – koordinate im jeweiligen Raum]
4. Der Gegner kehrt nach 5 Sekunden zur Startposition zurück, solange dieser keinen Spieler verfolgt.

2.2.3.2 Angriff:

1. Der Gegner muss eine leichte Attacke durchführen, wenn der Spieler weniger als 45px entfernt ist.

2.2.3.3 Health – Der Gegner hat eine Trefferpunktanzahl (TP) 0....MAX_HEALTH.

1. Der Gegner stirbt, wenn Health <= 0.
2. Die TP des Gegners muss 5px über den Gegnersprite sichtbar sein

2.2.3.4 Tod – Wenn der Gegner stirbt, bekommt der Spieler Sünden

[Anzahl Sünden = SIN_AMOUNT]

2.2.3.5 Kollision – Der Gegner muss mit folgenden Objekten kollidieren können:

1. Boden und Wände
2. Spieler

2.2.3.6 Bei 50% Leben des Bosses –

1. Der Boss wird goldfarben RGBA(1.0f, 0.84f, 0.0f, 1.0f), sobald 50 % seiner TP erreicht ist.
2. Der Boss erhält doppelte Rüstung [2x Armor] unter 50 % TP.
3. Der Boss belebt alle gestorbenen Gegner im Raum wieder, sobald er unter 50 % TP fällt.

2.2.4 Environment

2.2.4.1 Brunnen – Diese dienen als Checkpoint für den Spieler. Wenn dieser die Hitbox (200px × 200px) des Brunnen betritt, passiert folgendes:

1. Alle Blood Vials werden aufgefüllt [Anzahl BloodVials = BVMaxUses]
2. Der Spieler heilt sich zu 100%. [TP = MAX_HEALTH]
3. Stamina füllt sich zu 100%. [Stamina = MAX_STAMINA]

2.2.4.2 Kiste – Der Spieler kann mit Kisten interagieren. Kann Glasscherben oder Sünden enthalten.

2.2.4.3 Plattform - Der Spieler kann sich auf einer Plattform mit der Form 45px × 5px bewegen und wird bei Kontakt aus allen Richtungen (oben, unten, seitlich) durch eine CollisionShape2D zuverlässig gestoppt.

2.2.4.4 Spike – Der Spieler bekommt DMG bei betreten der Hitbox $30\text{px} \times 7\text{px}$. [TP -= DMG]

2.2.4.5 Tür - Diese dienen zum Wechseln der Räume. Sie teleportiert den Spieler zur eine anderen festgelegten Tür, sobald der Spieler die Hitbox $8\text{px} \times 50\text{px}$ betritt.

2.2.5 Spielmanagement

2.2.5.1 – Der Benutzer muss das Spiel über einen Button im Hauptmenü mit der Beschriftung „Continue“ starten und über den Button „Quit“ zuverlässig schließen können; beide Buttons müssen innerhalb von 1 Sekunde auf Eingaben reagieren.

2.2.5.2 Spielstand:

1. Der Benutzer muss in der Lage sein, den Spielstand zu speichern, über den Knopf „Save Game“
2. Der Benutzer muss in der Lage sein, den Spielstand zu löschen, über den Knopf „Delete“
3. Der Benutzer muss in der Lage sein, den Spielstand laden zu können, über den Knopf „Load Game“

2.2.5.3 Optionen – Diese Optionen dienen für die Konfiguration des Spiels.

1. Der Benutzer muss die Steuerung beliebig ändern können [Tastebelegung]
2. Der Benutzer muss die Grafik ändern können [Auflösung]
3. Der Benutzer muss die Sounds ändern können, zwischen 0% – 100% [Lautstärke]

2.3 Nonfunctional Requirements

2.3.1 User interface and human factors

- **UI-Reaktionszeit:** Alle Benutzeroberflächenelemente (Buttons, Menüpunkte, Interaktionen) müssen auf Benutzereingaben innerhalb von maximal **1 Sekunde** reagieren.
- **Button-Größe:** Interaktive Schaltflächen müssen eine Mindestgröße von **64 × 64 Pixel** besitzen, um auch bei niedriger Auflösung gut bedienbar zu sein.
- **HUD-Anzeige:** Die wichtigsten Werte (Health, Stamina, Sünden, Blood Vials) müssen dauerhaft sichtbar und innerhalb von **100 Millisekunden** nach einer Veränderung (z.B. Schaden, Heilung) aktualisiert werden.
- **Menüführung:** Alle Spielmenüs müssen logisch gegliedert und mit maximal **3 Klicks** erreichbar sein.

2.3.2 Documentation

- **Nachvollziehbarkeit:** Der Quellcode muss so strukturiert und kommentiert sein, dass neue Entwickler die Funktionalität einzelner Module innerhalb von **30 Minuten** nachvollziehen können.
- **Dokumentationstool:** Der gesamte Code muss mit **Doxygen**-kompatiblen Kommentaren versehen werden, sodass bei jeder Hauptversion eine HTML-Dokumentation automatisch generiert werden kann.
- **Abdeckung:** Mindestens **90 %** aller Klassen, Methoden und Konstanten müssen mit beschreibenden Doxygen-Kommentaren versehen sein (/// oder /** */).
- **Aktualität:** Die Dokumentation muss stets mit der aktuellen Codebasis übereinstimmen und bei Änderungen am Code spätestens im nächsten Commit angepasst werden.

2.3.3 Hardware considerations

- **Eingabegeräte:** Das Spiel muss vollständig mit **Tastatur und Maus** spielbar sein, ohne die Notwendigkeit zusätzlicher Peripheriegeräte wie Gamepads oder Touchscreens.

2.3.4 Performance characteristics

- **Bildrate:** Das Spiel muss bei normaler Nutzung eine stabile Bildrate von mindestens **30 FPS** erreichen, gemessen auf einem durchschnittlichen System (z.B. Mittelklasse-PC ohne dedizierte Grafikkarte).
- **Stabilität:** Während einer Spielsitzung von mindestens **60 Minuten** darf das Spiel zu keinem Zeitpunkt abstürzen oder ungewollt beendet werden.
- **Ladezeiten:** Das Laden eines Levels oder gespeicherten Spielstands darf nicht länger als **5 Sekunden** dauern.

2.4 Pseudo Requirements

- **Entwicklungsumgebung:** Die Projektstruktur ist kompatibel mit Versionierungssystemen wie Git und kann problemlos über Plattformen wie GitHub verwaltet werden.
- **Betriebssystem:** Die Entwicklung und der Ziel-Build richten sich primär an Windows-Systeme, sollen jedoch grundsätzlich auch unter Linux oder MacOS lauffähig sein.

2.5 System Models

2.5.1 Use Case Model

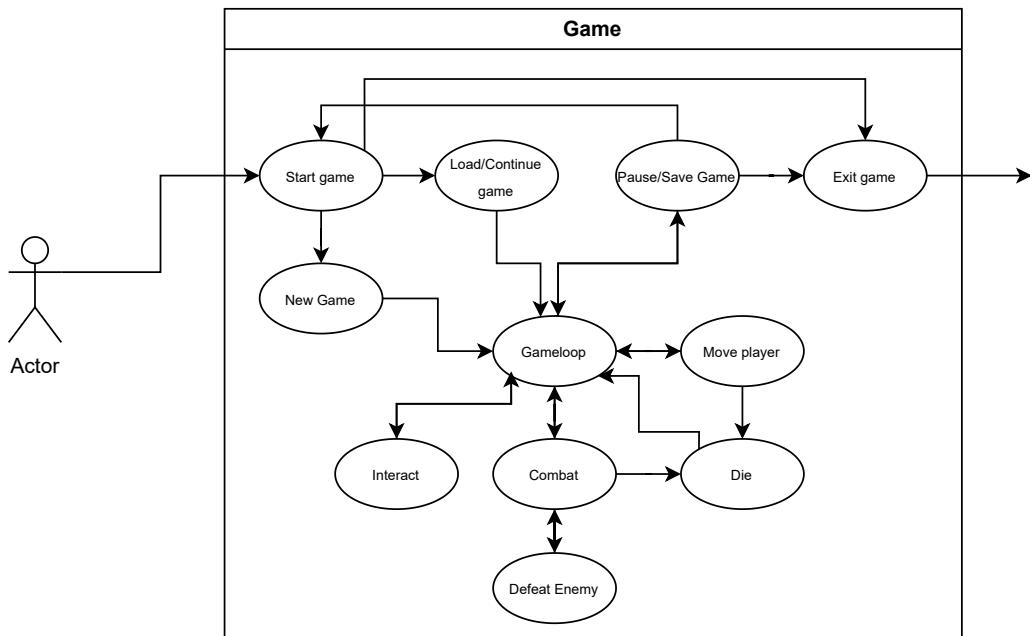


Abbildung 1: Finales Use-Case-Diagramm

2.5.2 Class Diagrams

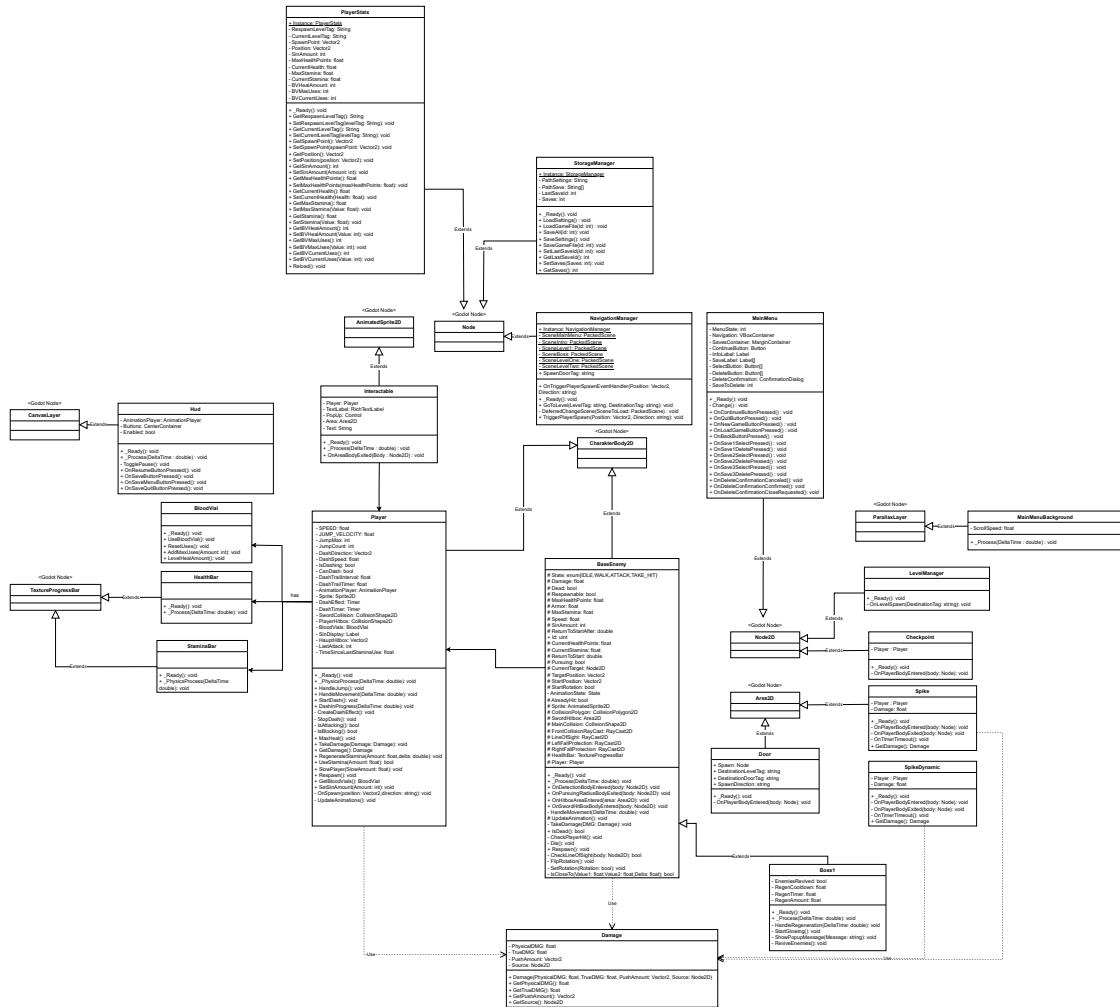


Abbildung 2: Finales UML-Klassendiagramm

2.5.3 User-Interface

Die Benutzeroberfläche ist in verschiedene, klar strukturierte Bereiche gegliedert und orientiert sich an intuitiver Bedienung.

