

Masterarbeit

**zur Erlangung des akademischen Grades
„Master of Engineering (M. Eng.)“
im Studiengang Informatik und Kommunikationssysteme
des Fachbereichs Informatik und Kommunikationssysteme
der Hochschule Merseburg**

**Konzeption, Umsetzung und Test eines Rollenspiels mit
Karteneditor im Rahmen der prototypischen Entwicklung für
verschiedene Plattformen**

Autoren: Julius Forner
Matrikel-Nr.: 18196
Michael Müller
Matrikel-Nr.: 17871

Version vom: 19. September 2016
Erstgutachter: Prof. Dr. rer. pol. Uwe Schröter (Hochschule Merseburg)
Zweitgutachter: M. Eng. Ulrich Borchert (Hochschule Merseburg)

Inhalt

Abkürzungsverzeichnis	III
Abbildungsverzeichnis.....	III
Tabellenverzeichnis.....	VI
Listingverzeichnis	VI
Formelverzeichnis	IX
1 Einleitung	1
2 Überblick über das Spielkonzept	3
3 Planung des Spiels	5
3.1 Anforderungen an das Spiel.....	5
3.2 Wahl der grafischen Umsetzung	9
3.3 Wahl des Entwicklungsframeworks und der Programmiersprache	14
3.4 Anpassungen für das Multiplattformkonzept	21
3.4.1 Ein- und Ausgabemöglichkeiten	21
3.4.2 Rechenleistung	23
4 Planung und prototypische Realisierung der Komponenten	25
4.1 Ablauf und Aufbau der Planung und der Realisierung	25
4.2 Basis- und Hintergrund-Komponenten	26
4.2.1 Steuerung	26
4.2.2 Navigationssystem.....	28
4.2.3 Mozo	37
4.2.4 Lokalisierung.....	42
4.2.5 Zwischenstand des Spiels – das Hauptmenü	46
4.3 Spielumgebung	49
4.3.1 Karteneditor	49
4.3.2 Bewegung und Animation	95

4.3.3	Audio.....	103
4.3.4	Zwischenstand des Spiels – die Spielwelt.....	107
4.4	UI-Systeme.....	110
4.4.1	Aufgabensystem.....	110
4.4.2	Inventar.....	125
4.4.3	Dialogsystem.....	146
4.4.4	Zwischenstand des Spiels – die UI-Systeme.....	153
5	Evaluierung und Test der Anwendung.....	155
5.1	Usability-Test auf verschiedenen Plattformen.....	155
5.2	Performance-Test.....	157
5.2.1	Tools zur Performance-Messung in Unity.....	157
5.2.2	Performance-Test.....	158
6	Zusammenfassung.....	161
	Glossar.....	IX
	Literaturverzeichnis.....	XI
	Bearbeiter der Abschnitte.....	XV

Abkürzungsverzeichnis

NPC..... *non-player character*

Abbildungsverzeichnis

Abbildung 1: Vergleich orthogonaler und perspektivischer Projektion.....	10
Abbildung 2: Beispiel eines 3D-Modells [2]	12
Abbildung 3: Eines der bekanntesten Side-Scroller (Super Mario Bros.) [31]	12
Abbildung 4: Eine typische Top-Down-Ansicht (The Legend of Zelda) [32]	13
Abbildung 5: Ausschnitt aus einem isometrischen Spiel (Landstalker) [30].....	13
Abbildung 6: Hierarchieansicht.....	17
Abbildung 7: Projekthierarchie.....	18
Abbildung 8: Szenenansicht.....	18
Abbildung 9: Detailansicht eines Objektes im Inspector.....	19
Abbildung 10: Aufbau eines GUI-Labels	20
Abbildung 11: Controller einer PlayStation 4-Konsole [11].....	21
Abbildung 12: Verlauf der Entwicklung.....	25
Abbildung 13: Anwendungsfälle für das Eingabesystem.....	26
Abbildung 14: Klassendiagramm für das Eingabesystem	27
Abbildung 15: Steuerungs-UI	28
Abbildung 16: Navigationssystem in Unity	29
Abbildung 17: Komponenten des neuen Navigationssystems.....	30
Abbildung 18: Kommunikation zwischen den Navigations-Komponenten	31
Abbildung 19: Klassendiagramm des Navigationssystems	32
Abbildung 20: Event Trigger mit einer übergebenen Methode	33
Abbildung 21: Klassendiagramm des Mozo	39
Abbildung 22: Klassendiagramm der Lokalisierung.....	43
Abbildung 23: Inhalt der Übersetzungsdatei.....	45
Abbildung 24: Hauptmenü des Spiels	46
Abbildung 25: Verknüpfungen der Elemente	46
Abbildung 26: An einen Text angehängte LocalizeText-Komponente.....	47
Abbildung 27: Ausführen des Hauptmenüs	48
Abbildung 28: Tileset mit verschiedenen Tiles für ein Top-Down-Gelände [19]	49

Abbildung 29: Darstellung eines Kachelrasters und eines isometrischen Rasters....	50
Abbildung 30: Das Kamera-Symbol (Gizmo) in Unity	51
Abbildung 31: Darstellung der Transform-Tools in der Szene (Handles)	51
Abbildung 32: Deklarieren der Variablen und die dazugehörige Abbildung im Inspector	52
Abbildung 33: Statische DrawLine()-Methode der Gizmos-Klasse	52
Abbildung 34: Statische DrawWireCube()-Methode der Gizmos-Klasse	52
Abbildung 35: Ablauf für das Zeichnen der Rasterlinien (1. Schritt).....	53
Abbildung 36: Ablauf für das Zeichnen der Rasterlinien (2. Schritt).....	54
Abbildung 37: Isometrisches Raster in der Szene nach der Implementierung.....	55
Abbildung 38: Vollständige Implementierung der eigenen Inspector-Ansicht für das Raster	61
Abbildung 39: Beispiel eines Tile-Objekts in verschiedenen Perspektiven	63
Abbildung 40: Anwendungsfalldiagramm für das Bearbeiten der Tile-Objekte	64
Abbildung 41: Gegenüberstellung der beiden Koordinatensysteme	69
Abbildung 42: Vergleich des Transform-Tools von Unity mit dem eigenen Transform-Handle	70
Abbildung 43: Die SpriteRenderer-Komponente von Unity	72
Abbildung 44: Importeinstellungen für Grafiken	72
Abbildung 45: Beispiel eines Spritesheets im Sprite Editor.....	74
Abbildung 46: Ein Mesh mit einem hervorgehobenen Triangle und den dazugehörigen Vertices	78
Abbildung 47: Eine Mesh-Collider-Komponente	78
Abbildung 48: Alle Eckpunkte (blau) des Custom Meshes inklusive Abstand vom Mittelpunkt (orange).....	82
Abbildung 49: Anwendungsfalldiagramm für das Erstellen eines Spielfeldes	85
Abbildung 50: Gegenüberstellung der Bedingungen an Beispielkoordinaten	91
Abbildung 51: Die Rigidbody-Komponente	95
Abbildung 52: Richtung der Gravitation	96
Abbildung 53: Spritesheet für die Laufanimationen der Spielfigur	98
Abbildung 54: Animieren mit dem Animation-Tool	99
Abbildung 55: Beispiel einer Animationsverknüpfung	100
Abbildung 56: Verknüpfung der Animationen der Spielfigur.....	100
Abbildung 57: Optionen für das importieren von Audiodaten.....	103

Abbildung 58: Einstellungsmöglichkeiten der Audio Source-Komponente	105
Abbildung 59: Untergrund des Beispiellevels	107
Abbildung 60: Auswählen eines Baums im Level Designer	108
Abbildung 61: Platzieren des Baumes	108
Abbildung 62: Platzieren eines (teilweise) verdeckten Objektes	109
Abbildung 63: Ausführen des Beispiellevels	109
Abbildung 64: Anwendungsfälle des Aufgabensystems	110
Abbildung 65: Entwurf des Aufgaben-UIs	111
Abbildung 66: Komponenten des Aufgabensystems	111
Abbildung 67: Klassen des Aufgabensystems und ihre Zusammenhänge	112
Abbildung 68: Beispielhafte Liste mit Objekten	115
Abbildung 69: Positionen bei der Anordnung eines Elementes	116
Abbildung 70: UI-Anordnung des Aufgabensystems	120
Abbildung 71: Valider Aufgabenschlüssel	124
Abbildung 72: Kampfschema	125
Abbildung 73: Normalschema	126
Abbildung 74: Anwendungsfalldiagramm des Inventars	126
Abbildung 75: Ablauf für Aktionen im Inventar	127
Abbildung 76: Komponentendiagramm des Inventars	127
Abbildung 77: Klassendiagramm des Inventars	128
Abbildung 78: UI-Entwurf des Inventars	130
Abbildung 79: Inventar beim Sortieren	131
Abbildung 80: Grafische Umsetzung des Inventars	141
Abbildung 81: Event Trigger des "Benutzen"-Toggles	142
Abbildung 82: UI-Prototyp eines Gegenstandes	144
Abbildung 83: Reihenfolge der Navigation zum Selektieren eines Mozo	145
Abbildung 84: Anwendungsfalldiagramm des Dialogsystems	146
Abbildung 85: Komponenten des Dialogsystems	146
Abbildung 86: Klassen des Dialogsystems	147
Abbildung 87: Ansicht des Dialogs im Unity-Inspektor	151
Abbildung 88: Erster Entwurf eines Dialogfensters	152
Abbildung 89: Der fertige Dialog	152
Abbildung 90 UI des Aufgabensystems während der Ausführung	153

Abbildung 91 UI des Inventars während der Ausführung.....	154
Abbildung 92 Dialog während der Ausführung.....	154
Abbildung 93: Profiler des Unity-Editors	157
Abbildung 94: Auswertung der Profiler-Daten.....	159
Abbildung 95: Verteilung der Auslastung	160

Tabellenverzeichnis

Tabelle 1: Überblick über die gängigsten Perspektiven [1]	10
Tabelle 2: Vergleich der Entwicklungs-Frameworks [8] [9] [10]	16
Tabelle 3: Basiseigenschaften eines Mozo.....	37
Tabelle 4: Hauptattribute eines Monsters	38
Tabelle 5: Kampfeigenschaften eines Monsters	38
Tabelle 6: Koordinaten für die platzierten Tile-Objekte in normaler und isometrischer Form	69
Tabelle 7: Bedingungen und Verknüpfungen der Idle-Zustände.....	101
Tabelle 8: Benötigte Zeit zum Sortieren mit OrderBy.....	138
Tabelle 9: Benötigte Zeit zum Sortieren mit einem eigenen Vergleich.....	140
Tabelle 10: UI-Handler des Inventars	142
Tabelle 11: Usability-Testteilnehmer.....	156
Tabelle 12: Ergebnisse des Usability-Tests.....	156

Listingverzeichnis

Listing 1: Erzeugen des Steuerungs-Prefabs.....	28
Listing 2: Manuelles Auslösen eines Events	33
Listing 3: Realisierung der globalen Erreichbarkeit.....	34
Listing 4: Instanziierung des Navigator-Handlers.....	35
Listing 5: Registrierung beim Navigator-Handler	35
Listing 6: Beispielabfrage bei einem Tastendruck auf die nach-oben-Taste.....	36
Listing 7: Definition des Namens eines Mozo	41
Listing 8: Ändern eines Attributes	41
Listing 9: Übersetzungsanfrage einer Text-Komponente	44

Listing 10: Bearbeiten einer Übersetzungsanfrage.....	45
Listing 11: Laden einer neuen Szene	48
Listing 12: Zeichnen der Linien im ersten Schritt.....	53
Listing 13: Klassendefinition des Editor-Scripts.....	57
Listing 14: Initialisierung der Rastervariable	58
Listing 15: Erzeugung einiger Layout-Elemente für die Darstellung des Schiebereglers der Rastergröße	59
Listing 16: Implementierung des Schiebereglers der Zellengröße.....	60
Listing 17: Erstellung des Vector3- und des Color-Feldes.....	60
Listing 18: Initialisieren einiger Variablen beim Selektieren des Tile-Objekts	66
Listing 19: Anpassung des Inspectors für die isometrische Positionsangabe	66
Listing 20: Konvertierung des isometrischen Vektors in einen normalen Vektor	67
Listing 21: Konvertierung der 2D-Position zur isometrischen Position	68
Listing 22: Erstellung eines isometrisch ausgerichteten Position-Handles	70
Listing 23: Runden der Handle-Position und Aktualisieren des Höhenwerts	71
Listing 24: Erzeugung eines Objektfeldes (Sprite) im Inspector	75
Listing 25: Erzeugung des Pivot-Feldes im Inspector.....	75
Listing 26: Auslesen und Anpassen der Importdaten/-einstellungen	76
Listing 27: Anpassung der Sprites und Zuweisen der Importdaten/-einstellungen	77
Listing 28: Erstellen eines Objektfeldes für das Mesh	79
Listing 29: Erstellen eines Buttons inklusive Methodenaufruf zur Mesh-Erzeugung..	80
Listing 30: Methode zur Erzeugung eines Meshes.....	80
Listing 31: Eigenschaften und Konstruktor der CustomBoxMesh-Klasse	81
Listing 32: Erzeugen eines neuen Mesh-Objektes und Definition der jeweiligen Eckpunkte.....	81
Listing 33: Festlegung der Vertices	82
Listing 34: Festlegung der Triangles	83
Listing 35: Rückgabe des Meshes.....	84
Listing 36: Initialisieren einiger Variablen beim Selektieren des Level-Objekts	86
Listing 37: Erzeugen der relevanten Felder für das Generieren einer Grundfläche im Inspector.....	87
Listing 38: Erzeugung des Buttons zur Generierung	87
Listing 39: Logik des Buttons zur Generierung der Grundfläche	88

Listing 40: Felder für die Platzieren-Funktion.....	89
Listing 41: Darstellung eines Gizmos für das Hervorheben einer Zelle	89
Listing 42: Erzeugen eines Ray-Objekts.....	90
Listing 43: Erstellen einer Vector3-Variable für die Mausposition	90
Listing 44: Weitere Bedingung für die korrekte Zellenselektierung	90
Listing 45: Aktuelle Zellenposition setzen und Szene neu zeichnen.....	92
Listing 46: Eingabeabfrage mit anschließenden Erstellen oder Löschen eines Tile-Objekts.....	92
Listing 47: Das bereits existierende Tile-Objekt an dieser Position zwischenspeichern	92
Listing 48: Vergleich aller Positionen der bereits platzierten Tile-Objekte mit der Zielposition.....	93
Listing 49: Keine erfolgreiche Platzierung.....	93
Listing 50: Vergleich der Sprite-Texturen.....	93
Listing 51: Funktion zum Löschen eines Tile-Objekts	94
Listing 52: Erzeugen des neuen Tile-Objekts	94
Listing 53: Klassendefinition für die Bewegungsabfrage.....	96
Listing 54: Abfrage der Sprungtaste	96
Listing 55: Hinzufügen der Sprungkraft zum Rigidbody.....	97
Listing 56: Abfragen der Richtungstasten	97
Listing 57: Klassendefinition und Start des Scripts zum Aktualisieren der Animationsparameter.....	101
Listing 58: Setzen der Bedingungsparameter	102
Listing 59: Setzen des Walk-Parameters bei keiner betätigten Taste.....	102
Listing 60: Abspielen einer Audiodatei mittels Code	107
Listing 61: Berechnung der Positionen	116
Listing 62: Zeichnen der Elemente	117
Listing 63: Neuberechnung der Parameter	118
Listing 64: Prüfen des QuestLogs des Spielers	121
Listing 65: Anhängen der Listener an die UI-Objekte	122
Listing 66: Aktualisieren der Aufgaben-Details	123
Listing 67: Anzeigen der Aufgabenbeschreibung.....	124
Listing 68: Beispielitem im XML-Format.....	132
Listing 69: Laden der XML-Dateien	133

Listing 70: Auswahl des Itemknotens	133
Listing 71: Unterscheidung der Kategorien	134
Listing 72: Laden einer Item-Eigenschaft	134
Listing 73: Rückgabe der Item-Details.....	135
Listing 74: Hinzufügen eines neuen Items.....	135
Listing 75: Tauschen zweier Items des Inventars.....	136
Listing 76: Implementierung der A bis Z-Sortierung	137
Listing 77: Füllen des Dictionarys.....	138
Listing 78: Sortierung des Inventars mit OrderBy	138
Listing 79: Vergleich zweier Unterkategorien	139
Listing 80: Prüfung auf dieselbe Unterkategorie.....	140
Listing 81: Registrierung des Item-List-Tokens	143
Listing 82: Ausführen eines Sortierschemas	143
Listing 83: Hinzufügen von neuen Objekten in die Gegenstandsliste.....	144
Listing 84: Anwenden des Item-Effektes	145
Listing 85: Wechseln des Dialogschrittes	148
Listing 86: Starten des Dialoges.....	149
Listing 87: Hinzufügen einer Komponente während der Laufzeit	149
Listing 88: Implementierung des Interfaces für neue Dialog-Komponenten	150
Listing 89: Registrierung der Komponente	150
Listing 90: Start einer zusätzlichen Dialog-Komponente	151

Formelverzeichnis

Formel 1: Berechnung der normalen Koordinaten (2D).....	67
Formel 2: Berechnung der isometrischen Koordinaten	68
Formel 3: Bedingungen für korrekte Zellenselektion	91
Formel 4: Berechnung der maximalen Anzahl der Elemente	115

1 Einleitung

Die immer größer werdende Plattformvielfalt stellt für die Spielentwickler in der heutigen Zeit ein immer größer werdendes Problem dar. Zusammen mit dem Zeitdruck ergibt sich daraus ein essenzielles Problem: Sollen die Spiele nativ entwickelt werden, also für jede Plattform mit den dafür vorgesehenen Sprachen und Werkzeugen, oder soll ein Spiel für eine Plattform entwickelt werden und dann auf die verschiedenen anderen Plattformen portiert werden? Die native Entwicklung verschlingt viel Zeit und Ressourcen, die Portierung bringt unter Umständen Inkompatibilität und Performanceeinbußen. Innerhalb der letzten Jahre ist jedoch ein Trend zu beobachten: Ein Spiel wird auf dem Computer entwickelt und dann mittels dafür geschaffener Entwicklungsumgebungen für die einzelnen Plattformen separat kompiliert. Die dafür verwendeten Programme sind größtenteils frei erhältlich und können von jedermann eingesetzt werden.

Ziel dieser Arbeit ist die Entwicklung eines Spiels mit einer solchen Entwicklungsumgebung. Dabei werden die Konzeption, die Realisierung und der anschließende Test auf multiplen Plattformen dokumentiert. In der Konzeption wird die Planung mittels verschiedener UML-Diagramme und theoretischen Überlegungen gestaltet. Die Realisierung beschreibt die praktische Entwicklung. Der anschließende Test auf verschiedenen Plattformen soll den Erfolg dieser Methode verdeutlichen. Das Spiel soll aus einzelnen Modulen bestehen, die separat entwickelt werden und unabhängig voneinander sind. Dadurch erreicht man eine gesteigerte Flexibilität. Jedes Modul wird in einem separaten Abschnitt beschrieben, welcher Konzeption und Umsetzung gleichermaßen enthält.

Nicht in dieser Arbeit dokumentiert werden dabei die nach der Fertigstellung des Spiels folgende Vermarktung und der Vertrieb. Ebenso ist die Arbeitsverwaltung beziehungsweise -Organisation kein Bestandteil, obgleich er in einer normalen Spiele- oder Softwareentwicklung von großer Bedeutung ist.

Der zweite Abschnitt gibt einen allgemeinen Überblick über das Spielkonzept und führt Verknüpfungen zwischen verschiedenen Begrifflichkeiten ein, die im Laufe dieser Arbeit genutzt werden.

Im dritten Abschnitt, der Planung des Spiels, werden grundlegende Fragen zum Entwicklungsablauf beantwortet. Neben den Anforderungen zählen dazu unter anderem die Wahl des zu verwendenden Frameworks und der Entwicklungsumgebung sowie der Programmiersprache. Hinzu kommen Überlegungen zu grundlegenden Spielmechaniken und Anpassungen, welche sich aus den Anforderungen ergeben.

Der vierte Abschnitt handelt von der Planung und prototypischen Realisierung der Komponenten des Spiels. Hier erfolgt die Vereinigung von Konzepten aus dem dritten Abschnitt und den Anforderungen zur Planung. Diese Planung leitet dann nahtlos zur Realisierung weiter. Jede Komponente wird in einem eigenen Abschnitt erläutert. Zwischen den Komponenten werden die Zwischenstände erläutert, um ein Bild von den zuvor erläuterten Komponenten in Aktion zu sehen.

Der fünfte Abschnitt beschreibt anschließend den Test des Prototyps auf verschiedenen Plattformen. Im Test wird auf Kompatibilität und Spielbarkeit geprüft. Die folgende Auswertung des Tests zeigt Optimierungspotenzial. Zusätzlich zum Usability-Test geht der Abschnitt auf die Performance des Spiels ein. Ein Fazit beendet die Arbeit, welches gestellte und tatsächlich erreichte Ziele vergleicht, Optimierungsmöglichkeiten aufzeigt und einen Ausblick auf die zukünftige Entwicklung des Spiels gibt.

2 Überblick über das Spielkonzept

Der folgende Abschnitt gibt einen ersten Überblick über das Spielkonzept. Dabei werden die Spielwelt sowie einige wichtige Komponenten und Abläufe beschrieben. Des Weiteren werden essenzielle Begrifflichkeiten geklärt.

Das Spiel lässt den Spieler in eine fiktive Welt eintauchen, die nicht ausschließlich von Menschen dominiert wird. Überall leben neben den Menschen auch Monster, die **Mozos** genannt werden. Die Mozos leben zum Teil friedlich mit den Menschen, Seite an Seite, unterstützen die Menschen bei ihren Tätigkeiten oder sind einfach nur Begleiter. Allerdings gibt es auch unzählige Mozos, die nicht domestiziert wurden. Sie leben in der Wildnis, abseits der menschlichen Zivilisation. Sie sind meist sehr aggressiv gegenüber anderen Kreaturen und wollen sich ständig mit Ihnen rivalisieren. Dieses stark ausgeprägte Konkurrenzverhalten liegt in der Natur der Mozos und wurde schon oftmals zum Verhängnis für Leute, die ohne ausreichenden Schutz unterwegs waren. Diesen Schutz bieten unter anderem gezähmte Mozos, die dem Trainer im **Kampf** zur Seite stehen. Die Menschen haben jedoch auch eine Art Sport aus diesen Kämpfen gemacht. Sie lassen ihre Mozos bewusst gegen wilde Mozos oder Mozos von anderen Trainern kämpfen, um deren natürliche Art zu unterstützen und die Mozos zu stärken. Dabei geben die Trainer ihren Mozos Befehle, etwa welcher Angriff ausgeführt werden soll. Jedes Mozo hat neben seiner Standardattacke noch spezielle **Fähigkeiten** zur Verfügung. Fähigkeiten sind besondere Aktionen, die dem Rivalen schaden oder den Anwender stärken. Sie sind bei jedem Mozo unterschiedlich und können rein physisch oder elementar sein, beispielsweise ein Feuerball. Dabei kann das Mozo besonders jene Fähigkeiten gut beherrschen, die dem eigenen **Element** entsprechen. Im Vergleich könnte zum Beispiel ein Feuer-Mozo Fähigkeiten vom Typ Feuer effektiver einsetzen als ein Luft-Mozo. Beide Trainer geben ihren Mozos abwechselnd Befehle, es ist also ein **rundenbasiertes** Kampfsystem. Während des Kampfes kann der Trainer **Gegenstände (Items)** einsetzen, die den Kampfablauf beeinflussen. Diese Gegenstände, welche im **Inventar** des Spielers gesammelt werden, ändern bestimmte Merkmale des Mozos im Kampf. So erhöht ein Heiltrank die Lebenspunkte des Mozos. Lebenspunkte gehören zu den wichtigsten **Attributen** im Kampf. Sie sagen aus, wie viele Treffer ein Mozo noch einstecken kann, ehe es in Ohnmacht fällt und nicht mehr weiterkämpfen kann. Sollte eins der beiden kämpfenden Mozos

in Ohnmacht fallen, dann ist der Kampf vorbei. Nach einem erfolgreich gewonnenen Kampf erhält der siegreiche Trainer eine **Belohnung**. Diese kann unterschiedlich ausfallen. Der siegreiche Trainer kann entweder Geld, Gegenstände oder auch eine Information erhalten. Zusätzlich gewinnt das Mozo, welches den Kampf gewonnen hat, **Erfahrung** hinzu. Wenn es genug Erfahrung gesammelt hat, steigt es im Level auf und kann neue Fähigkeiten erlernen. Außerdem steigen die Anzahl der Lebenspunkte und die Wertigkeit anderer Attribute des Mozo.

Natürlich besteht die Spielwelt nicht nur aus Kämpfen. So kann der Spieler abseits von Duellen die Welt erkunden und dabei allerlei Aufgaben finden, welche abzarbeiten sind. Diese Aufgaben, im nachfolgenden auch **Quests** genannt, leiten den Spieler durch die Welt. Eine Quest kann beispielsweise daraus bestehen, dass der Spieler einen bestimmten Ort aufsucht und dort verschiedene Dinge erledigt. Er soll etwa einen Gegenstand suchen oder mit einer Person sprechen. Im Gespräch mit dieser Person, dem **Dialog**, ergeben sich eventuell neue Informationsquellen oder Aufgaben für den Spieler. Damit der Spieler seinen Spielstil verbessern oder anpassen kann, gibt es ein **Statistik- und Erfolgssystem**. Mittels dieses Systems können bestimmte Statistiken, wie die Anzahl der gewonnenen oder verlorenen Kämpfe, das gesammelte Geld oder die gefangenen Mozos festgehalten und vom Spieler ausgewertet werden. Das Erfolgssystem basiert auf dem Statistiksystem und schaltet bei bestimmten Meilensteinen einen Erfolg frei. Hat der Spieler beispielsweise 10 Mozos vom Element Wasser gefangen, so erhält er als Belohnung eine Haifischmütze, welche er sich aufsetzen kann. Diese Mütze verbessert dann die Effektivität seiner Wasser-Mozos im Kampf.

Das grundlegend fertig gestellte Spiel bietet bei einer Weiterentwicklung verschiedene Erweiterungsmöglichkeiten. Die Kämpfe könnten dynamischer gestaltet werden, indem nicht immer nur ein Mozo gegen ein anderes antritt, sondern drei Mozos gegen drei. Zur weiteren Steigerung des dynamischen Kampfes könnten bei diesen Mehrfachkämpfen auch Positionen dienen. Jedes Mozo steht dabei auf einer bestimmten Position auf dem Spielfeld und hat von dieser Position aus eine spezifische Reichweite. Des Weiteren könnte die Mechanik des Fangens beziehungsweise des Zähmens wilder Mozos in das Spiel gebracht werden, damit der Spieler die unterschiedlichsten Mozos aus der gesamten Spielwelt sammeln und zähmen kann.

3 Planung des Spiels

3.1 Anforderungen an das Spiel

Das Spielprinzip bringt eine Menge Funktionen mit, welche der Benutzer erwartet. In diesem Abschnitt wird beschrieben, welche Funktionen essentiell sind und umgesetzt werden sollen. Auch wird beschrieben, welche nicht implementiert werden können, da sie über den Rahmen dieser Arbeit hinausgehen würden.

Plattform

Das Spiel soll nicht für eine spezielle Plattform entwickelt werden. Stattdessen soll eine Multiplattform-Anwendung erstellt werden, welche auf verschiedenen Plattformen lauffähig ist. Dazu zählen natürlich der PC sowie Laptops mit Windows und Linux-Betriebssystem, mobile Geräte mit den dominierenden Betriebssystemen Android, iOS und Windows Phone und ebenso Konsolen - unter anderem die Playstation 4 und XBOX 360. Dadurch, dass das Spiel nicht auf eine Plattform und damit auch nicht auf eine Bildschirmauflösung limitiert wird, muss das Spiel abhängig von der Auflösung skalieren. Das heißt, bei höheren Auflösungen wie Ultra-HD (3840*2160), soll das Spiel genauso aussehen, wie auf niedrigen Smartphone-Auflösungen (960*640). Dazu müssen die verschiedenen Spielelemente dynamisch skaliert werden. Auch die Steuerung muss auf die Unabhängigkeit vom System angepasst werden. So existieren auf den verschiedenen Plattformen diverse Steuerungsmöglichkeiten – es soll eine Möglichkeit gefunden werden, diese Steuerungsmöglichkeiten einheitlich zu nutzen.

Funktional soll das Spiel aus verschiedenen Komponenten bestehen. Im Folgenden werden diese Komponenten erläutert.

Spieler

Der Spieler ist die Figur, die der Anwender im Spiel steuert. Sie bewegt sich durch die Spielwelt, erledigt Aufgaben, sammelt Gegenstände und spricht mit anderen NPCs (Not-Player-Character).

Spielwelt

Die Spielwelt ist die virtuelle Umgebung, in der sich der Spieler zum großen Teil befindet, und mit zahlreichen Objekten und NPCs interagieren kann. Außerdem werden hier die Handlung des Spiels vorangetrieben und Regeln festgelegt, an die sich der Spieler halten muss. Die Relevanz hängt dabei meist vom **Genre** des Spiels ab. Bei Denk- und Geschicklichkeitsspielen beispielsweise, ist die Spielumgebung meist nebensächlich und vernachlässigbar, wohingegen bei Abenteuer- und **Rollenspielen** die Erforschung der Spielwelt und die Mechanismen der Umgebung eines der Kernelemente darstellt und maßgeblich die Qualität des Spiels bestimmt. Dementsprechend soll im Laufe dieser Arbeit auch ein hoher Wert auf die Gestaltung der Spielwelt gelegt und mit Hilfe eines eigenen, im Framework integrierten, **Karteneditors** umgesetzt werden. Dies bedeutet zwar einen deutlichen Mehraufwand zu Beginn der Entwicklung, doch später lässt sich so die Spielwelt, mit wenigen Handgriffen, beliebig erweitern und verändern.

Folgende Funktionalitäten sollen hierfür implementiert werden:

- Das Einblenden eines Rasters in der Szene.
- Das Visualisierung des Feldes, auf der sich die Maus im Raster befindet.
- Die Übersicht aller platzierbaren Objekte.
- Die Möglichkeit zur Generierung von großen Kartenflächen (z.B. 20x20 Objekte).
- Das Platzieren einzelner/mehrerer Objekte in der Szene.
- Das Löschen einzelner/mehrerer Objekte in der Szene.
- Eine schnelle Bearbeitung der platzierbaren Objekte.

Mozos

Mozos sind eine zentrale Komponente im Spiel. Im Rahmen dieser Arbeit werden die Mozos nur im Team des Spielers dargestellt, in einer späteren Version wird der Spieler sie auch fangen und trainieren können. Sie besitzen spezielle Attribute, mit welchen sich jedes Mozo von einem anderen unterscheidet.

Inventar

Im Inventar findet der Spieler seine Gegenstände wieder, welche er im Laufe des Spiels gesammelt hat. Diese Gegenstände soll der Spieler verwalten können.

Zu diesem Zweck soll Inventar folgende Funktionen bieten:

- Items benutzen – Der Spieler soll die Items, welche er gefunden hat, benutzen können. Items haben Effekte, welche auf das gewählte Ziel angewendet werden.
- Items sortieren – Die Item-Liste soll zwecks Übersichtlichkeit sortiert werden. Als Sortier-Schema kann zwischen dem „Manuellen“- , „A bis Z“- , „Normal“- und dem „Kampf“-Schema gewählt werden. Die Auswirkungen der einzelnen Optionen werden im Inventarabschnitt näher erläutert.
- Item ablegen – Wenn der Spieler ein Item nicht mehr benötigt oder möchte, dann soll er es ablegen können.
- Zwischen den Item-Arten wechseln – Die Items können in verschiedene Kategorien unterteilt werden. Der Spieler soll zwischen diesen Kategorien wählen können.

Des Weiteren soll das Team des Spielers, welches aus verschiedenen Mozos besteht, angezeigt werden.

Aufgabensystem

Das Aufgabensystem (oder Questsystem) soll den Spieler durch die Spielwelt leiten. Dazu werden ihm von verschiedenen Quellen Aufgaben übertragen. Wenn der Spieler diese Aufgaben erfüllt, erhält er dafür Belohnungen. Das Aufgabensystem soll eine GUI-Oberfläche haben, in welcher der Spieler zwischen den Aufgaben wechseln kann. In dieser GUI-Oberfläche Details werden auch zur aktuellen Aufgabe angezeigt, etwa das Ziel und die Belohnungen.

Bewegungssystem

Der Spieler agiert in einer Welt, in welcher er sich selbst bewegen kann. Je nach ausgewähltem Steuerungssystem könnte hier beispielsweise optional noch ein Algorithmus zur Wegfindung mit eingebaut werden.

Dialoge

Dialoge dienen zur Interaktion zwischen Spieler, Mozos, NPCs (non-player character) und Gegenständen.

Zu diesem Zweck soll das Dialogsystem folgende Anforderungen erfüllen:

- Jeder Dialog besteht aus n Dialogschritten. Jeder dieser Schritte beinhaltet einen Text.
- Ein Dialogschritt ist standardmäßig ein Monolog, in dem der Spieler mit einem „Ok“-Button zum nächsten Dialogschritt wechseln kann. Optional kann der Dialogschritt als Frage deklariert werden. Damit erhält der Spieler die Möglichkeit, mittels „Ja“- und „Nein“-Buttons auf eine Frage zu Antworten.
- Das Drücken auf die einzelnen Buttons löst eine Weiterleitung zum nächsten Dialogschritt aus.
- Dem Dialog sollen optionale Komponenten hinzugefügt werden können – etwa der Name oder ein Bild des Dialogpartners.

Grafik und Animation

Die Spielwelt soll isometrisch dargestellt werden. Es müssen Grafiken für die Welt, Charaktere, Kreaturen und Gegenstände erstellt werden. Bestimmte Komponenten, wie die Duelle, müssen animiert werden. Die Grafiken sollen anhand der gewählten Auflösung dynamisch skalieren.

Audio

Für das Hauptmenü und das Spiel ist eine Hintergrundmusik zu implementieren. Zusätzlich sollen beispielhaft Effekte bei bestimmten Aktionen, etwa wenn der Spieler eine Aktion bestätigt, abgespielt werden.

Lokalisierung

Die Lokalisierung soll dafür sorgen, dass das Spiel auch in unterschiedlichen Sprachen gespielt werden kann. Der Spieler vermag dann mittels verschiedener Optionen die jeweilige Sprache zu wählen. Zusätzlich könnte die Systemsprache ausgelesen und dann als Standardsprache gesetzt werden.

Abgrenzungskriterien

Das Spiel stellt einen Prototyp dar, welcher nicht dieselbe Funktionsfähigkeit wie ein fertiges Spiel aufweist. Beispielsweise können bestimmte Einstellungen zur

Personalisierung und die Hilfe fehlen. Einige Grafiken und Bilder entsprechen noch nicht denen, die in der finalen Version ihren Einsatz finden.

3.2 Wahl der grafischen Umsetzung

Ein weiterer wichtiger Aspekt der Spielentwicklung ist die grafische Umsetzung. In diesem Abschnitt wird beschrieben, wie genau die Spielinhalte dem Spieler dargestellt werden sollen. Grundsätzlich sind hierfür zwei Faktoren verantwortlich, die **Dimensionalität** und die **Perspektivität**.

Dimensionalität

Im Allgemeinen kann eine Spielwelt **dreidimensional (3D)** oder **zweidimensional (2D)** visualisiert werden. Das bedeutet, dass auf einer 3D-Ebene die Elemente (Modelle) in drei Dimensionen, also auf der x-, y- und z-Achse, im Raum dargestellt werden können. Dabei werden weiter entfernte Elemente (z-Achse) kleiner dargestellt. Man spricht dann von einer **perspektivischen Projektion**. Auf der gezeichneten 2D-Ebene hingegen können die Elemente (Sprites) nur auf der x- und y-Achse abgebildet werden. Hier ist die perspektivische Tiefe nicht vorhanden und die Elemente behalten dementsprechend auch immer die gleiche Größe, egal wie weit oder nah sie von der Kamera positioniert sind. Dies wird als **orthogonale Projektion** bezeichnet.

In Abbildung 1 wurden beide Projektionen gegenübergestellt. Dabei sind die Koordinaten der jeweiligen vier Würfel in beiden Szenen identisch, lediglich die Projektionsart unterscheidet sich.

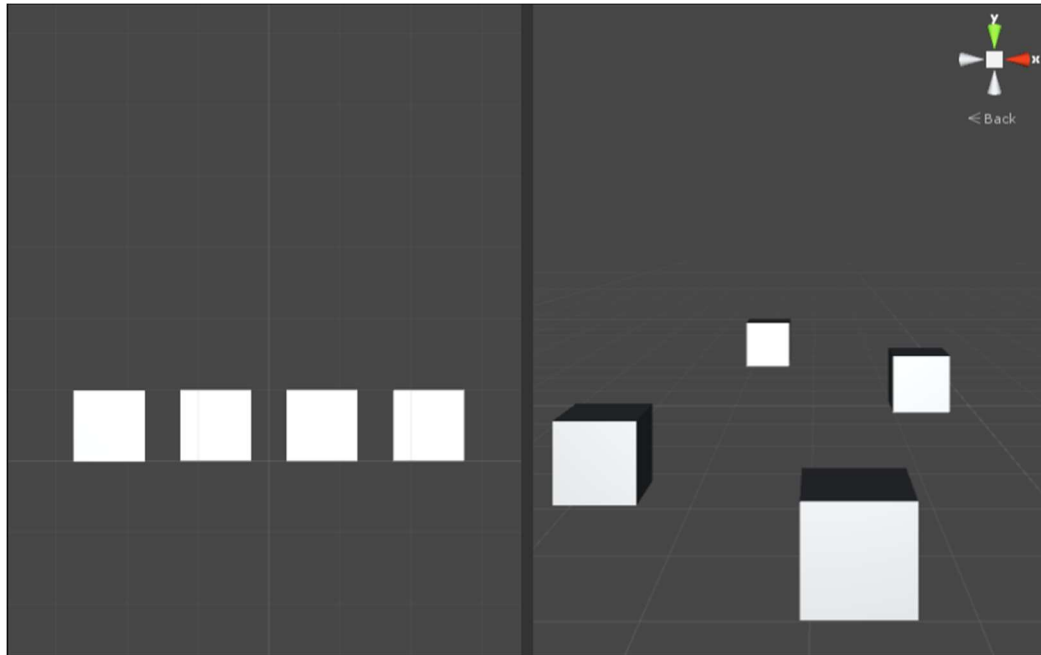


Abbildung 1: Vergleich orthogonaler und perspektivischer Projektion

Perspektivität

Die Perspektive, auch Kameraperspektive, legt fest, in welcher Form der Spieler auf das Spielgeschehen schaut und nach welchem Steuerungsprinzip grundsätzlich navigiert wird. Es gibt, abhängig vom verwendeten Genre und von der Dimensionalität, unterschiedlich geeignete Darstellungsmöglichkeiten (siehe Tabelle 1).

Perspektive	Ansicht (Kamera)	Dimension	Genre (Beispiel)
First-Person Ich-Perspektive	Aus Sicht der Spielfigur	3D	Shooter, Simulation
Third-Person Verfolgerperspektive	Hinter der Spielfigur	3D	Action, Adventure, Rollenspiel
Side-Scrolling Seitenansicht	Von der Seite	2D, 3D	Jump 'n' Run, Beat 'em up
Top-Down Vogelperspektive	Von oben	2D, 3D	Rollenspiele, Shooter Action-Adventure
Isometrie Überblicksperspektive	Von schräg oben	2D, 3D	Rollenspiele, Strategie

Tabelle 1: Überblick über die gängigsten Perspektiven [1]

Hierbei ist zu beachten, dass die Ego- und Third-Person-Perspektive reine 3D-Darstellungformen sind und nicht in 2D gerenderten Spielwelten eingesetzt werden können. Jedoch ist es technisch möglich, alle 2D-Darstellungen auch in 3D zu adaptieren. So lässt sich zum Beispiel ein Top-Down-Spiel auch mit 3D-Grafiken umsetzen.

Mit Hilfe von Perspektiven ist es zudem möglich, in einer zweidimensionalen Visualisierung einen dreidimensionalen Effekt zu erzielen. Dabei wird dem Spieler eine räumliche Tiefe vorgetäuscht, die es technisch gesehen gar nicht gibt. Solch eine Kombination aus 2D und 3D wird auch **zweieinhalbdimensional** (2.5D) genannt und lässt sich zum Beispiel durch eine **isometrische Perspektive** erzielen.

Wahl der Darstellung

Welche grafische Umsetzung ausgewählt werden soll, hängt vom verwendeten Genre, von der Ästhetik und ausschlaggebend von der notwendigen Einarbeitungszeit ab.

Vergleicht man zwecks Entscheidungsfindung zunächst die Dimensionalitäten miteinander, wird ersichtlich, dass 3D viele Vorteile und grafische Möglichkeiten bietet. Dazu gehören z.B. freie Kamerafahrten, Partikeleffekte, dynamische Schattenverläufe sowie verschiedene Lichtquellen. Außerdem lassen sich die Modelle, im Gegensatz zu 2D-Elementen, ohne großen Aufwand animieren. Der Grund dafür ist, dass sich alle Teile des 3D-Modells komplett frei im Raum bewegen und rotieren lassen. Beim 2D-Modell hingegen muss für jede Bewegungsphase, also für jeden einzelnen Frame einer Animation, eine Zeichnung erstellt werden. Je flüssiger die Animation sein soll, desto mehr Frames müssen gezeichnet werden. Das macht die 2D-Lösung in diesem Punkt wesentlich zeitintensiver als die 3D-Lösung. Der Aufwand für die Einarbeitungszeit und für die Objektmodellierung bei 3D fällt jedoch deutlich höher aus als bei 2D und würde den Rahmen dieser Arbeit bei weitem übersteigen. Dies gilt vor allem dann, wenn dabei ein hoher Detailbeziehungsweise Realismusgrad angestrebt wird.

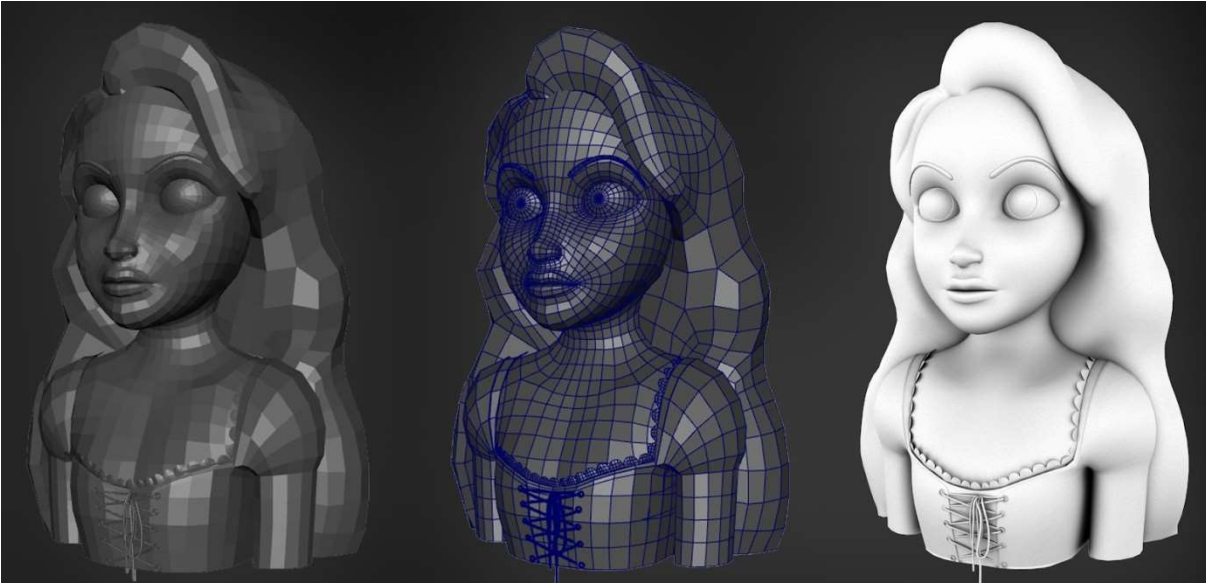


Abbildung 2: Beispiel eines 3D-Modells [2]

Aus den oben genannten Gründen wurde eine 2D-Spielumgebung gewählt. In einer 2D-Spielumgebung wird eine überschaubare Vielfalt an grafischen Optionen und Freiheiten geboten, die eine relativ schnelle Einarbeitung ermöglichen, und zugleich der angestrebten klassischen Rollenspieloptik entgegenkommt.

In der gewählten Dimensionalität stehen die **Side-Scrolling**, die **Top-Down** und die **isometrische Ansicht** als Perspektive zur Auswahl (siehe Tabelle 1). Diese werden nachfolgend näher beschrieben.

Bei der **Side-Scrolling** Ansicht wird das Spielgeschehen von der Seite betrachtet. Das bedeutet, die Spielfigur lässt sich in der Spielumgebung tatsächlich auch nur in zwei Dimensionen, also auf der x- und y-Ebene, bewegen (links-rechts und unten-oben). Diese Art der Darstellung lässt sich zwar sehr leicht umsetzen, schränkt jedoch die Übersicht sowie die Freiheiten zur Interaktion und Erkundung der Spielwelt



Abbildung 3: Eines der bekanntesten Side-Scroller (Super Mario Bros.) [31]

enorm ein. Aus diesem Grund ist diese Ansicht meist auch nur für Spiele mit linearem Ablauf und klaren Levelstrukturen geeignet.



Abbildung 4: Eine typische Top-Down-Ansicht (The Legend of Zelda) [32]

Die **Top-Down**-Ansicht ist vor allem bei den klassischen Rollenspielen eine sehr oft genutzte Darstellungsform. Hier schaut der Spieler von oben auf die Spielwelt. Zwar gibt es bei dieser Form grundsätzlich auch nur zwei mögliche Bewegungsrichtungen, jedoch sind hier die y- und z-Ebene miteinander vertauscht, was dazu führt, dass sich, im Gegensatz zur Seitenansicht, mehr Elemente von der Spielumgebung darstellen lassen. Die Vorteile sind

eine gute Übersicht über die Spielumgebung und eine sowohl grafisch als auch technisch relativ einfache Umsetzung. Da es jedoch unter Umständen recht schwer sein kann, bei solch einer direkten Draufansicht die Umgebung ästhetisch und detailreich darzustellen, wird die Perspektive oft etwas schräg abgebildet, meist auch auf Kosten einer korrekten perspektivischen Ansicht.

Bei der **isometrischen** Perspektive schaut der Spieler von schräg-oben auf das Spielgeschehen. Dadurch ist es möglich, die Spielumgebung mit einer zusätzlichen sichtbaren Raumtiefe (z-Ebene) darzustellen. Anders als bei der Top-Down Ansicht wird hier aber keine Ebene ersetzt,



Abbildung 5: Ausschnitt aus einem isometrischen Spiel (Landstalker) [30]

sondern eine zusätzliche Ebene durch Kombination der x- und y-Koordinaten ermittelt. Dies ermöglicht ein höheres Potential an Steuerungs- und Interaktionsmöglichkeiten sowie eine bessere Übersicht der Spielwelt. Deshalb eignet sich diese Darstellungsform auch für Spiele mit komplexem Spieldesign, so wie es oft bei Strategie- und Rollenspielen vorkommt. Die isometrische Perspektive ist die grafisch und technisch aufwendigste Perspektive, da mit dem speziellen Blickwinkel bzw. der zusätzlichen Tiefe weitere Regeln und Besonderheiten zu berücksichtigen sind. Hierzu mehr bei der Implementierung der Spielwelt (Abschnitt 4.3.1).

Die isometrische Perspektive wird an dieser Stelle als Spielperspektive gewählt. Sie überzeugt mit ihren Vorteilen gegenüber der Side-Scrolling- und Top-Down-Ansicht und bietet die beste Alternative zur Dreidimensionalität.

3.3 Wahl des Entwicklungsframeworks und der Programmiersprache

In diesem Abschnitt wird die Frage geklärt, mit welchem Entwicklungsframework und welcher Programmiersprache das Spiel implementiert werden soll. Dazu werden verschiedene Frameworks hinsichtlich ihrer Tauglichkeit für das Projekt verglichen. Jedes der Frameworks hat dabei folgende Bestandteile in der Engine:

Grafik

Die Grafik-Engine ist für die Darstellung der Spielinhalte am Bildschirm verantwortlich. In allen modernen Frameworks werden dabei 3D-Engines verwendet, die ein dreidimensionales Bild erzeugen. Jede Grafik-Engine bringt verschiedene Elemente und Effekte mit, welche die Darstellung stark ändern können. Zu diesen Effekten zählen beispielsweise die Beleuchtung von Objekten und Räumen, das Hervorheben von Objekten oder die Anpassbarkeit der Grafik.

Physik

Eine Physik-Engine „ist eine Computer-Software, mit der die Gesetzmäßigkeiten der Physik simuliert werden können“ [3]. Jede Engine versucht, möglichst nah an die Physik der realen Welt heranzureichen. Dafür berechnet die Engine mithilfe von physikalischen Gesetzen die gegebene Situation und gibt einen entsprechenden

Output. Aufgrund dieser Berechnungen ist die Physik-Engine zusammen mit der Grafik eine der Komponenten, die die meiste Rechenlast erzeugen.

Sound

Das Soundsystem gibt die Hintergrundmusik und Geräusche eines Spiels aus. So werden bei definierten Ereignissen Effekte erstellt, um dem Spiel mehr Intensität zu geben.

Steuerung

Beim Steuerungssystem werden die verschiedenen Eingabemöglichkeiten verarbeitet, etwa Eingaben per Touchscreen, Gamepad oder Tastatur und Maus. Diese Eingaben müssen präzise verarbeitet werden, damit der Spieler keine Verzögerung zwischen Eingabe und Reaktion bemerkt.

Netzwerk

Die Netzwerk-Engine bietet verschiedene Komponenten, um ein Spiel so zu erstellen, dass verschiedene Spieler zusammen über das Internet spielen können.

Engine-Vergleich

Folgende Engines sollen auf ihre Tauglichkeit hinsichtlich des Projektes geprüft werden:

1. Die Unity3D-Engine des Herstellers Unity Technologies [4].
2. Die Unreal-Engine von Epic Games [5].
3. Die Frostbite-Engine von Frostbite [6].
4. Die CryEngine des Herstellers CryTek [7].

Die Engines wurden aufgrund ihrer Popularität gewählt. Um eine möglichst gute Vergleichbarkeit erreichen zu können, werden folgende Punkte untersucht:

1. Plattformen: Welche Plattformen können über die Engine erreicht werden?
Wird der Code nativ implementiert?
2. Lizenz: Was kostet die Entwicklung mit der Engine, erwartet der Hersteller eine Gewinnbeteiligung?

3. Grafik/Physik/Sound-Engine: Wie gut sind die verschiedenen Engine-Komponenten für das Projekt und seine Anforderungen gerüstet?
4. Programmiersprachen: Mit welchen Sprachen kann das Projekt innerhalb des Entwicklungsframeworks umgesetzt werden?

Dabei sollen natürlich möglichst viele Plattformen erreicht werden. Auch Lizenzkosten sind im Idealfall nicht vorhanden. Die Engine-Komponenten sollen Methoden zum Verwenden von 2D-Grafiken und –Physik-Berechnungen mitbringen. Als Programmiersprache wäre eine moderne Hochsprache wie C# oder Java wünschenswert. Je mehr dieser optimalen Eigenschaften eine Engine mitbringt, desto wahrscheinlicher ist der praktische Einsatz in dieser Arbeit.

Die zu untersuchenden Punkte wurden auf den jeweiligen Hersteller-Websites recherchiert, mit folgendem Ergebnis:

	Unity3D	Unreal	Frostbite	CryEngine
Plattformen	Mobile Geräte, Konsolen, PC, Web-Browser	PlayStation, Xbox, PC	PlayStation, Xbox, PC	PlayStation, Xbox, PC, Wii
Lizenz	Unter 100.000\$ Umsatz gratis, Ansonsten 80€/Monat	Ab 3000\$ Quartals-Umsatz 5% Beteiligung, Ansonsten Gratis	Geschlossen	10€
Grafik	3D- und 2D-Engine	3D- und 2D-Engine	3D-Engine	3D-Engine
Physik	3D- und 2D-Physik	3D- und 2D-Physik	3D-Engine	3D-Engine
Sound	Vorhanden	Vorhanden	Vorhanden	Vorhanden
Programmiersprachen	C#, Python, JavaScript	C++	C++	C++

Tabelle 2: Vergleich der Entwicklungs-Frameworks [8] [9] [10]

Betrachtet man nur die Funktionen, welche eine Game-Engine bieten soll, so wird recht schnell deutlich, dass alle verglichenen Frameworks die nötigen Komponenten enthalten und in ihrer Mächtigkeit recht ähnlich sind. Kommt es allerdings zum Lizenzmodell und den Plattformen, für die mithilfe der Engine Spiele entwickelt werden können, so zeigen sich Unterschiede auf. Dabei erreicht man mit der Unity-Engine die meisten Spieler, da hier im Gegensatz zu den anderen Engines auch mobile Geräte, sämtliche Konsolen und auch das Spielen per Webbrowser unterstützt wird. Auch beim Lizenzmodell hat Unity3D bei kleineren Projekten Vorteile. Des Weiteren wird unter anderem die Programmiersprache C# unterstützt, welche weitere Vorteile wie eine Einbindung des .NET-Frameworks von Microsoft bietet. Aufgrund der gegebenen Vorteile der UnityEngine gegenüber den anderen Engines wird Unity als Entwicklungsumgebung für das Projekt genutzt.

Unity bietet eine separate Engine für die Entwicklung von 2D- und 3D-Spielen. Die beiden Engines unterscheiden sich etwa bei der Physik. Eine Kugel würde also in einer 3D-Welt anders von einem Gegenstand abprallen als in einer 2D-Welt. Die Entwicklungsumgebung der UnityEngine, welche ebenfalls den Namen Unity trägt, ist das zentrale Element in der Spielentwicklung. Innerhalb dieser Entwicklungsumgebung werden Assets¹, Projektklassen, Spielobjekte, Animationen und alle restlichen Komponenten der Spielentwicklung verwaltet. Dazu setzt die Unity-Entwicklungsumgebung auf verschiedene GUI-Komponenten, die sich modular ein- und ausblenden lassen. Die wichtigsten werden an dieser Stelle kurz erläutert.

Hierarchieansicht

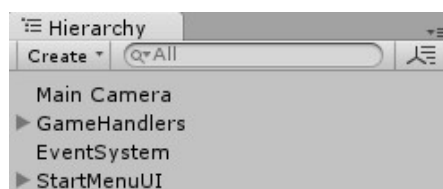


Abbildung 6: Hierarchieansicht

¹ Assets sind vorgefertigte Dateien wie Grafiken oder Textdateien.

Abbildung 6 zeigt die Hierarchieansicht. In dieser Hierarchie werden die Spielobjekte angezeigt, welche in der aktuellen Szene² verwendet werden. Jedes Objekt kann Kinderobjekte beinhalten, welche dann ebenfalls weiter verzweigt sein können. Die Hierarchieansicht hilft, den Überblick über diese Objekte zu behalten. Es können neue Objekte erzeugt werden, zusätzlich können die bereits bestehenden durchsucht werden.

Projekthierarchie

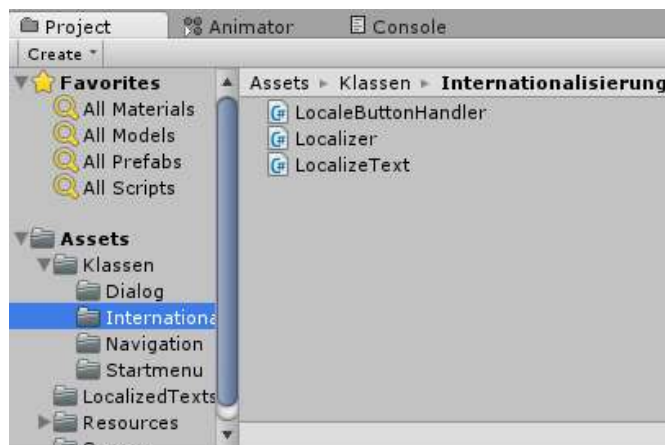


Abbildung 7: Projekthierarchie

Die Projekthierarchie, welche in Abbildung 7 zu sehen ist, stellt die im Projekt vorhandenen Assets wie Klassen, Bilder oder Textdateien dar. Des Weiteren kann der Animator für die Animationen und die Konsole geöffnet werden.

Szenenansicht



Abbildung 8: Szenenansicht

² Eine Szene kann in Unity als ein Level verstanden werden, in dem sich der Spieler bewegen kann. Aber auch andere Spielbestandteile, beispielsweise das Hauptmenü, werden in einer separaten Szene gespeichert. Jede Szene hat ihre eigenen, von den anderen Szenen getrennten Spieleobjekte.

In der Szenenansicht (siehe Abbildung 8) wird eine Vorschau für den Entwickler erzeugt, damit Spielobjekte wie Buttons und Textfelder selektiert, angeordnet und zueinander positioniert werden können. Sobald ein einzelnes Objekt selektiert ist, werden die zugehörigen Details im Inspector aufgeführt. Eine Szene kann ein Menü, aber auch ein Level sein. Der Entwickler entscheidet, was sich in einer Szene abspielt.

Inspector

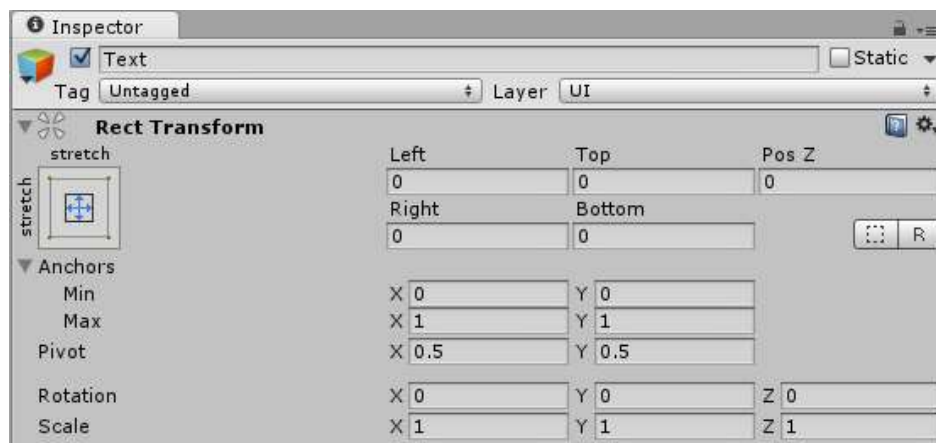


Abbildung 9: Detailansicht eines Objektes im Inspector

Abbildung 9 zeigt den Inspector. Im Inspector werden alle Komponenten inklusive ihrer veränderbaren Eigenschaften angezeigt. Diese Komponenten sind für das Verhalten eines Spielobjektes zuständig.

Natürlich können noch weitere Fenster zum Unity-Editor hinzugefügt werden.

Diese sind dann allerdings auf spezielle Aufgabenbereiche zugeschnitten und werden daher an dieser Stelle nicht näher erläutert.

Mithilfe dieser grundlegenden Komponenten kann die Spielentwicklung gestartet werden. Wie bereits beschrieben nutzt Unity Spielobjekte und Komponenten, welche an diese Objekte angehängt werden. Das ist die wichtigste Mechanik in Unity - jedes Objekt nutzt Komponenten, die das Verhalten und die Interaktion mit anderen Komponenten definieren. Ein einfaches Label, welches auf dem UI des Spiels einen

Text anzeigen soll, wird aus verschiedenen Komponenten zusammengesetzt. Diese Komponenten ergänzen sich und ergeben in der Gesamtheit das UI-Label.

Dabei ist, wie in Abbildung 10 zu sehen, ein Spielobjekt beziehungsweise *GameObject*, immer das zu Grunde liegende Objekt.

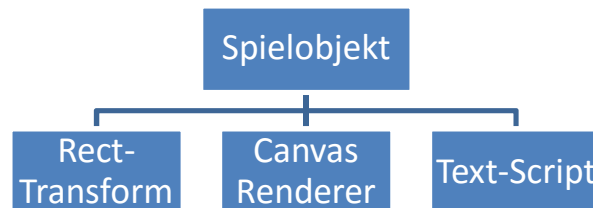


Abbildung 10: Aufbau eines GUI-Labels

An dieses Objekt werden dann weitere Komponenten angehängt. Die Rect-Transform-Komponente kommt bei allen GUI-Objekten in Unity vor, sie dient zur Positionierung, Skalierung und Rotation von Spielobjekten. Die Canvas-Renderer-Komponente dient zur Darstellung. Schließlich hat das Objekt eine Text-Script-Komponente, welche benutzerdefinierte Texte auf das Spieleobjekt zeichnet. Diese Anordnung zieht sich durch die gesamte Spielentwicklung mittels Unity. Wenn zum Beispiel ein Spieler implementiert werden soll, könnte ein neues Spielobjekt erzeugt werden. Anschließend werden selbst implementierte Komponenten wie ein Inventar oder eine Geldbörse angehängt.

Aber auch bei der eigentlichen Implementierung der Klassen bringt die Unity-Engine einige nützliche Funktionen mit. So kann der Entwickler spezielle Methoden implementieren, welche dann bei bestimmten Ereignissen aufgerufen werden. Die wichtigsten beiden Methoden sind *Start()* und *Update()*. Die *Start()*-Methode wird aufgerufen, sobald die Komponente, an die die Klasse angehängt wurde, initialisiert wird. Danach wird, so oft es geht, die *Update()*-Methode aufgerufen. Hier kann dann Logik implementiert werden, die während der Laufzeit ausgeführt werden soll.

3.4 Anpassungen für das Multiplattformkonzept

3.4.1 Ein- und Ausgabemöglichkeiten

Nachdem eine Engine für das Entwickeln von Spielen für multiple Plattformen gefunden wurde, muss erörtert werden, welche Konsequenzen aus diesem Konzept für die unterschiedlichen Zielplattformen folgen.

Bei den Eingaberäten unterscheiden sich die verschiedenen Möglichkeiten am deutlichsten. So haben alle drei Zielplattformen, und zwar Konsole, PC und Smartphone unterschiedliche Wege zur Eingabe.

Konsolen

Alle Konsolen lassen sich über Controller bedienen. Diesen Controller nimmt der Spieler in beide Hände, um damit das Spielgeschehen zu steuern.



Abbildung 11: Controller einer PlayStation 4-Konsole [11]

Grundsätzlich sind die Controller aller aktuellen Konsolen gleich aufgebaut: Es gibt auf der linken Seite ein Steuerkreuz (1), welches zur Bewegung des Spielecharakters dient. Auf der rechten Seite sind meist vier weitere Tasten angeordnet, die Aktionstasten (2). Mit diesen Tasten kann der Spieler dem Spielcharakter in der Spielumgebung agieren lassen. Er nimmt beispielsweise Gegenstände auf oder spricht mit NPCs. Oft stehen mit den Schultertasten (3) dem Spieler auch noch zusätzliche Aktionen zur Verfügung. Obendrein besitzen die Controller noch zwei Analog-Sticks (4), die zur genaueren Steuerung genutzt werden können, etwa um ein Fadenkreuz zu bewegen. Meist gibt es auch noch Knöpfe (5) die nur indirekt oder gar nicht das Spielgeschehen beeinflussen, um beispielsweise das Hauptmenü der Konsole zu öffnen.

PCs

Die Steuerung von Spielen an einem PC kann mittels Maus und Tastatur erfolgen. Je nach Spiel wird eines dieser Eingabemedien stärker oder ausschließlich benutzt. So wird die Maus zum Steuern des Charakters und zur Interaktion mit der Spielwelt genutzt, mit der Tastatur können dann spezielle Funktionen oder Menü-Aufrufe gestaltet werden.

Smartphones

Ein Smartphone wird ausschließlich via Touch bedient. Das heutige Smartphone besitzt meist nur noch wenige physische Tasten, etwa um das Hauptmenü aufzurufen. Hier wird die Steuerung also entweder über das direkte Behandeln der Touch-Bedienung oder über für mobile Geräte speziell zugeschnitten UI-Bedienkonzepte erledigt. Bei der Bedienung über UI-Elemente werden etwa virtuelle Joysticks eingeblendet, damit der Spieler die Spielfigur bewegen kann.

Nachdem nun die unterschiedlichen Bedienmöglichkeiten der Zielplattformen vorgestellt wurden, muss ein Konzept gefunden werden, welches erlaubt, alle Plattformen mit einer guten Steuerungsmöglichkeit zu versehen. Dabei ist zu beachten, dass der PC und die Konsolen theoretisch über dasselbe Konzept gesteuert werden könnten. Die Richtungstasten des Controllers können auf die Tastatur und der Joystick kann auf die Maus appliziert werden. Dabei bleiben allerdings die mobilen Geräte unbeachtet. Um auch eine Steuerung für ein Smartphone anbieten zu können, gibt es nun mehrere Möglichkeiten. Zum einen könnte das Spiel je nach Zielplattform eine native Steuerung anbieten. Dann allerdings müssten für jede Plattform eigene Versionen des Spiels implementiert werden, zumindest für die Steuerung. Ein anderes Konzept wurde bereits bei der Vorstellung der Smartphone-Steuerung erwähnt. Während Konsole und PC relativ gleich bedient werden können, wird beim Smartphone eine virtuelle, über das UI erstellte Steuerungseinheit angezeigt. So lassen sich alle Plattformen mit einem einzigen Steuerungskonzept spielen.

Um eine adäquate Steuerung zu ermöglichen, muss die virtuelle Steuerungseinheit mindestens folgende Komponenten bieten:

- Vier Richtungstasten, um den Spielcharakter zu bewegen oder in Dialogen eine Auswahl zu tätigen,
- eine Taste zum Bestätigen und eine zum Abbrechen und
- eine Taste um das Menü aufzurufen.