Hauptmenü: Nach dem Start des Spiels erscheint das Hauptmenü mit folgenden Optionen:

- **Continue** – lädt den zuletzt automatisch gespeicherten Spielstand, sofern vorhanden.
 - **New Game** – startet ein neues Spiel von Anfang an.
 - **Load Game** – öffnet ein Untermenü mit drei Speicherplätzen (Slots), aus denen ein Spielstand geladen oder gelöscht werden kann.
 - **Quit** – beendet das Spiel vollständig.

Pause-Menü: Wenn das Spiel pausiert wird über die ESC-Taste, erscheint ein Menü mit folgenden Optionen:

- **Resume** – kehrt zurück ins laufende Spiel.
- **Save Game** – speichert den aktuellen Fortschritt im gewählten Slot.
- **Save & Return to Title** – speichert den Fortschritt und kehrt ins Hauptmenü zurück.
- **Save & Quit Game** – speichert den Fortschritt und beendet das Spiel.

3 Glossary

A.2.2. Erste Version

Anforderungsanalyse

Funktionale Anforderungen

2.1 Use Cases

- **Spiel management**
 - Der Benutzer muss das Spiel starten können.
 - Der Benutzer muss das Spiel verlassen können.
 - Der Benutzer muss den aktuellen Spielstand speichern können.
 - Der Benutzer muss ein zuvor gespeichertes Spiel laden können.
 - Der Benutzer muss ein Spielstand neuerstellen können.
 - Der Benutzer muss ein Spielstand löschen können.
 - Der Benutzer muss Spieloptionen(Sound, Steuerung, Grafik) ändern können.
- **Spielerinteraktionen**
 - Der Benutzer muss den Spieler im Spiel bewegen können.
 - Der Benutzer muss in der Lage kämpfen zu können.
 - Der Benutzer muss mit verschiedenen Umgebungsobjekten interagieren können.
 - Der Benutzer muss sich heilen können.
 - Der Benutzer muss sterben können.
 - Der Benutzer muss in der Lage sein die Storyline zu beenden.
 - Der Benutzer muss Items einsammeln können.
 - Der Benutzer muss Kisten öffnen können.
 - Der Benutzer muss Respawnpoints aktivieren/verwenden können.
 - Der Benutzer muss zwischen Räume laufen können.
- **Gegnerinteraktionen**
 - Ein Gegner muss in der Lage sein sich frei zu bewegen.
 - Ein Gegner muss den Spieler angreifen können.
 - Ein Gegner muss Sünden dropen.
 - Ein Gegner muss respawnen können.

Nicht-funktionale Anforderungen

- **Leistung**
 - Das Spiel muss flüssig laufen, ohne merkliche Verzögerungen oder Abstürze.
- **Benutzerfreundlichkeit**
 - Die Benutzeroberfläche muss intuitiv und einfach zu bedienen sein.
- **Skalierbarkeit**
 - Das System sollte erweiterbar sein, um neue Funktionen oder Inhalte hinzuzufügen.

A.3. UML-Klassendiagramm

A.3.1. Finale Version

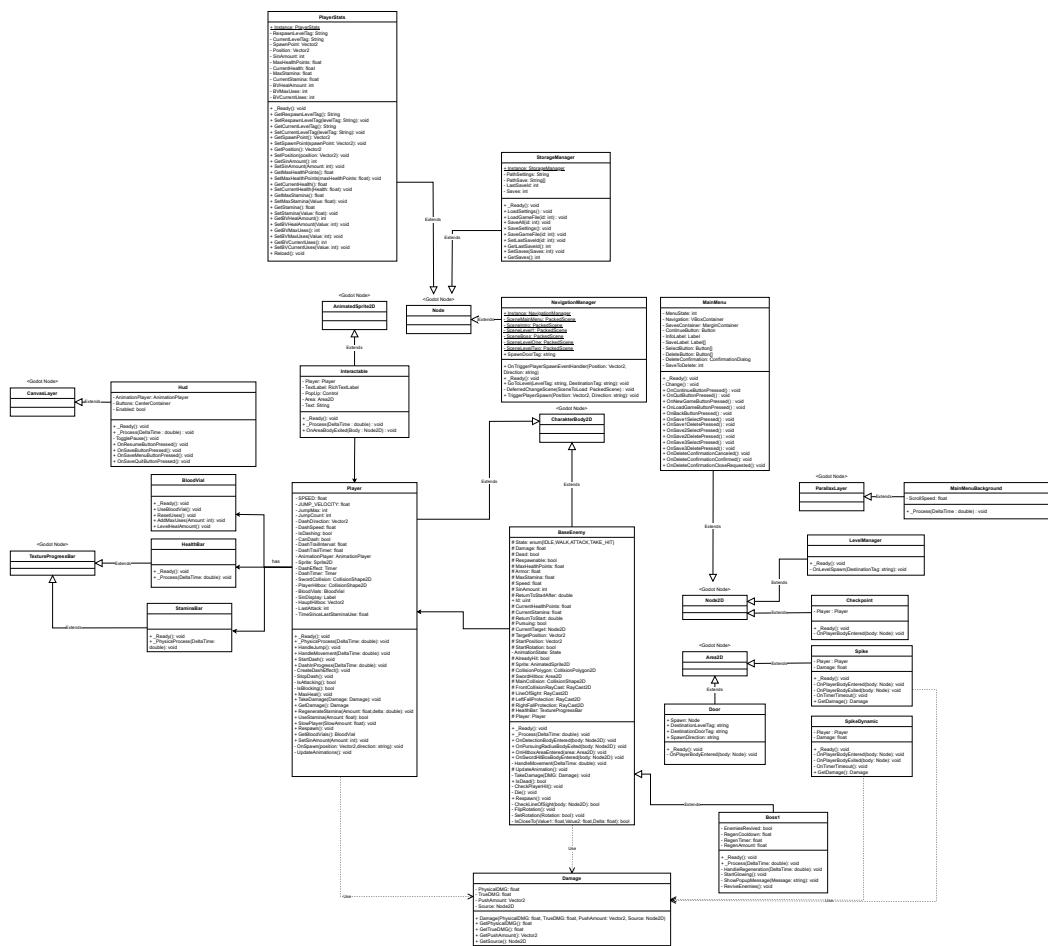


Abbildung 8: Finales UML-Klassendiagramm

A.3.2. Erste Version

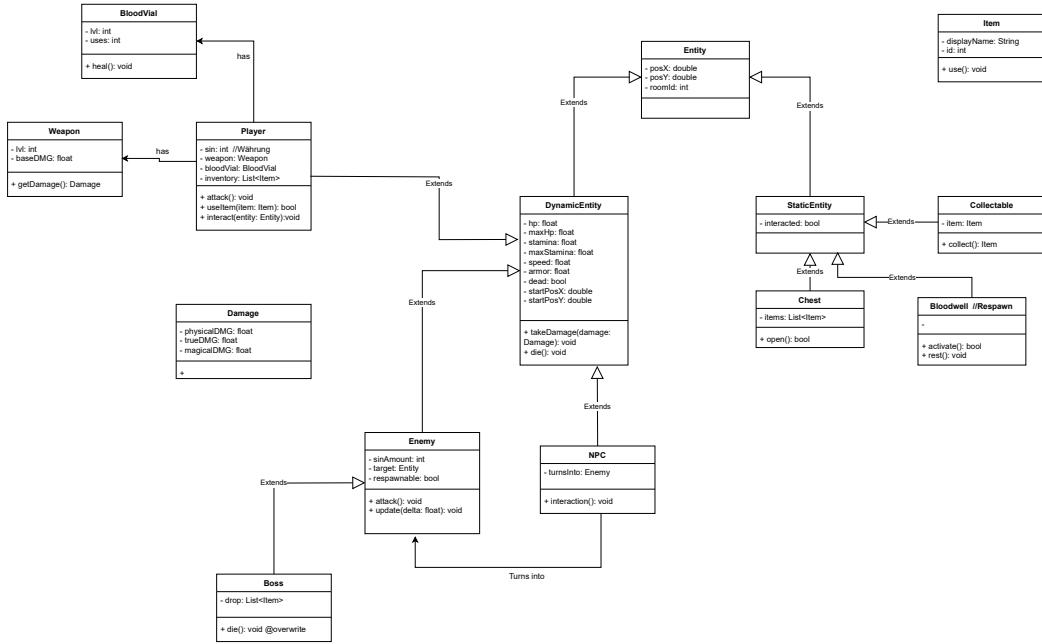


Abbildung 9: Erstes UML-Klassendiagramm

A.4. Tests

A.4.1. Unit-Tests GdMUT

```

1  using Godot;
2
3  namespace GdMUT
4  {
5      public class PlayerTestsGdMUT
6      {
7 #if TOOLS
8          private static Player SetupPlayer()
9          {
10             Player player = new Player();
11
12             // Sprite2D hinzufügen
13             var sprite = new Sprite2D();
14             sprite.Name = "Sprite2D";
15             player.AddChild(sprite);
16
17             // SwordHit und SwordCollision hinzufügen

```

```

18     var swordHit = new Node2D();
19     swordHit.Name = "SwordHit";
20     sprite.AddChild(swordHit);
21
22     var swordCollision = new CollisionShape2D();
23     swordCollision.Name = "SwordCollision";
24     swordHit.AddChild(swordCollision);
25
26     // AnimationPlayer hinzufügen
27     var animationPlayer = new AnimationPlayer();
28     animationPlayer.Name = "AnimationPlayer";
29     player.AddChild(animationPlayer);
30
31     // PlayerHitbox hinzufügen
32     var playerHitbox = new CollisionShape2D();
33     playerHitbox.Name = "PlayerHitbox";
34     player.AddChild(playerHitbox);
35
36     // DashTimer hinzufügen
37     var dashTimer = new Timer();
38     dashTimer.Name = "DashTimer";
39     player.AddChild(dashTimer);
40
41     // DashEffect hinzufügen
42     var dashEffect = new Timer();
43     dashEffect.Name = "DashEffect";
44     player.AddChild(dashEffect);
45
46     // _Ready() aufrufen, um den Spieler zu initialisieren
47     player._Ready();
48
49     return player;
50 }
51
52 // Test, ob sich der Spieler nach links bewegen kann, bei "Taste
53 // → a"
54 [CSTestFunction]
55 public static Result PlayerCanMoveRight()
56 {
57     // Spieler mit Setup-Methode initialisieren
58     Player player = SetupPlayer();
59     player.Velocity = Vector2.Zero;
60
61     // Bewegung nach links simulieren
62     Input.ActionPress("ui_left");
63     player._PhysicsProcess(0.016f); // Simulierte einen Frame
64     Input.ActionRelease("ui_left");