Ausgehend vom eigentlichen Ziel, dem Erstellen eines Spiels für multiple Plattformen, werden sich die Smartphones über eine virtuelle UI-Steuerung und die Konsolen bzw. PCs über das gewohnte Steuerungskonzept bedienen lassen. Eine native Steuerung für jede Zielplattform würde dem Ziel „Ein Spiel für alle Plattformen“ entgegenwirken. Wie passend die UI-Steuerung für das Smartphone ist, wird in Abschnitt 5.1 Usability-Test auf verschiedenen Plattformen erörtert.

Ausgabemöglichkeiten

Die verschiedenen Plattformen verwenden Bildschirme als Ausgabemöglichkeit, die unterschiedliche Auflösungen besitzen können. Die gängigste Auflösung unter Konsolen- und PC-Bildschirmen ist Full-HD mit 1920 mal 1080 Pixeln [12] [13]. Bei den mobilen Geräten sieht es dagegen differenziert aus: Full-HD kommt hier nur auf eine Verbreitung von 4,6 Prozent. Die iPhone-Standard-Auflösung von 960 mal 640 Pixel kommt auf 8,8 Prozent [14]. Damit sich Entwickler nicht um viele verschiedene Auflösungen separat kümmern müssen, skaliert die Unity-Engine UI-Elemente und Grafiken so, dass sie für die aktuelle Auflösung passend sind. Der Entwickler muss nur noch festlegen, welche Elemente sich in welchem Verhältnis zu anderen Elementen skalieren.

3.4.2 Rechenleistung

Ebenso wie die Auflösungen von Bildschirmen unterscheidet sich die Leistung der Hardware, welche die einzelnen Plattformen verbaut haben. Während Konsolen innerhalb derselben Generation die gleiche Hardware nutzen, unterscheidet sich beim PC die Hardware meist von Gerät zu Gerät. Mobile Geräte haben zwar wie Konsolen in einer Generation die gleiche Hardware, jedoch ist die Situation, ausgehend von den vielen verschiedenen Geräten, mit dem PC vergleichbar. In der

Leistungsfähigkeit sind gute PCs die Geräte mit der höchsten Leistung. Konsolen sind nicht so leistungsfähig wie die PCs. Mobile Geräte erreichen in der Regel die niedrigste Leistung, was bei der Größe der Geräte nicht verwunderlich ist [15]. Die Unity-Engine bietet zur Verringerung der Rechenlast verschiedene Optionen an. Eine Option besteht im Wählen einer Auflösung, welche nicht der nativen Auflösung des Endgerätes entspricht. Hat ein Smartphone beispielsweise eine Auflösung von $1920 * 1080$, dabei allerdings schwache Hardware, kann eine niedrigere Auflösung wie $1024 * 768$ gewählt werden. Damit sind die grafischen Ausgaben des Spiels weniger aufwändig und können schneller berechnet werden. Eine weitere Möglichkeit ist die Textur-Kompression. Dabei werden Texturen weniger detailreich dargestellt, was wiederum zugunsten der Rechenlast erfolgt. Auch die Audio-Daten, wie etwa Hintergrundmusik, haben Einsparpotenzial. Solange Audiodaten im Smartphone-Arbeitsspeicher liegen, ist dort weniger Platz für andere Daten. Diese Audiodaten können auch direkt von dem Festspeicher des Geräts abgespielt werden, was weniger Auslastung verursacht. Natürlich gibt es noch weitere Möglichkeiten, aus Gründen des Umfangs wurden hier jedoch nur die wichtigsten vermerkt.

4 Planung und prototypische Realisierung der Komponenten

4.1 Ablauf und Aufbau der Planung und der Realisierung

Der folgende Abschnitt beschreibt die Planung und prototypische Realisierung der einzelnen, in den Anforderungen definierten, Komponenten. Dazu werden zuerst die Anforderungen in UML-Diagrammen (Unified Modeling Language) dargestellt. Diese Diagramme dienen der Spezifikation von einzelnen Software-Komponenten und erlauben einen ersten Einblick in die zu entwickelnde Software. Die UML-Spezifikation bezeichnet viele verschiedene Diagrammtypen, die jedoch nicht alle in dieser Arbeit Verwendung finden werden. Hier wird nur eine grobe Darstellung vorgenommen, die jedoch ausreicht, um ein Softwareprojekt auf Prototypbasis zu entwickeln. Dabei wird zuerst das Anwendungsfalldiagramm erstellt. Aus dem Anwendungsfalldiagramm werden die benötigten Komponenten abgeleitet und ein Komponentendiagramm erstellt. Aus diesem Komponentendiagramm wird dann wiederum das Klassendiagramm entwickelt. Nachdem die Planung abgeschlossen ist, kann ein erster UI-Entwurf entstehen. In diesen Entwürfen wird Anordnung und Aussehen des UIs sichtbar. Danach folgt die praktische Implementierung der Komponenten. Hier werden die in den Klassendiagrammen vorgestellten Klassen und Beziehungen realisiert und wichtige Codebestandteile kurz vorgestellt. Im letzten Abschnitt jeder Komponente wird die UI-Implementierung erläutert.

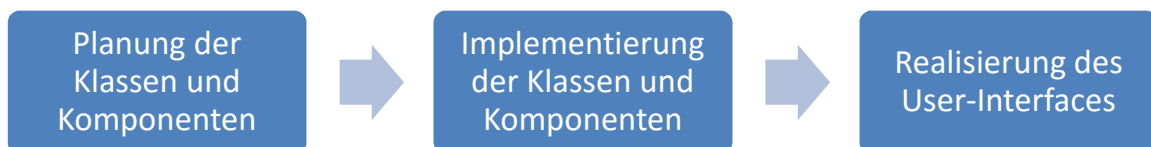


Abbildung 12: Verlauf der Entwicklung

Abbildung 12 zeigt den Verlauf der Entwicklung für jede einzelne Komponente mit den einzelnen Schritten. Die Entwicklung der Komponenten beginnt dabei bei den Basis- und Hintergrund-Komponenten. Hier werden essenzielle Komponenten erläutert, welche für die spätere Entwicklung relevant sind und damit zu Beginn erläutert werden. Darauf folgt die Entwicklung des Karteneditors und seiner Module. Abschließend werden UI-Systeme geplant und implementiert, damit der Spieler mit dem Spiel interagieren kann. Abgerundet werden diese Kapitel durch

Zwischenkapitel, in denen auf die in den vorherigen Abschnitten entwickelten Komponenten eingegangen wird, und wie diese ihren Weg in das fertige Spiel finden.

4.2 Basis- und Hintergrund-Komponenten

4.2.1 Steuerung

Der erste Abschnitt der Planung und prototypischen Realisierung beschreibt das Input-Handling. Wie in Abschnitt 3.4.1 erläutert, wird für das Spiel eine einheitliche Steuerung angestrebt, so dass sich

- Konsolen mit dem Controller,
- PCs mit der Tastatur und
- Smartphones mit einem UI-Overlay steuern lassen.

Um die entsprechenden Anwendungsfälle abzudecken, wurde das in Abbildung 13 dargestellte Use-Case-Diagramm entworfen.

Anwendungsdiagramm

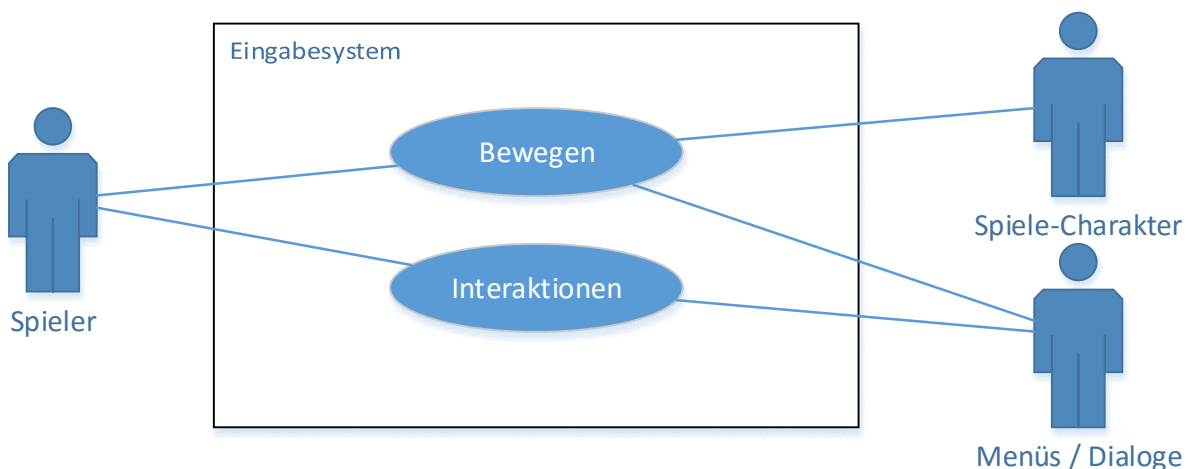


Abbildung 13: Anwendungsfälle für das Eingabesystem

Der Spieler soll mit den Bewegungstasten beziehungsweise UI-Richtungstasten den Spielcharakter in die jeweilige Richtung bewegen. Nachfolgend werden alle Eingabemöglichkeiten als „Tasten“ bezeichnet. Sollte im Spiel ein UI-System geöffnet sein, so wird entsprechend in diesem UI navigiert – es wird etwa der nächste Menüpunkt ausgewählt. Während der Spieler in diesem UI navigiert, steht der Spielcharakter still.

Damit alle Plattformen mit ihren unterschiedlichen Eingabemöglichkeiten dasselbe Resultat im Spiel hervorrufen, muss ein Eingabesystem geschaffen werden, welches die unterschiedlichen Eingaben in einheitliche Signale wandelt. Diese werden dann vom Spiel verarbeitet. Wie bereits beschrieben, sollen zur Steuerung die Tasten von Controller und Tastatur genutzt werden, beim Smartphone soll ein UI eingeblendet werden. Für die Eingabesteuerung wurde die in Abbildung 14 dargestellte Klasse *UINavigatorHandler* entworfen.

Klassendiagramm



Abbildung 14: Klassendiagramm für das Eingabesystem

Das boolesche Attribut *checkKeyPress* signalisiert dem System, dass es auf Eingaben mittels physischer Tasten prüfen soll. Die dafür benötigte Prüfung, ob es sich bei der ausführenden Plattform um eine Konsole beziehungsweise einen PC oder ein Smartphone handelt, übernimmt die *Start()*-Methode. Sollte es sich um ein Smartphone handeln, so wird die Methode *showGUI()* aufgerufen, welche das UI zur Steuerung einblendet. Sollte dies nicht der Fall sein, so wird in der *Update()*-Methode auf das Drücken von Tasten geprüft. Die restlichen Methoden *pressed...()* leiten schließlich ein einheitliches Signal an das Spiel weiter.

Implementierung

Um zu prüfen, welche Plattform die Applikation ausführt, bietet die UnityEngine die Eigenschaft *Application.Plattform* an [16]. Die zu Grunde liegende Aufzählungs-Variable *RuntimePlattform* kann diverse Zustände annehmen [17], darunter auch die benötigten Zustände für Konsolen, PC und Smartphones. Die jeweiligen Systeme

können präzise unterschieden werden, damit Anpassungen vorgenommen werden können. Für Smartphones beispielsweise unterscheidet Unity zwischen den Betriebssystemen Android, Windows Phone und iOS. Wenn es sich bei dem ausführenden Betriebssystem um eines der genannten Systeme handelt, so wird das Steuerungs-UI für Smartphones eingeblendet, welches in Abbildung 15 zu sehen ist. Dieses UI besteht aus Buttons, um die jeweilige Richtung oder Interaktionsmöglichkeit zu bestätigen. Es wird nicht per Code erzeugt, sondern per Prefab.

Prefabs sind im Vorfeld erstellte Spieleobjekte, welche als Einheit gespeichert werden und dann bei Bedarf im Code erzeugt und vervielfältigt werden können.

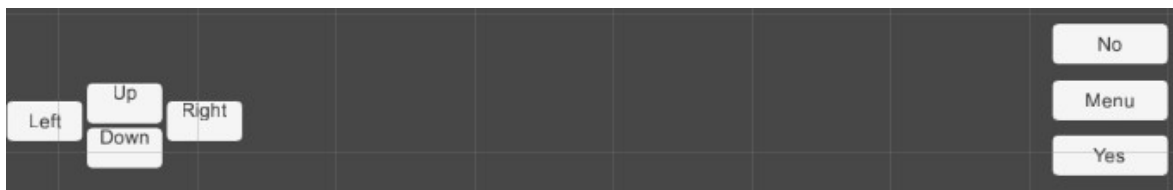


Abbildung 15: Steuerungs-UI

Die Erzeugung des UI-Prefabs ist in Listing 1 zu sehen:

```

192     Instantiate(Resources.Load("Prefabs/ControlCanvas"));
193
194     //UI-Events werden hinzugefügt
195     GameObject.Find("UpButton").GetComponent<Button>().onClick.
196     AddListener(() => { pressedUp(); });

```

1

2

Listing 1: Erzeugen des Steuerungs-Prefabs

Dabei wird jedoch nicht nur das Steuerungs-UI erzeugt, sondern es werden auch gleich Listener angehängt (1). Diese Listener prüfen in diesem Fall, ob der Benutzer auf den jeweiligen Button drückt. Als Parameter wird eine Methode angegeben, welche nach dem Auslösen des Listeners aufgerufen wird (2). Für den Nach-Oben-Button ist das also die *pressedUp()*-Methode. In dieser Methode wird dann der Spielecharakter bewegt.

4.2.2 Navigationssystem

Das Navigationssystem kümmert sich nicht, wie der Name vermuten lässt, um die Navigation des Spielers oder um eine Wegfindung, sondern um die Navigation zwischen UI-Elementen eines UI-Systems. Ein solches UI-System ist beispielsweise das Inventar. Es besteht aus verschiedenen UI-Elementen wie Labels, Buttons, Radiobuttons oder Bildern. Die UnityEngine enthält ein eigenes Navigationssystem,

mit welchem sich verschiedene UI-Elemente verknüpfen lassen. Zu sehen ist eine solche Navigation in Abbildung 16.

Diese Navigation erlaubt das Auswählen von *Selectable*-Elementen, welche beim Drücken bestimmter Tasten selektiert werden.

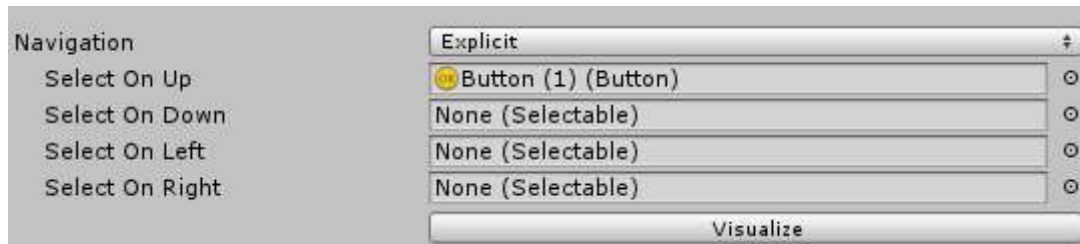


Abbildung 16: Navigationssystem in Unity

Ist etwa aktuell der Button „Spielen“ selektiert und der Spieler betätigt auf der Tastatur die Pfeiltaste nach oben, so wird ein anderer Button selektiert. Für Grundaufgaben genügt dieses System, allerdings ist es für dieses Projekt ungenügend. So können die Tasten, die zum Navigieren genutzt werden, nicht individualisiert werden. Für PCs oder Konsolen als Zielplattform hätte dies keine gravierenden Einschnitte, außer dass für die Navigation eben nur die von UnityEngine festgelegten Tasten genutzt werden können. Nicht genutzt werden kann das System hingegen von Smartphones, denn dort existieren die genutzten Tasten nicht, beziehungsweise die Steuerung erfolgt über das für das Smartphone erzeugte Steuerungs-UI. Eine Möglichkeit wäre, das Drücken der von der UnityEngine unterstützten Tasten zu simulieren. Da dies allerdings eine separate Implementation oder Nutzung von bereits vorhandenen Simulationsprojekten voraussetzen würde, wurde diese Möglichkeit verworfen. Ein weiterer Negativpunkt des hauseigenen Navigationssystems ist die Unterstützung von selektierbaren Elementen. So werden neben Buttons auch andere Elemente wie Label unterstützt, jedoch wird dem Spieler bei einem Label nicht grafisch signalisiert, dass dieses Element aktuell selektiert ist. Ein Button signalisiert dies etwa, indem er die Farbe ändert. Bei einem Label wird jedoch keine Veränderung vorgenommen. Um diese negativen Punkte und weitere zu beheben, wurde ein eigenes Navigationssystem entworfen.

Ein eigenes Anwendungsfalldiagramm ist an dieser Stelle nicht nötig, da es sich um eine Implementierungsproblematik handelt und sich an dem zu Grunde liegenden

Modell aus Abschnitt 4.2.1 nichts ändert. Es wurde ein System mit den folgenden, in Abbildung 17 zu sehenden, Komponenten entworfen.

Komponentendiagramm

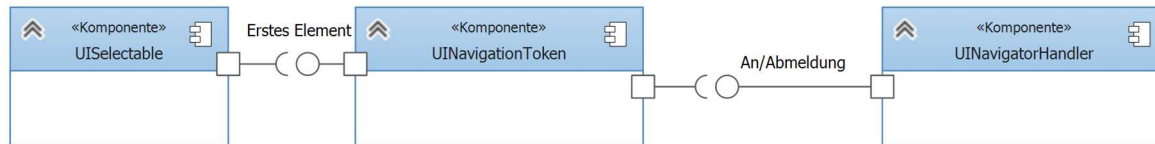


Abbildung 17: Komponenten des neuen Navigationssystems

Die Komponente *UISelectable* kann an UI-Elemente angehängt werden. In dieser Komponente kann bestimmt werden, welches *UISelectable* bei bestimmten Aktionen aktiviert werden soll.

Die Hauptkomponente, erweitert dabei den in Abschnitt 4.2.1 eingeführten Navigator-Handler. Der Handler soll zusätzlich prüfen, wo der aktuelle Fokus liegt: Ist ein UI-System wie das Inventar oder ein Dialog geöffnet, so wird in den jeweiligen Elementen des Systems navigiert. Sollte keines geöffnet sein, so wird wie gehabt der Spielcharakter bewegt. Ein UI-System muss sich, damit es navigierbar wird, beim *NavigatorHandler* anmelden. Diese Anmeldung übernimmt die *UINavigationToken*-Komponente. Sie wird an das UI-System angehängt und impliziert, dass es mindestens ein UI-Element gibt, an welchem die *UISelectable*-Komponente angehängt ist. In dieser Komponente kann, wie beim Pendant der UnityEngine, bestimmt werden, welches Element als nächstes ausgewählt werden soll. Die *UINavigationToken*-Komponente übergibt nun das gespeicherte Element zusammen mit der Anmeldung an den *NavigatorHandler*. Dieser nimmt die Anmeldung entgegen, legt den Fokus auf das UI-System, welches sich angemeldet hat, und navigiert infolge dessen dort. Diese Navigation kann nur unterbrochen werden, indem der UI-Navigation-Token sich beim Handler wieder abmeldet. Dieser stellt daraufhin seinen Ursprungszustand wieder her und die Richtungstasten werden zum Steuern des Spielcharakters genutzt. Um die Kommunikation der einzelnen Komponenten besser zu verdeutlichen, wurde das in Abbildung 18 gezeigte Sequenzdiagramm entworfen.

Sequenzdiagramm

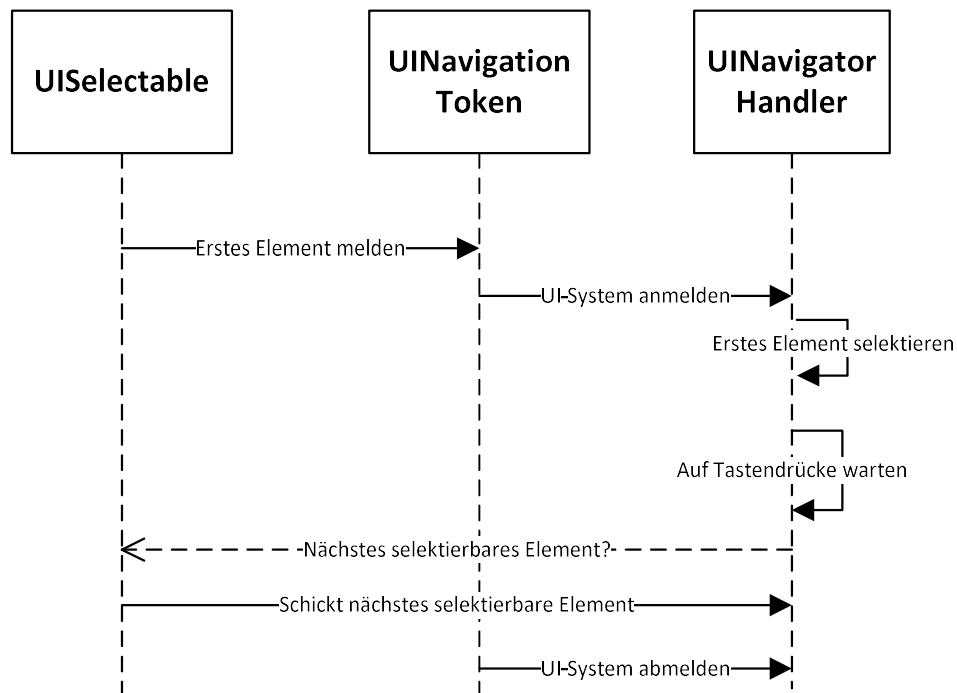


Abbildung 18: Kommunikation zwischen den Navigations-Komponenten

Nachdem ein UI-Element selektiert ist, kann solange zwischen den anderen verschiedenen selektierbaren Elementen gewechselt werden, bis sich das UI-System wieder vom *NavigatorHandler* abmeldet.

Implementierung

Mithilfe der vorhandenen Entwurfsdiagramme kann nun das Klassendiagramm entwickelt werden, welches in Abbildung 19 zu sehen ist. Die gezeigten Klassen werden anschließend mit Logik versehen und implementiert. Diese Implementierung wird anhand der einzelnen Komponenten beschrieben.

Klassendiagramm

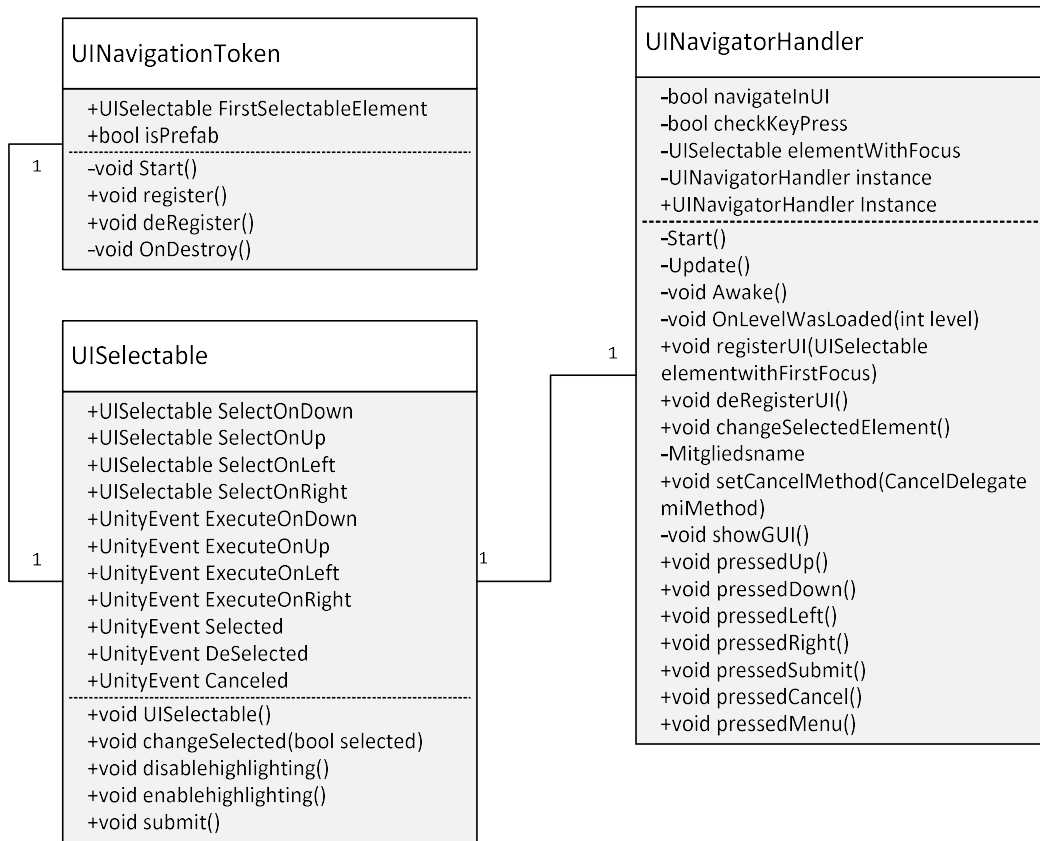


Abbildung 19: Klassendiagramm des Navigationssystems

UISelectable

Die zu Grunde liegende Klasse *UISelectable* implementiert vier Elemente, welche selektiert werden, wenn die entsprechenden Tasten gedrückt wurden. Die Klasse bietet einen leeren Konstruktor sowie Methoden, um das UI-Element, an dem die Komponente angehängt ist, hervorzuheben. Mit der Methode *changeSelected()* wird diese Hervorhebung ein- und ausgeschaltet. Die Methode *submit()* wird schließlich ausgelöst, wenn der Benutzer bei dem aktuell selektierten Element die Akzeptieren-Taste drückt. Die Logik hinter dieser Methode wird an anderer Stelle hinterlegt. Die UnityEngine erlaubt das Anhängen von Event Triggern zu einzelnen Spielobjekten, wie in Abbildung 20 zu sehen.

In diesen Triggern kann definiert werden, was bei bestimmten Aktionen, wie einem Klick auf das Element, passieren soll.

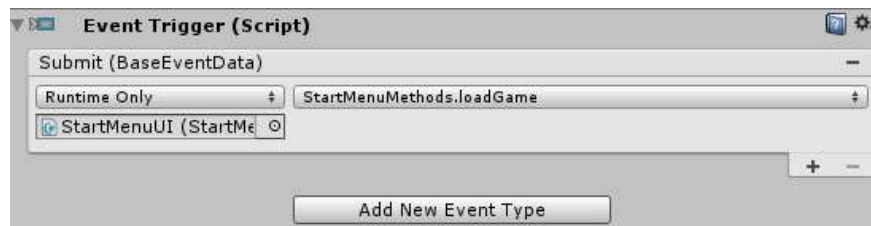


Abbildung 20: Event Trigger mit einer übergebenen Methode

Es werden Methoden hinterlegt, welche aktionsabhängig genutzt werden. Beispielsweise löst die *submit()*-Methode das Submit-Event auf dem UI-Element aus, wie in Listing 2 zu dargestellt ist.

```

128 public void submit()
129 {
130     ExecuteEvents.Execute(gameObject, new BaseEventData(EventSystem.current),
131         ExecuteEvents.submitHandler);
132 }

```

Listing 2: Manuelles Auslösen eines Events

Mittels der *ExecuteEvents*-Klasse können alle Events angesprochen werden, die auch der Event Trigger verwaltet. Allerdings muss auch hier wieder händisch die Funktionalität erweitert werden. So bietet die Unity-Engine natürlich keine Trigger für Events, welche nicht alltäglich sind. Zum besseren Verwalten von *UISelectable*-Elementen wurden daher noch *UnityEvents* hinzugefügt. Ein Attribut vom Typ *UnityEvent* nimmt eine Liste von Methoden auf, die aufgerufen werden, sobald die *Invoke()*-Methode ausgeführt wird. Als Beispiel sei an dieser Stelle das *Canceled*-Feld einer *UISelectable*-Komponente genannt. Nicht alle UI-Systeme benötigen die selbe Funktionalität, wenn auf eine Useraktion reagiert wird. Bei einem Dialog soll das Abbrechen beispielsweise wie ein „Nein“ gehandhabt werden. Im Inventar dagegen soll ein Schritt im UI-System zurücknavigiert werden. Dies ist zu bewerkstelligen, indem das jeweilige System eigene Methoden in die *Canceled*-Liste übergibt. In diesen Methoden kann dann die Logik ausgeführt werden, die für das UI-System nötig ist. Zusätzlich wurden für jede Aktionstaste ein *UnityEvent*-Attribut implementiert. Dem Entwickler wird dadurch ermöglicht, individuell auf jeden Tastendruck des Spielers zu reagieren.

UINavigationToken

Diese Klasse kümmert sich um die An- und Abmeldung beim *NavigatorHandler*. Dazu wird die Komponente an ein Spielobjekt angehängt. Das Attribut *isPrefab* zeigt, ob es sich bei dem UI-System, welches verarbeitet werden soll, um ein per Code erzeugtes oder ein fertiges System handelt. Ein Beispiel für ein fertiges System ist das Hauptmenü. Es wird in einer separaten Szene erstellt und in dieser dauerhaft angezeigt. Prefabs werden in verschiedenen Szenen per Code erzeugt und dann verarbeitet. Das bedeutet, fertige UI-Systeme, wie das Hauptmenü, melden sich direkt beim *NavigatorHandler* an, während sich die über Prefabs erzeugte UI-Systeme erst anmelden, wenn sie von Hand aktiviert werden. Das *UISelectable*, welches als erstes zu markierendes Element übergeben werden muss, wird zusammen mit der Anmeldung an den *UINavigationHandler* weitergegeben. Für die An- und Abmeldung stehen die beiden Methoden *register()* und *deRegister()* bereit. Sollte das zugrundeliegende Spielobjekt, an dem die Token-Komponente angehängt ist, zerstört werden, so meldet sich das UI-System automatisch ab.

UINavigationController

Der *UINavigationController* ist bereits aus Abschnitt 4.2.1 bekannt, allerdings wurde die Klasse um weitere Eigenschaften und Methoden erweitert. Die zusätzliche boolesche Variable *navigateInUI* zeigt an, ob aktuell in einem UI-System navigiert wird oder ob die Steuerung direkt an den Spielcharakter weitergeleitet wird. Wenn in einem UI-System navigiert wird, dann muss immer ein Element selektiert sein. Dieses Element wird in *elementWithFokus* gespeichert (siehe Klassendiagramm in Abbildung 19).

Da der *UINavigationController* nur einmal über alle Szenen aktiv sein soll, wurde ein Singleton-Pattern eingeführt. Dieses Pattern „sorgt dafür, dass es von einer Klasse nur eine einzige Instanz gibt und diese global zugänglich ist“ [18].

```

37 public static UINavigationController Instance
38 {
39     get { return instance; }
40 }

```

Listing 3: Realisierung der globalen Erreichbarkeit

Konkret realisiert wird dies dadurch, dass es in der Klasse eine Eigenschaft gibt, welche die aktuelle Instanz der Klasse zurückgibt, wie in Listing 3 sichtbar ist.

Nachdem die Klasse global erreichbar ist, muss sichergestellt werden, dass immer nur eine Instanz der Klasse existiert. Dazu wurde die *Awake()*-Methode implementiert, welche immer aufgerufen wird, wenn eine Komponente aktiv wird.

In dieser Methode wird geprüft, ob bereits eine Instanz existiert – sollte dies der Fall sein, dann wird die überschüssige Instanz zerstört. Sollte keine Instanz bestehen, so wird eine neue erzeugt.

```

46     void Awake()
47     {
48         if (instance != null && instance != this)
49         {
50             Destroy(gameObject);
51             return;
52         }
53         else
54         {
55             instance = this;
56         }
57         DontDestroyOnLoad(transform.root.gameObject);
58     }

```

Listing 4: Instanziierung des Navigator-Handlers

Listing 4 zeigt in Codezeile 57, dass das Objekt, an welchem der Handler angefügt ist, nicht zerstört wird, wenn im Spiel eine neue Szene geladen wird. Dadurch ist die Klasse nun global verfügbar und es ist sichergestellt, dass immer nur eine Instanz der Klasse existiert.

Damit sich UI-Systeme, wie das Inventar, beim Navigator-Handler anmelden können, stellt der Handler die Methode *registerUI()* bereit, welche in Listing 5 zu sehen ist.

```

76     public void registerUI(UINavigationToken token)
77     {
78         navigateInUI = true;
79         tokenWithFocus = token;
80         tokenWithFocus.firstSelectableElement.
81             changeSelected(true, tokenWithFocus.highlightMode);
82     }

```

Listing 5: Registrierung beim Navigator-Handler

Die Methode erwartet ein *UINavigationToken*. Wie in Zeile 78 zu sehen ist, wird der *UINavigationController* angewiesen, dass ab jetzt in einem UI-System navigiert werden soll. Anschließend wird das aktuelle Element gesetzt. Der übergebene Token bringt dieses Element mit. Das Element wird danach selektiert und damit hervorgehoben.

Sollte jetzt ein Tastendruck erfolgen, so wird beim aktuell selektierten Element geprüft, ob es für die jeweilige Richtung ein Ziel gespeichert hat.