```

```

65     // Test: Velocity X sollte kleiner als 0 sein
66     return player.Velocity.X < 0 ? Result.Success :
67         Result.Failure;
68 }
69
70     // Test, ob sich der Spieler nach rechts bewegen kann, bei
71     // "Taste d"
72 [CSTestFunction]
73     public static Result PlayerCanMoveLeft()
74 {
75     // Spieler initialisieren
76     Player player = SetupPlayer();
77     player.Velocity = Vector2.Zero;
78
79     // Bewegung nach links simulieren
80     Input.ActionPress("ui_right");
81     player._PhysicsProcess(0.016f); // Simuliere einen Frame bei
82         // 60 FPS
83     Input.ActionRelease("ui_right");
84
85     // Test: Velocity X sollte grösser als 0 sein
86     return player.Velocity.X > 0 ? Result.Success :
87         Result.Failure;
88 }
89
90     // Test, ob sich der Spieler springen kann, bei "Taste w"
91 [CSTestFunction]
92     public static Result PlayerCanJump()
93 {
94     // Spieler initialisieren
95     Player player = SetupPlayer();
96     player.Velocity = Vector2.Zero;
97
98     // Springen simulieren
99     Input.ActionPress("ui_up");
100    player._PhysicsProcess(0.016f); // Simuliere einen Frame bei
101        // 60 FPS
102    Input.ActionRelease("ui_up");
103
104    // Test: Velocity Y sollte kleiner als 0 sein (nach oben
105    // gerichtet)
106    return player.Velocity.Y < 0 ? Result.Success :
107        Result.Failure;
108 }
109
110     // Test, ob sich der Spieler dashen kann, bei "Taste L-Shift"
111 [CSTestFunction]
112     public static Result PlayerCanDash()

```

```

106    {
107        Player player = SetupPlayer();
108        player.Velocity = Vector2.Zero;
109
110        // Simuliere einen Dash
111        Input.ActionPress("dash");
112        player._PhysicsProcess(0.016f);
113        Input.ActionRelease("dash");
114
115        // Test: Spieler sollte sich mit Dash-Geschwindigkeit
116        → bewegen
117        return player.Velocity.Length() > 100f ? Result.Success :
118        → Result.Failure;
119    }
120
121    // Test, ob sich der Spieler blocken kann, bei "Taste Space".
122    → NUR GESCHWINDIGKEITSTEST
123    [CSTestFunction]
124    public static Result PlayerCanBlock()
125    {
126        Player player = SetupPlayer();
127
128        // Simuliere Blocken
129        Input.ActionPress("block");
130        player._PhysicsProcess(0.016f);
131
132        // Test: Velocity sollte 0 sein (Spieler bleibt stehen)
133        return player.Velocity.X == 0 ? Result.Success :
134        → Result.Failure;
135    }
136 #endif
137 }
138 }
```

A.4.2. Unit-Tests GdUnit4

Test testen:

```

1 using GdUnit4;
2 using Godot;
3
4
5 [TestSuite]
6 public class TestTestsGdUnit {
7
8     [TestCase]
9     public void Test(){
10        Assertions.AssertBool(true).IsEqual(true);
11    }
12 }
```

```
11     }
12 }
```

Player-Tests:

```
1  using GdUnit4;
2  using Godot;
3  using NUnit.Framework.Internal;
4  using System.Threading;
5  using System.Threading.Tasks;
6
7  // Testklasse für den Player
8
9  [TestSuite]
10 public class PlayerTestGdUnit
11 {
12     private ISceneRunner Runner;
13     private Player Player;
14     private SemaphoreSlim Semaphore = new SemaphoreSlim(1, 1);
15
16     [BeforeTest]
17     public async Task Setup()
18     {
19         Runner = ISceneRunner.Load("res://scenes/level1.tscn");
20         await Runner.AwaitIdleFrame();
21
22         Assertions.AssertThat(Runner).IsNotNull();
23
24         Player = GD.Load<PackedScene>("res://scenes/player.tscn").Instantiate<Player>();
25         Player.Name = "Player";
26         Runner.Scene().AddChild(Player);
27         Player.Position = new Vector2(150, 288);
28
29         Assertions.AssertThat(Player).IsNotNull();
30         await Runner.AwaitIdleFrame();
31     }
32
33     [TestCase]
34     [RequireGodotRuntime]
35     public async Task TestWMovement()
36     {
37         // Test for W
38         await TestMovement(Key.W, Vector2.Up);
39
40         await Setup();
41 }
```

```

42
43 [TestCase]
44 [RequireGodotRuntime]
45 public async Task TestAMovement()
46 {
47     // Test for A
48     await TestMovementApprox(Key.A, Vector2.Left);
49
50     await Setup();
51 }
52
53 [TestCase]
54 [RequireGodotRuntime]
55 public async Task TestDMovement()
56 {
57     // Test for D
58     await TestMovementApprox(Key.D, Vector2.Right);
59
60     await Setup();
61 }
62
63 [TestCase]
64 [RequireGodotRuntime]
65 public async Task TestDashWithNoDirectionMovement()
66 {
67     // Test L-Shift for dash if no direction is pressed
68     Player.Position = new Vector2(120, 288);
69
70     await TestMovement(Key.Shift, Vector2.Zero);
71
72     await Setup();
73 }
74
75 [TestCase]
76 [RequireGodotRuntime]
77 public async Task TestDashWithDirectionLeftMovement()
78 {
79     // Test L-Shift for dash if Left direction is pressed
80
81     Runner.SimulateKeyPress(Key.A);
82     await Runner.SimulateFrames(1, 1);
83     await TestMovementApprox(Key.Shift, Vector2.Left);
84
85     await Setup();
86 }
87
88 [TestCase]
89 [RequireGodotRuntime]

```

```

90     public async Task TestDashWithDirectionRightMovement()
91     {
92         // Test L-Shift for dash if Right direction is pressed
93
94         Runner.SimulateKeyPress(Key.D);
95         await Runner.SimulateFrames(1, 1);
96         await TestMovementApprox(Key.Shift, Vector2.Right);
97
98         await Setup();
99     }
100
101    [TestCase]
102    [RequireGodotRuntime]
103    public async Task TestDashWithDirectionUpMovement()
104    {
105        // Test L-Shift for dash if "W" direction is pressed
106
107        Runner.SimulateKeyPress(Key.W);
108        await Runner.SimulateFrames(1, 1);
109        await TestMovementApprox(Key.Shift, Vector2.Up);
110
111        await Setup();
112    }
113
114    [TestCase]
115    [RequireGodotRuntime]
116    public async Task TestDashWithLightAttackMovement()
117    {
118        // Test L-Shift for dash if light attack is pressed
119        Player.Position = new Vector2(120, 288); // Reset position to
120        // avoid interference
121
122        Runner.SimulateMouseButtonPress(MouseButton.Left);
123        await Runner.SimulateFrames(1, 100);
124        await TestMovementApprox(Key.Shift, Vector2.Zero);
125
126        await Setup();
127    }
128
129    [TestCase]
130    [RequireGodotRuntime]
131    public async Task TestDashWithHeavyAttackMovement()
132    {
133        // Test L-Shift for dash if heavy attack is pressed
134        Player.Position = new Vector2(120, 288); // Reset position to
135        // avoid interference
136
137        Runner.SimulateMouseButtonPress(MouseButton.Right);

```

```

136     await Runner.SimulateFrames(1, 100);
137     await TestMovementApprox(Key.Shift, Vector2.Zero);
138
139     await Setup();
140 }
141
142 [TestCase]
143 [RequireGodotRuntime]
144 public async Task TestHeavyAttack()
145 {
146     Player.Position = new Vector2(120, 288);
147     // Test heavy attack with right mouse button
148
149     Runner.SimulateMouseButtonPress(MouseButton.Right);
150     await Runner.SimulateFrames(1, 100);
151
152     Assertions.AssertThat(Player.IsAttacking()).IsTrue();
153
154     await Setup(); // Reset player state after attack
155 }
156
157 [TestCase]
158 [RequireGodotRuntime]
159 public async Task TestLightAttack()
160 {
161     // Test light attack with right mouse button
162
163     Player.Position = new Vector2(120, 288); // Reset position to
164     // avoid interference
165
166     Runner.SimulateMouseButtonPress(MouseButton.Left);
167     await Runner.SimulateFrames(1, 100);
168
169     Assertions.AssertThat(Player.IsAttacking()).IsTrue();
170
171     await Setup();
172 }
173
174 [TestCase]
175 [RequireGodotRuntime]
176 public async Task TestBlock()
177 {
178     // Test block with space Key
179
180     Runner.SimulateKeyPress(Key.Space);
181     await Runner.SimulateFrames(5, 100);
182     Assertions.AssertThat(Player.IsBlocking()).IsTrue();

```

```

183     await Setup();
184 }
185
186 [TestCase]
187 [RequireGodotRuntime]
188 public async Task TestMaxHeal()
189 {
190     // Test if player has maximum health
191     float InitialHealth = PlayerStats.Instance.GetCurrentHealth();
192
193     PlayerStats.Instance.SetCurrentHealth(50.0f);
194     Player.MaxHeal();
195     await Runner.SimulateFrames(1, 100);
196     Assertions.AssertThat(PlayerStats.Instance.GetCurrentHealth()).Is
197         → sEqual(InitialHealth);
198     await Setup();
199 }
200
201 [TestCase]
202 [RequireGodotRuntime]
203 public async Task TestTakeDMG()
204 {
205     // Test if player has maximum health
206     Spike Spike = GD.Load<PackedScene>("res://scenes/spike.tscn").In
207         → stantiate<Spike>();
208     Runner.Scene().AddChild(Spike);
209     float InitialHealth = PlayerStats.Instance.GetCurrentHealth();
210
211     Damage Damage = Spike.GetDamage(); // Get 10 damage from Spike
212     Player.TakeDamage(Damage);
213
214     await Runner.SimulateFrames(1, 1);
215     Assertions.AssertThat(PlayerStats.Instance.GetCurrentHealth()).Is
216         → sEqual(InitialHealth -
217             → 10.0f);
218     await Setup();
219 }
220
221 [TestCase]
222 [RequireGodotRuntime]
223 public async Task TestTakeDMGWhileBlocking()
224 {
225     // Test if player has maximum health
226     BaseEnemy BaseEnemy = GD.Load<PackedScene>("res://scenes/base_en
227         → emy.tscn").Instantiate<BaseEnemy>();
228     Runner.Scene().AddChild(BaseEnemy);
229     PlayerStats.Instance.SetCurrentHealth(100.0f); // Set initial
230         → health

```

```

225
226     Runner.SimulateKeyPress(Key.Space); // Simulate blocking
227     await Runner.SimulateFrames(5, 100);
228     Damage DMG = new Damage(50.0f, 0, Vector2.Zero, BaseEnemy);
229     Player.TakeDamage(DMG);
230
231     Assertions.AssertThat(PlayerStats.Instance.GetCurrentHealth()).IsEqual
232         → sEqual(100.0f); // Health should remain the same due to
233         → blocking
234         await Setup();
235     }
236
237     [TestCase]
238     [RequireGodotRuntime]
239     public async Task TestRegenerateStamina()
240     {
241         // Test stamina regeneration
242         PlayerStats.Instance.SetStamina(50.0f); // Set initial stamina
243         float StaminaRegenRate = 10f;
244         float Delta = 0.5f; // Simulate half a second
245
246         Player.RegenerateStamina(StaminaRegenRate, Delta);
247         await Runner.SimulateFrames(5, 100);
248
249         Assertions.AssertThat(PlayerStats.Instance.GetStamina()).IsEqual
250             → (100.0f); // Assuming max stamina is
251             → 100
252
253         await Setup();
254     }
255
256     [TestCase]
257     [RequireGodotRuntime]
258     public async Task TestUseBloodVial()
259     {
260         // Test using a blood vial
261         PlayerStats.Instance.SetCurrentHealth(50.0f); // Set initial
262             → health
263
264         await Runner.AwaitIdleFrame();
265         Runner.SimulateKeyPress(Key.Q); // Simulate using blood vial
266         Assertions.AssertThat(PlayerStats.Instance.GetCurrentHealth()).IsEqual
267             → sEqual(50.0f);
268
269         await Setup();
270     }
271
272     [TestCase]

```

```

267 [RequireGodotRuntime]
268 public async Task TestUseBloodVialWithMaxHealth()
269 {
270     // Test using a blood vial when health is already at maximum
271     PlayerStats.Instance.SetCurrentHealth(100.0f); // Set initial
272     // health
273
274     await Runner.AwaitIdleFrame();
275     Runner.SimulateKeyPress(Key.Q); // Simulate using blood vial
276     Assertions.AssertThat(PlayerStats.Instance.GetCurrentHealth()).Is
277     // Equal(100.0f);
278
279     await Setup();
280 }
281
282 private async Task TestMovement(Key Key, Vector2 EcpectedDirection)
283     //KeyboardInput
284 {
285     await Semaphore.WaitAsync(); // wait for semaphore
286
287     try
288     {
289         Vector2 InitialPosition = Player.Position;
290
291         Runner.SimulateKeyPress(Key);
292         await Runner.SimulateFrames(1, 100);
293         Runner.SimulateKeyRelease(Key);
294         await Runner.AwaitIdleFrame();
295
296         Vector2 NewPosition = Player.Position;
297
298         Vector2 MovementDirection = (NewPosition -
299             InitialPosition).Normalized();
300
301         Assertions.AssertThat(MovementDirection).IsEqual(EcpectedDir
302             ection);
303     }
304     finally
305     {
306         Semaphore.Release(); // release the semaphore when task is
307             // done
308     }
309 }
310
311 private async Task TestMovementApprox(Key Key, Vector2
312     EcpectedDirection) //KeyboardInput
313 {
314     await Semaphore.WaitAsync(); // wait for semaphore

```

```

308
309     try
310     {
311         Vector2 initialPosition = Player.Position;
312
313         Runner.SimulateKeyPress(Key);
314         await Runner.SimulateFrames(1, 100);
315         Runner.SimulateKeyRelease(Key);
316         await Runner.AwaitIdleFrame();
317
318         Vector2 newPosition = Player.Position;
319
320         Vector2 MovementDirection = (newPosition -
321             → initialPosition).Normalized();
322
323         Assertions.AssertThat(MovementDirection).IsEqualApprox(Epsilon.J
324             → tedDirection, new Vector2(0.1f,
325             → 0.1f));
326
327     }
328
329 }
```

BaseEnemy-Tests:

```

1  using Godot;
2  using System;
3  using GdUnit4;
4  using System.Threading.Tasks;
5  using Microsoft.CodeAnalysis.CSharp;
6
7
8  [TestSuite]
9  public class BaseEnemyTestGdUnit
10 {
11
12     private BaseEnemy Enemy;
13     private ISceneRunner Runner;
14
15     [BeforeTest]
16     public async Task Setup()
17     {
18 }
```

```

19     Runner = ISceneRunner.Load("res://scenes/level1.tscn");
20     await Runner.AwaitIdleFrame();
21
22     Enemy = GD.Load<PackedScene>("res://scenes/base_enemy.tscn").Instantiate();
23     Runner.Scene().AddChild(Enemy);
24
25     Enemy.Position = new Vector2(300, 288);
26     Enemy.StartPosition = new Vector2(300, 288);
27     await Runner.AwaitIdleFrame();
28
29
30 }
31
32 public async Task Reset()
33 {
34
35     Enemy.Position = new Vector2(300, 288);
36     Enemy.StartPosition = new Vector2(300, 288);
37     Enemy.TargetPosition = Vector2.Infinity;
38     Enemy.Pursuing = false;
39     Enemy.Dead = false;
40     await Runner.AwaitIdleFrame();
41
42 }
43
44
45 [TestCase]
46 [RequireGodotRuntime]
47 public async Task TestMovementIfDead()
48 {
49     await Reset();
50
51     Vector2 InitialPosition = Enemy.Position;
52     Enemy.TargetPosition = new Vector2(130, 288);
53     Enemy.Pursuing = true;
54     Enemy.Dead = true;
55     await Runner.SimulateFrames(5, 100);
56     await Runner.AwaitIdleFrame();
57     Vector2 MovementDirection = (Enemy.Position - InitialPosition).Normalized();
58     Assertions.AssertThat(MovementDirection).IsEqual(Vector2.Zero);
59
60
61 }
62
63 [TestCase]
64 [RequireGodotRuntime]

```

```

65     public async Task TestMovementIfNotPursuing()
66     {
67         await Reset();
68
69         Vector2 InitialPosition = Enemy.Position;
70         Enemy.TargetPosition = new Vector2(130, 288);
71         Enemy.Pursuing = false;
72         await Runner.SimulateFrames(5, 100);
73         await Runner.AwaitIdleFrame();
74         Vector2 MovementDirection = (Enemy.Position -
75             → InitialPosition).Normalized();
76         Assertions.AssertThat(MovementDirection).IsEqual(Vector2.Zero);
77
78     }
79
80     [TestCase]
81     [RequireGodotRuntime]
82     public async Task TestMovementIfNoTargetPosition()
83     {
84         await Reset();
85
86         Vector2 InitialPosition = Enemy.Position;
87         Enemy.TargetPosition = Vector2.Infinity;
88         Enemy.Pursuing = true;
89         await Runner.SimulateFrames(5, 100);
90         await Runner.AwaitIdleFrame();
91         Vector2 MovementDirection = (Enemy.Position -
92             → InitialPosition).Normalized();
93         Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.Z)
94             → ero, new Vector2(0.1f,
95             → 0.1f));
96
97
98     [TestCase]
99     [RequireGodotRuntime]
100    public async Task TestMovementToStartPositionLeft()
101    {
102        await Reset();
103
104        Vector2 InitialPosition = Enemy.Position;
105        Enemy.TargetPosition = new Vector2(130, 288);
106        Enemy.StartPosition = new Vector2(130, 288);
107        Enemy.Pursuing = false;
108        Enemy.ReturnToStart = -1;

```

```

109     await Runner.SimulateFrames(5, 100);
110     Vector2 MovementDirection = (Enemy.Position -
111         → InitialPosition).Normalized();
112     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.L_];
113         → eft, new Vector2(0.1f,
114         → 0.1f));
115
116 }
117
118 [TestCase]
119 [RequireGodotRuntime]
120 public async Task TestMovementToTargetPositionRight()
121 {
122     await Reset();
123
124     Vector2 InitialPosition = Enemy.Position;
125     Enemy.TargetPosition = new Vector2(450, 288);
126     Enemy.StartPosition = new Vector2(450, 288);
127     Enemy.Pursuing = false;
128     Enemy.ReturnToStart = -1;
129     await Runner.SimulateFrames(5, 100);
130     Vector2 MovementDirection = (Enemy.Position -
131         → InitialPosition).Normalized();
132     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.R_];
133         → ight, new Vector2(0.1f,
134         → 0.1f));
135
136 }
137
138 [TestCase]
139 [RequireGodotRuntime]
140 public async Task TestTakeDamageIfDead()
141 {
142     await Reset();
143
144     Enemy.Dead = true;
145     Vector2 InitialPosition = Enemy.Position;
146     float InitialHealth = Enemy.CurrentHealthPoints;
147     Damage Damage = new Damage(10, 10, Vector2.Zero, null);
148     Enemy.TakeDamage(Damage);
149     await Runner.AwaitIdleFrame();
150     Assertions.AssertThat(Enemy.CurrentHealthPoints).IsEqual(Initial_];
151         → Health);
152     Vector2 MovementDirection = (Enemy.Position -
153         → InitialPosition).Normalized();

```

```

149     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.Z)
150         ↵ ero, new Vector2(0.1f,
151         ↵ 0.1f));
152
153 }
154
155 [TestCase]
156 [RequireGodotRuntime]
157 public async Task TestTakeDamageOnlyPhysical()
158 {
159
160     await Reset();
161
162     Vector2 InitialPosition = Enemy.Position;
163     float InitialHealth = Enemy.CurrentHealthPoints;
164     Damage Damage = new Damage(30, 0, Vector2.Zero, null);
165     Enemy.TakeDamage(Damage);
166     await Runner.AwaitIdleFrame();
167     float CalculatedHealth = InitialHealth - Damage.GetPhysicalDMG()
168         ↵ * (1 - Enemy.Armor / 100.0f) - Damage.GetTrueDMG();
169     Assertions.AssertThat(Enemy.CurrentHealthPoints).IsEqual(Calcula
170         ↵ tedHealth);
171
172     Vector2 MovementDirection = (Enemy.Position -
173         ↵ InitialPosition).Normalized();
174     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.Z)
175         ↵ ero, new Vector2(0.1f,
176         ↵ 0.1f));
177
178 }
179
180
181 [TestCase]
182 [RequireGodotRuntime]
183 public async Task TestTakeDamageOnlyTrue()
184 {
185
186     await Reset();
187
188     Vector2 InitialPosition = Enemy.Position;
189     float InitialHealth = Enemy.CurrentHealthPoints;
190     Damage Damage = new Damage(0, 30, Vector2.Zero, null);
191     Enemy.TakeDamage(Damage);
192     await Runner.AwaitIdleFrame();
193     float CalculatedHealth = InitialHealth - Damage.GetPhysicalDMG()
194         ↵ * (1 - Enemy.Armor / 100.0f) - Damage.GetTrueDMG();
195     Assertions.AssertThat(Enemy.CurrentHealthPoints).IsEqual(Calcula
196         ↵ tedHealth);

```

```

187     Vector2 MovementDirection = (Enemy.Position -
188         → InitialPosition).Normalized();
189     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.Z_]
190         → ero, new Vector2(0.1f,
191         → 0.1f));
192 }
193
194 [TestCase]
195 [RequireGodotRuntime]
196 public async Task TestTakeDamageOnlyPushRight()
197 {
198
199     await Reset();
200
201     Vector2 InitialPosition = Enemy.Position;
202     float InitialHealth = Enemy.CurrentHealthPoints;
203     Damage Damage = new Damage(0, 0, new Vector2(10f, 0f), null);
204     Enemy.TakeDamage(Damage);
205     await Runner.AwaitIdleFrame();
206     Assertions.AssertThat(Enemy.CurrentHealthPoints).IsEqual(Initial_]
207         → Health);
208     Vector2 MovementDirection = (Enemy.Position -
209         → InitialPosition).Normalized();
210     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.R_]
211         → ight, new Vector2(0.1f,
212         → 0.1f));
213
214 }
215
216 [TestCase]
217 [RequireGodotRuntime]
218 public async Task TestTakeDamageOnlyPushLeft()
219 {
220
221     await Reset();
222
223     Vector2 InitialPosition = Enemy.Position;
224     float InitialHealth = Enemy.CurrentHealthPoints;
225     Damage Damage = new Damage(0, 0, new Vector2(-10f, 0f), null);
226     Enemy.TakeDamage(Damage);
227     await Runner.AwaitIdleFrame();
228     Assertions.AssertThat(Enemy.CurrentHealthPoints).IsEqual(Initial_]
229         → Health);
230     Vector2 MovementDirection = (Enemy.Position -
231         → InitialPosition).Normalized();

```

```

224     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.L_]
225         ↵   eft, new Vector2(0.1f,
226         ↵   0.1f));
227
228     }
229
230     [TestCase]
231     [RequireGodotRuntime]
232     public async Task TestTakeDamageCombined()
233     {
234
235         await Reset();
236
237         Vector2 InitialPosition = Enemy.Position;
238         float InitialHealth = Enemy.CurrentHealthPoints;
239         Damage Damage = new Damage(30, 10, new Vector2(-10f, 0f), null);
240         Enemy.TakeDamage(Damage);
241         await Runner.AwaitIdleFrame();
242         float CalculatedHealth = InitialHealth - Damage.GetPhysicalDMG()
243             ↵   * (1 - Enemy.Armor / 100.0f) - Damage.GetTrueDMG();
244         Assertions.AssertThat(Enemy.CurrentHealthPoints).IsEqual(Calcula_]
245             ↵   tedHealth);
246         Vector2 MovementDirection = (Enemy.Position -
247             ↵   InitialPosition).Normalized();
248         Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.L_]
249             ↵   eft, new Vector2(0.1f,
250             ↵   0.1f));
251
252     }
253
254     [TestCase]
255     [RequireGodotRuntime]
256     public async Task TestTakeDamageTillDead()
257     {
258
259         await Reset();
260
261         Vector2 InitialPosition = Enemy.Position;
262         float InitialHealth = Enemy.CurrentHealthPoints;
263         Damage Damage = new Damage(0, 200, Vector2.Zero, null);
264         Enemy.TakeDamage(Damage);
265         await Runner.AwaitIdleFrame();
266         float CalculatedHealth = InitialHealth - Damage.GetPhysicalDMG()
267             ↵   * (1 - Enemy.Armor / 100.0f) - Damage.GetTrueDMG();
268         Assertions.AssertThat(Enemy.CurrentHealthPoints).IsEqual(Calcula_]
269             ↵   tedHealth);
270         Vector2 MovementDirection = (Enemy.Position -
271             ↵   InitialPosition).Normalized();