Listing 6 zeigt diese Funktionsweise beispielhaft für einen Tastendruck auf die nach-oben-Taste.

```
231 |         if(tokenWithFocus.elementWithFocus.SelectOnUp != null)
232 |             changeSelectedElement(tokenWithFocus.elementWithFocus.SelectOnUp);
```

Listing 6: Beispielabfrage bei einem Tastendruck auf die nach-oben-Taste.

Sollte ein Ziel gespeichert sein, so wird das Element, welches selektiert ist, gewechselt. Wenn kein Ziel gespeichert ist, passiert nichts.

Das Navigationssystem zum Wechseln zwischen UI-Elementen ist jetzt vollständig. Es ist möglich, UI-Systeme beim *UINavigationControllerHandler* an- und abzumelden, in diesen UI-Systemen wird mithilfe der *UISelectable*s navigiert. Wenn sich das UI-System schließlich abmeldet, wird die Steuerung wieder direkt auf den Spielcharakter übertragen.

4.2.3 Mozo

Wie in Abschnitt 3.1, den Anforderungen, beschrieben, ist das Fangen und Trainieren von Mozos eine der Hauptsäulen im Spiel. Jedes Mozo besitzt unterschiedliche Eigenschaften, Attribute und Fähigkeiten, die dessen Stärken und Schwächen definieren. Dabei verfügt ein Mozo über folgende Basiseigenschaften:

Basiseigenschaft	Beschreibung
Name	Der vom Spiel vergebene Name für das Mozo. Dieser kann vom Spieler geändert werden.
Stufe	Die Stufe repräsentiert die Erfahrung des Mozo im Kampf. Je höher die Stufe, desto erfahrener ist es. Mit jedem Stufenanstieg erhält das Mozo einen Bonus auf bestimmte Attribute.
Erfahrungspunkte	Jedes Mozo erhält bei bestimmten Ereignissen Erfahrungspunkte – etwa bei einem Sieg gegen andere Mozos. Wenn diese Erfahrungspunkte einen bestimmten Wert erreichen, dann steigt die Stufe des Mozo.
Entwicklung	Von jedem Mozo existieren mehrere Entwicklungsformen, die bei bestimmten Stufen erreicht werden können. Beim Erreichen einer neuen Entwicklungsform steigen die Attribute enorm an und neue Fähigkeiten können erlernt werden.
Element	Ein Mozo gehört einem bestimmten Element an. Dieses Element kann gegenüber anderen Elementen stärker oder schwächer sein – beispielsweise ist Feuer im Nachteil gegen Wasser.
Fähigkeiten	Eine Fähigkeit besteht aus unterschiedlichen Eigenschaften wie den benötigten Energiepunkten und Schaden.

Tabelle 3: Basiseigenschaften eines Mozo

Neben den Basiseigenschaften verfügt ein Mozo über Hauptattribute, welche die Aktionen im Kampf beeinflussen:

Attribut	Beschreibung
Lebenspunkte	Gibt an, wie viel Schaden das Mozo erleiden kann, bevor es kampfunfähig wird.
Energiepunkte	Wird für das Einsetzen von Fähigkeiten benötigt.
Stärke	Bestimmt die physische Kraft der Mozos.
Beherrschung	Bestimmt die Effektivität elementarer Fähigkeiten.

Abwehr	Beeinflusst wie viel physischen Schaden das Mozo erleidet.
Elementarwiderstand	Beeinflusst die Resistenz gegen andere Elemente.
Schnelligkeit	Je schneller ein Mozo ist, desto mehr Aktionen kann es im Kampfgeschehen ausführen.

Tabelle 4: Hauptattribute eines Monsters

Diese Hauptattribute können vom Spieler entweder durch Training oder bestimmte Items verändert werden. Ein solches Item kann etwa ein Heiltrank sein, welcher die Lebenspunkte eines Mozo wiederherstellt.

Mozos haben zudem Kampfeigenschaften, die unter anderem durch die Hauptattribute beeinflusst werden:

Kampfeigenschaft	Beschreibung
Regeneration Lebenspunkte	Die Lebenspunkte, welche ein Mozo nach jeder Runde regeneriert.
Regeneration Energiepunkte	Die Energiepunkte, welche ein Mozo nach jeder Runde regeneriert.
Treffergenauigkeit	Je höher die Treffergenauigkeit ist, desto niedriger ist die Chance, dass das gegnerische Mozo ausweichen kann.
Ausweichchance	Je höher die Ausweichchance ist, desto eher weicht das Mozo einem gegnerischen Angriff aus.
Schadensreduktion physisch	Erlittene physische Schäden werden um diesen Wert verringert.
Schadensreduktion elementar	Erlittene Elementarschäden werden um diesen Wert verringert.
Schadensmultiplikator physisch	Physische Angriffe werden um diesen Wert erhöht.
Schadensmultiplikator elementar	Elementarangriffe werden um diesen Wert erhöht.

Tabelle 5: Kampfeigenschaften eines Monsters

Die Kampfeigenschaften beeinflussen bestimmte Aktionen im Kampf, etwa wie treffsicher ein Mozo ist oder wie viel Schaden es austeilen kann.

Mit Hilfe dieser Eigenschaften kann nun mit der Planung der Implementierung begonnen werden.

Auf ein Anwendungsfall- und Komponentendiagramm wird an dieser Stelle verzichtet, da es sich bei den Mozos um eine Basis-Komponente des Spiels handelt. Andere Komponenten wie das Kampfsystem oder die Teamansicht bauen auf diese Basis-Komponente auf.

Klassendiagramm

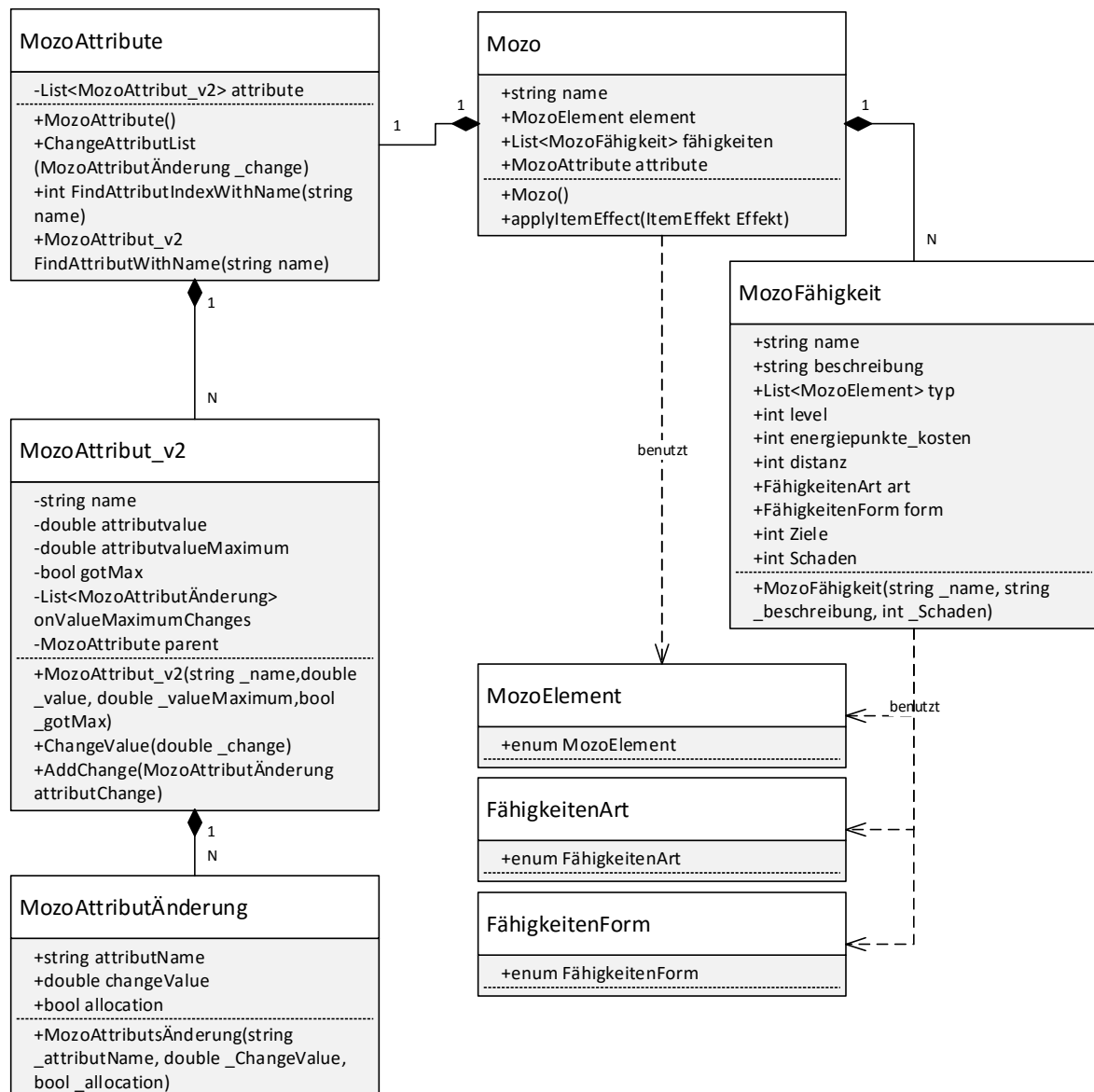


Abbildung 21: Klassendiagramm des Mozo

Abbildung 21 zeigt das Klassendiagramm für ein Mozo. Die Klassen werden rund um die Basisklasse *Mozo* aufgebaut, welche das zentrale Element darstellt. Jedes Mozo hat n verschiedene Fähigkeiten. Dies wird durch eine Liste von Objekten vom Typ

MozoFähigkeit repräsentiert. Die Fähigkeiten werden durch diverse Attribute definiert, etwa durch einen Namen, ein Level und die Distanz. Des Weiteren werden Aufzählungen für das Element, die Art und die Form einer Fähigkeit genutzt. Um die Attribute eines Mozo zu definieren, wurden die Klassen *MozoAttribute*, *MozoAttribut* und *MozoAttributÄnderung* genutzt. Die Klasse *MozoAttribute* hält eine Liste von Attributen und Methoden bereit, um bestimmte Attribute zu finden. Das eigentliche Attribut besteht aus dem Wert, dem Maximum des Wertes und einer booleschen Variable die anzeigt, ob der Attributwert ein Maximum besitzt. Diese Variable wird relevant, wenn der Wert eines Attributes erhöht wird: Dann wird geprüft, ob das Attribut einen Maximalwert besitzt. Sollte diese Bedingung erfüllt sein, dann ruft das Attribut eine Methode der Elternklasse auf, welche die Werte anderer Attribute ändert. In der Praxis ist dies zum Beispiel bei einem Item relevant, welches die Erfahrungspunkte des Mozo erhöht. Wenn die Erfahrungspunkte etwa den Wert von 900/1000 haben und der Spieler ein Item zur Steigerung nutzt, welches 200 Punkte hinzufügt, dann muss zum einen das Level-Attribut des Mozo um eins erhöht und das Erfahrungspunkte-Attribut zurückgesetzt werden. So können die Attribute leicht geändert werden und jedes Attribut kann eine eigene Liste von Effekten bereithalten, welche ausgeführt werden, wenn das Attribut sein Maximum erreicht.

Damit ist die Planung der Mozos abgeschlossen. Für die Mozos ist kein eigenes GUI erforderlich, so dass die Beschreibung der Implementierung der Klassen folgt.

Implementierung

Fähigkeit

Die Implementation der Klasse *MozoFähigkeit* ist trivial. Die Klasse setzt sich aus Strings für den Namen, der Beschreibung und anderen elementaren Datentypen zusammen. Form, Art und Element einer Fähigkeit sind als Aufzählung definiert. Die Klasse erwartet im Konstruktor einen Namen, eine Beschreibung und den Schaden.

Mozo

Die Eigenschaften des Mozo (siehe Einleitung zur Komponente Mozo) wurden, wie im Klassendiagramm auf Seite 40 beschrieben, implementiert.

In Listing 7 ist beispielhaft zu sehen, wie der Name des Mozo definiert³ wurde.

```

8  |  /// <summary>
9  |  /// DE: Der Name des Mozo.
10 |  /// EN: The name of the mozo.
11 |  /// </summary>
    |  3 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
12 |  public string Name
13 |  {
14 |      get
15 |      {
16 |          return name;
17 |      }
18 |
19 |      set
20 |      {
21 |          name = value;
22 |      }
23 |  }

```

Listing 7: Definition des Namens eines Mozo

Des Weiteren wurde eine Methode integriert, um die einen Gegenstandseffekt auf ein Mozo anwenden zu können.

MozoAttribute/MozoAttribut/Attributänderung

Die Klasse *MozoAttribute* beinhaltet eine Liste von Attributen. Listing 8 zeigt die Methode zum Ändern eines Attributes:

```

135 |  public void ChangeValue(double change)
136 |  {
137 |      //Wenn der Maximalwert erreicht wird,
138 |      //wird die Methode MaxErreicht vom Parent aufgerufen.
139 |      if (attributvalue + change > attributvalueMaximum && gotMax.Equals(true))
140 |      {
141 |          attributvalue = attributvalueMaximum;
142 |
143 |          foreach(MozoAttributÄnderung AE in onValueMaximumChanges)
144 |          {
145 |              parent.ChangeAttributeList(AE);
146 |          }
147 |      }
148 |      else
149 |      {
150 |          attributvalue += change;
151 |      }
152 |  }

```

Listing 8: Ändern eines Attributes

³In C# können Attribute gekapselt werden, indem sie wie exemplarisch gezeigt über eine öffentlich verfügbare Schnittstelle verfügen, welche dann wiederum auf das klasseninterne Attribut zugreift. Diese Vorgehensweise hat den Vorteil, dass der Zugriff auf das Attribut geregelt und kontrolliert stattfindet [28]. Ein Entwickler kann beispielsweise in der öffentlichen Schnittstelle prüfen, ob die übergebenen Daten einem validen Format vorliegen.

Dabei wird in Zeile 139 geprüft, ob das Maximum des Attributes erreicht ist und ob das Attribut überhaupt einen Maximalwert besitzt. Für diesen Fall wird, wie ab Zeile 143 zu sehen ist, die gespeicherte Liste von Änderungen an die Elternklasse übergeben. Dort werden die Änderungen ausgeführt. Damit ist die Mozo-Komponente fertig gestellt. Sie kann jetzt von anderen Komponenten wie dem Inventar genutzt werden.

4.2.4 Lokalisierung

Damit das Spiel nicht nur auf verschiedenen Systemen, sondern auch mit verschiedenen Sprachen gespielt werden kann, muss eine Lokalisierung der Textinhalte geschehen. Auch an dieser Stelle wird wieder auf ein Anwendungsfall- und Komponentendiagramm verzichtet, da diese trivial wären.

Klassendiagramm

Damit möglichst flexibel mit zu übersetzenden Texten umgegangen werden kann, wurden vier Klassen entworfen (siehe nachfolgende Abbildung 22). Diese Klassen werden jeweils verschiedenen Spielobjekten angehängt. Hierbei fungiert der *Localizer* als Zentrum. Er nimmt Übersetzungsanfragen sowie Sprachänderungen entgegen. Die Komponenten *LocaleButtonHandler*, *LocalizeText* und *LanguageFile* initiieren diese Anfragen. Nachfolgend wird jede Komponente kurz erläutert. Auch wird auf die Kommunikation mit dem *Localizer* eingegangen.

Abbildung 22 zeigt das Klassendiagramm des Lokalisierungssystems und die Beziehungen der Komponenten zueinander.

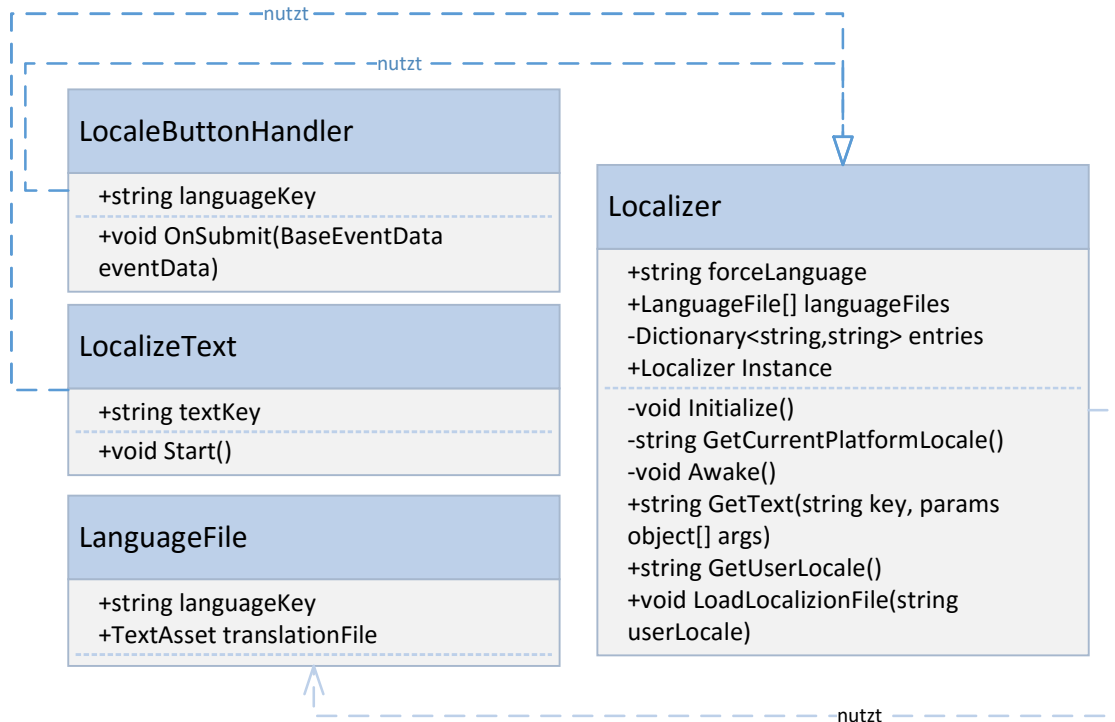


Abbildung 22: Klassendiagramm der Lokalisierung

LocaleButtonHandler

Die Klasse *LocaleButtonHandler* wird an Objekte angehängt, welche die aktuelle Sprache wechseln sollen. Dazu könnten etwa Buttons im Haupt- oder Optionsmenü dienen. Um zwischen den Sprachen zu wechseln, erhält jede Instanz der Klasse einen *languageKey*, welcher signalisiert, zu welcher Sprache gewechselt werden soll, sobald der Button betätigt wurde. Die *OnSubmit()*-Methode reagiert dann auf das Drücken des Buttons und sendet eine Lokalisierungsanfrage an den *Localizer*.

LocalizeText

Damit einzelne Textbausteine übersetzt werden können, müssen sie über eine *LocalizeText*-Komponente verfügen. In der zugrundeliegenden Klasse wird ein *textKey* definiert, mit welchem sich die Komponente „ausweist“. Dieser Key ist also ein eindeutiger Name für eine Text-Komponente, um den zu übersetzenden Text in der Sprachdatei zu finden. Zusätzlich wird die *Start()*-Methode implementiert, so dass der Text beim Start der Komponente automatisch übersetzt wird.

Dazu wird, wie in Listing 9 zu sehen ist, eine Übersetzungsanfrage an den *Localizer* gesendet.

```
15 public void Start () {  
16     Text label = GetComponentInChildren<Text>();  
17     if(label != null)  
18     {  
19         label.text = Localizer.Instance.GetText(textKey);  
20     }  
21 }
```

Listing 9: Übersetzungsanfrage einer Text-Komponente

Für die Übersetzungsanfrage wird in Zeile 16 zunächst die Text-Komponente geladen, welche den entsprechenden Text bereithält. Anschließend wird, nach einer obligatorischen Prüfung, ob die Text-Komponente vorhanden ist, die Anfrage gesendet. Dazu wird die *GetText()*-Methode der global verfügbaren Instanz der *Localizer*-Klasse aufgerufen und der *textKey* übergeben. Der Rückgabewert ist dabei der übersetzte Text.

LanguageFile

Diese Klasse repräsentiert eine Sprachdatei. Jede Sprachdatei besteht aus einem *languageKey*, welcher für die Sprache der Übersetzung steht und den eigentlichen Übersetzungsdaten.

Localizer

Die Klasse *Localizer* spielt die wichtigste Rolle, denn dort werden die Sprachdateien geladen und die eingehenden Übersetzungsanfragen einzelner Objekte beantwortet. Um diese Klasse über alle Szenen verfügbar zu machen, wurde wie beim UI-Handler aus Abschnitt 4.2.2 auf das Singleton-Pattern gesetzt. Beim Initialisieren der Klasse sucht der *Localizer* zunächst nach der Sprache, mit welcher das Spiel gestartet werden soll. Dazu wird geprüft, welche Systemsprache das ausführende System aufweist. Für diesen Zweck stellt die UnityEngine die öffentlich abrufbare Eigenschaft *Application.systemLanguage* bereit. Abhängig von der aktuellen Systemsprache wird dann die Sprache des Spiels gesetzt. Nachdem die Sprache gesetzt ist, kann die dazugehörige Sprachdatei geladen werden. Zum Laden einer Sprachdatei wurde die Methode *LoadLocalizionFile()* implementiert.

Eine solche Datei sieht beispielsweise folgendermaßen aus:

```
Button.Play=SpieleN
Button.Quit=Beenden
GameMenu.Description=Beschreibung
```

Abbildung 23: Inhalt der Übersetzungsdatei

Wie zu erkennen ist, besteht jede Komponente einer Sprachdatei aus zwei Komponenten. Zum einen der Schlüssel beziehungsweise der eindeutige Name, zum anderen die Übersetzung für den jeweiligen Schlüssel. Diese beiden Werte werden durch ein Erkennungszeichen getrennt. Beim Laden der Datei werden diese Werte dann getrennt und in einem Dictionary gespeichert. Damit ist die Klasse bereit, Anfragen zur Übersetzung eines Textes entgegenzunehmen. Um solche Anfragen bearbeiten zu können, wurde die Methode `getText()` implementiert. Als Parameter erwartet die Methode unter anderem den `textKey` der Text-Komponente. Dieser Key wird in den gespeicherten Sprachdateien nachgeschlagen und anschließend die angeforderte Übersetzung zurückgegeben.

```
54 public string GetText(string key, params object[] args) {
55     if (entries.Count == 0) {
56         Initialize();
57     }
58     if (entries.ContainsKey(key)) {
59         return string.Format(entries[key], args);
60     } else {
61         return string.Format("Key: '{0}' not found!", key);
62     }
63 }
```

Listing 10: Bearbeiten einer Übersetzungsanfrage

Wie in Listing 10 zu sehen ist, wird dazu in den Zeilen 55 bis 57 zuerst geprüft, ob das Nachschlagewerk, in welchem alle Übersetzungen gespeichert werden, leer ist. Wenn dies der Fall ist, wird die Klasse initialisiert und das Nachschlagewerk gefüllt. Anschließend wird nach dem Schlüssel, den die Text-Komponente übergeben hat, gesucht. Ist dieser gefunden, wird der zugehörige Eintrag als Ergebnis zurückgegeben. Sollte die Sprache während der Laufzeit noch einmal gewechselt werden, so wird die Klasse einfach mit dem neu gesetzten Sprachkürzel neu initialisiert. Mit der Initialisierung wird die neue Sprachdatei geladen und es können erneut Übersetzungsanfragen entgegengenommen werden.

4.2.5 Zwischenstand des Spiels – das Hauptmenü

Nachdem einige der grundlegenden Komponenten fertig gestellt sind, kann nun ein kleiner Zwischenstand des Spiels aufgezeigt werden, welcher die bisherigen Komponenten und ihre Benutzung zeigt. Es soll ein Menü entwickelt werden, in welchem der Spieler das Hauptspiel startet, beendet oder die Sprache wechseln kann. Dazu werden die Navigations- und die Lokalisier-Komponente genutzt. Abbildung 24 zeigt eine erste Version der GUI-Implementierung des Hauptmenüs. Zu sehen sind die Buttons zum Steuern der Sprache (1) und zum Starten beziehungsweise Beenden (2) des Spiels. Des Weiteren wurde eine kleine Beschreibung hinzugefügt (3).

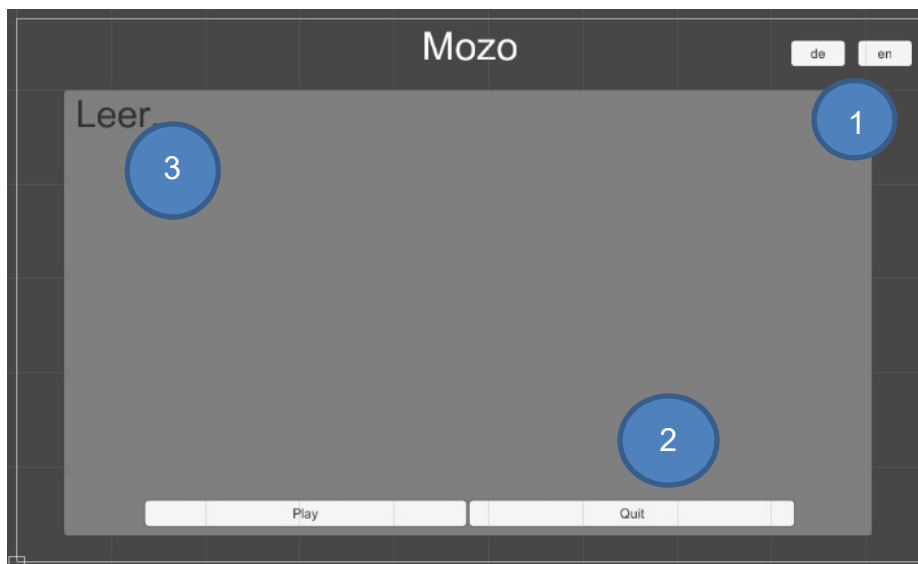


Abbildung 24: Hauptmenü des Spiels

Zum Navigieren zwischen den interaktionsfähigen GUI-Elementen wird das in Abschnitt 4.2.2 implementierte Navigationssystem genutzt. Dabei wurden die GUI-Elemente folgendermaßen verknüpft:

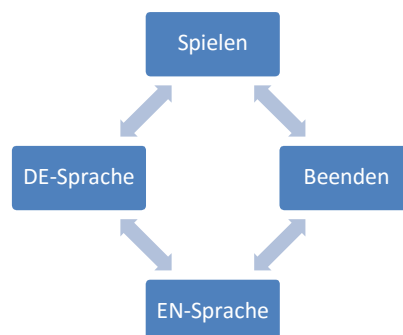


Abbildung 25: Verknüpfungen der Elemente

Zur Verknüpfung wurden den Elementen *UISelectable*-Komponenten angehängt. Dort ist definiert, zu welchem Element nach einem bestimmten Tastendruck gewechselt wird. Im Hauptmenü wird nur mit der nach-oben-Taste beziehungsweise der ambivalenten nach-unten-Taste navigiert.

Natürlich müssen diese Elemente auch mit Logik ausgestattet werden. Die Sprach-Buttons sollen etwa zur jeweiligen Sprache wechseln und der „Spielen“-Button soll das Hauptspiel starten. Damit die in Abschnitt 4.2.4 implementierte Lokalisation genutzt werden kann, werden Text-Elementen, die übersetzt werden sollen, *LocalizeText*-Komponenten angehängt. Diese zeichnen sich wie bereits beschrieben durch einen *TextKey* aus, mit dem die jeweilige Übersetzung gesucht wird.

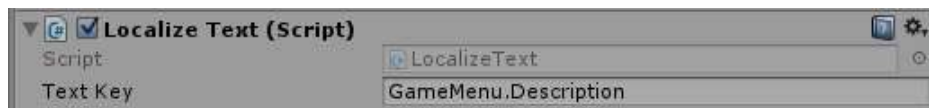


Abbildung 26: An einen Text angehängte *LocalizeText*-Komponente

Abbildung 26 zeigt den *TextKey* für die Beschreibung: „*GameMenu.Description*“. In der Übersetzungsdatei steht der zugehörige Text für diesen Schlüssel. Nach dem Start der Lokalisierungs-Komponente wird dieser Text automatisch in der aktuell ausgewählten Sprache geladen. Um diese Sprache zu wechseln, werden die Sprach-Buttons ebenfalls mit Logik versehen. An diese Buttons wird die in Abschnitt 4.2.4 implementierte Komponente *LocaleButtonHandler* angehängt, welche für die Änderung der aktuellen Sprache zuständig ist. Für jeden Button wird ein Sprachkürzel definiert, welches für eine Sprache steht. Beispielsweise erhält der „DE“-Button das „de“-Sprachkürzel. Wenn der Nutzer jetzt auf diesen Button drückt, wird die zugehörige Sprachdatei geladen und alle Texte werden fortan in diese Sprache übersetzt.

Damit der Spieler jetzt das Hauptspiel starten kann, muss der „Spielen“-Button mit Logik versehen werden. Das Hauptspiel wird in einer neuen Szene gespeichert. In der UnityEngine reicht, wie in Listing 11 zu sehen ist, ein Befehl, um die neue Szene zu laden, und das Hauptspiel zu starten. Dieser Befehl wurde in einer neuen Funktion verpackt und über die bekannten Event Trigger an den „Spielen“-Button angehängt. Dadurch wird der Spieler zur Haupt- beziehungsweise Spieleszene weitergeleitet.

Diese Weiterleitung erfolgt über den *SceneManager*, welcher für das Laden von Szenen die Methode *LoadScene()* bereitstellt.

```
18 public void loadGame()  
19 {  
20     SceneManager.LoadScene("GameScene");  
21 }
```

Listing 11: Laden einer neuen Szene

Damit ist das vorläufige Hauptmenü hinsichtlich der Logik komplett implementiert. Abbildung 27 zeigt mit beispielhaften Texten, wie das ausgeführte Hauptmenü aussehen könnte. Zu sehen ist auch das UI-Overlay, welches in Abschnitt 4.2.1 implementiert wurde und zur Steuerung auf mobilen Geräten und PCs genutzt wird.

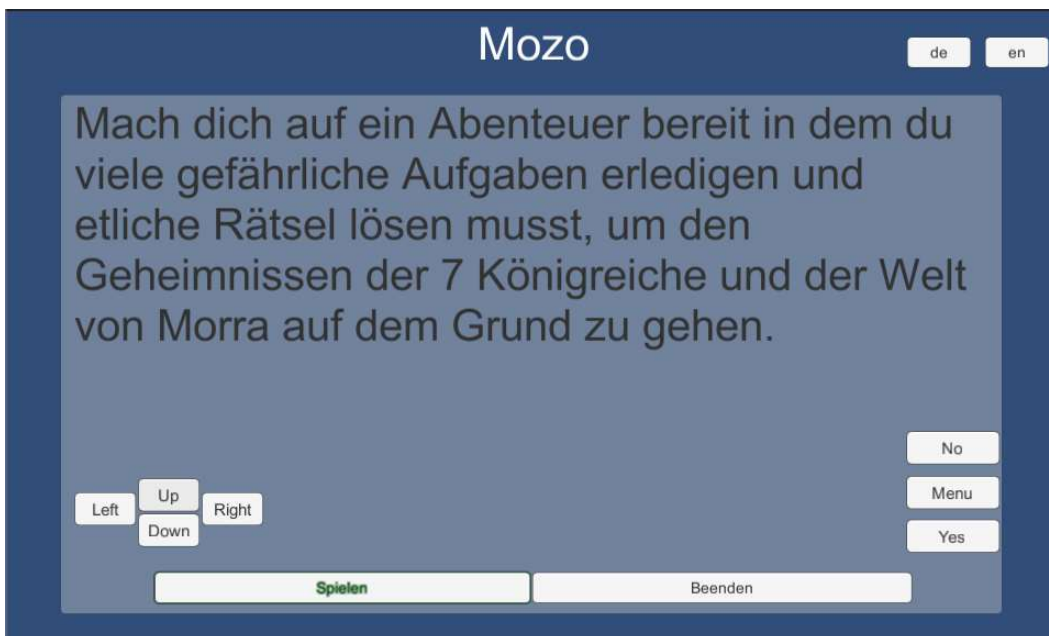


Abbildung 27: Ausführen des Hauptmenüs

Es wurden alle bisher vorgestellten Komponenten beispielhaft implementiert und ihre Nutzung beschrieben.

Die nachfolgenden Abschnitte beschreiben die Entwicklung von Komponenten des Hauptspiels. Die Komponenten des Hauptspiels basieren zumeist auf den bereits vorgestellten Komponenten. Beispielsweise benötigen alle UI-Systeme des Hauptspiels die Navigations- und Lokalisierungs-Komponente.

4.3 Spielumgebung

4.3.1 Karteneditor

4.3.1.1 Darstellungssystem

In diesem Abschnitt wird die Realisierung des Karteneditors behandelt. Hierfür muss jedoch zunächst ein grundlegendes Prinzip zum Organisieren und Darstellen der Texturen, insbesondere für das Gelände, festgelegt werden. Dafür gibt es das sogenannte **Tile-System**. Beim Tile-System wird das Spielfeld (**Tilemap**) in viele kleine Kacheln (**Tiles**) unterteilt, die sich in einem kachelartigen Raster nahtlos miteinander kombinieren und austauschen lassen. Dieses Prinzip der Komprimierung wurde ursprünglich bei Grafikdateien (**Tilesets**) angewandt, um die Auslastung des Arbeitsspeichers zu minimieren und die Zugriffszeiten zu optimieren. Auch heute noch ist diese Art der Grafikverwaltung ein übliches Verfahren und wird im Rahmen dieses Projektes ebenfalls eingesetzt. Ein Beispiel, wie so ein Tileset aussehen kann, ist in Abbildung 28 zu sehen.