```

```

262     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.Z |
263         ↳ ero, new Vector2(0.1f,
264         ↳ 0.1f));
265     Assertions.AssertThat(Enemy.Dead).IsTrue();
266
267
268     [TestCase]
269     [RequireGodotRuntime]
270     public async Task TestDie()
271     {
272
273         await Reset();
274
275         Enemy.Die();
276         await Runner.AwaitIdleFrame();
277
278         Assertions.AssertThat(Enemy.Dead).IsTrue();
279         Assertions.AssertThat(Enemy.Velocity).IsEqual(Vector2.Zero);
280         Assertions.AssertThat(Enemy.MainCollision.Get(CollisionShape2D.P |
281             ↳ propertyName.Disabled)).IsTrue();
282     }
283
284 }
```

Spike-Tests:

```

1  using Godot;
2  using GdUnit4;
3  using System.Threading.Tasks;
4  using static GdUnit4.Assertions;
5
6
7  [TestSuite]
8  public class SpikeTestGdUnit
9  {
10     private ISceneRunner Runner;
11     private Spike Spike;
12     private Player Player;
13
14     [BeforeTest]
15     public async Task Setup()
16     {
17         Runner = ISceneRunner.Load("res://scenes/level1.tscn");
18         await Runner.AwaitIdleFrame();
```

```

19     Assertions.AssertThat(Runner).IsNotNull();
20     // Laedt die Spike-Szene und instanziiert sie
21     Spike = GD.Load<PackedScene>("res://scenes/spike.tscn").Instantiate();
22     // Laedt die Spieler-Szene und instanziiert sie
23     Player = GD.Load<PackedScene>("res://scenes/player.tscn").Instantiate();
24     Player.Name = "Player"; // Wichtig fuer die Erkennung im Spiel
25     Runner.Scene().AddChild(Player);
26
27     Runner.Scene().AddChild(Spike);
28
29     // Initialisiert den ISceneRunner mit der neuen Szene
30     // Runner = ISceneRunner.Load(scene, true);
31
32     // Warte einen Frame, damit die Szene vollstaendig geladen ist
33
34 }
35
36
37 [TestCase]
38 [RequireGodotRuntime]
39 public async Task
40     TestWhenPlayerEntersSpikePlayerHealthShouldDecrease()
41 {
42     // Test if player has maximum health
43     Runner.Scene().AddChild(Spike);
44     float InitialHealth = PlayerStats.Instance.GetCurrentHealth();
45
46     await Runner.SimulateFrames(1, 1);
47     Assertions.AssertThat(PlayerStats.Instance.GetCurrentHealth()).IsEqual(InitialHealth -
48         Spike.GetDamage().GetTrueDMG());
49     await Setup();
50 }
51
52 [TestCase]
53 [RequireGodotRuntime]
54 public async Task TestPlayerTakesRepeatedDamageWhileOnSpike()
55 {
56     // Setze die Startgesundheit des Spielers
57     PlayerStats.Instance.SetCurrentHealth(100.0f);
58     var InitialHealth = PlayerStats.Instance.GetCurrentHealth();
59
60     // Positioniere den Spieler auf dem Spike
61     Player.GlobalPosition = Spike.GlobalPosition;
62
63     // Simuliere das Betreten des Spikes

```

```

62     var Area2D = Spike.GetNode<Area2D>("StaticBody2D/Area2D");
63     Runner.Scene().AddChild(Area2D);
64     Area2D.EmitSignal(Area2D.SignalName.BodyEntered, Player);
65
66
67     // Warte auf den ersten Schaden
68     await Runner.AwaitIdleFrame();
69
70     // Simuliere eine Verzögerung, um wiederholten Schaden zu
71     // ermöglichen
72     await Runner.SimulateFrames(1100);
73
74     // Überprüfe, ob die Gesundheit des Spielers weiter reduziert
75     // wurde
76     var NewHealth = PlayerStats.Instance.GetCurrentHealth();
77     AssertThat(InitialHealth -
78     // Spike.GetDamage().GetTrueDMG()).IsGreater(NewHealth);
79     await Setup();
80 }
81
82 [TestCase]
83 [RequireGodotRuntime]
84 public async Task TestTimerStopsWhenPlayerLeavesSpike()
85 {
86     // Setze die Startgesundheit und positioniere den Spieler
87     PlayerStats.Instance.SetCurrentHealth(100.0f);
88     Player.Position = Spike.Position;
89
90
91     // Simuliere das Betreten und sofortige Verlassen des Spikes
92     var Area2D = Spike.GetNode<Area2D>("StaticBody2D/Area2D");
93     Runner.Scene().AddChild(Area2D);
94
95     Area2D.EmitSignal(Area2D.SignalName.BodyEntered, Player);
96     await Runner.AwaitIdleFrame();
97
98     Player.Position = new Vector2(1000, 1000); // Bewege den Spieler
99     // weg vom Spike
100
101    Area2D.EmitSignal(Area2D.SignalName.BodyExited, Player);
102    await Runner.AwaitIdleFrame();
103
104    // Speichere die Gesundheit nach dem Verlassen
105    var HealthAfterExit = PlayerStats.Instance.GetCurrentHealth();
106
107    // Warte eine Weile, um sicherzustellen, dass kein weiterer
108    // Schaden auftritt
109    await Runner.SimulateFrames(1, 1);

```

```

105     // Ueberpruefe, ob die Gesundheit unveraendert geblieben ist
106     AssertThat(PlayerStats.Instance.GetCurrentHealth()).IsEqual(Heal ↵
107         → thAfterExit);
108     await Setup();
109 }
110
111 [TestCase]
112 [RequireGodotRuntime]
113 public async Task TestTimerNotStartedForNonPlayer()
114 {
115     // Erstelle ein anderes Objekt, das kein Spieler ist
116     var OtherObject = new CharacterBody2D();
117     Runner.Scene().AddChild(OtherObject);
118
119     // Hole den Timer aus dem Spike
120     var Timer = Spike.GetNode<Timer>("StaticBody2D/Area2D/Timer");
121
122     // Simuliere, dass das andere Objekt den Spike betritt
123     var Area2D = Spike.GetNode<Area2D>("StaticBody2D/Area2D");
124     Area2D.EmitSignal(Area2D.SignalName.BodyEntered, OtherObject);
125     await Runner.AwaitIdleFrame();
126
127     // Ueberpruefe, ob der Timer nicht gestartet wurde
128     AssertThat(Timer.IsStopped()).IsTrue();
129     await Setup();
130 }
131
132 [TestCase]
133 [RequireGodotRuntime]
134 public async Task
135     → TestWhenBaseEnemyEntersSpikeBaseEnemyHealthShouldNotChange()
136 {
137     // Erstelle einen Gegner und setze seine Gesundheit
138     var BaseEnemy = new BaseEnemy();
139     BaseEnemy.CurrentHealthPoints = 50.0f;
140     Runner.Scene().AddChild(BaseEnemy);
141     var InitialHealth = BaseEnemy.CurrentHealthPoints;
142
143     // Simuliere, dass der Gegner den Spike betritt
144     var Area2D = Spike.GetNode<Area2D>("StaticBody2D/Area2D");
145     Area2D.EmitSignal(Area2D.SignalName.BodyEntered, BaseEnemy);
146     await Runner.AwaitIdleFrame();
147
148     // Ueberpruefe, ob die Gesundheit des Gegners unveraendert ist
149     AssertThat(BaseEnemy.CurrentHealthPoints).IsEqual(InitialHealth);
150     await Setup();
151 }
```

150 }

Checkpoint-Tests:

```
1  using Godot;
2  using GdUnit4;
3  using System.Threading.Tasks;
4  using static GdUnit4.Assertions;
5
6  [TestSuite]
7  public class CheckpointTestGdUnit
8  {
9      private ISceneRunner Runner;
10     private Checkpoint Checkpoint;
11     private Player Player;
12
13     [BeforeTest]
14     public async Task Setup()
15     {
16         Runner = ISceneRunner.Load("res://scenes/level_one.tscn", true);
17         Checkpoint = new Checkpoint();
18         Runner.Scene().AddChild(Checkpoint);
19         Player = GD.Load<PackedScene>("res://scenes/player.tscn").Instantiate();
20         Runner.Scene().AddChild(Player);
21         var BloodVial = new BloodVial();
22         Player.AddChild(BloodVial);
23         var BloodVialsField = Player.GetType().GetField("BloodVials",
24             System.Reflection.BindingFlags.NonPublic |
25             System.Reflection.BindingFlags.Instance);
26         if (BloodVialsField != null)
27             BloodVialsField.SetValue(Player, BloodVial);
28         await Runner.AwaitIdleFrame();
29     }
30
31     [TestCase]
32     [RequireGodotRuntime]
33     public async Task TestPlayerSetsSpawnpointOnCheckpointEnter()
34     {
35         // Test if player spawnpoint is set correctly when entering
36         // checkpoint
37         Checkpoint.GlobalPosition = new Vector2(100, 200);
38         PlayerStats.Instance.SetSpawnPoint(Vector2.Zero);
39         Checkpoint.Call("OnPlayerBodyEntered", Player);
40         AssertThat(PlayerStats.Instance.GetSpawnPoint()).IsEqual(new
41             Vector2(100, 200));
42         await Setup();
```

```

39 }
40
41 [TestCase]
42 [RequireGodotRuntime]
43 public async Task TestPlayerIsHealedAndStaminaRestoredOnCheckpoint()
44 {
45     // Test if player health and stamina are restored when entering
46     // checkpoint
47     PlayerStats.Instance.SetCurrentHealth(10);
48     PlayerStats.Instance.SetStamina(5);
49     Checkpoint.Call("OnPlayerBodyEntered", Player);
50     AssertThat(PlayerStats.Instance.GetCurrentHealth()).IsEqual(PlayerStat
51     ↵ erStats.Instance.GetMaxHealthPoints());
52     AssertThat(PlayerStats.Instance.GetStamina()).IsEqual(PlayerStat
53     ↵ s.Instance.GetMaxStamina());
54     await Setup();
55 }
56
57 [TestCase]
58 [RequireGodotRuntime]
59 public async Task TestPlayerBloodVialsAreResetOnCheckpoint()
60 {
61     // Test if player blood vials are reset when entering
62     // checkpoint
63     var BloodVials = Player.GetBloodVials();
64     BloodVials.ResetUses();
65     Checkpoint.Call("OnPlayerBodyEntered", Player);
66     AssertThat(PlayerStats.Instance.GetBVCurrentUses()).IsEqual(PlayerStat
67     ↵ erStats.Instance.GetBVMaxUses());
68     await Setup();
69 }
70
71 [TestCase]
72 [RequireGodotRuntime]
73 public async Task TestNoEffectWhenNonPlayerEntersCheckpoint()
74 {
75     // Test if entering checkpoint with non-player object has no
76     // effect
77     var dummy = new Node2D();
78     PlayerStats.Instance.SetSpawnPoint(Vector2.Zero);
79     PlayerStats.Instance.SetCurrentHealth(10);
80     PlayerStats.Instance.SetStamina(5);
81     Runner.Scene().AddChild(dummy);
82     Checkpoint.Call("OnPlayerBodyEntered", dummy);
83     AssertThat(PlayerStats.Instance.GetSpawnPoint()).IsEqual(Vector2
84     ↵ .Zero);
85     AssertThat(PlayerStats.Instance.GetCurrentHealth()).IsEqual(10);
86     AssertThat(PlayerStats.Instance.GetStamina()).IsEqual(5);

```

```

80         await Setup();
81     }
82
83     [TestCase]
84     [RequireGodotRuntime]
85     public async Task TestMultipleCheckpointsOverwriteSpawnpoint()
86     {
87         // Test if multiple checkpoints overwrite player spawnpoint
88         // correctly
89         var Checkpoint2 = new Checkpoint();
90         Runner.Scene().AddChild(Checkpoint2);
91         Checkpoint.GlobalPosition = new Vector2(100, 200);
92         Checkpoint2.GlobalPosition = new Vector2(300, 400);
93         Checkpoint.Call("OnPlayerBodyEntered", Player);
94         AssertThat(PlayerStats.Instance.GetSpawnPoint()).IsEqual(new
95             // Vector2(100, 200));
96         Checkpoint2.Call("OnPlayerBodyEntered", Player);
97         AssertThat(PlayerStats.Instance.GetSpawnPoint()).IsEqual(new
98             // Vector2(300, 400));
99         await Setup();
100    }
101}

```

Boss-Tests:

```

1  using Godot;
2  using System;
3  using GdUnit4;
4  using System.Threading.Tasks;
5  using Microsoft.CodeAnalysis.CSharp;
6
7
8  [TestSuite]
9  public class BossTestGdUnit
10 {
11
12     private Boss1 Boss;
13     private ISceneRunner Runner;
14
15     [BeforeTest]
16     public async Task Setup()
17     {
18
19         Runner = ISceneRunner.Load("res://scenes/bossRoom.tscn");
20         await Runner.AwaitIdleFrame();
21
22         Boss = Runner.Scene().GetNode<Boss1>("enemies/Boss");

```

```

23     // Runner.Scene().AddChild(Boss);
24
25     Boss.Position = new Vector2(300, 288);
26     Boss.StartPosition = new Vector2(300, 288);
27     await Runner.AwaitIdleFrame();
28
29 }
30
31 public async Task Reset()
32 {
33
34     Boss.Position = new Vector2(300, 288);
35     Boss.StartPosition = new Vector2(300, 288);
36     Boss.TargetPosition = Vector2.Inf;
37     Boss.Pursuing = false;
38     Boss.Dead = false;
39     await Runner.AwaitIdleFrame();
40
41 }
42
43
44 [TestCase]
45 [RequireGodotRuntime]
46 public async Task TestMovementIfDead()
47 {
48
49     await Reset();
50
51     Vector2 InitialPosition = Boss.Position;
52     Boss.TargetPosition = new Vector2(130, 288);
53     Boss.Pursuing = true;
54     Boss.Dead = true;
55     await Runner.SimulateFrames(5, 100);
56     await Runner.AwaitIdleFrame();
57     Vector2 MovementDirection = (Boss.Position -
58         InitialPosition).Normalized();
59     Assertions.AssertThat(MovementDirection).IsEqual(Vector2.Zero);
60
61 }
62
63 [TestCase]
64 [RequireGodotRuntime]
65 public async Task TestMovementIfNotPursuing()
66 {
67
68     await Reset();
69
70     Vector2 InitialPosition = Boss.Position;

```

```

70     Boss.TargetPosition = new Vector2(130, 288);
71     Boss.Pursuing = false;
72     await Runner.SimulateFrames(5, 100);
73     await Runner.AwaitIdleFrame();
74     Vector2 MovementDirection = (Boss.Position -
75         → InitialPosition).Normalized();
76     Assertions.AssertThat(MovementDirection).IsEqual(Vector2.Zero);
77
78 }
79
80 [TestCase]
81 [RequireGodotRuntime]
82 public async Task TestMovementIfNoTargetPosition()
83 {
84     await Reset();
85
86     Vector2 InitialPosition = Boss.Position;
87     Boss.TargetPosition = Vector2.Inf;
88     Boss.Pursuing = true;
89     await Runner.SimulateFrames(5, 100);
90     await Runner.AwaitIdleFrame();
91     Vector2 MovementDirection = (Boss.Position -
92         → InitialPosition).Normalized();
93     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.Zero,
94         → ero, new Vector2(0.1f,
95         → 0.1f));
96
97 }
98
99 [TestCase]
100 [RequireGodotRuntime]
101 public async Task TestMovementToStartPositionLeft()
102 {
103     await Reset();
104
105     Vector2 InitialPosition = Boss.Position;
106     Boss.TargetPosition = new Vector2(130, 288);
107     Boss.StartPosition = new Vector2(130, 288);
108     Boss.Pursuing = false;
109     Boss.ReturnToStart = -1;
110     await Runner.SimulateFrames(5, 100);
111     Vector2 MovementDirection = (Boss.Position -
112         → InitialPosition).Normalized();

```

```

111     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.L_]
112         ↵   eft, new Vector2(0.1f,
113         ↵   0.1f));
114
115
116     }
117
118     [TestCase]
119     [RequireGodotRuntime]
120     public async Task TestMovementToTargetPositionRight()
121     {
122         await Reset();
123
124         Vector2 InitialPosition = Boss.Position;
125         Boss.TargetPosition = new Vector2(450, 288);
126         Boss.StartPosition = new Vector2(450, 288);
127         Boss.Pursuing = false;
128         Boss.ReturnToStart = -1;
129         await Runner.SimulateFrames(5, 100);
130         Vector2 MovementDirection = (Boss.Position -
131             ↵  InitialPosition).Normalized();
132         Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.R_]
133             ↵  ight, new Vector2(0.1f,
134             ↵  0.1f));
135
136     }
137
138     [TestCase]
139     [RequireGodotRuntime]
140     public async Task TestTakeDamageIfDead()
141     {
142
143         await Reset();
144
145         Boss.Dead = true;
146         Vector2 InitialPosition = Boss.Position;
147         float InitialHealth = Boss.CurrentHealthPoints;
148         Damage Damage = new Damage(10, 10, Vector2.Zero, null);
149         Boss.TakeDamage(Damage);
150         await Runner.AwaitIdleFrame();
151         Assertions.AssertThat(Boss.CurrentHealthPoints).IsEqual(InitialH_]
152             ↵  ealth);
153         Vector2 MovementDirection = (Boss.Position -
154             ↵  InitialPosition).Normalized();
155         Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.Z_]
156             ↵  ero, new Vector2(0.1f,
157             ↵  0.1f));

```

```

150
151    }
152
153    [TestCase]
154    [RequireGodotRuntime]
155    public async Task TestTakeDamageOnlyPhysical()
156    {
157
158        await Reset();
159
160        Vector2 InitialPosition = Boss.Position;
161        float InitialHealth = Boss.CurrentHealthPoints;
162        Damage Damage = new Damage(30, 0, Vector2.Zero, null);
163        Boss.TakeDamage(Damage);
164        await Runner.AwaitIdleFrame();
165        float CalculatedHealth = InitialHealth - Damage.GetPhysicalDMG()
166        ↪ * (1 - Boss.Armor / 100.0f) - Damage.GetTrueDMG();
167        Assertions.AssertThat(Boss.CurrentHealthPoints).IsEqual(Calculat ↪ edHealth);
168        Vector2 MovementDirection = (Boss.Position -
169        ↪ InitialPosition).Normalized();
170        Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.Z ↪ ero, new Vector2(0.1f,
171        ↪ 0.1f));
172
173    }
174
175    [TestCase]
176    [RequireGodotRuntime]
177    public async Task TestTakeDamageOnlyTrue()
178    {
179
180        await Reset();
181
182        Vector2 InitialPosition = Boss.Position;
183        float InitialHealth = Boss.CurrentHealthPoints;
184        Damage Damage = new Damage(0, 30, Vector2.Zero, null);
185        Boss.TakeDamage(Damage);
186        await Runner.AwaitIdleFrame();
187        float CalculatedHealth = InitialHealth - Damage.GetPhysicalDMG()
188        ↪ * (1 - Boss.Armor / 100.0f) - Damage.GetTrueDMG();
189        Assertions.AssertThat(Boss.CurrentHealthPoints).IsEqual(Calculat ↪ edHealth);
190        Vector2 MovementDirection = (Boss.Position -
191        ↪ InitialPosition).Normalized();

```

```

188     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.Z)
189         ↳ ero, new Vector2(0.1f,
190         ↳ 0.1f));
191
192 }
193
194 [TestCase]
195 [RequireGodotRuntime]
196 public async Task TestTakeDamageOnlyPushRight()
197 {
198
199     await Reset();
200
201     Vector2 InitialPosition = Boss.Position;
202     float InitialHealth = Boss.CurrentHealthPoints;
203     Damage Damage = new Damage(0, 0, new Vector2(10f, 0f), null);
204     Boss.TakeDamage(Damage);
205     await Runner.AwaitIdleFrame();
206     Assertions.AssertThat(Boss.CurrentHealthPoints).IsEqual(InitialH
207         ↳ ealth);
208     Vector2 MovementDirection = (Boss.Position -
209         ↳ InitialPosition).Normalized();
210     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.R
211         ↳ ight, new Vector2(0.1f,
212         ↳ 0.1f));
213
214 }
215
216
217 [TestCase]
218 [RequireGodotRuntime]
219 public async Task TestTakeDamageOnlyPushLeft()
220 {
221
222     await Reset();
223
224     Vector2 InitialPosition = Boss.Position;
225     float InitialHealth = Boss.CurrentHealthPoints;
226     Damage Damage = new Damage(0, 0, new Vector2(-10f, 0f), null);
227     Boss.TakeDamage(Damage);
228     await Runner.AwaitIdleFrame();
229     Assertions.AssertThat(Boss.CurrentHealthPoints).IsEqual(InitialH
230         ↳ ealth);
231     Vector2 MovementDirection = (Boss.Position -
232         ↳ InitialPosition).Normalized();
233     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.L
234         ↳ eft, new Vector2(0.1f,
235         ↳ 0.1f));
236
237 }
```

```

226 }
227
228 [TestCase]
229 [RequireGodotRuntime]
230 public async Task TestTakeDamageCombined()
231 {
232
233     await Reset();
234
235     Vector2 InitialPosition = Boss.Position;
236     float InitialHealth = Boss.CurrentHealthPoints;
237     Damage Damage = new Damage(30, 10, new Vector2(-10f, 0f), null);
238     Boss.TakeDamage(Damage);
239     await Runner.AwaitIdleFrame();
240     float CalculatedHealth = InitialHealth - Damage.GetPhysicalDMG()
241         * (1 - Boss.Armor / 100.0f) - Damage.GetTrueDMG();
242     Assertions.AssertThat(Boss.CurrentHealthPoints).IsEqual(Calculat ]
243         edHealth);
244     Vector2 MovementDirection = (Boss.Position -
245         InitialPosition).Normalized();
246     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.L ]
247         eft, new Vector2(0.1f,
248         0.1f));
249
250 }
251
252 [TestCase]
253 [RequireGodotRuntime]
254 public async Task TestTakeDamageTillDead()
255 {
256
257     await Reset();
258
259     Vector2 InitialPosition = Boss.Position;
260     float InitialHealth = Boss.CurrentHealthPoints;
261     Damage Damage = new Damage(0, 400.0f, Vector2.Zero, null);
262     Boss.TakeDamage(Damage);
263     float CalculatedHealth = InitialHealth - Damage.GetPhysicalDMG()
264         * (1 - Boss.Armor / 100.0f) - Damage.GetTrueDMG();
265     Assertions.AssertThat(Boss.CurrentHealthPoints).IsEqual(Calculat ]
266         edHealth);
267     Vector2 MovementDirection = (Boss.Position -
268         InitialPosition).Normalized();
269     Assertions.AssertThat(MovementDirection).IsEqualApprox(Vector2.Z ]
270         ero, new Vector2(0.1f,
271         0.1f));
272     Assertions.AssertThat(Boss.Dead).IsTrue();
273 }