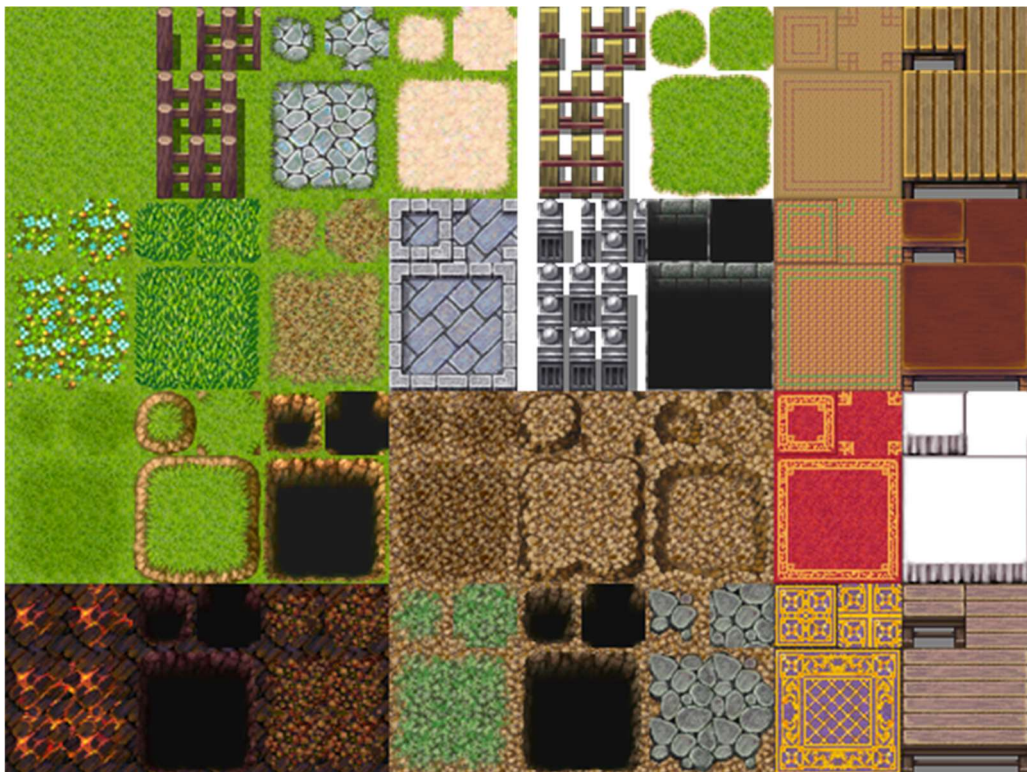


Abbildung 28: Tileset mit verschiedenen Tiles für ein Top-Down-Gelände [19]

Das Tile-System wird zwar häufig für Spiele aus der Top-Down-Ansicht, wie in der oberen Abbildung zu sehen, verwendet, jedoch lässt sich dieses Prinzip auch leicht für den isometrischen Gebrauch adaptieren. Statt eines Kachelmusters wird die Fläche eines Tiles als Raute betrachtet. Dadurch ist es nicht mehr möglich, dass sich die Tiles nahtlos nebeneinander positionieren lassen. Stattdessen müssen für einen passenden Übergang die Tiles nun so auf dem Spielfeld positioniert werden, dass diese halbzeilig versetzt sind. Das Raster für solch eine isometrische Aufteilung wird in Abbildung 29 gezeigt.

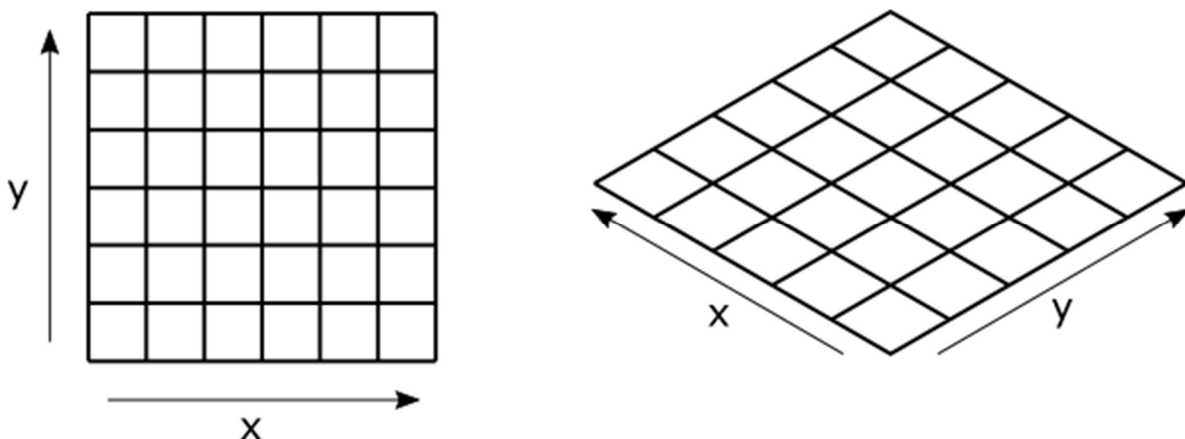


Abbildung 29: Darstellung eines Kachelrasters und eines isometrischen Rasters

4.3.1.2 Erweiterung des Unity-Editors

Da die Funktionen des Karteneditors direkt in Unity implementiert und während des Entwicklungsmodus nutzbar sein sollen, ist es notwendig, kurz auf die Erweiterbarkeit der Unity Engine einzugehen. In Unity gibt es eine Vielzahl von Möglichkeiten, die Entwicklungsumgebung zu erweitern und umzugestalten. An dieser Stelle wird jedoch nur auf die Prozeduren eingegangen, welche für die vorliegende Arbeit relevant sind. Diese Prozeduren werden anhand der Implementierung des **isometrischen Rasters** näher beschrieben.

Gizmos und Handles

Die wichtigsten Features, um etwas aktiv in der Szene einblenden zu können, sind sogenannte **Gizmos** und **Handles**.

Gizmos sind Hilfselemente mit denen man Objekte, die nicht tatsächlich existieren, in der Szene darstellen lassen kann. Ein einfaches Beispiel hierfür wäre das Kamera-Symbol, welches standardmäßig eingeblendet wird, um anzuzeigen, an welcher Stelle das Kameraobjekt positioniert ist (siehe Abbildung 30).



Abbildung 30: Das Kamera-Symbol (Gizmo) in Unity

Neben Icons und Texturen lassen sich mitunter auch noch Linien, Würfel sowie Kugeln als Gizmos darstellen. Diese Anzeigeelemente dienen jedoch an erster Stelle nur als Information für den Entwickler. Handles sind dagegen richtige Kontrollelemente, mit denen direkt interagiert werden kann. Bei der Verwendung von Transform-Tools, die in der Toolbar von Unity zu finden sind, werden beispielsweise Handles eingeblendet, um ein Objekt in der Szene positionieren (1), rotieren (2) oder skalieren (3) zu können (siehe Abbildung 31).

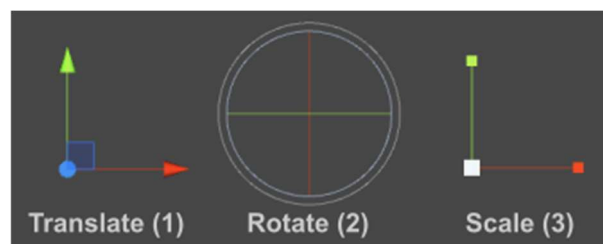


Abbildung 31: Darstellung der Transform-Tools in der Szene (Handles)

Da das Raster nicht unmittelbar in der Szene angepasst werden muss, ist ein Gizmo für die Implementierung ausreichend.

Für das Erstellen des Gizmos zunächst ein Script erzeugt werden. Jedes Script, welches in Unity erstellt wird, leitet sich grundsätzlich von *MonoBehaviour* ab und ist die Basis, um Scripte als GameObject-Komponente verwenden zu können. Wird einem GameObject solch ein Script zugewiesen, werden alle öffentlichen Variablen (*public*) im Inspektor angezeigt und können, selbst während der Laufzeit, dynamisch verändert werden. Neben der Größe des Rasters und deren Zellen, soll auch die Position des Rasters angepasst werden können (siehe Abbildung 32).

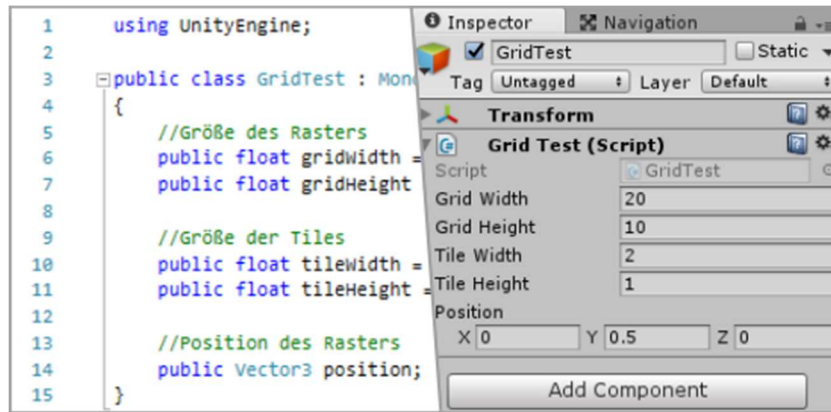


Abbildung 32: Deklarieren der Variablen und die dazugehörige Abbildung im Inspector

Als nächstes wird die *OnDrawGizmos()*-Funktion implementiert. Diese Methode ist eine von vielen Methoden der *MonoBehaviour*-Klasse. Sie wird, ähnlich wie *Update()*, mehrmals aufgerufen und aktualisiert Gizmos in der Szene automatisch, falls diese sich verändert. Anders als bei *Update()* wird die *OnDrawGizmos()*-Methode jedoch auch vor der Laufzeit des Spiels ausgeführt.

In dieser Methode wird als nächstes, über die *Gizmos*-Klasse, auf die *DrawLine()*-Methode zugegriffen (siehe Abbildung 33).

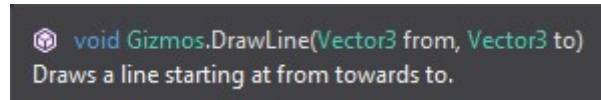


Abbildung 33: Statische DrawLine()-Methode der Gizmos-Klasse

Die *DrawLine()*-Methode zeichnet eine Linie in der Szene. Die übergebenen Parameter geben dabei die Start- und Endposition an.

Für das Raster werden, je nach Größe, mehrere Linien benötigt, die in der richtigen Länge und an der richtigen Stelle positioniert werden müssen. Dafür ist zunächst ein Rechteck nötig, an dessen Rahmen sich die Start- und Endpunkte ausrichten sollen. So ein Rahmen eines Rechtecks kann mit der Methode *DrawWireCube()* erzeugt werden (siehe Abbildung 34).

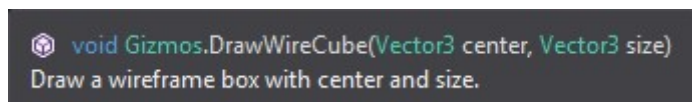


Abbildung 34: Statische DrawWireCube()-Methode der Gizmos-Klasse

Hierbei wird zum einem die Position des Mittelpunkts sowie die Gesamtgröße des Rechtecks übergeben.

Im ersten Schritt müssen nun die Startpunkte der Linien grundlegend so positioniert werden, dass diese beginnend von der oberen linken bis zur rechten unteren Seite des Rechtecks liegen. Simultan werden die jeweiligen Endpunkte entlang der linken oberen bis zur unteren rechten Seite angeordnet (siehe Abbildung 35).

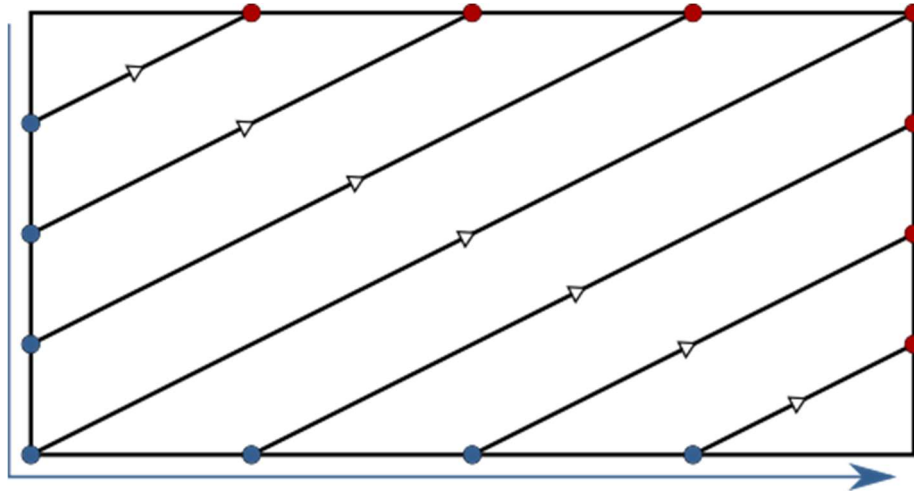


Abbildung 35: Ablauf für das Zeichnen der Rasterlinien (1. Schritt)

Die Abstände der Punkte an der linken und rechten Seite entsprechen der Tile-Höhe. Die Tile-Breite gibt den Abstand der oberen und unteren Seite an.

In Listing 12 wird die Implementierung für diesen Schritt dargestellt.

```

23     Gizmos.DrawWireCube(position, new Vector3(gridwidth, gridHeight, 1));
24
25     //Anzahl der Linien (pro Seite)
26     float lineCount = (gridwidth / tilewidth + gridHeight / tileHeight) - 1;
27
28     //Startpunkt (linke Seite)
29     Vector3 fromLeft = new Vector2(-gridwidth / 2, (gridHeight / 2) - tileHeight);
30
31     //Schleife zum Zeichnen der Linien von Links nach Rechts
32     for (float x = 0.0f; x < lineCount; x += 1.0f)
33     {
34         Gizmos.DrawLine(fromLeft + position, new Vector3(-fromLeft.y * 2, -fromLeft.x / 2) + position);
35
36         //Gehe vom Startpunkt solange runter, bis die untere Rastergrenze erreicht ist
37         //Die Schritte entsprechen jeweils der Tile-Höhe
38         if (fromLeft.y - tileHeight >= -gridHeight / 2)
39         {
40             fromLeft += new Vector3(0, -tileHeight);
41         }
42         //Gehe von der unteren Rastergrenze solange nach rechts bis die rechte Rastergrenze erreicht ist
43         //Die Schritte entsprechen jeweils der Tile-Breite
44         else
45         {
46             fromLeft += new Vector3(tilewidth, 0);
47         }
48     }

```

Listing 12: Zeichnen der Linien im ersten Schritt

Hierbei wird zuallererst das Rechteck gezeichnet und dabei die Position sowie Rastergröße als Vector3-Parameter übergeben (Zeile 23). Der z-Wert bleibt dabei standardmäßig auf 1, da die Tiefe des Rasters irrelevant ist.

Als nächstes werden für die Abbruchbedingung die Anzahl der benötigten Linien sowie der erste Startpunkt ermittelt. Für beide Berechnungen sind die Gesamtgröße des Rechtecks und auch die Größe der Tiles essenziell. Anschließend werden die entsprechenden Linien nacheinander in der Szene gezeichnet. Von der y-Koordinate des ermittelten Startpunktes wird dabei in jedem Durchlauf die Tile-Höhe subtrahiert, bis die untere linke Ecke erreicht ist. Von dort an bleiben die y-Koordinaten gleich und die Tile-Breite wird hinzuaddiert, bis alle Linien gezeichnet wurden.

Im nächsten Schritt wird grundsätzlich auf dieselbe Weise vorgegangen. Die jeweiligen Startpunkte sind lediglich horizontal gespiegelt und befinden sich nun an der rechten statt der linken Seite des Rechtecks. Die oberen und unteren Punkte bleiben gleich (siehe Abbildung 36).

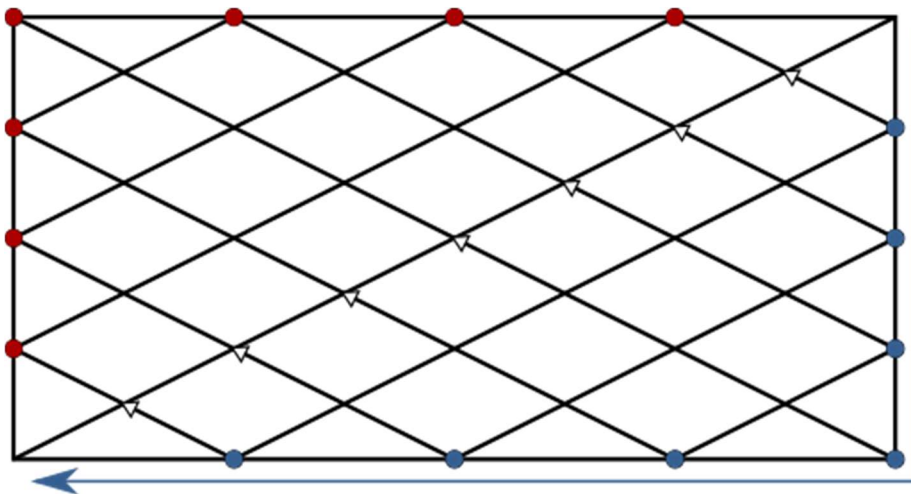


Abbildung 36: Ablauf für das Zeichnen der Rasterlinien (2. Schritt)

Für den Implementierungsschritt ändert sich demnach nicht viel. Lediglich die x-Koordinaten müssen negiert werden, der Rest bleibt gleich. Hierfür ist also kein weiteres Listing nötig.

Wenn beide Implementierungsschritte im Script ausgeführt sind, ist schließlich auch das isometrische Raster vollständig in der Szene abgebildet (siehe Abbildung 37).

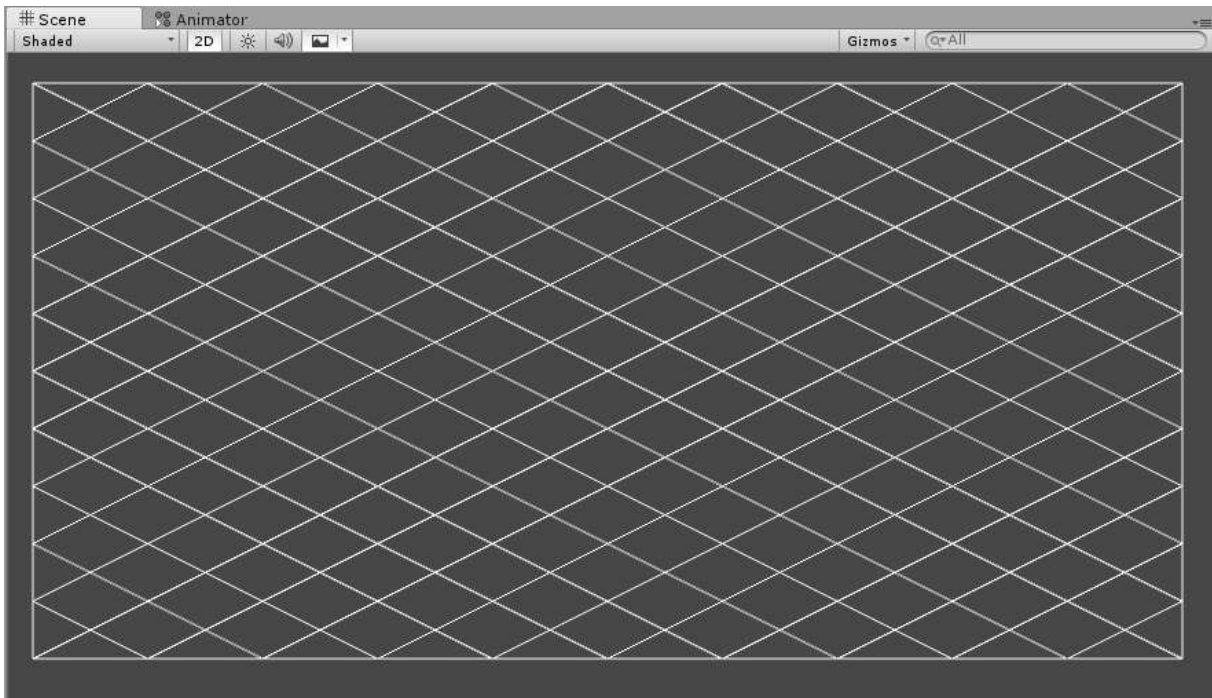


Abbildung 37: Isometrisches Raster in der Szene nach der Implementierung

Die Größe des Rasters und dessen Zellen (Tiles) können jederzeit im Inspector angepasst werden. Die Breite muss dabei jedoch stets dem doppelten Höhenwert entsprechen. Das bedeutet, dass das Raster nicht mehr korrekt dargestellt werden kann, wenn hier willkürliche Werte eingetragen werden. Demzufolge sind einige Einschränkungen für den Entwickler sinnvoll, damit er weiß, in welchem Rahmen sich die Werte befinden dürfen. Außerdem wäre eine Einstellmöglichkeit für die Rasterfarbe (besonders für die Transparenz) praktisch, da Gizmos immer vor allen Spielobjekten in der Szene gezeichnet werden und Spielobjekte sonst von den Gizmos überdeckt werden. An dieser Stelle wird nun auf das zweite große Feature zur Erweiterbarkeit eingegangen - den **Custom Inspector**.

Custom Inspector

Mit einem benutzerdefinierten Inspector (Custom Inspector) hat man in Unity die Möglichkeit, das Inspector-Fenster so zu erweitern und umzugestalten, dass, im Vergleich zur Standard-Ansicht, komplexe Einstellungsmöglichkeiten für Spielobjekte realisierbar sind.

Folgende Anpassungen lassen sich hierbei beispielsweise an dem Inspector vornehmen:

- Alle relevanten Daten- und Wertetypen sind normal als Felder darstellbar, darunter:
 - Positionsvektoren (Vector2 und Vector3),
 - Zahlenwerte (float, double und int),
 - Label- und Textfelder (String) und
 - Farbpaletten (Color).
- Zur besseren Übersicht und Organisation können Layout-Elemente erzeugt werden:
 - einfache Gruppencontainer (horizontal, vertikal, ein- und ausklappbar),
 - Container mit Scroll-Balken und
 - Separatoren.
- Für eine bessere Bedienung gibt es zudem einige Eingabelemente:
 - Schieberegler für Zahlenwerte,
 - Popups für verschieden Daten- und Wertetypen sowie
 - Buttons mit Funktionen.
- Layout-Eigenschaften und GUI-Styles lassen sich für die meisten Elemente nochmals anpassen:
 - Schriftart und -farbe,
 - Hintergrundfarbe und -textur,
 - Ausrichtung (absolut oder automatisch),
 - Rahmen.
- Events lassen sich durch das Selektieren eines GUI-Elements ausführen.
- Tools und Komponenten sind gezielt ausblendbar (z.B. die Transform-Komponente).

Damit ein Custom Inspector für ein Spielobjekt erzeugt werden kann, ist es zunächst notwendig, ein zusätzliches Editor-Script zu erzeugen. Dieses Script muss jedoch weder dem Spielobjekt zugewiesen, noch von MonoBehaviour abgeleitet werden, da es sich nur auf ein MonoBehaviour-Script beziehen soll. Dafür muss von der Klasse Editor geerbt und der Ausdruck `[CustomEditor (typeof (typ))]` vor der Klassendefinition geschrieben werden. Dieser Ausdruck sorgt dafür, dass alle Anpassungen, die hier festgelegt werden, für das Spielobjekt des übergebenen Script-Typs (Klasse) übernommen werden.

In diesem Fall würde also dieses Editor-Script auf unser vorher erstelltes Raster-Script verweisen (siehe Listing 13).

```

1  using UnityEditor;
2  using UnityEngine;
3
4  [CustomEditor(typeof(GridTest))]
5  public class GridTestEditor : Editor

```

Listing 13: Klassendefinition des Editor-Scripts

Auch in diesem Script stehen wieder einige relevante Methoden zur Verfügung, die die Editor-Klasse mit sich bringt:

- ***OnEnable()***

Diese Methode wird aufgerufen, wenn das Script aktiv wird. Dies geschieht beim Selektieren des entsprechenden Spielobjekts und wird meist dafür verwendet, um Variablen zu initialisieren und Einstellungen vorzunehmen.

- ***OnDisable()***

OnDisable() ist Gegenpart zur *OnEnabled()*-Methode. *OnDisable()* wird aufgerufen, wenn das Spielobjekt den Fokus verliert. Hier können Variablen und Einstellungen zurückgesetzt werden.

- ***OnInspectorGUI()***

Hiermit wird die Inspector-Darstellung neu gezeichnet. *OnInspectorGUI()* wird immer dann neu aufgerufen, wenn mit dem Inspector interagiert wird. Hier werden auch demzufolge fast alle, der bereits genannten, Anpassungen implementiert.

- ***OnSceneGUI()***

Mit dieser Methode lässt sich die Szene anpassen und erweitern. *OnSceneGUI()* wird immer dann ausgeführt, wenn sich etwas in der Szene ändert. Hier kann beispielsweise die aktuelle Mausposition abgefragt werden oder es lassen sich Handles darstellen.

- ***DrawDefaultInspector()***

Diese Methode wird üblicherweise innerhalb von *OnInspectorGUI()* aufgerufen und sorgt dafür, dass der Inspector normal dargestellt wird. Dies ist nützlich, wenn lediglich ein paar zusätzliche Elemente zum normalen Inspector-Ansicht hinzugefügt werden wollen.

Einige dieser Methoden werden nun genutzt, um den Inspector für das Rasterobjekt so anzupassen, dass der Entwickler das richtige Verhältnis beim Einstellen der Raster- und Zellengröße einhalten kann. Dafür werden für beide Größen Schieberegler (Slider) genutzt. Die Schieberegler geben bestimmte Werte zur Auswahl vor. Die Vorgabe gewährleistet eine schnelle und korrekte Veränderung des Rasters, ohne beliebige Werte berücksichtigen zu müssen. Außer dem Verändern von Raster- und Zellengröße sollen auch Position und Farbe des Rasters verändert werden können.

Bevor jedoch die *OnInspectorGUI()*-Methode mit den entsprechenden Anpassungen versehen wird, muss zunächst eine neue Variable vom Typ *GridTest* erzeugt und in der *OnEnable()*-Methode initialisiert werden. Hierfür gibt es ein Objekt namens *target*, welches das Ziel des Editor-Scripts enthält und mittels Typenumwandlung (*GridTest*) der Variable zugewiesen werden kann (siehe Listing 14).

```

7      GridTest grid;
8
9      public void OnEnable()
10     {
11         grid = (GridTest)target;
12     }

```

Listing 14: Initialisierung der Rastervariable

Nun kann jederzeit auf das Raster-Script zugegriffen werden und die entsprechenden Werte für die Schieberegler sind auslesbar. Im nächsten Schritt geht es schließlich darum, den Inspector mit Hilfe der *OnInspectorGUI()*-Methode zu zeichnen.

Für die einzelnen GUI-Elemente wird über die Klasse *EditorGUILayout* auf verschiedene Funktionen zugegriffen. Sie enthält auch die Schieberegler.

Im Listing 15 ist die Implementierung für den Schieberegler der Rastergröße und einiger Layout-Elemente zu sehen.

```

14 public override void OnInspectorGUI()
15 {
16     EditorGUILayout.Separator();
17     EditorGUILayout.BeginHorizontal();
18
19     EditorGUILayout.PrefixLabel("Grid Size");
20
21     float gridSizeSliderStep = grid.gridWidth / 40;
22     gridSizeSliderStep = EditorGUILayout.IntSlider((int)gridSizeSliderStep, 1, 12);
23
24     grid.gridWidth = gridSizeSliderStep * 40;
25     grid.gridHeight = grid.gridWidth / 2;
26
27     EditorGUILayout.EndHorizontal();
28     EditorGUILayout.LabelField("Width: " + grid.gridWidth + " Height: " + grid.gridHeight);

```

Listing 15: Erzeugung einiger Layout-Elemente für die Darstellung des Schiebereglers der Rastergröße

Hier werden zunächst zwei Layout-Elemente verwendet, um die Übersicht optisch etwas zu verbessern. Der Separator erzeugt zu Beginn eine Leerstelle. Als nächstes wird mit *BeginHorizontal()* eine horizontale Gruppe aufgebaut, damit das Label und der Schieberegler nebeneinander dargestellt werden können. Für den Schieberegler wird, noch bevor dieser erzeugt wird, eine Zwischenvariable für die Stufen angelegt. Eine Stufe soll hierbei einer Größe von 40x20 entsprechen. Aus diesem Grund wird die aktuelle Rasterbreite durch 40 dividiert. Die Funktion *IntSlider()* stellt schließlich den Schieberegler im Inspector dar und erwartet dafür die aktuelle Stufe sowie eine untere und eine obere Grenze als Parameter. Die minimal einstellbare Größe des Rasters wäre dementsprechend, mit den oben festgelegten Parametern, 40x20 (Stufe 1) und die maximale Größe 480x220 (Stufe 12).

Die Funktion *IntSlider()* gibt zusätzlich die aktuelle Stufe zurück und überschreibt die Zwischenvariable wieder, falls sich im Inspector etwas verändert.

Anschließend muss die aktuelle Rasterbreite nochmals ermittelt und dem Raster-Script zugewiesen werden. Dafür wird die Stufe wieder mit 40 multipliziert. Um die Rasterhöhe zu berechnen wird lediglich die Breite halbiert. Zuletzt wird die horizontale Gruppe geschlossen, damit in der nächsten Zeile der aktuelle Wert für die Rastergröße ausgegeben werden kann.

Für die Implementierung des Schiebereglers der Zellengröße wird im Grunde ähnlich verfahren. Jedoch wird bei der Berechnung der Stufen und Werte anders vorgegangen, da nicht alle Größen sinnvoll und problemlos darstellbar sind. Listing 16 zeigt dazu an dieser Stelle den entsprechenden Code im Script.

```

35     float tileSizeSliderStep = Mathf.Log(grid.tilewidth)/Mathf.Log(2) + 1;
36     tileSizeSliderStep = EditorGUILayout.IntSlider((int)tileSizeSliderStep, 1, 4);
37
38     grid.tilewidth = Mathf.Pow(2, tileSizeSliderStep - 1);
39     grid.tileHeight = grid.tilewidth / 2;

```

Listing 16: Implementierung des Schiebereglers der Zellengröße

In diesem Teil des Scripts werden die Werte der einzelnen Stufen so gewählt, dass sich diese aus einer Zweierpotenz ergeben ($2^0, 2^1, 2^2$ und 2^3). Der Exponent entspricht dabei der aktuellen Stufe. Dafür wird wieder zunächst eine Zwischenvariable angelegt (Zeile 35), die sich dementsprechend mit der Formel $\log(\text{Zellenbreite})/\log(2)$ ermitteln lässt. Für eine einheitliche Darstellung, wird hier noch zusätzlich 1 hinzuaddiert, damit beide Schieberegler auch bei Stufe 1 beginnen. In Zeile 36 wird dann der Schieberegler wie gewohnt erzeugt. Dabei wurde eine maximale Stufe von 4, was einer Größe von 8x4 entspricht, festgelegt. Anschließend wird aus der aktuellen Stufe mittels Zweierpotenz die Zellenbreite bestimmt und wieder die Hälfte des Wertes der Zellenhöhe zugewiesen.

Zuletzt müssen noch Felder für die Position und die Farbe platziert werden (siehe Listing 17).

```

46     grid.position = EditorGUILayout.Vector3Field("Position", grid.position);
47
48     EditorGUILayout.PrefixLabel("Farbe");
49
50     EditorGUI.indentLevel++;
51     grid.color = EditorGUILayout.ColorField(grid.color);
52     EditorGUI.indentLevel--;
53
54     EditorGUILayout.Separator();

```

Listing 17: Erstellung des Vector3- und des Color-Feldes

Hierbei muss auf den Positionsvektor sowie auf die *Color*-Variable, die vorher deklariert werden muss, vom Raster-Script zugegriffen werden. Die *indentLevel*-Variable sorgt dafür, dass die nachfolgenden Elemente mehr oder weniger im Inspector eingerückt sind. Der Standardwert entspricht hierbei 0.

Anschließend muss am Anfang der *OnDrawGizmos()*-Methode, im Raster-Script, der Befehl *Gizmos.color = color* ausgeführt werden, damit sich die Farbe des Rasters auch tatsächlich ändert.

Das Inspector-Fenster ist nun vollständig angepasst und der Entwickler kann damit leicht die wichtigsten Einstellungen am Raster vornehmen (siehe Abbildung 38).