```

```

264
265
266 [TestCase]
267 [RequireGodotRuntime]
268 public async Task TestDie()
269 {
270
271     await Reset();
272
273     Boss.Die();
274     await Runner.AwaitIdleFrame();
275
276     Assertions.AssertThat(Boss.Dead).IsTrue();
277     Assertions.AssertThat(Boss.Velocity).IsEqual(Vector2.Zero);
278     Assertions.AssertThat(Boss.MainCollision.Get(CollisionShape2D.PropertyName.Disabled)).IsTrue();
279
280 }
281
282
283 [TestCase]
284 [RequireGodotRuntime]
285 public async Task TestStartGlowing()
286 {
287     await Reset();
288
289     Boss.StartGlowing();
290     await Runner.AwaitIdleFrame();
291
292     Assertions.AssertThat(Boss.Sprite.Modulate).IsEqual(new
293         Color(1.0f, 0.84f, 0.0f, 1.0f));
294
295
296 [TestCase]
297 [RequireGodotRuntime]
298 public async Task TestHPUnder50Percent()
299 {
300     await Reset();
301
302     Boss.CurrentHealthPoints *= 0.49f;
303     await Runner.AwaitIdleFrame();
304
305     Assertions.AssertThat(Boss.Sprite.Modulate).IsEqual(new
306         Color(1.0f, 0.84f, 0.0f, 1.0f));
307     Assertions.AssertThat(Boss.EnemiesRevived).IsTrue();
308     Assertions.AssertThat(Boss.Armor).IsEqual(60f);
309
310 }

```

```
309 [TestCase]  
310 [RequireGodotRuntime]  
311 public async Task TestRegenerateHealth()  
{  
    Boss.CurrentHealthPoints = 350.0f;  
    float Delta = 20f;  
  
    for (int i = 0; i < 5; i++)  
    {  
        Boss.HandleRegeneration(Delta);  
    }  
    await Runner.SimulateFrames(10, 100);  
  
    Assertions.AssertThat(Boss.CurrentHealthPoints).IsEqual(400.0f);  
  
    await Setup();  
}  
}  
}
```

A.4.3. Excel-Tests

Tests für Spieler					
Nr.	Anforderung	Testfallbeschreibung	Erwartetes Ergebnis	Tasteneingabe	Ergebnis
2.2.1.1.1	Der Spieler muss sich nach rechts bewegen können [Taste 'D']	Prüft, ob der Spieler sich bei gedrückter Taste 'D' nach rechts bewegt.	Der Spieler bewegt sich nach rechts.	D	Positiv
2.2.1.1.2	Der Spieler muss sich nach links bewegen können [Taste 'A']	Prüft, ob der Spieler sich bei gedrückter Taste 'A' nach links bewegt.	Der Spieler bewegt sich nach links.	A	Positiv
2.2.1.1.3	Der Spieler muss springen können [Taste 'W']	Prüft, ob der Spieler bei gedrückter Taste 'W' springt.	Der Spieler springt nach oben.	W	Positiv
2.2.1.1.4	Der Spieler muss doppelt springen können [2x Taste 'W'], wenn Stamina > 15	Prüft, ob der Spieler in der Luft ein zweites Mal springen kann.	Der Spieler springt in der Luft ein zweites Mal nach oben.	W, W	Positiv
2.2.1.1.4	Der Spieler kann nicht doppelt springen, wenn Stamina < 15	Prüft, ob Doppelsprung blockiert wird, wenn Stamina < 15	Der Spieler kann keinen zweiten Sprung ausführen	W, W	Positiv
2.2.1.1.5	Bewegungsrichtung dashen können [Taste 'L-Shift']	gedrückter Taste 'L-Shift' einen Dash in die Bewegungsrichtung	Dash in die aktuelle Bewegungsrichtung aus.	L-Shift	Positiv
2.2.1.1.5	Der Spieler kann dashen, wenn Stamina > 20	Prüft, ob Dash korrekt ausgeführt wird, wenn Stamina > 20	Der Spieler führt einen Dash aus	L-Shift	Positiv
2.2.1.1.5	Der Spieler kann nicht dashen, wenn Stamina < 20 ist	Prüft, ob Dash nicht ausgeführt wird, wenn Stamina < 0	Der Spieler führt keinen Dash aus	L-Shift	Positiv
2.2.1.1.5	Der Spieler kann nicht dashen, wenn keine Richtung angegeben ist	Prüft, ob Dash blockiert wird, wenn keine Richtung eingegeben wird	Der Spieler führt keinen Dash aus	L-Shift	Positiv
2.2.1.1.5	Der Spieler kann nicht durch eine Wand dashen	Prüft, ob Dash nicht durch die Wand funktioniert	Der Spieler dasht gegen die Wand	L-Shift	Positiv
2.2.1.1.5	Der Spieler kann nicht dashen, wenn er am angreifen ist	Prüft, ob Dash nicht ausgeführt wird, wenn der Spieler eine Attacke ausführt (Leichte/Schwere)	Der Spieler führt keinen Dash aus	L-Shift + Linke/Rechte Maustaste	Positiv

2.2.1.2.1	Der Spieler muss eine leichte Attacke durchführen können [Linke Maustaste], wenn Stamina > 10	Prüft, ob der Spieler eine leichte Attacke durchführt, wenn die linke Maustaste gedrückt wird.	Der Spieler führt eine leichte Attacke aus.	Linke Maustaste	Positiv
2.2.1.2.2	Der Spieler muss eine schwere Attacke durchführen können [Rechte Maustaste], wenn Stamina > 25	Prüft, ob der Spieler eine schwere Attacke durchführt, wenn die rechte Maustaste gedrückt wird.	Der Spieler führt eine schwere Attacke aus.	Rechte Maustaste	Positiv
2.2.1.2.1	Der Spieler kann keine leichte Attacke durchführen [Linke Maustaste], wenn Stamina > 10	Prüft, ob der Spieler keine leichte Attacke durchführt, wenn die linke Maustaste gedrückt wird.	Der Spieler führt eine leichte Attacke nicht aus.	Linke Maustaste	Positiv
2.2.1.2.2	schwere Attacke durchführen [Rechte Maustaste], wenn	schwere Attacke durchführt, wenn die rechte Maustaste	Der Spieler führt eine schwere Attacke nicht aus.	Rechte Maustaste	Positiv
2.2.1.2.2	Der Spieler fügt Schaden zu, wenn er einen Gegner trifft	Prüft, ob ein Gegner Schaden nimmt, wenn der Spieler ihn trifft	Der Gegner nimmt Schaden	Angriff durch Gegner	Positiv
2.2.1.2.3	Der Spieler muss einen Angriff blocken können [Taste 'Leertaste']	Prüft, ob der Spieler einen Angriff blockt, wenn die Leertaste gedrückt wird.	Der Spieler blockt den Angriff.	Leertaste	Positiv
2.2.1.2.3	Der Spieler blockt Schaden, wenn die Leertaste gedrückt wird und er getroffen wird	Prüft, ob der Spieler keinen Schaden nimmt, wenn er blockt und getroffen wird	Der Spieler nimmt keinen Schaden	Leertaste und Angriff durch Gegner	Positiv
2.2.1.2.4	Der Spieler heilt sich um 25HP, wenn er ein Bloodvial verwendet	Prüft, ob der Spieler sich heilt, wenn Bloodvial verwendet wird	Der Spieler heilt sich	Q	Positiv
2.2.1.2.4	Die Bloodvials werden zurückgesetzt wenn der Spieler stirbt	Prüft, ob die Bloodvials zurückgesetzt werden, wenn der Spieler stirbt	Bloodvials setzen sich zurück	KEINE	Positiv
2.2.1.3	Stamina regeneriert sich nach 3 Sekunden, wenn keine Schwere/Leichte Attacke oder Doppelsprung oder Dash ausgeführt wird	Prüft, ob die Stamina sich nach 3 Sekunden regeneriert	Die Stamina regeneriert sich	KEINE	Positiv
2.2.1.4.1	Der Spieler stirbt wenn Health <= 0	Prüft, ob der Spieler stirbt wenn er kein Leben mehr hat	Spieler stirbt	KEINE	Positiv

2.2.1.4.5	Health regeneriert sich nicht über die Zeit	Prüft, ob Health sich nicht regeneriert	Health des Spielers ändert sich nicht	KEINE	Positiv
2.2.1.5.1	Der Sündenzähler steigt um 10, wenn ein Gegner getötet wird	Prüft, ob Sündenzähler steigt	Sündenzähler ist um 10 gestiegen	KEINE	Positiv

1	Die Animation 'jump' wird abgespielt, wenn der Spieler springt	Prüft, ob die 'jump'-Animation korrekt abgespielt wird, wenn der Spieler springt	Die Animation 'jump' wird abgespielt	W	Positiv
2	Die Animation 'run' wird abgespielt, wenn der Spieler läuft	Prüft, ob die 'run'-Animation korrekt abgespielt wird, wenn der Spieler sich bewegt	Die Animation 'run' wird abgespielt	D oder A	Positiv
3	Die Animation 'light_attack' wird abgespielt, wenn der Spieler angreift [Linke Maustaste]	Prüft, ob die 'light_attack'-Animation korrekt abgespielt wird, wenn angegriffen wird	Die Animation 'light_attack' wird abgespielt	Linke Maustaste	Positiv
4	'heavy_attack' wird abgespielt, wenn der Spieler angreift [Rechte Maustaste]	Prüft, ob die 'heavy_attack'-Animation korrekt abgespielt wird, wenn angegriffen wird	Die Animation 'heavy_attack' wird abgespielt	Rechte Maustaste	Positiv
5	Die Animation 'block' wird abgespielt, wenn der Spieler blockt	Prüft, ob die 'block'-Animation korrekt abgespielt wird, wenn der Spieler blockt	Die Animation 'block' wird abgespielt	Leertaste	Positiv
6	Die Animation 'dash' wird abgespielt, wenn der Spieler dasht	Prüft, ob die 'dash'-Animation korrekt abgespielt wird, wenn der Spieler dasht	Die Animation 'dash' wird abgespielt	L-Shift	Positiv

Tests für HUD				
Nr.	Anforderung	Testfallbeschreibung	Erwartetes Ergebnis	Ergebnis
2.2.2.1	Im Display muss die Healthbar oben links angezeigt werden. Diese muss die momentane TP des Spielers abbilden.	Wenn der Spieler leben verliert oder regeneriert, soll die Healthbar das anzeigen können	Healthbar wird kleiner, wenn der Spieler Leben verliert oder größer, wenn der Spieler Leben regeneriert	Positiv
2.2.2.2	Im Display muss die Staminabar oben links unter der Healthbar angezeigt werden. Diese muss die momentane Stamina des Spielers abbilden.	Wenn der Spieler Stamina verliert oder regeneriert, soll die Staminabar das anzeigen können	Staminabar wird kleiner, wenn der Spieler Stamina verliert oder größer, wenn der Spieler Stamina regeneriert	Positiv
2.2.2.3	Im Display muss die Anzahl der Blood Vials unten links des HUDs angezeigt werden. Diese muss die momentane Anzahl der Blood Vials abbilden.	Wenn der Spieler "Q" drückt (ein Bloodvial verwendet), soll der BloodVialZähler um 1 kleiner werden	BloodVialzähler wird um 1 kleiner, wenn "Q" gedrückt wird	Positiv
2.2.2.4	Im Display muss die Anzahl der Sünden unten rechts des HUDs angezeigt werden. Diese muss die momentane Anzahl der Sünden abbilden.	Wenn der Spieler einen Gegner tötet, erhöht sich der Zähler um die Sündenanzahl des Gegners	Sündenzähler erhöht sich um die Anzahl der Sünden des Gegners	Positiv