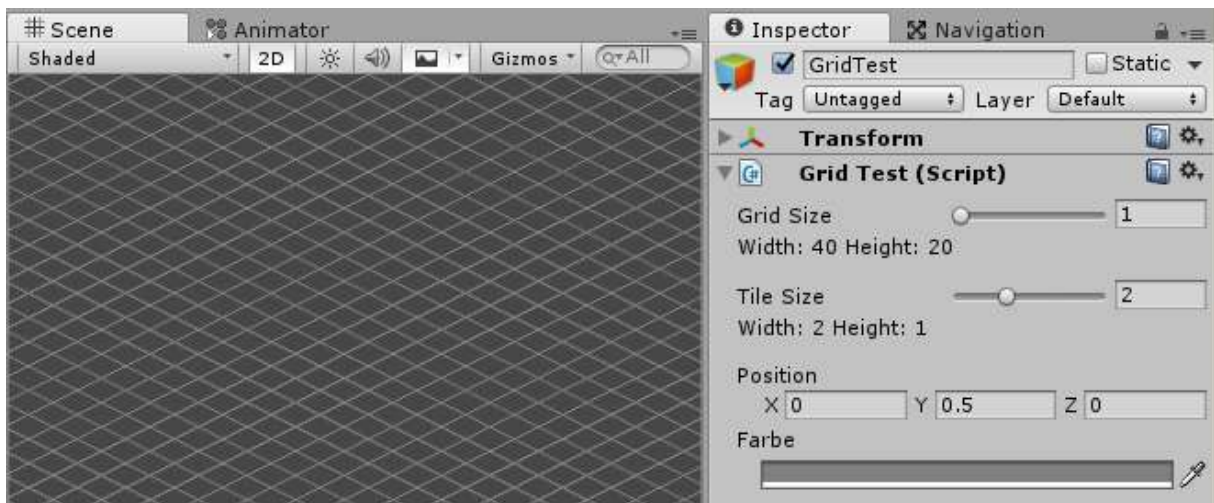


Abbildung 38: Vollständige Implementierung der eigenen Inspector-Ansicht für das Raster

Attribute in Unity

Die letzte Möglichkeit, auf die in diesem Abschnitt eingegangen wird, um den Inspector bzw. ein Script anzupassen, sind Attribute. Der Ausdruck **[CustomEditor (typeof (typ))]**, der bereits beim Erstellen des Custom Inspectors verwendet wurde, war so ein Attribut und sorgte dafür, dass das Editor-Script auf ein anderes Script verwiesen werden konnte.

Ein anderes, oft genutztes Attribut, ist **[ExecuteInEditMode]**. Es bewirkt, dass Scripte bzw. Methoden, die normalerweise nur während der Laufzeit aufrufbar sind (z.B. *Start()* und *Update()*-Methode), auch im Editormodus laufen.

Unity stellt noch viele weitere solcher Attribute bereit, um Variablen, Methoden oder, wie in diesen beiden Fällen, Klassen mit zusätzlichen Eigenschaften zu versehen.

Andere wichtige Attribute sind unter anderem:

- **[HideInInspector]**

Variablen die als *public* deklariert sind, können mit diesem Ausdruck im Inspector ausgeblendet werden.

- **[Range (von, bis)]**

Legt einen Wertebereich für eine *int*- oder *float*-Variable fest. Statt des üblichen Zahlenfeldes wird ein Schieberegler im Inspector dargestellt.

- **[RequireComponent (typeof (typ))]**

Wenn ein Script mit diesem Ausdruck einem GameObject zugewiesen wird, wird zusätzlich die angegebene Komponente hinzugefügt, falls diese noch nicht vorhanden ist.

- **[SerializeField]**

Private Variablen können hiermit serialisiert und im Inspector angezeigt werden.

- **[CanEditMultipleObjects]**

Damit wird dem Editor erlaubt, mehrere selektierte Spielobjekte im Inspector bearbeiten zu können.

Eine Liste mit allen bereitgestellten Attributen ist in der technischen Dokumentation von Unity zu finden. [20]

4.3.1.3 Erstellen und Bearbeiten eines Tile-Objekts

Ein Tile-Objekt soll, wie bereits erwähnt, ein platzierbares Objekt auf dem Spielfeld sein. Grundsätzlich soll dabei jedes Tile-Objekt eine **Position** im Spielfeld, eine **Sprite-Textur** für die zweidimensionale Darstellung und ein **Collider** für die Kollisionsabfragen besitzen. Da Unity ein simultanes Arbeiten von 2D und 3D fast⁴ vollständig unterstützt, werden, aus praktischen Gründen, hierfür 3D-Collider und dementsprechend auch ein dreidimensionales Grundgerüst (**Mesh**) genutzt. Die grafische Umsetzung bleibt dennoch für 2D ausgelegt.

In der Abbildung 39 wird gezeigt, wie so ein Tile-Objekt in Kombination, mit 3D-Collider und 2D-Textur in der Szene aussehen kann.

⁴ Alle physischen 3D-Komponenten von Unity funktionieren nur mit anderen 3D-Komponenten. Ein 3D-Collider registriert beispielsweise keine Kollision mit einem 2D-Collider.

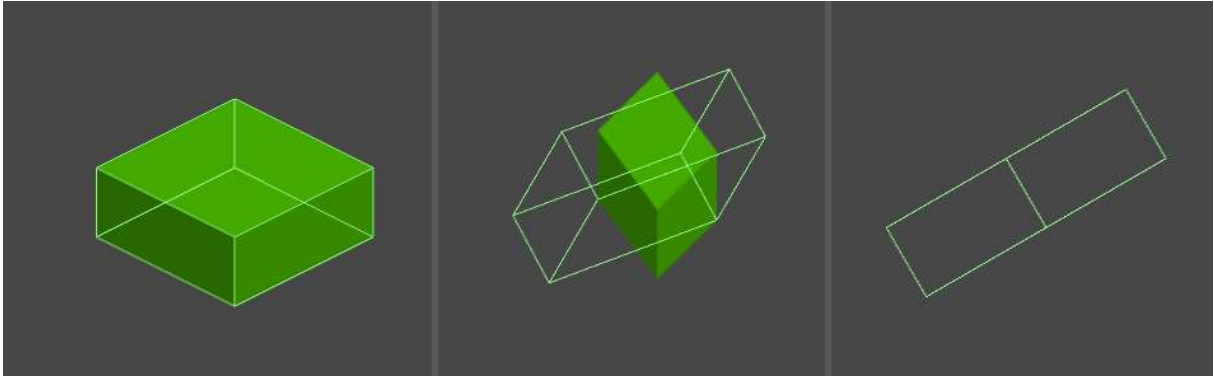


Abbildung 39: Beispiel eines Tile-Objekts in verschiedenen Perspektiven

Für die isometrische Perspektive wird wie in Abschnitt 3.2 beschrieben, ein dreidimensionales Koordinatensystem genutzt.

Da sich die Perspektive im 2D-Modus von Unity (orthogonale Projektion) jedoch nicht ändern lässt, orientiert sich die Platzierung der Objekte an einem normalen zweidimensionalen Koordinatensystem (Drauf- und Seitenansicht). Das bedeutet, die isometrische Position muss so angepasst werden, dass sie in dem vereinfachten Koordinatensystem korrekt platziert wird. Wie dabei genau vorgegangen wurde, wird bei der Implementierung der Transform-Funktion beschrieben.

Um nun die Tile-Objekte so schnell und komfortabel wie möglich zu bearbeiten, sollen alle relevanten Komponenten in einem Custom Inspector zugreif- und anpassbar gemacht werden, da Unity von Haus aus nur begrenzte Bearbeitungsmöglichkeiten bietet. Diese Aufgabe übernimmt der **Tile Designer**.

Im Nachfolgenden werden die Anwendungsfälle für die Bearbeitung eines Tile-Objekts dargestellt (siehe Abbildung 40).

a) Anwendungsfalldiagramm

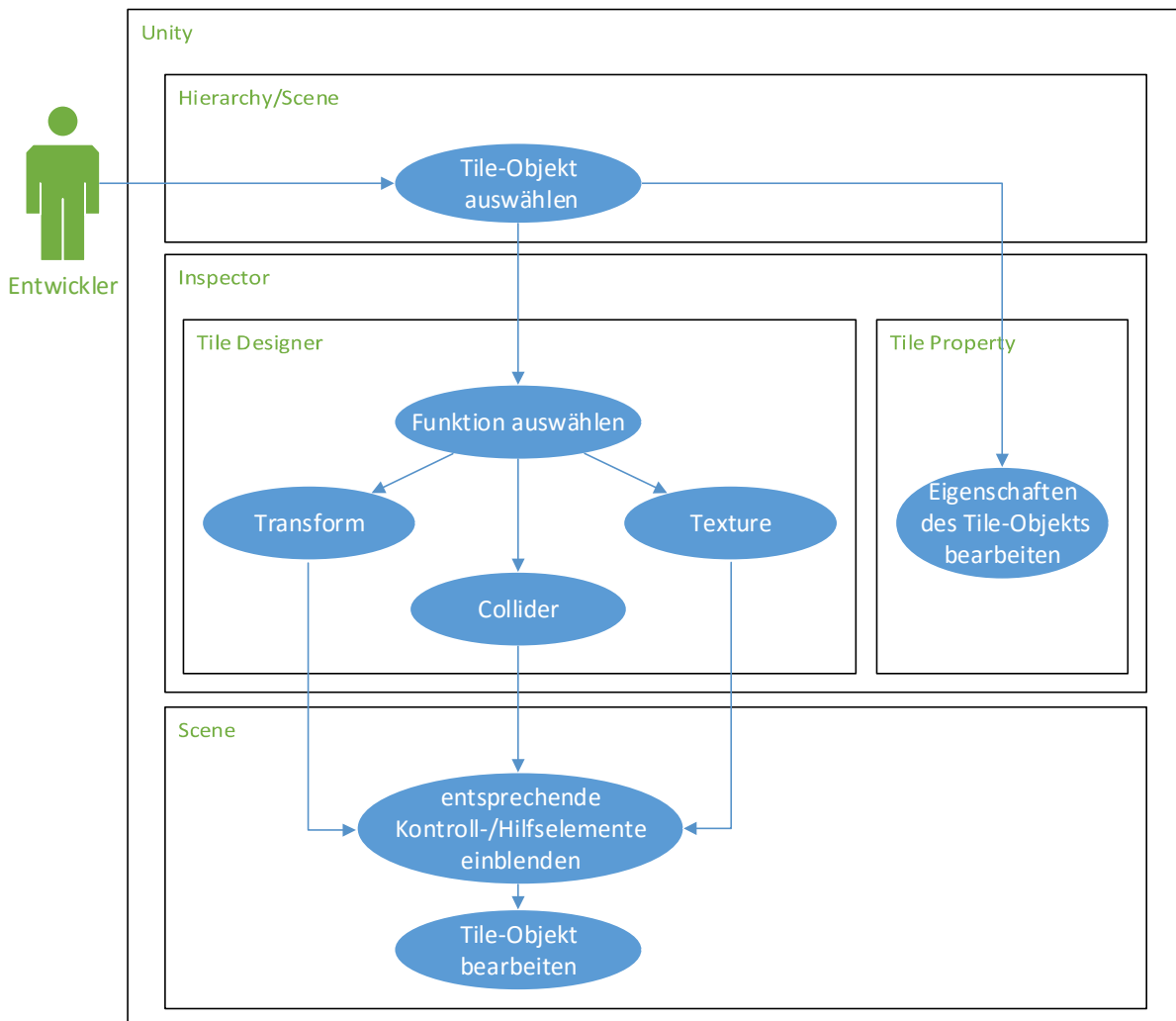


Abbildung 40: Anwendungsfalldiagramm für das Bearbeiten der Tile-Objekte

Dem Entwickler sollen hierbei, nachdem er ein beliebiges Tile-Objekt in der Hierarchie oder in der Szene ausgewählt hat, folgende Funktionen in dem Inspector zur Verfügung stehen:

- **Transform**

Mit der Transform-Funktion soll sich die Position des Tile-Objekts in der Szene verändern lassen. Dabei soll entschieden werden können, ob die Platzierung frei oder fest im Raster erfolgt. Das ermöglicht, je nach Wahl, eine schnellere oder präzisere Positionierung der Objekte in der Szene.

- **Collider**

Die Collider-Funktion soll die Möglichkeit bieten, dem Tile-Objekt einen bereits vorhandenen 3D-Mesh zuzuordnen oder einen neuen 3D-Mesh zu erstellen. In beiden Fällen soll sich anschließend die Form anpassen lassen.

- **Texture**

Mittels der Texture-Funktion soll dem Tile-Objekt eine Textur festgelegt und zusätzlich die Pivot-Koordinate (Mittelpunkt) verändert werden können.

Wird eine dieser Funktionalitäten vom Entwickler aufgerufen, sollen die entsprechenden Einstellungsoptionen im Inspector dargestellt werden. Zusätzlich sollen in der Level-Szene Kontroll- und Hilfselemente (Handles/Gizmos) eingeblendet werden, die die Bedienung vereinfachen - beispielsweise durch das Einblenden von Koordinatenpfeile bei der Transform-Funktion.

Neben den grundsätzlichen Eigenschaften (Tile Designer) sollen ebenso die spielinternen Eigenschaften (Tile Property) im Inspector verändert werden können. Es soll sich hier um ein separates MonoBehaviour-Script handeln. Deshalb kann es unabhängig von den anderen Funktionalitäten genutzt werden. Diese Komponente wird jedoch erst einmal außer Acht gelassen, da hier lediglich einige Variablen gesetzt werden, die erst im späteren Verlauf der Entwicklung des Spiels relevant werden.

b) Implementierung

Da bereits im Abschnitt 4.3.1.2 erklärt wurde, wie man einen Custom Inspector entwirft, wird auf eine nochmals genaue Beschreibung verzichtet. Stattdessen wird hauptsächlich auf die Implementierung der Hauptfunktionen des Tile-Designers sowie auf die wichtigsten Stellen im Editor-Script eingegangen.

Zuallererst müssen einige Variablen in der *OnEnable()*-Methode des Editor-Scripts initialisiert werden (siehe Listing 18).

```

31     tileDesigner = (TileDesigner)target;
32     Debug.Log("OnEnable " + tileDesigner.gameObject.name);
33
34     btnTransform = (Texture2D)Resources.Load("LevelEditor/tileTransform");
35     btnCollider = (Texture2D)Resources.Load("LevelEditor/tileCollider");
36     btnTexture = (Texture2D)Resources.Load("LevelEditor/tileTexture");
37     btnRotate = (Texture2D)Resources.Load("LevelEditor/doubleRotate_low");
38
39     //Blendet die Transform-Tools aus
40     Tools.hidden = true;
41     tileDesigner.transform.hideFlags = HideFlags.HideInInspector;
42
43     tileDesigner.functionSelection = 1;

```

Listing 18: Initialisieren einiger Variablen beim Selektieren des Tile-Objekts

Um auf das Ziel-Script, also auf den Tile Designer, zugreifen zu können, muss als erstes über `target` die entsprechende Variable gesetzt werden (Zeile 31).

Des Weiteren werden mehrere Bilder, über die `Resources`-Klasse von Unity, für die Buttons geladen. Hierfür wird die `Load()`-Methode verwendet, die den entsprechenden Pfad zur Bild-Datei erwartet (Zeile 34 bis 37).

Da die eigentlichen Transform-Tools und –Komponenten nicht benötigt werden, können diese, mit den in Zeile 40 und 41 gezeigten Variablen, ausgeblendet werden. Anschließend wird noch die `functionSelection`-Variable gesetzt, die bestimmt, welche Funktion zuerst eingeblendet werden soll.

Transform

Für die Implementierung der Transform-Funktion wird grundsätzlich auf eine `Vector3`-Variable zugegriffen, die die Position des Tile-Objekts enthält. Da die Positionierung der Tile-Objekte auf dem bereits erstellten, isometrischen Raster erfolgen soll, wird im Inspector die Position so angegeben, dass sie der isometrischen Form entspricht (siehe Listing 19).

```

152     EditorGUI.BeginChangeCheck();
153
154     tileDesigner.isoPosition = EditorGUILayout.Vector3Field("Iso Position", tileDesigner.isoPosition);
155
156     if (EditorGUI.EndChangeCheck())
157     {
158         tileDesigner.transform.localPosition = tileDesigner.ConvertIsoTo2D(tileDesigner.isoPosition);
159         tileDesigner.height = tileDesigner.isoPosition.z;
160         Repaint();
161     }

```

Listing 19: Anpassung des Inspectors für die isometrische Positionsangabe

Wie hier zu sehen ist, wird für die isometrische Position ein `Vector3`-Feld erzeugt (Zeile 154). Die Methode `BeginChangeCheck()` prüft dabei, ob das Feld im Inspector

verändert wird. Sobald das geschieht, gibt die Methode *EndChangeCheck()* den Wert *true* zurück und weitere Befehle können simultan ausgeführt werden.

Wenn sich die aktuelle Position (*localPosition*) des Tile-Objekts ändert, wird sie durch die isometrische Position ersetzt (Zeile 158). Die Positionsvariable *localPosition* ist hierbei ausschlaggebend für die tatsächliche Position in der Szene. Bevor diese jedoch gesetzt werden kann, muss die isometrische Position konvertiert werden. Der Grund dafür ist, wie bereits beschrieben, dass sich die isometrische und die von Unity genutzte Koordinatenform unterscheiden. Die Höhe der isometrischen Position (*isoPosition.z*) muss in einer separaten Variable gespeichert werden (Zeile 159), da es keine Höhenangabe in der Standard-Koordinatenform gibt.

Hierbei wird die Methode *ConvertIsoTo2D()* für die Konvertierung der isometrischen zur normalen Form genutzt (siehe Listing 20).

```

268 // Convert 3D isometric position to 2D position.
269 4 Verweise public Vector3 ConvertIsoTo2D(Vector3 positionIso)
270 {
271     Vector3 convertedPos;
272     convertedPos.x = positionIso.y - positionIso.x;
273     convertedPos.y = (positionIso.y + positionIso.x) / 2 + positionIso.z;
274     convertedPos.z = (positionIso.y - positionIso.x) * ISOSCALEZ - positionIso.z;
275
276     return convertedPos;
277 }

```

Listing 20: Konvertierung des isometrischen Vektors in einen normalen Vektor

Die Tiefe (z) der isometrischen Form wird entsprechend der nachfolgenden Formel mit der Konstante $\sqrt{3}$ konvertiert (*ISOSCALEZ*).

$$\begin{aligned}
 x_{2D} &= y_{iso} - x_{iso} \\
 y_{2D} &= y_{iso} + x_{iso}/2 + z_{iso} \\
 z_{2D} &= (y_{iso} - x_{iso}) * \sqrt{3} - z_{iso}
 \end{aligned}$$

Formel 1: Berechnung der normalen Koordinaten (2D)

Bei der normalen Koordinatenform wird oft von einer 2D-Position gesprochen. Dennoch besitzt diese auch eine z-Ebene. Im Gegensatz zur isometrischen

Koordinatenform entspricht diese jedoch nicht der Höhe, sondern der Tiefe des Tile-Objekts. Der z-Wert sorgt dafür, dass die Tile-Objekte, oder genauer gesagt die dazugehörigen 3D-Meshes, auch in auf der z-Achse korrekt im dreidimensionalen Raum aneinander positioniert werden. Die Darstellung der Objekte bleibt dabei jedoch unverändert (siehe orthogonale Projektion, Abschnitt 3.2).

Da es an verschiedenen Stellen nötig ist, aus der normalen Koordinatenform eine isometrische Form zu erhalten, wird auch dafür eine Methode zu Konvertierung implementiert (siehe Listing 21).

```

279 // Convert 2D position to isometric position.
280 1 Verweis public Vector3 Convert2DToIso(Vector3 position2D, float height, float off)
281 {
282     Vector3 convertedPos;
283     convertedPos.x = ((position2D.y - height) * 2 - position2D.x) / 2;
284     convertedPos.y = ((position2D.y - height) * 2 + position2D.x) / 2;
285     convertedPos.z = height + off;
286
287     return convertedPos;
288 }

```

Listing 21: Konvertierung der 2D-Position zur isometrischen Position

Hierbei wird der 2D-Koordinate zusätzlich auch die Höhe, inklusive Offset-Wert, nachfolgender Berechnung entsprechend übergeben.

$$\begin{aligned}
 x_{iso} &= \frac{(y_{2D} - \text{Platzierungshöhe}) * 2 - x_{2D}}{2} \\
 y_{iso} &= \frac{(y_{2D} - \text{Platzierungshöhe}) * 2 + x_{2D}}{2} \\
 z_{iso} &= \text{Platzierungshöhe} + \text{Offset}
 \end{aligned}$$

Formel 2: Berechnung der isometrischen Koordinaten

Im Allgemeinen ist die normale Form immer dann nötig, wenn etwas direkt in der Szene platziert oder eingeblendet werden soll. Die isometrische Form dagegen ist neben der praktischen Handhabung auch für logische Berechnungen nützlich. Ein Beispiel dafür ist die Tiefensortierung, die jedoch erst im späteren Verlauf der Entwicklung implementiert wird.

In der nachfolgenden Darstellung werden, anhand mehrerer unterschiedlich platzierter Tile-Objekte, beide Koordinatensysteme gegenübergestellt (siehe Abbildung 41).

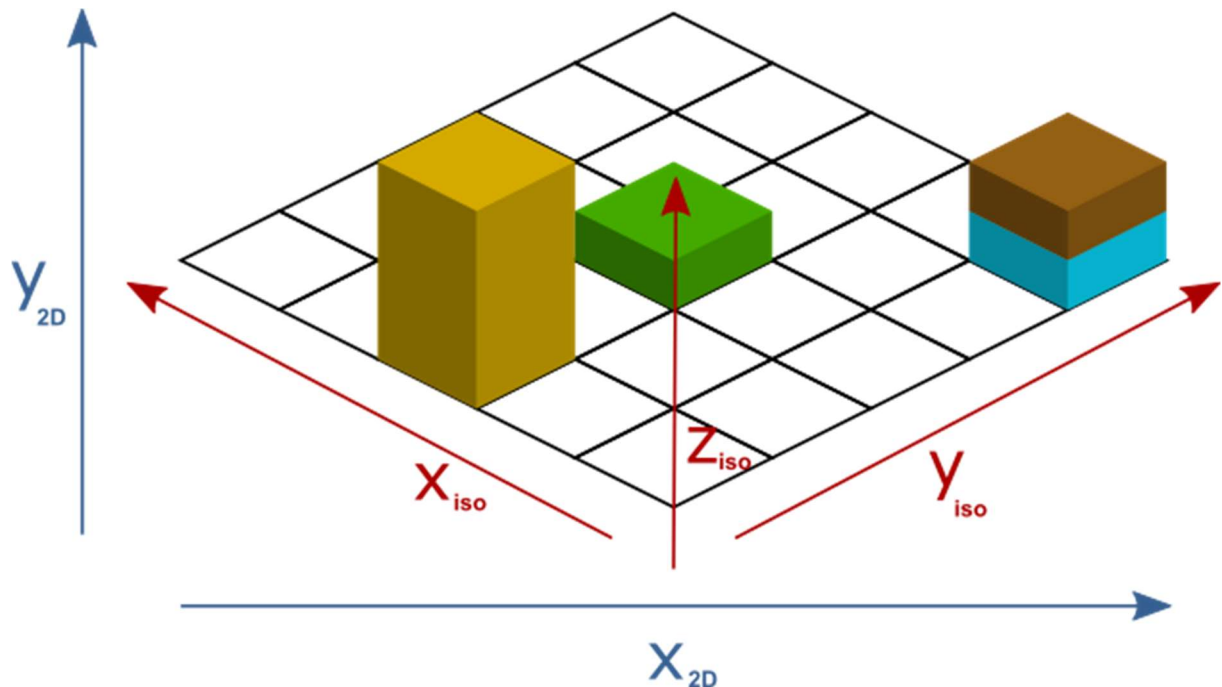


Abbildung 41: Gegenüberstellung der beiden Koordinatensysteme

Wie genau sich die beiden Koordinatensysteme auf die unterschiedlichen Positionswerte der Tile-Objekte auswirken, zeigt Tabelle 6.





Tile-Objekt	normale Position	isometrische Position
	x: 0 y: 0 z: 0	x: 0 y: 0 z: 0
	x: -2 y: -1 z: -1.732	x: 0 y: -2 z: 0
	x: 4 y: 0 z: 0	x: -2 y: 2 z: 0
	x: 4 y: 0.5 z: -0.289	x: -2 y: 2 z: 0.5

Tabelle 6: Koordinaten für die platzierten Tile-Objekte in normaler und isometrischer Form

Die letzte relevante Methode der Transform-Funktion wird in der *OnSceneGUI()* im Editor-Script aufgerufen. Sie sorgt dafür, dass isometrisch ausgerichtete Koordinatenpfeile (Handles) am Tile-Objekt eingeblendet werden. Diese sollen das, im Rahmen der isometrischen Positionierung, nutzlose (2D) Transform-Tool von Unity ersetzen. Damit lässt sich die Position direkt in der Szene verändern. Wobei *localPosition* dabei stetig aktualisiert wird.

Die verwendete Methode *IsometricPositionHandle()* erfordert die eigentliche Position des Tile-Objekts (*localPosition*) und ein Offset als Parameter (siehe Listing 22). Das Offset dient hierbei zur genaueren Anpassung der Handle-Position, da diese nicht immer am Mittelpunkt des Objekts dargestellt werden soll.

```

452 Vector3 IsometricPositionHandle(Vector3 position, Vector3 offset)
453 {
454     Vector3 handlePos = position + offset;
455
456     EditorGUI.BeginChangeCheck();
457     handlePos = Handles.DoPositionHandle(handlePos, Quaternion.Euler(-120, 0, -135));
458     handlePos = handlePos - offset;

```

Listing 22: Erstellung eines isometrisch ausgerichteten Position-Handles

Zu Beginn der Methode wird zunächst die Position mit einem Offset addiert (Zeile 454). Diese neue Position entspricht nun dem Mittelpunkt der Objektgrundfläche und wird für das Erstellen des Handles, der mit der Methode *DoPositionHandle()* von der *Handles*-Klasse erzeugt wird, verwendet (Zeile 457). Außerdem muss zusätzlich die entsprechende Rotation für die isometrische Ausrichtung übergeben werden. Hierfür wird auf Quaternion zugegriffen, die diese Rotation repräsentiert. Die *Euler()*-Funktion der Quaternion lässt schließlich das Objekt im Beispiel (siehe Listing 22) auf der x-Achse um -120° und auf der z-Achse um -135° rotieren.

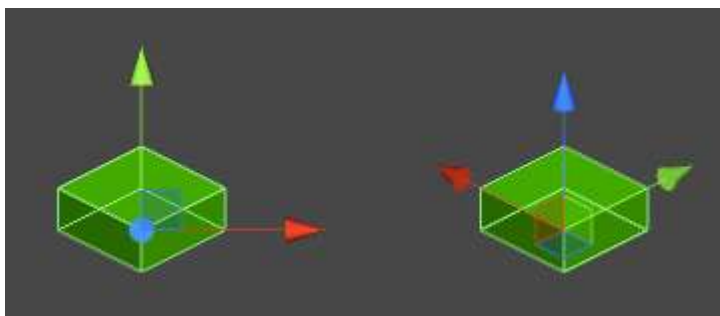


Abbildung 42: Vergleich des Transform-Tools von Unity mit dem eigenen Transform-Handle

Nachdem das Handle in der Szene erzeugt wurde, kann es an den Pfeilen der *handlePos*-Vektoren verändert werden. Anschließend wird der Offset-Vektor wieder von dem *handlePos*-Vektor subtrahiert, damit nachfolgend wieder mit der eigentlichen Objektposition gearbeitet werden kann.

Die zuvor aufgerufene *BeginChangeCheck()*-Methode überprüft dabei erneut, ob sich die Position des Handles ändert. Listing 23 zeigt die darauffolgenden Prozesse, wenn dieser Fall eintritt.

```

460         if (EditorGUI.EndChangeCheck())
461         {
462             handlePos.x = Mathf.Round(handlePos.x * 100000) / 100000;
463             handlePos.y = Mathf.Round(handlePos.y * 100000) / 100000;
464             handlePos.z = Mathf.Round(handlePos.z * 100000) / 100000;
465
466             //Changed only in Y Directon
467             if (handlePos.x == position.x)
468             {
469                 tileDesigner.height += (handlePos.y - position.y); ;
470             }

```

Listing 23: Runden der Handle-Position und Aktualisieren des Höhenwerts

Zunächst werden die neuen Koordinaten auf fünf Nachkommastellen gerundet (Zeilen 462 bis 464). Ansonsten würden die Werte (bei zu langen Zahlen) im Inspector nicht richtig dargestellt werden. Anschließend ist die Höhe zu speichern, da es sich hier wieder um die normale Koordinatenform handelt. Hierbei wird überprüft, ob sich bei der veränderten Position (*handlePos*), verglichen mit der Ausgangsposition (*position*), nur der y-Wert unterscheidet. Falls ja, kann davon ausgegangen werden, dass lediglich die Höhe verändert wurde. Die Differenz der beiden y-Werte, wird anschließend der Höhe hinzuaddiert (Zeile 469).

Sprite

Mit Sprite-Texturen, kurz Sprites, werden alle Grafiken bezeichnet, die für reine 2D-Darstellungen genutzt und direkt auf Objekte gezeichnet werden. Normalerweise sind Sprites Teil einer Animation und befinden sich in einer Grafikdatei mit mehreren anderen Sprites (Spritesheet). Jedoch werden in Unity auch die Grafiken so bezeichnet, die nicht zu einer Animation gehören und beispielsweise für die Umgebung genutzt werden (Tileset).

Grundsätzlich werden Sprites in Unity über die SpriteRenderer-Komponente zugewiesen (siehe Abbildung 43).



Abbildung 43: Die SpriteRenderer-Komponente von Unity

Neben der Auswahl des Sprites, lässt sich hier zusätzlich:

- das Sprite einfärben,
- das Sprite horizontal oder vertikal spiegeln,
- verschiedene Filter über das Sprite legen (Material),
- einen Sorting Layer festlegen,
- einen Order-Wert definieren.

Das wichtigste hierbei sind die Einstellungen des Sorting Layer und die des Order-Wertes. Damit legt man fest, in welcher Reihenfolge die verschiedenen Sprites in der Szene gerendert werden sollen. Der Sorting Layer gibt hierbei die Sortierebene an und der Order-Wert die Sortierung innerhalb dieser Ebene.

Bevor Sprites jedoch überhaupt einem Objekt zugewiesen werden können, müssen diese zunächst importiert werden (siehe Abbildung 44).

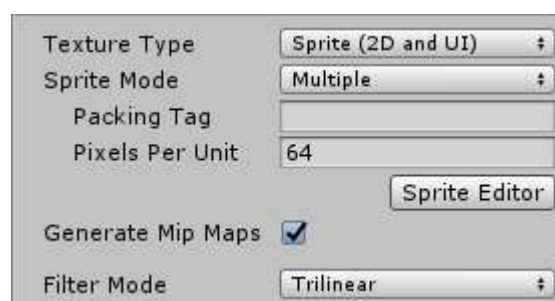


Abbildung 44: Importeinstellungen für Grafiken

Hierbei können folgende Importeinstellungen vorgenommen werden:

- **Texture Type**
 - legt den Texturtyp fest. Je nach Wahl werden zusätzliche Einstellungen eingeblendet. Andere Texturtypen sind an dieser Stelle nur für 3D-Darstellungen relevant.
- **Sprite Mode**
 - definiert, ob es sich in der Grafik um einen einzelnen Sprite oder um mehrere handelt.
- **Packing Tag**
 - kennzeichnet das Sprite mittels ID (optional).
- **Pixel Per Unit**
 - spezifiziert, wieviel Pixel einer Rastereinheit von Unity entsprechen soll. Als Standardwert wird hier 64 Pixel angegeben. Damit entspricht $y=1$ der Tile-Höhe und $x=2$ der Tile-Breite. Dadurch werden alle Koordinatenangaben nachvollziehbarer und leichter zu berechnen.
- **Sprite Editor**
 - legt die einzelnen Sprites und dessen Pivot-Punkte fest.
- **Generate Mip Maps**
 - definiert, ob Mipmaps generiert werden sollen. Eine Mipmap ist eine kompakte Abbildung des originalen Sprites und wird verwendet, um eine bessere Performance bei kleinen Darstellungen, wenn Grafiken beispielsweise weiter weg positioniert sind, zu erzielen.
- **Filter Mode**
 - stellt den Darstellungsmodus ein. Dies bewirkt, dass Sprites eher blockartig (Point) oder mehr verschwommen (Bilinear und Trilinear) dargestellt werden. Mit Trilinear lässt sich hierbei die beste Darstellung erzielen, da neben der Berechnung der Zwischenwerte, die bei Bilinear erfolgt, auch die Mipmaps einbezogen werden.

In der nachfolgenden Abbildung 45 wird zum einem dargestellt, wie so eine Grafik mit mehrere Sprites aussehen kann und zum anderen wird die Festlegung der einzelnen Sprite-Bereiche im Sprite Editor verdeutlicht.

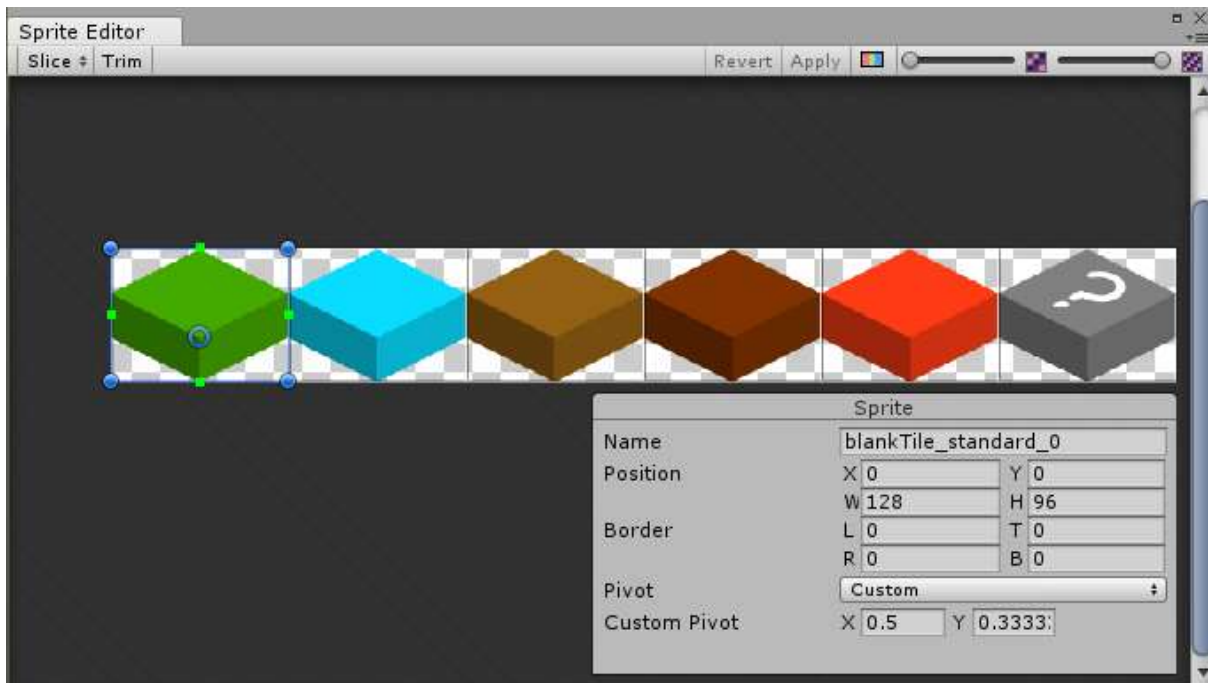


Abbildung 45: Beispiel eines Spritesheets im Sprite Editor

Beim Sprite-Editor bietet das Slice-Tool die Möglichkeit, das Spritesheet automatisch anhand der Farben und der Zwischenräume aufteilen zu lassen. Alternativ wird das Spritesheet in mehrere gleichgroße Flächen unterteilt, zu sehen in obiger Abbildung. In beiden Fällen werden dabei Position und Größe der einzelnen Sprite-Elemente automatisch festgelegt. Im Nachhinein können diese Werte manuell angepasst werden.

Nachdem alle Sprite-Bereiche definiert wurden, können die Pivot-Punkte festgelegt werden. Der Pivot-Punkt soll im Rahmen dieses Projektes jedoch nicht den Mittelpunkt des Sprites sondern dessen Grundfläche definieren, da sich auch die Platzierung der Tile-Objekte in der Szene immer am Pivot-Punkt orientiert.

Die Abmessung eines Pivot-Punkts kann nur im Verhältnis zu Breite beziehungsweise Höhe festgelegt werden. Das bedeutet, ein x-Wert von 0.5 entspräche, in diesem Fall, der Hälfte der Sprite-Breite und ein y-Wert von $0.\overline{33}$ ein Drittel der Sprite-Höhe.

An dieser Stelle wird ein Problem ganz klar ersichtlich und damit auch die grundlegende Intention für die Implementierung der eigenen Sprite-Funktion.

Denn die Bestimmung der Grundflächenmitte eines Sprites sowie der dazugehörigen Verhältnisse, kann vor allem bei komplexeren Objekten zu einem sehr zeitintensiven Vorgang werden. Deshalb wird mit der Sprite-Funktion nicht nur eine schnelle Möglichkeit geboten das Sprite eines Tile-Objektes zu ändern, sondern auch dessen Pivot-Punkt in isometrischer Form festzulegen.

Für das Einstellen des Sprites, wurde zunächst wieder ein Objektfeld im Inspector erzeugt. Diesmal jedoch für den Objekttyp [Sprite](#) (siehe Listing 24).

```

305     SpriteRenderer spriteRenderer = tileDesigner.SpriteRenderer;
306
307     EditorGUILayout.PrefixLabel("Sprite");
308     EditorGUI.indentLevel++;
309     spriteRenderer.sprite = (Sprite)EditorGUILayout.ObjectField(spriteRenderer.sprite,
310                                                                typeof(Sprite), false);
311     EditorGUI.indentLevel--;

```

Listing 24: Erzeugung eines Objektfeldes (Sprite) im Inspector

Das Feld greift dabei auf die *SpriteRenderer*-Komponente des Tile-Objekts zu, um das aktuelle Sprite zu erhalten und zuzuweisen.

Anschließend wird nun das Feld für den Pivot-Punkt erstellt (siehe Listing 25).

```

321     EditorGUI.BeginChangeCheck();
322
323     Vector2 pivotField = spriteRenderer.sprite.pivot / spriteRenderer.sprite.pixelsPerUnit;
324     pivotField = EditorGUILayout.Vector2Field("Pivot", pivotField, GUILayout.Expandwidth(true));
325
326     if (EditorGUI.EndChangeCheck())
327     {
328         Bounds bounds = spriteRenderer.sprite.bounds;
329
330         Vector2 pivotPoint;
331         pivotPoint.x = pivotField.x / bounds.size.x;
332         pivotPoint.y = pivotField.y / bounds.size.y;
333
334         SetPivot(spriteRenderer.sprite, pivotPoint);
335     }

```

Listing 25: Erzeugung des Pivot-Feldes im Inspector

Da sich die Position des Pivot-Punktes an die Größe der originalen Grafik (128x64 Pixel) orientiert, muss diese zunächst durch 64 dividiert werden (Zeile 323).

Wenn das Pivot-Feld angepasst wird, wird die Position nochmals dividiert, diesmal jedoch durch die Größe des Sprites. Der erhaltene Vektor entspricht nun dem den Breiten- und Höhenverhältnis (Zeile 328 bis 332).

Anschließend muss die neue Pivot-Position dem Sprite zugewiesen werden. Unity bietet dafür jedoch weder Methoden noch sonstige Alternativen an, die das Zuweisen des Pivots nach dem Importieren erlaubt. Deshalb musste an dieser Stelle die Methode *SetPivot()* implementiert werden, die alle Importvorgänge für das Sprite nochmals durchführt und dabei die aktuelle Pivot-Position setzt (siehe Listing 26).

```

400 void SetPivot(Sprite sprite, Vector2 pivotPoint)
401 {
402     //Pfad des Assets
403     string path = AssetDatabase.GetAssetPath(sprite);
404
405     //TexturImportDaten von dem Asset-Pfad laden und sie per Script auslesbar machen
406     TextureImporter textureImporter = AssetImporter.GetAtPath(path) as TextureImporter;
407     textureImporter.isReadable = true;
408
409     //Hole die Textur-Import-Einstellung and setze die Sprite Eigenschaften
410     TextureImporterSettings tis = new TextureImporterSettings();
411     textureImporter.ReadTextureSettings(tis);
412
413     tis.spriteMode = (int)SpriteImportMode.Multiple;
414     tis.spritePixelsPerUnit = 64f;
415     tis.filterMode = FilterMode.Trilinear;
416     textureImporter.textureType = TextureImporterType.Sprite;

```

Listing 26: Auslesen und Anpassen der Importdaten/-einstellungen

Im ersten Schritt muss der zugehörige Pfad zu Grafik ausgelesen werden (Zeile 403). Aus dem daraus erhaltenen Asset (Datei) wird ein *TextureImporter* erzeugt und so angepasst, dass dieser per Script auslesbar ist (Zeile 406 und 407).

Als nächstes wird ein Objekt für die Importeinstellungen (*TextureImporterSettings*) erzeugt und von dem *TextureImporter* ausgelesen. Die Importeinstellungen (Zeile 413 bis 416) entsprechen der in Abbildung 22 dargestellten Konfiguration.

Anschließend müssen die Einstellungen für die Sprites, die normalerweise im Sprite Editor durchgeführt werden, vorgenommen werden (siehe Listing 27).


```

418 //Eine Liste mit den Metadaten der Sprites (SpriteSheet) erstellen
419 List<SpriteMetaData> changeSpriteSheet = new List<SpriteMetaData>(textureImporter.spritesheet);
420 int spriteIndex = changeSpriteSheet.FindIndex(spriteData => spriteData.name == sprite.name);
421
422 //Das entsprechende Sprite zwischenspeichern, dessen Pivot bearbeiten und
423 //anschließend zurück in die Liste packen
424 SpriteMetaData smd = changeSpriteSheet[spriteIndex];
425 smd.alignment = (int)SpriteAlignment.Custom;
426 smd.pivot = pivotPoint;
427 changeSpriteSheet[spriteIndex] = smd;
428
429 //Die Liste mit den originalen Sprite austauschen
430 textureImporter.spritesheet = changeSpriteSheet.ToArray();
431
432 //Setze Textur-Import-Einstellungen
433 textureImporter.SetTextureSettings(tis);
434
435 //Das Asset aktualisieren
436 AssetDatabase.ImportAsset(path, ImportAssetOptions.ForceUpdate);

```

Listing 27: Anpassung der Sprites und Zuweisen der Importdaten/-einstellungen

Zunächst wird eine Liste mit Metadaten für die Sprites erstellt (Zeile 419). Mittels des Namens des Sprites ist der zugehörige Index aus der Liste zu ermitteln (Zeile 420).

Daraufhin kann die relevante Sprite-Information (*SpriteMetaData*) ausgelesen und angepasst werden. Nachfolgend wird die Ausrichtung des Pivots auf Custom gestellt und anschließend die aktuelle Pivot-Position gesetzt (Zeile 425 und 426).

Als Nächstes werden die entsprechenden Metadaten aktualisiert und dem Importer zugewiesen (Zeile 427 und 430). Zuletzt sind alle Einstellungen vom Importer zu setzen (Zeile 433) und das Asset wird aktualisiert (Zeile 436).

Collider und Mesh

Collider sind die Komponenten eines Spielobjekts, die dafür sorgen, dass Kollisionen mit anderen Objekten registriert werden. Wie bereits zu Beginn des Abschnitts erörtert, werden für die Entwicklung 3D-Collider genutzt. 2D-Collider können Kollisionen nicht auf der z-Ebene, also in der Höhe (isometrische Perspektive), erkennen. Die Positionierung sowie Bewegung (beispielsweise durch Springen der Spielfigur) auf unterschiedlich hohen Tile-Objekten, soll jedoch eine wichtige Mechanik der Spielwelt darstellen. Deshalb fiel, im Rahmen dieses Projektes, die Entscheidung gegen die Nutzung von 2D-Collidern.

Grundsätzlich gibt es in Unity folgende Komponenten um 3D-Collider zu erstellen:

- **Box Collider:** Erstellt ein Collider in der Form einer Box.
- **Sphere Collider:** Erstellt ein Collider in der Form einer Kugel.

- **Capsule Collider:** Erstellt ein Collider in der Form einer Kapsel.
- **Mesh Collider:** Erstellt einen Collider anhand der Form eines Meshes.

Bei den ersten drei Komponenten, handelt es sich um sogenannte „primitive Collider“. Diese erzeugen Collider in einer der entsprechenden Grundformen, die sich anschließend grob anpassen lassen.

Mit dem Mesh Collider hat man dagegen die Möglichkeit, mittels Meshes, die Collider frei zu formen.

Ein Mesh ist das Grundgerüst eines 3D-Modells. Es besteht aus mehreren Vektoren (Vertices), wobei davon immer jeweils drei Vektoren eine Fläche bilden (Triangle oder Polygon). Die daraus entstehenden Flächen bilden schließlich eine komplexe Struktur, die als Polygonnetz oder eben auch als Mesh bezeichnet wird (siehe Abbildung 46).

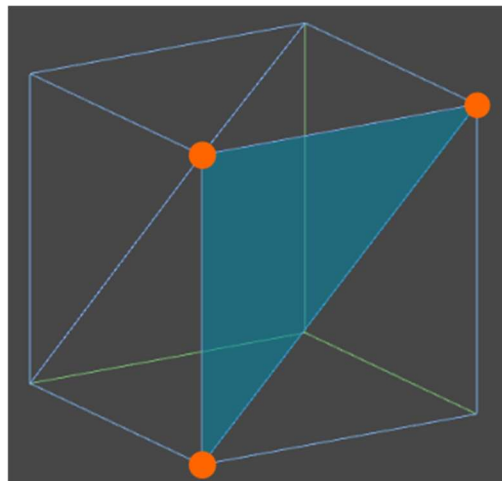


Abbildung 46: Ein Mesh mit einem hervorgehobenen Triangle und den dazugehörigen Vertices

Im aktuellen Stand der Entwicklung würde zwar ein Box-Collider für die Tile-Objekte ausreichen, dennoch fiel die Wahl auf den Mesh Collider, damit bereits der Grundstein für die Implementierung von komplexeren Objekten gelegt werden kann (siehe Abbildung 47).



Abbildung 47: Eine Mesh-Collider-Komponente

Folgende Parameter lassen sich einstellen:

- **Convex**
Ermöglicht dem Collider mit anderen Mesh Collidern zu kollidieren.
- **Is Trigger**
gibt an, ob es sich bei dem Collider um einen Auslöser (engl. Trigger) handelt. Falls dem so ist, registriert das Tile-Objekt nur noch den Kontakt und kollidiert, physikalisch gesehen, nicht mehr mit anderen Objekten.
- **Material**
ist ein Verweis auf ein *Physic Material*, welches physikalische Eigenschaften definiert (z.B. die Reibung des Objekts). Es beeinflusst die Interaktion mit anderen Collidern.
- **Mesh**
referenziert auf einen Mesh. Es bestimmt die Form für den Kollisionsbereich.

Nachfolgend wird erläutert, wie es möglich gemacht wird, im Custom Inspector ein Mesh zu erstellen, es zuzuweisen und es bearbeiten zu können.

Hierfür wird zunächst ein Feld im Inspector erzeugt, mit dem man über den Mesh Collider auf das Mesh zugreifen und es aktualisieren kann (siehe Listing 28).

```

189 |         EditorGUILayout.PrefixLabel("Mesh");
190 |         EditorGUI.indentLevel++;
191 |         tileDesigner.Mesh = EditorGUILayout.ObjectField(tileDesigner.Mesh, typeof(Mesh), false) as Mesh;
192 |         EditorGUI.indentLevel--;

```

Listing 28: Erstellen eines Objektfeldes für das Mesh

Da es sich um ein Objektfeld handelt, muss zusätzlich noch der Objekttyp *Mesh* bei der Erstellung angegeben werden.

Anschließend wird ein Button im Inspector platziert, der dafür sorgt, dass ein neuer Mesh erzeugt wird (siehe Listing 29).

```

210     if (GUILayout.Button("Create Mesh", GUILayout.ExpandWidth(true)))
211     {
212         string name = "customMesh_";
213         int i = 0;
214
215         while(AssetDatabase.LoadAssetAtPath("Assets/Resources/Meshes/" + name + i + ".asset", typeof(Mesh)) != null)
216         {
217             i++;
218         }
219
220         tileDesigner.CreateBoxMesh(name + i);
221     }

```

Listing 29: Erstellen eines Buttons inklusive Methodenaufruf zur Mesh-Erzeugung

Für die Button-Erzeugung wird ein String für die Bezeichnung und eine Layout-Eigenschaft verwendet. Die Layout-Eigenschaft sorgt dafür, dass die Breite des Buttons sich der Breite des Inspectors anpasst.

Da die neu erstellten Meshes nicht alle den gleichen Namen haben können, muss innerhalb der Button-Logik überprüft werden, ob der Name bereits im Meshes-Ordner existiert. Der String des Namens enthält dabei eine Zählvariable, die solange hochgezählt wird, bis kein weiteres Asset per Pfad geladen werden kann. Anschließend wird mit diesem Namen die Methode *CreateBoxMesh()* ausgeführt.

```

144     public void CreateBoxMesh(string name)
145     {
146         customMesh = new CustomBoxMesh(name, 1, 1, new float[] { 0.5f, 0.5f, 0.5f, 0.5f });
147
148         Mesh = customMesh.Mesh;
149
150         AssetDatabase.CreateAsset(Mesh, "Assets/Resources/Meshes/" + name + ".asset");
151         AssetDatabase.SaveAssets();
152         AssetDatabase.Refresh();
153     }

```

Listing 30: Methode zur Erzeugung eines Meshes

Hier wird zuallererst ein CustomBoxMesh-Objekt erzeugt, dem folgende Parameter übergeben werden:

- Name des Meshes
- Die Länge des Meshes (x-Ebene)
- Die Breite des Meshes (y-Ebene)
- Vier Höhenwerte für die oberen Eckpunkte des Meshes (z-Ebene)

Die Höhe ist separat für alle vier oberen Eckpunkte einstellbar, um beispielsweise auch abfallendes Gelände zu erzeugen, damit sich die Spielfigur eine Ebene höher oder tiefer bewegen kann. Die Größenangaben entsprechen hierbei alle der isometrischen Form.

Der generierte Box-Mesh wird anschließend dem Mesh Collider zugewiesen (Zeile 148) und als Asset im Meshes-Ordner erstellt (Zeile 150 bis 152).

Als Nächstes wird ein kurzer Blick in die Klasse *CustomBoxMesh()* geworfen und im Folgenden dargestellt, wie genau ein Mesh erzeugt wird (siehe Listing 31).

```

5 public class CustomBoxMesh
6 {
7     12 Verweise
8     public Mesh Mesh { get; set; }
9     11 Verweise
10    public float Length { get; set; }
11    11 Verweise
12    public float Width { get; set; }
13    12 Verweise
14    public float[] Height { get; set; }
15
16    3 Verweise
17    public CustomBoxMesh(string name, float _length, float _width, float[] _height)
18    {
19        Length = _length;
20        Width = _width;
21        Height = _height;
22        Mesh = GenerateMesh(name);
23    }

```

Listing 31: Eigenschaften und Konstruktor der CustomBoxMesh-Klasse

Zu Beginn der Klasse werden die entsprechenden Properties zur Größenangabe und zum erstellenden Mesh deklariert. Diese werden dann beim Instanzieren eines *CustomBoxMesh*-Objekts im Konstruktor initialisiert. Für das Mesh wird hierbei die *GenerateMesh()*-Methode verwendet (siehe Listing 32).

```

20 public Mesh GenerateMesh (string name)
21 {
22     Mesh boxMesh = new Mesh();
23     boxMesh.name = name;
24
25     #region Vertices
26     Vector3 p0 = new Vector3(-0.5f, -0.5f, Height[0]);
27     Vector3 p1 = new Vector3(Length - 0.5f, -0.5f, Height[1]);
28     Vector3 p2 = new Vector3(Length - 0.5f, -0.5f, 0);
29     Vector3 p3 = new Vector3(-0.5f, -0.5f, 0);
30
31     Vector3 p4 = new Vector3(-0.5f, Width - 0.5f, Height[2]);
32     Vector3 p5 = new Vector3(Length - 0.5f, Width - 0.5f, Height[3]);
33     Vector3 p6 = new Vector3(Length - 0.5f, Width - 0.5f, 0);
34     Vector3 p7 = new Vector3(-0.5f, Width - 0.5f, 0);

```

Listing 32: Erzeugen eines neuen Mesh-Objektes und Definition der jeweiligen Eckpunkte

Als erstes wird ein neues Mesh-Objekt erzeugt und dem übergebenen Namen zugewiesen. Im Anschluss werden die Eckpunkte der Box mit Hilfe der

Größenangaben definiert. Welcher Vektor für welchen Eckpunkt steht, ist in Abbildung 48 ersichtlich.

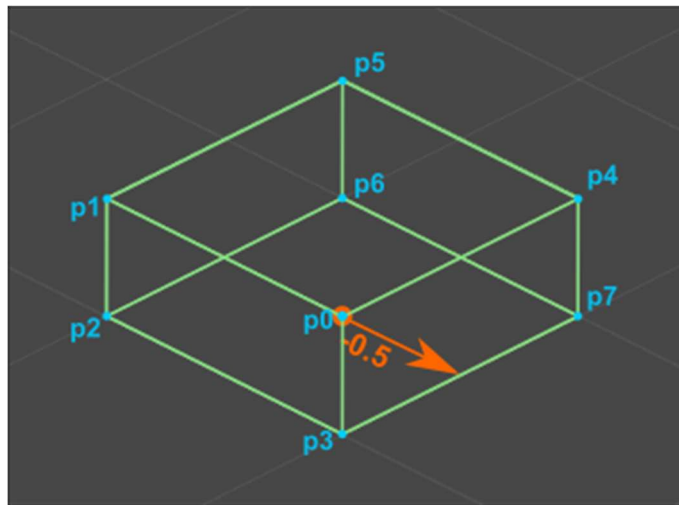


Abbildung 48: Alle Eckpunkte (blau) des Custom Meshes inklusive Abstand vom Mittelpunkt (orange)

Im Vergleich mit Listing 32 wird ersichtlich, dass eine Einstellung der Länge nur die Punkte p_1 , p_2 , p_5 sowie p_6 und eine Einstellung der Breite die Punkte p_4 , p_5 , p_7 sowie p_6 beeinflusst. Die Höhenangaben korrelieren mit den Punkten p_0 , p_1 , p_4 und p_5 . Außerdem ist der Wert -0.5 zu beachten, da sich die Grenzen des Meshes nicht vom Zellenmittelpunkt aus orientieren sollen, sondern vom Zellenrand des Rasters.

```

36  Vector3[] vertices = new Vector3[]
37  {
38      // Front
39      p0, p1, p2, p3,
40
41      // Right
42      p7, p4, p0, p3,
43
44      // Top
45      p4, p5, p1, p0,
46
47      // Bottom
48      p6, p7, p3, p2,
49
50      // Left
51      p5, p6, p2, p1,
52
53      // Back
54      p7, p6, p5, p4
55  };
56  #endregion

```

Listing 33: Festlegung der Vertices

In Listing 33 werden nun die Punkte in einen Array für die Vertices gespeichert. Hierbei werden alle Punkte, die eine Fläche der Box bilden, separat gesetzt. Das bedeutet, dass die Menge der Vertices, das Vierfache der Eckpunkte entspricht.

Anschließend müssen die Triangles aus den entsprechenden Vertices gebildet werden (siehe Listing 34).

```

58 #region Triangles
59 int[] triangles = new int[]
60 {
61     // Front
62     3, 1, 0,
63     3, 2, 1,
64
65     // Right    7, 5, 4 / 7, 6, 5
66     3 + 4 * 1, 1 + 4 * 1, 0 + 4 * 1,
67     3 + 4 * 1, 2 + 4 * 1, 1 + 4 * 1,
68
69     // Top     11, 9, 8 / 11, 10, 9
70     3 + 4 * 2, 1 + 4 * 2, 0 + 4 * 2,
71     3 + 4 * 2, 2 + 4 * 2, 1 + 4 * 2,
72
73     // Bottom  15, 13, 12 / 15, 14, 13
74     3 + 4 * 3, 1 + 4 * 3, 0 + 4 * 3,
75     3 + 4 * 3, 2 + 4 * 3, 1 + 4 * 3,
76
77     // Left    19, 17, 16 / 19, 18, 17
78     3 + 4 * 4, 1 + 4 * 4, 0 + 4 * 4,
79     3 + 4 * 4, 2 + 4 * 4, 1 + 4 * 4,
80
81     // Back    23, 21, 20 / 23, 22, 21
82     3 + 4 * 5, 1 + 4 * 5, 0 + 4 * 5,
83     3 + 4 * 5, 2 + 4 * 5, 1 + 4 * 5,
84 };
85 #endregion

```

Listing 34: Festlegung der Triangles

Hierfür wird ein weiteres Array erzeugt, das die Indizes der entsprechenden Vertices beinhalten soll. Alle drei Vektoren bilden dabei ein Triangle. Da eine Fläche der Box aus mindesten zwei Triangles besteht, müssen pro Fläche sechs Vertices angegeben werden.

Die Anordnung der Vertices in Listing 34 wurde dabei so gewählt, dass für die Triangles ein Algorithmus eingesetzt werden kann, der die zusammengehörenden Vertices ermittelt [21].

Zum Abschluss müssen die Vertices und Triangles dem Mesh-Objekt zugewiesen werden. Mit den Methoden *RecalculateBounds()* und *Optimize()* wird das Mesh aktualisiert (siehe Listing 35).

```
87     boxMesh.vertices = vertices;
88     boxMesh.triangles = triangles;
89
90     boxMesh.RecalculateBounds();
91     boxMesh.Optimize();
92
93     return boxMesh;
```

Listing 35: Rückgabe des Meshes

Der Ablauf zur Bearbeitung des Meshes unterscheidet sich kaum von der Erstellung eines Meshes. Im Inspector werden, wie bisher, die entsprechenden Felder für die Größenangaben erzeugt. Mit diesen wird auf die entsprechenden Properties des *CustomBoxMesh*-Objekts zugegriffen und eine Methode ausgeführt, die das Array für die Vertices neu initialisiert sowie das Mesh anschließend wieder aktualisiert (siehe Listing 33 und Listing 35).

4.3.1.4 Platzieren der Tile-Objekte auf dem Spielfeld

Ein Spielfeld ist ein Abschnitt der Spielwelt (auch als Level bezeichnet). Es beinhaltet eine Grundfläche sowie mehrere Tile-Objekte, die sich darauf befinden.

Auch hier sollen die Einstellungsoptionen für das Bearbeiten des Spielfeldes über ein Custom Inspector realisiert werden. Der Fokus soll auf dem schnellen Platzieren der Tile-Objekte liegen.

Der entsprechende Inspector wird durch den **Level Designer** erstellt. Die Anwendungsfälle hierfür werden in Abbildung 49 dargestellt.

a) Anwendungsfalldiagramm

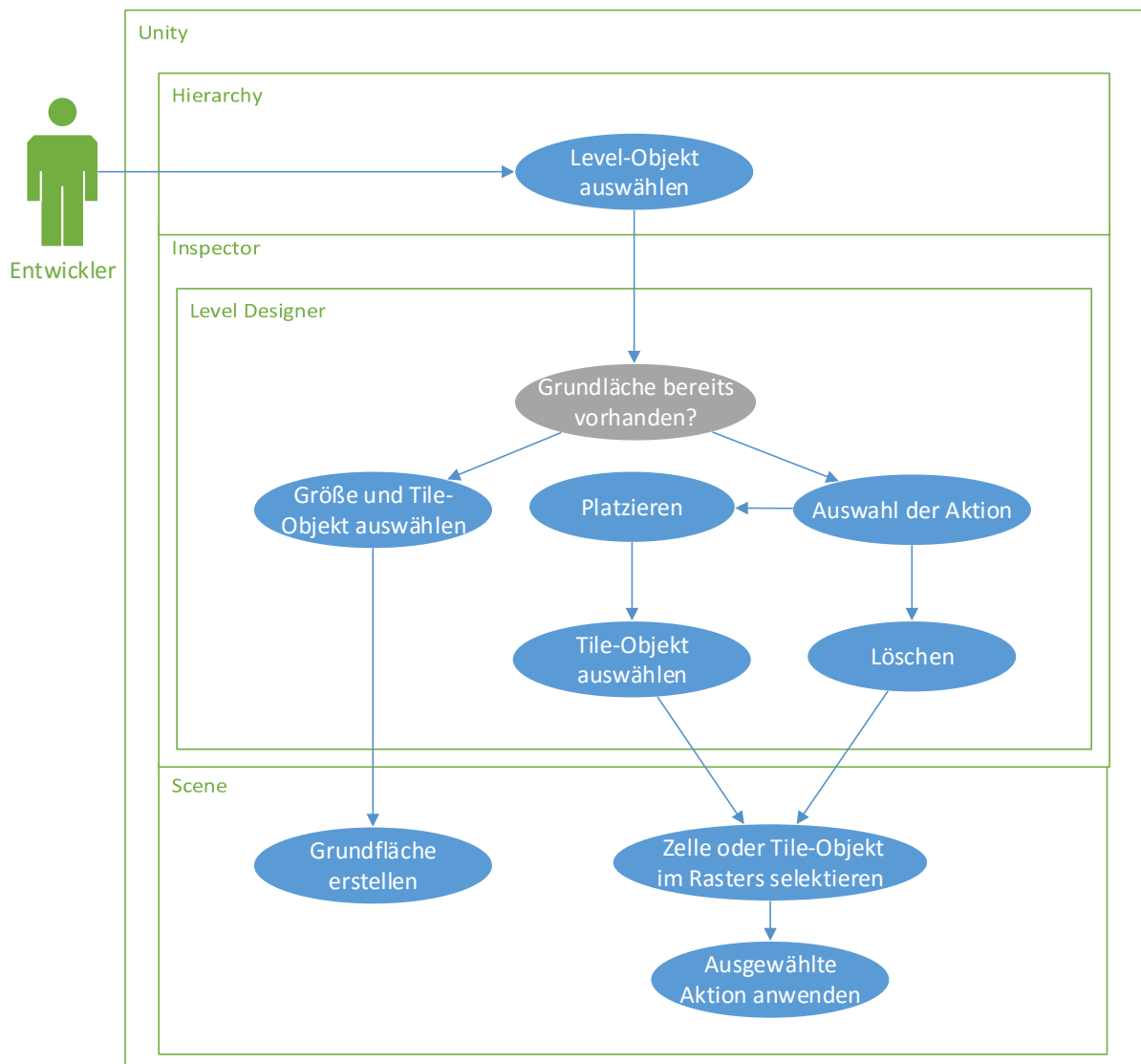


Abbildung 49: Anwendungsfalldiagramm für das Erstellen eines Spielfeldes

Nach der Auswahl des Spielfeldobjekts, sind die Einstellungen für die Grundfläche im Inspector anzuzeigen. Sobald eine Größe (Breite x Höhe) und ein Tile-Objekt ausgewählt und die Grundfläche erzeugt wurde, sollen die folgenden Funktionen des Level Designers verwendbar sein:

- **Platzieren**

Die Platzieren-Funktion gibt dem Entwickler die Möglichkeit mittels Mausklick die Tile-Objekte in der Szene bzw. auf der Grundfläche zu platzieren. Dabei soll zusätzlich eingestellt werden können, ob das Tile-Objekt auch auf bereits belegten Positionen platziert werden kann. Das bereits platzierte Tile-Objekt wird dabei überschrieben. Die Auswahl des Tile-Objekts findet über eine eigene Liste statt.

- **Löschen**

Mit der Löschen-Funktion werden bereits platzierte Tile-Objekte wieder entfernt.

Auch hier sollen wieder passende Hilfselemente zur komfortableren Bedienung eingeblendet werden. Beim Platzieren soll beispielsweise, das entsprechende Mesh des ausgewählten Tile-Objekts an der Mausposition angezeigt werden.

Die Platzieren-Funktion entspricht hierbei auch der Primärfunktion und nimmt dementsprechend den größten Anteil des folgenden Implementierungsabschnitts ein.

b) Implementierung

Zunächst werden die Variablen in der *OnEnable()*-Methode des Editor-Scripts initialisiert (siehe Listing 36).

```

41         levelDesigner = (LevelDesigner)target;
42         Debug.Log("OnEnable " + levelDesigner.gameObject.name);
43
44         btnPlace = (Texture2D)Resources.Load("LevelEditor/tilePlace");
45         btnDelete = (Texture2D)Resources.Load("LevelEditor/tileDelete");
46         btnEdit = (Texture2D)Resources.Load("LevelEditor/tileEdit");
47
48         //Blendet die Transform-Tools aus
49         Tools.hidden = true;
50         levelDesigner.transform.hideFlags = HideFlags.HideInInspector;
51
52         levelDesigner.functionSelection = 1;

```

Listing 36: Initialisieren einiger Variablen beim Selektieren des Level-Objekts

Bei der Initialisierung wird analog zum Tile Designer vorgegangen: erst die Variable für das *target* setzen (Zeile 41), die entsprechenden Bilder für die Buttons laden (Zeile 44 bis 46), die Transform-Tools und -Komponenten ausblenden (Zeile 49 und 50) sowie die aktuell dargestellte Funktion festlegen (Zeile 52).