Tests für Gegner				
Anforderung	Beschreibung	Test	Beobachtung	Ergebnis
2.2.3.1.1	Der Gegner muss den Spieler anfangen zu verfolgen, wenn dieser sich in seinem Detectionbereich befindet.	Wenn der Spieler in den Detectionbereich eines Gegners läuft, muss der Gegner den Spieler anfangen zu verfolgen	Gegner verfolgt Spieler, wenn dieser seinen Detectionbereich betritt.	Positiv
2.2.3.1.2	Der Gegner muss aufhören den Spieler zu verfolgen, wenn dieser den Pursuingbereich verlässt.	Wenn der Spieler den Pursuingbereich eines Gegners verlässt, muss der Gegner den Spieler aufhören zu verfolgen	Gegner stoppt die Verfolgung, sobald der Spieler den Pursuingbereich verlassen hat.	Positiv
2.2.3.1.3	Der Gegner hat eine Startposition [x,y – koordinate]	Wenn eine Szene geladen wird, hat der Gegner eine fixe deterministische Startposition.	Beim mehrmaligen Laden der gleichen Szene startet der Gegner immer am selben Ort.	Positiv
2.2.3.1.4	Der Gegner kehrt nach einer Zeit zur Startposition zurück, wenn er sich dort noch nicht befindet	Wenn der Gegner sich nicht auf der Startposition befindet, muss er nach 5s zu seiner Startposition zurück laufen.	Gegner kehrt nach 5s zu Startposition zurück.	Positiv
2.2.3.2.1	Der Gegner muss eine leichte Attacke durchführen können	Wenn der Spieler sich in Schlagreichweite des Gegners befindet, muss der Gegner den Spieler angreifen.	Der Gegner greift den Spieler an, sobald dieser in Schlagreichweite kommt.	Positiv
2.2.3.3.1	Der Gegner stirbt, wenn Health == 0.	Wenn die Trefferpunkte des Gegners auf Null abfallen, dann muss er Sterben.	Gegner stirbt, sobald seine Trefferpunkte kleiner gleich Null sind	Positiv
2.2.3.3.2	Die TP des Gegners muss über den Gegnersprite sichtbar sein	Eine Anzeige der momentanen Trefferpunkte des Gegners, muss immer über dessen Sprite sichtbar sein, solange er lebt.	Die momentanen Trefferpunkte eines Gegner sind immer Sichtbar, solange er lebt.	Positiv
2.2.3.4	Wenn der Gegner stirbt, bekommt der Spieler Sünden [Anzahl Sünden: SIN_AMOUNT]	Wenn ein Gegner stirbt, muss der Spieler die beim Gegner festgelegte Anzahl an Sünden bekommen.	Der Spieler bekommt die korrekte Anzahl an Sünden, sobald der Gegner stirbt.	Positiv
2.2.3.5.1	Der Gegner muss mit Boden und Wände kollidieren können	Wenn ein Gegner vor einer Wand oder auf dem Boden stehe muss er mit diesen Objekten kollidieren.	Der Gegner kollidiert mit Wänden und mit dem Boden.	Positiv
2.2.3.5.2	Der Gegner muss mit Spieler kollidieren können	Wenn ein Spieler vor dem Gegner steht, darf der Gegner nicht durch den Spieler laufen.	Der Gegner kollidiert mit dem Spieler und er kann nicht durch den Spieler hindurchschreiten.	Positiv

Tests für Boss				
Anforderung	Beschreibung	Test	Beobachtung	Ergebnis
2.2.3.1.1	Der Boss muss den Spieler anfangen zu verfolgen, wenn dieser sich in seinem Detectionbereich befindet.	Wenn der Spieler in den Detectionbereich des Bosses läuft, muss der Boss den Spieler anfangen zu verfolgen	Boss verfolgt Spieler, wenn dieser seinen Detectionbereich betritt.	Positiv
2.2.3.1.2	Der Boss muss aufhören den Spieler zu verfolgen, wenn dieser den Pursuingbereich verlässt.	Wenn der Spieler den Pursuingbereich eines Bosses verlässt, muss der Boss den Spieler aufhören zu verfolgen	Boss stoppt die Verfolgung, sobald der Spieler den Pursuingbereich verlassen hat.	Positiv
2.2.3.1.3	Der Boss hat eine Startposition [x,y – koordinate]	Wenn eine Szene geladen wird, hat der Boss eine fixe deterministische Startposition.	Beim mehrmaligen Laden der gleichen Szene startet der Boss immer am selben Ort.	Positiv
2.2.3.1.4	Der Boss kehrt nach einer Zeit zur Startposition zurück, wenn er sich dort noch nicht befindet	Wenn der Boss sich nicht auf der Startposition befindet, muss er nach 5s zu seiner Startposition zurück laufen.	Boss kehrt nach 5s zu Startposition zurück.	Positiv
2.2.3.2.1	Der Boss muss eine leichte Attacke durchführen können	Schlagreichweite des Boss befindetm, muss der Boss den Spieler angreifen.	Der Boss greift den Spieler an, sobald dieser in Schlagreichweite kommt.	Positiv
2.2.3.3.1	Der Boss stirbt, wenn Health == 0.	Wenn die Trefferpunkte des Boss auf Null abfallen, dann muss er Sterben.	Boss stirbt, sobald seine Trefferpunkte kleiner gleich Null sind	Positiv
2.2.3.3.2	Die TP des Boss muss über den Bosssprite sichtbar sein	Eine Anzeige der momentanen Trefferpunkte des Boss, muss immer über dessen Sprite sichtbar sein, solange er lebt.	Die momentanen Trefferpunkte eines Boss sind immer Sichtbar, solange er lebt.	Positiv
2.2.3.4	Wenn der Boss stirbt, bekommt der Spieler Sünden [Anzahl Sünden: SIN_AMOUNT]	Wenn ein Boss Stirbt, muss der Spieler die beim Boss festgelegte Anzahl an Sünden bekommen.	Der Spieler bekommt die korrekte Anzahl an Sünden, sobald der Boss stirbt.	Positiv
2.2.3.5.1	Der Boss muss mit Boden und Wände kollidieren können	Wand oder auf dem Boden stehe muss er mit diesen Objekten kollidieren.	Der Boss kollidiert mit Wänden und mit dem Boden.	Positiv
2.2.3.5.2	Der Boss muss mit Spieler kollidieren können	Wenn ein Spieler vor dem Boss steht, darf der Boss nicht durch den Spieler laufen.	Der Boss kollidiert mit dem Spieler und er kann nicht durch den Spieler hindurchschreiten.	Positiv

2.2.3.6.1	Der Boss wird goldfarben, sobald die 50 % seiner Lebenspunkte erreicht ist.	Dem Boss wird Schaden zugefügt, bis er unter 50 % Lebenspunkte fällt.	Die Farbe des Bosses ändert sich sichtbar zu Gold.	Positiv
2.2.3.6.2	Der Boss erhält mehr Rüstung unter 50 % Lebenspunkten.	Dem Boss wird Schaden vor und nach 50 % zugefügt.	Nach Erreichen von 50 % Lebenspunkten wird weniger Schaden pro Treffer verursacht.	Positiv
2.2.3.6.3	Der Boss belebt alle gestorbenen Gegner im Raum wieder, sobald er unter 50 % Lebenspunkte fällt.	Vor dem Kampf werden Gegner getötet. Der Boss wird dann auf unter 50 % gebracht.	Die zuvor besiegten Gegner stehen wieder auf und greifen den Spieler erneut an.	Positiv

Tests für Environment				
Nr	Anforderung	Testfallbeschreibung	Erwartetes Ergebnis	Ergebnis
2.2.4.1.1	Wenn der Player mit dem Brunnen interagiert passiert folgendes: Alle Blood Vials werden aufgefüllt	Wenn der Spieler den Brunnen berührt dann werden seine Blood Vials auf die maximale Anzahl gesetzt	Anzahl der Vials wird auf 5 gesetzt	Positiv
2.2.4.1.2	Wenn der Player mit dem Brunnen interagiert passiert folgendes: Der Spieler heilt sich zu 100%	Wenn der Spieler den Brunnen berührt dann ist sein Leben wieder auf maximalen Wert	Healthbar ist wieder voll gefüllt	Positiv
2.2.4.1.3	Wenn der Player mit dem Brunnen interagiert passiert folgendes: Alle toten Gegner die Respawnable sind,	Wenn der Spieler den Brunnen berührt dann werden wieder alle respawnbaren Gegner neu geladen	Alle respawnbaren Gegner werden wieder neu geladen	Positiv
2.2.4.1.4	Wenn der Player mit dem Brunnen interagiert passiert folgendes: Stamina füllt sich	Wenn der Spieler den Brunnen berührt dann ist sein Leben wieder auf maximalen Wert	Staminabar ist wieder voll gefüllt	Positiv
2.2.4.4	Der Spieler kann sich darauf bewegen und stößt von jeder Seite dagegen	Der Spieler kann sich drauf bewegen und stößt von den Seiten und von unten dagegen	Spieler kann sich auf der Platform bewegen aber nicht durch	Positiv
2.2.4.5	Wie eine Platform nur das diese sich in einem festen Bereich bewegt	Der Spieler bewegt sich auf der Platform in einem festen Bereich mit	Der Spieler bewegt sich auf der Platform mit	Positiv
2.2.4.6	Spieler kann durchlaufen und Schaden bekommen	Wenn der Spieler durch die Spikes läuft bekommt er Schaden	Der Spieler kann durch einen Spike durchlaufen und bekommt Schaden	Positiv
2.2.4.7	Wie dynamische Platform und Spike	Wenn der Spieler durch die Spikes läuft bekommt er Schaden und bewegt sich in einem festen Bereich auf den Spikes mit	Der Spieler kann sich im Spike bewegen und bewegt sich mit dem Spike mit und bekommt dabei Schaden	Positiv
2.2.4.8	Diesen dienen zum Teleportieren des Spielers, es gilt folgendes: Nach dem Tod wird der Spieler an den Teleporter am Brunnen teleportiert	Wenn der Spieler stirbt wird er an den Teleporter am Brunnen teleportiert	Der Spieler teleportiert sich beim Sterben zum Teleporter des letzten Brunnen den er berührt hat	Positiv
2.2.4.8	Diesen dienen zum Teleportieren des Spielers, es gilt folgendes: Der Spieler kann durch Teleporter zu einer anderen Map transferiert werden mit allen aktuellen Werten	Wenn ein Spieler einen Teleporter betritt wird er an eine andere Map transferiert mit all seinen Werten	Der Spieler teleportiert sich beim berühren eines Teleportes zu einer anderen Map und behält die Werte seines Lebens, Staminas, Sünden und Anzahl seiner Bloodvials bei	Positiv

Tests für Spielmanagement				
Anforderung	Beschreibung	Test	Beobachtung	Ergebnis
2.2.5.1.1	Der Benutzer muss das Spiel im Menu, durch ein Knopfdruck, starten und verlassen können	Wenn der Benutzer im Menü ist, muss dieser das Spiel beenden können.	Der Benutzer kann mit dem Quit Game Button das Spiel verlassen.	Positiv
2.2.5.2.1	Der Benutzer muss in der Lage sein, den Spielstand zu speichern	Wenn der Benutzer sich in einem Spielstand befindet, muss er diesen Speichern können.	Der Benutzer kann durch den Save Game Button den Spielstand speichern.	Positiv
2.2.5.2.2	Der Benutzer muss in der Lage sein, den Spielstand zu löschen	Wenn ein Spielstand existiert muss ihn der Benutzer löschen können.	Der Benutzer kann einen existierenden Spielstand im Menü löschen.	Positiv
2.2.5.2.3	Der Benutzer muss in der Lage sein, den Spielstand laden zu können	Wenn der Benutzer einen Spielstand gespeichert hat, muss er diesen im Menü auch wieder laden können.	Der Benutzer kann im Menü einen gespeicherten Spielstand laden.	Positiv

A.5. Doxygen mit Quellcode