Anschließend werden die Einstellungen für die Generierung der Grundfläche im Inspector dargestellt. Hierfür sind wieder geeignete Felder in der *OnInspectorGUI()*-Methode zu erzeugen (siehe Listing 37).

```

233     sliderX = EditorGUILayout.IntSlider("width (x)", sliderX, 5, 50);
234     sliderY = EditorGUILayout.IntSlider("Length (Y)", sliderY, 5, 50);
235
236     levelDesigner.grndObj = (GameObject)EditorGUILayout.ObjectField(levelDesigner.grndObj,
237                             typeof(GameObject), true);

```

Listing 37: Erzeugen der relevanten Felder für das Generieren einer Grundfläche im Inspector

Wie bereits bei der Implementierung des Rasters, werden auch hier zwei Schieberegler für die Größe (isometrisch) erzeugt (Zeile 227 und 228). Hinzu kommt hier noch ein Objektfeld, das angibt, aus welchen Tile-Objekten die Grundfläche bestehen soll (Zeile 236).

Als nächstes wird der Button, mittels der bereits bekannten Parameter, erzeugt (Listing 38).

```

238     if (GUILayout.Button("Generate", GUILayout.ExpandWidth(true)) && levelDesigner.grndObj)

```

Listing 38: Erzeugung des Buttons zur Generierung

Die Logik innerhalb des Buttons wird dabei jedoch nur ausgeführt, wenn auch ein Tile-Objekt ausgewählt wurde.

```

240     GameObject undergroundObject = new GameObject("Underground");
241     undergroundObject.transform.SetParent(levelDesigner.transform, false);
242
243     int fromtoX = Mathf.FloorToInt(sliderX / 2f);
244     int fromtoY = Mathf.FloorToInt(sliderY / 2f);
245
246     for (int x = -fromtoX; x < fromtoX + (sliderX - fromtoX*2); x++)
247     {
248         for (int y = -fromtoY; y < fromtoY + (sliderY - fromtoY*2); y++)
249         {
250             GameObject obj = (GameObject)Instantiate(levelDesigner.grndObj);
251             obj.name = levelDesigner.grndObj.name;
252             obj.transform.position = TEST2ISO.ConvertIsoTo2D(new Vector3(x, y, -0.5f));
253             obj.transform.SetParent(undergroundObject.transform, false);
254             obj.GetComponentInChildren<SpriteRenderer>().sortingLayerName = "Underground";
255         }
256     }

```

Listing 39: Logik des Buttons zur Generierung der Grundfläche

In Listing 39 wird zunächst ein *GameObject* als Child des Level-Objektes erzeugt (Zeile 240 und 241), das im Anschluss alle Tile-Objekte erhalten soll. Da sich der Koordinatenmittelpunkt auch in der Mitte des Grundfläche befinden soll, werden als nächstes die Werte des Schiebereglers halbiert und, mit Hilfe der *FloorToInt()*-Methode, abgerundet (Zeile 243 und 244). Die erhaltenen Werte ergeben den Start- bzw. Endpunkt für die verschachtelten Schleifen.

Die Schleifen werden dabei so ausgeführt, dass mit dem untersten Punkt (-x und -y) begonnen und anschließend die Tile-Objekte, beginnend mit der y-Reihe (Länge), nach und nach instanziiert werden. Die x-Reihe (Breite) wird weitergeführt sobald die zugehörige y-Reihe fertig ist.

Die Instanziierung erfolgt mit Hilfe des übergebenen Tile-Objekts. Diesem werden nachfolgend ein Name (Zeile 251), die entsprechende 2D Position (Zeile 252), das Parent-Objekt (Zeile 253) sowie ein Layer (Zeile 254) zugeordnet.

Im Anschluss sollen nun die Tile-Objekte auf dem eigentlichen Spielfeld platziert werden können.

Dafür müssen zuallererst ein Objektfeld, für das zu platzierende Tile-Objekt, ein *boolean*-Feld, für den Überschreibungsmodus, und ein Schieberegler, für die Platzierungshöhe (Ebene) erstellt werden (siehe Listing 40).

```

293 private void DrawPlaceTiles()
294 {
295     levelDesigner.currentTileObject = (TileDesigner)EditorGUILayout.ObjectField(levelDesigner.
296         currentTileObject,typeof(TileDesigner), true);
297     levelDesigner.override_Tile = EditorGUILayout.Toggle("Override", levelDesigner.override_Tile);
298     levelDesigner.atPlain = EditorGUILayout.IntSlider("Draw at Plain", levelDesigner.atPlain, 0, 20);
299 }

```

Listing 40: Felder für die Platzieren-Funktion

Der Überschreibungsmodus sorgt dafür, dass beim Platzieren andere Tile-Objekte überschrieben werden können. Mittels Schieberegler lassen sich Tile-Objekte bis zu einer Höhe von $z=10$ (Stufe 20) platzieren.

Einer der wichtigsten Bestandteile zum Bearbeiten des Spielfelds ist das korrekte Selektieren und Erfassen der Position im isometrischen Raster der Szene.

Zunächst wird hierzu im Ziel-Script (*LevelDesigner*) die Darstellung für das Hervorheben einer Zelle (*Gizmo*) implementiert. Dafür müssen in der *OnDrawGizmos()*-Methode folgende Befehle ausgeführt werden (siehe Listing 41).

```

55     Vector3 p1 = gizmoPosition + new Vector3(-cellwidth / 2, 0);
56     Vector3 p2 = gizmoPosition + new Vector3(0, -cellHeight / 2);
57     Vector3 p3 = gizmoPosition + new Vector3(cellwidth / 2, 0);
58     Vector3 p4 = gizmoPosition + new Vector3(0, cellHeight / 2);
59
60     Gizmos.DrawLine(p1, p2);
61     Gizmos.DrawLine(p2, p3);
62     Gizmos.DrawLine(p3, p4);
63     Gizmos.DrawLine(p4, p1);

```

Listing 41: Darstellung eines Gizmos für das Hervorheben einer Zelle

Zuerst werden die vier Punkte einer Zelle definiert. Diese lassen sich mittels Höhe und Breite leicht ermitteln. Hinzu wird jeweils die aktuelle Zellenposition (*gizmoPosition*) addiert. Die Punkte werden mit der *DrawLine()*-Methode verbunden und in der Szene dargestellt.

Nun geht es darum, die korrekte Zellenposition anhand der Mausposition zu ermitteln und zu aktualisieren. Hierbei wird in der *OnSceneGUI()*-Methode des Editor-Scripts als erstes auf die Klasse *HandleUtility* zugegriffen. Diese Klasse enthält unter anderem die *GUIPointToWorldRay()*-Funktion. Mit dieser Funktion kann ein Punkt in der Unity-Umgebung in Welt- bzw. Szenenkoordinaten, in Form eines *Ray*-Objekts, umgewandelt werden (siehe Listing 42).

```
309 | Ray ray = HandleUtility.GUIPointToWorldRay(Event.current.mousePosition);
```

Listing 42: Erzeugen eines Ray-Objekts

Ein *Ray*-Objekt stellt hierbei einen Strahl mit einem Ursprung und einer Richtung dar. An dieser Stelle werden jedoch nur die Ursprungskordinaten verwendet, die der aktuellen Mausposition in der Szene entsprechen.

```
316 | Vector3 mousePos = new Vector3();
317 | mousePos.x = Mathf.Round(ray.origin.x);
318 | mousePos.y = Mathf.Round(ray.origin.y * 2) / 2;
```

Listing 43: Erstellen einer Vector3-Variable für die Mausposition

In Listing 43 werden die Ursprungskordinaten des *Ray*-Objekts einer neuen *Vector3*-Variablen zugewiesen. Die x-Koordinate wird dabei auf eine volle und die y-Koordinate auf eine halbe Nachkommastelle gerundet. Damit kann die Mausselektionen im Raster schon einmal grob eingeschränkt werden. Zusätzlich sind weitere Abfragen nötig, damit die Zellen tatsächlich korrekt selektiert werden (siehe Listing 44).

```
320 | Vector3 tilePos = new Vector2();
321 |
322 | if (Mathf.Abs(mousePos.x) % 2 == 0 && (Mathf.Abs(mousePos.y)) % 1 == 0)
323 | {
324 |     tilePos = mousePos;
325 | }
326 |
327 | else if (Mathf.Abs(mousePos.x) % 2 == 1 && Mathf.Abs(mousePos.y) % 1 == 0.5f)
328 | {
329 |     tilePos = mousePos;
330 | }
331 |
332 | else
333 | {
334 |     tilePos = levelDesigner.gizmoPosition;
335 | }
```

Listing 44: Weitere Bedingung für die korrekte Zellenselektierung

Hierfür muss eine neue Zwischenvariable (*tilePos*) erzeugt werden (Zeile 320), die mit der jeweiligen Mausposition gesetzt wird, wenn eine der beiden Bedingungen erfüllt ist:

<p>1. Bedingung</p> $ x_{Maus} \bmod 2 = 0 \text{ und } y_{Maus} \bmod 1 = 0$ <p>2. Bedingung</p> $ x_{Maus} \bmod 2 = 1 \text{ und } y_{Maus} \bmod 1 = 0.5$

Formel 3: Bedingungen für korrekte Zellenselektion

Die erste Bedingung entspricht hierbei einer Zellenselektion, bei dem der y-Wert einer ganzen Zahl entspricht. Der zugehörige x-Wert darf dabei nur aus geraden Koordinaten bestehen.

Die zweite Bedingung entspricht dagegen einer Selektion, bei der der y-Wert einer 0.5er Stelle entspricht. Der zugehörige x-Wert muss ungerade sein.

In Abbildung 50 werden die Bedingungen an einigen Beispielkoordinaten dargestellt.

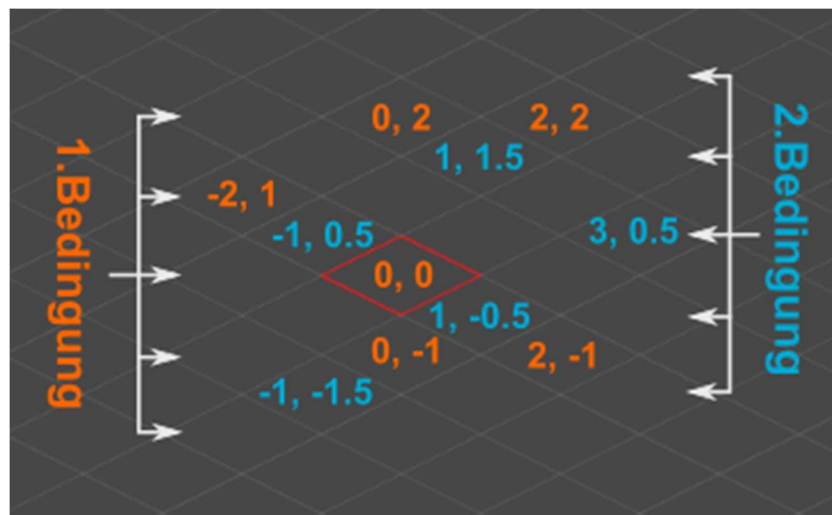


Abbildung 50: Gegenüberstellung der Bedingungen an Beispielkoordinaten

Sollte keiner der beiden Bedingungen erfüllt werden, handelt es sich um keine legitime Zellenposition und *tilePos* wird mit der alten Zellenposition (*gizmoPosition*) belegt.

Falls jedoch eine der beiden Bedingungen zutrifft und *tilePos* die aktuelle Mausposition enthält, wird *tilePos* anschließend der *gizmoPosition* zugewiesen. Die Szene wird neu gezeichnet, wenn es sich nicht um dieselben Koordinaten handelt (siehe Listing 45).

```

337     if (tilePos != levelDesigner.gizmoPosition)
338     {
339         levelDesigner.gizmoPosition = tilePos;
340         SceneView.RepaintAll();
341     }

```

Listing 45: Aktuelle Zellenposition setzen und Szene neu zeichnen

Nun kann jede Zelle im Raster selektiert werden, in dem sich der Mauszeiger über der Zelle befindet. Anschließend müssen noch die Tile-Objekte mittels Mausklick platziert werden können. Hierfür wird über *Event.current* auf alle aktuell verarbeiteten Ereignisse zugegriffen und verglichen, welche Art der Eingabe erfolgt (siehe Listing 46).

```

382     if (current.type == EventType.mouseDown || current.type == EventType.mouseDrag)
383     {
384         Vector3 pos = new Vector3(tilePos.x, tilePos.y, tilePos.y * ISOSCALEZ);
385         pos += TileDesigner.ConvertIsoTo2D(new Vector3(0, 0, levelDesigner.atPlain * 0.5f));
386
387         if ((current.button == 0) && (levelDesigner.functionSelection == 1))
388         {
389             Debug.Log("Create: " + CreateTile(pos));
390         }
391
392         if ((current.button == 0) && (levelDesigner.functionSelection == 2))
393         {
394             Debug.Log("Delete: " + DeleteTile(pos));
395         }
396     }

```

Listing 46: Eingabeabfrage mit anschließenden Erstellen oder Löschen eines Tile-Objekts

Als erstes wird überprüft, ob es sich bei der erfassten Eingabe um einen Mausklick oder eine gedrückte Maustaste handelt (Zeile 382). Anschließend wird die Position des Tile-Objekts neu kalkuliert. Hierfür wird der z-Wert mittels $y * \sqrt{3}$ ermittelt (Zeile 384) und die aktuell eingestellte isometrische Höhe (*atPlain*) halbiert, konvertiert und der Position hinzuaddiert (Zeile 385).

Im Anschluss wird überprüft, ob es sich um die linke Maustaste handelt und welche der beiden Funktionen aktiv ist. Je nachdem, wird entweder ein neues Tile-Objekt an dieser Position platziert (Zeile 389) oder ein bereits vorhandenes gelöscht (Zeile 394).

Entspricht also *functionSelection* dem Wert 1, so handelt es sich um die Platzieren-Funktion und die Methode *CreateTile()* wird ausgeführt (siehe Listing 47).

```

405     bool CreateTile(Vector3 pos)
406     {
407         TileDesigner targetTile = levelDesigner.GetTileBySpot(pos);

```

Listing 47: Das bereits existierende Tile-Objekt an dieser Position zwischenspeichern

Zu Beginn der *CreateTile()*-Methode wird das *TileDesigner*-Objekt des bereits an dieser Position liegenden Tile-Objekts zwischengespeichert (siehe Listing 48).

```

79  public TileDesigner GetTileBySpot(Vector3 spot)
80  {
81      foreach (Transform child in transform.Find("Foreground").transform)
82      {
83          if (child.localPosition == spot)
84              return child.GetComponent("TileDesigner") as TileDesigner;
85      }
86
87      return null;
88  }

```

Listing 48: Vergleich aller Positionen der bereits platzierten Tile-Objekte mit der Zielposition

Um das Tile-Objekt auf der Zielposition zu ermitteln, werden die Positionen aller platzierten Tile-Objekte verglichen. Stimmt eine Position mit der Zielposition überein, wird das zugehörige *TileDesigner*-Objekt der *CreateTile()*-Methode zurückgegeben (Zeile 84) und die Variable *targetTile* gesetzt.

Wurde an der Zielposition ein Tile-Objekt gefunden und der Überschreibungsmodus ist deaktiviert, gibt die *CreateTile()*-Methode *false* zurück und das Tile-Objekt wird weder erzeugt noch platziert (siehe Listing 49).

```

409      if (targetTile != null && !levelDesigner.override_Tile)
410      {
411          return false;
412      }

```

Listing 49: Keine erfolgreiche Platzierung

Ein aktivierter Überschreibmodus sorgt dagegen dafür, dass der Sprite-Name des gefundenen Tile-Objekts, mit dem Sprite-Namen des zu platzierenden Tile-Objekts verglichen wird (siehe Listing 50).

```

414      if (targetTile != null && levelDesigner.override_Tile)
415      {
416          String oldTile = targetTile.SpriteRenderer.sprite.name;
417          String newTile = levelDesigner.currentTileObject.GetComponent<TileDesigner>()
418                          .SpriteRenderer.sprite.name;
419
420          if (oldTile.Equals(newTile))
421              return false;
422
423          DeleteTile(pos);
424      }

```

Listing 50: Vergleich der Sprite-Texturen

Sollten beide Namen identisch sein, wird hier ebenfalls *false* zurückgegeben und der Platziervorgang wird abgebrochen. Falls sich die Namen jedoch unterscheiden, wird das alte Tile-Objekt mittels *DeleteTile()*-Methode gelöscht (siehe Listing 51).

```

446  |   bool DeleteTile(Vector3 tilePos)
447  |   {
448  |       TileDesigner targetTile = levelDesigner.GetTileBySpot(tilePos);
449  |
450  |       if (targetTile != null)
451  |       {
452  |           DestroyImmediate(targetTile.gameObject);
453  |           return true;
454  |       }
455  |
456  |       return false;
457  |   }

```

Listing 51: Funktion zum Löschen eines Tile-Objekts

Die *DeleteTile()*-Funktion ist an dieser Stelle dieselbe, wie die bereits in Listing 36 gezeigte Methode und speichert, genau wie zu Beginn der *CreateTile()*-Methode, das *TileDesigner*-Objekt, mit Hilfe der *GetTileBySpot()*-Methode, zwischen (Zeile 448).

Sollte hier ein Tile-Objekt gefunden werden, wird dieses mit der *DestroyImmediate()*-Funktion gelöscht (Zeile 452).

Sobald die Zielposition nicht mehr belegt ist, muss schließlich nur noch das neue Tile-Objekt erzeugt werden (siehe Listing 52).

```

429  |   GameObject newGO = (GameObject)Instantiate(levelDesigner.currentTileObject.gameObject);
430  |   newGO.transform.position = pos;
431  |
432  |   Transform foregroundObject = levelDesigner.transform.Find("Foreground");
433  |
434  |   if (!foregroundObject)
435  |   {
436  |       foregroundObject = new GameObject("Foreground").transform;
437  |       foregroundObject.SetParent(levelDesigner.transform, false);
438  |   }
439  |
440  |   newGO.name = levelDesigner.currentTileObject.name;
441  |   newGO.transform.SetParent(foregroundObject, false);
442  |   newGO.GetComponent<TileDesigner>().height = levelDesigner.atPlain * 0.5f;

```

Listing 52: Erzeugen des neuen Tile-Objekts

Als erstes wird das neue *GameObject*, mittels des zugewiesenen Tile-Objekts (*currentTileObject*), instanziiert und die aktuelle Zellenposition wird übergeben (Zeile 429 und 430). Anschließend wird das *Foreground*--Spielobjekt über die *Find()*-Methode geholt (Zeile 432). Da diese Methode das *Foreground*-Objekt als *Transform*

zurückgibt, wird dementsprechend auch ein Transform-Objekt erzeugt. Im Großen und Ganzen gibt es zwischen Transform und *GameObject* keinen wesentlichen Unterschied, beide enthalten grundsätzlich die gleichen Daten des Objekts. Wurde kein *Foreground*-Spielobjekt gefunden, so wird zunächst eines erstellt und dem *Level*-Spielobjekt zugewiesen. Abschließend wird dem Tile-Objekt ein Name gegeben (Zeile 440), das *Foreground*-Objekt als Parent eingestellt (Zeile 441) und die entsprechende Höhe zugewiesen (Zeile 442).

4.3.2 Bewegung und Animation

Für die Implementierung des Bewegungssystems muss zunächst einmal dem Tile-Objekt, welches als Spielfigur fungieren soll, die Komponente *Rigidbody* hinzugefügt werden (siehe Abbildung 51).

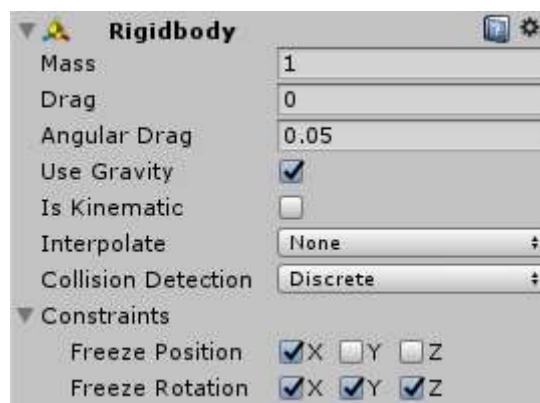


Abbildung 51: Die Rigidbody-Komponente

Diese *Rigidbody*-Komponente verleiht dem Objekt physikalische Eigenschaften. Beispielsweise kann festgelegt werden, wie groß die Masse (*Mass*) des Objektes ist und ob sich das Objekt von der Gravitation anziehen lässt (*Use Gravity*). Beide Parameter beeinflussen das Aufprallen des Objektes. Außerdem werden die Transform-Werte des Objekts bei Kollisionen mit anderen physischen Objekten verändert. Dies lässt sich zum Beispiel mit dem Parameter *is Kinematic*, der das Objekt zu einem starren Körper werden lässt, oder gezielt durch das Sperren der einzelnen Transform-Werte (*Freeze Position* und *Freeze Rotation*), unterbinden.

Damit die Richtung der Gravitation für die isometrische Perspektive korrekt funktioniert, muss der Richtungs-Vektor in den Projekteinstellungen von Unity richtig eingestellt werden (siehe Abbildung 52).

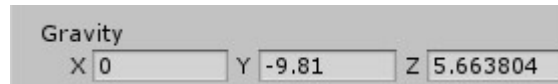


Abbildung 52: Richtung der Gravitation

Die y-Koordinate entspricht dabei weiterhin dem Ursprungswert. Lediglich die z-Koordinate muss, mittels des bereits verwendeten Faktors, angepasst werden ($z = y * 1/\sqrt{3}$).

Nachfolgend wird ein Script für das Abfragen der Bewegungstasten vorgestellt (siehe Listing 53).

```

3 public class PlayerController : MonoBehaviour {
4
5     const float ISOSCALEZ = 1.73205f;
6
7     public float speed = 5;
8     public float jumpForce = 250;
9
10    private bool jump = false;

```

Listing 53: Klassendefinition für die Bewegungsabfrage

Hierbei handelt es sich wieder um ein normales *MonoBehaviour*-Script, das während der Laufzeit ausgeführt werden soll.

Zu Beginn wird eine Variable für die Bewegungsgeschwindigkeit und eine für die Sprungkraft der Spielfigur initialisiert. Beide werden als *public*-Variable deklariert, da sie jederzeit im Inspector anpassbar sein sollen. Außerdem wird eine *boolean*-Variable für den Sprungvorgang erzeugt, die in der *Update()*-Methode mittels der Sprungtaste (Leertaste) gesetzt wird (siehe Listing 54).

```

12 void Update()
13 {
14     if (Input.GetKeyDown(KeyCode.Space) && !jump)
15     {
16         jump = true;
17     }
18 }

```

Listing 54: Abfrage der Sprungtaste

Im Anschluss wird in der *FixedUpdate()*-Methode auf das *Rigidbody* zugegriffen und eine nach oben gerichtete Kraft hinzugefügt, die dafür sorgt, dass die Spielfigur springt (siehe Listing 55).

```

20 void FixedUpdate ()
21 {
22     if (jump)
23     {
24         Vector3 force = new Vector3(0, jumpForce, jumpForce * -ISOSCALEZ / 3);
25         GetComponent<Rigidbody>().AddForce(force);
26         jump = false;
27     }

```

Listing 55: Hinzufügen der Sprungkraft zum Rigidbody

Die *Update()*-Methode wird ein Mal pro gezeichneten Frame aufgerufen. Da Frames mal schneller oder langsamer gezeichnet werden, können die Aufrufzeiten variieren. Die *FixedUpdate()*-Methode wird stattdessen in einem festen Intervall aufgerufen und berechnet außerdem die Physik. Für eine korrekte Berechnung finden daher auch die Zugriffe auf die *Rigidbody*-Komponente in der *FixedUpdate()*-Methode statt. Präzise Abfragen werden stattdessen in der *Update()*-Methode aufgerufen.

Lediglich die Richtungstasten können ebenfalls in der *FixedUpdate()*-Methode abgefangen werden, da diese sowieso meist ständig gedrückt gehalten werden und somit eine präzise Abfrage überflüssig ist (siehe Listing 56).

```

29     if (Input.GetKey(KeyCode.DownArrow))
30     {
31         transform.Translate((new Vector3(-1, -0.5f, -ISOSCALEZ/2) * speed) * Time.deltaTime);
32     }
33
34     else if (Input.GetKey(KeyCode.UpArrow))
35     {
36         transform.Translate((new Vector3(1, 0.5f, ISOSCALEZ/2) * speed) * Time.deltaTime);
37     }
38
39     else if (Input.GetKey(KeyCode.LeftArrow))
40     {
41         transform.Translate((new Vector3(-1, 0.5f, ISOSCALEZ/2) * speed) * Time.deltaTime);
42     }
43
44     else if (Input.GetKey(KeyCode.RightArrow))
45     {
46         transform.Translate((new Vector3(1, -0.5f, -ISOSCALEZ/2) * speed) * Time.deltaTime);
47     }

```

Listing 56: Abfragen der Richtungstasten

Für die Umsetzung der Bewegung wird schließlich auf die *Translate()*-Funktion zugegriffen. Die Richtung wird hier als Parameter übergeben und die Bewegungsgeschwindigkeit (*speed*) hinzumultipliziert. *Time.deltaTime* gibt dabei die Zeitdifferenz zum letzten gezeichneten Frame an und wird zusätzlich hinzumultipliziert, um einen frame-unabhängigen Wert zu erhalten [22].

Die Spielfigur kann nun im Spielfeld bewegt werden. Jetzt muss nur noch eine entsprechende Animation abgespielt werden, je nachdem in welche Richtung sich die Spielfigur bewegt.

Dafür wird ein Spritesheet mit den erforderlichen Frames für die Animationen benötigt. Umso mehr Frames ein Sprite dabei hat, umso flüssiger und realistischer sieht im Endeffekt die Bewegung aus. In der folgenden Abbildung 53 wird ein einfaches Spritesheet für die Laufanimationen in vier Richtungen gezeigt.

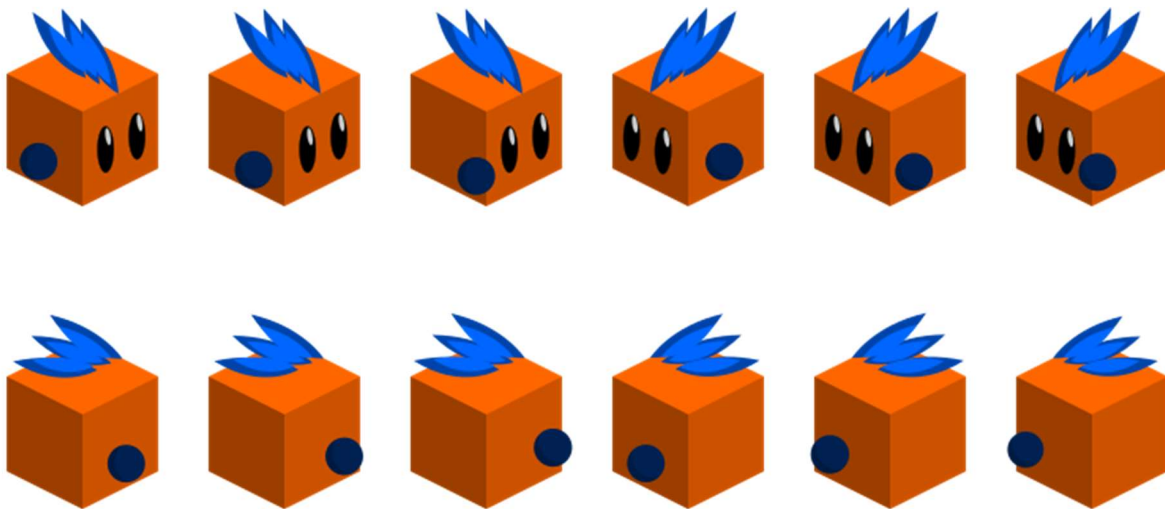


Abbildung 53: Spritesheet für die Laufanimationen der Spielfigur

Das Spritesheet stellt dabei eine stark vereinfachte Form einer Laufanimation dar. Normalerweise werden für solche Bewegungsabläufe weitaus mehr Frames verwendet (mindestens 8 Frames pro Laufrichtung). Zur Veranschaulichung ist die vereinfachte Form jedoch ausreichend.

Um nun aus den einzelnen Sprites (Frames) eine Animation zu erzeugen, bietet Unity das Animation-Tool an, mit denen man Animation-Clips für Spielobjekte anlegen kann. Um einen Animation-Clip zu erstellen, müssen die entsprechenden

Sprites in einer Zeitachse platziert und die Geschwindigkeit (*Sample*), in der die Sprites durchgeschaltet werden sollen, eingestellt werden (siehe Abbildung 54).

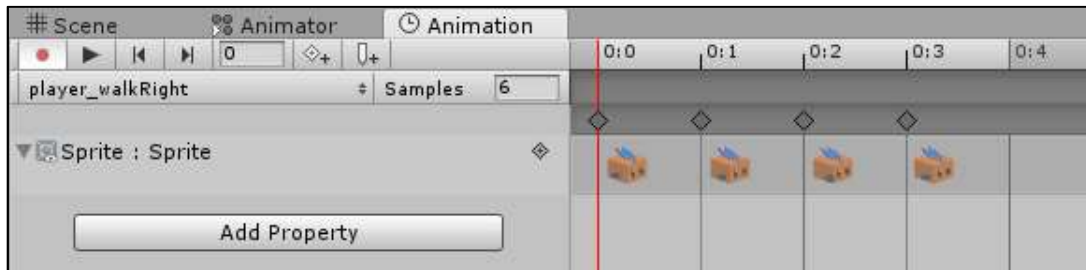


Abbildung 54: Animieren mit dem Animation-Tool

Das Animation-Tool bietet weitere Funktionen. Zum einen lassen sich Methoden zu jedem Zeitpunkt der Animation mittels *Event*-Button aufrufen. Zum anderen lassen sich Sprites manipulieren und verändern. Für das Ändern von Größe oder Farbe in einer Animation muss die Größe bzw. Farbe des ersten und des letzten Frames angegeben werden, und schon berechnet das Tool die zugehörigen Werte der dazwischenliegenden Frames, so dass eine flüssige Animation entstehen kann.

Auf diese Weise muss für die vier Bewegungsrichtungen jeweils ein Animation-Clip erstellt werden. Zusätzlich werden vier weitere Clips benötigt, in denen die Spielfigur auf der Stelle steht. Das bedeutet, dort wird jeweils nur ein Frame benötigt. Der Spielfigur stehen dann also insgesamt acht Animationen zur Verfügung, die nun zum richtigen Zeitpunkt ausgelöst werden müssen.

Hierfür bietet Unity ein weiteres nützliches Werkzeug an – den Animator. Dieser stellt eine Übersicht über alle Animationen, die auch als Zustände betrachtet werden können, eines Spielobjekts dar und erlaubt es diese miteinander zu verknüpfen (Transition). Verknüpfte Animationen laufen nacheinander ab, das bedeutet, wenn die erste Animation (A) mit einer zweiten Animation (B) verknüpft wird, dann wird die zweite Animation (B) gestartet, sobald alle Frames der ersten Animation (A) durchgelaufen sind (siehe Abbildung 55).

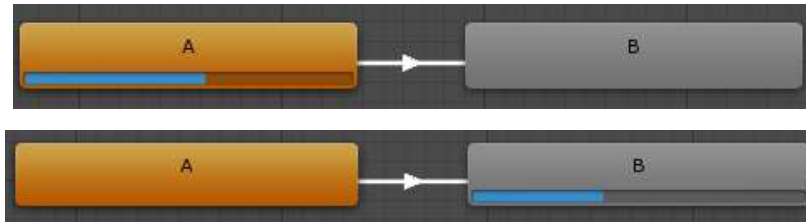


Abbildung 55: Beispiel einer Animationsverknüpfung

Solche Verknüpfungen können auch mit Bedingungen realisiert werden, wodurch sich Animationen unter anderem auch mehrmals verknüpfen lassen. In der folgenden Abbildung 56 wurden alle Animationen der Spielfigur miteinander verknüpft.

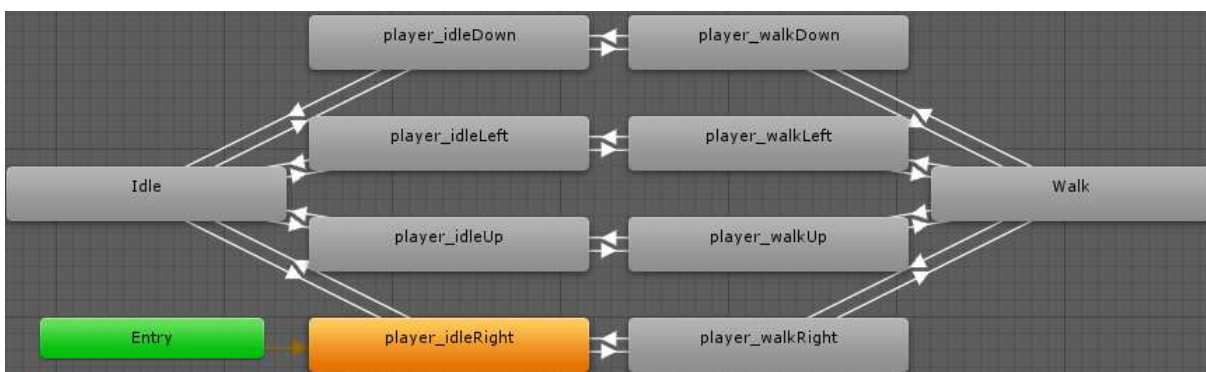


Abbildung 56: Verknüpfung der Animationen der Spielfigur

Entry stellt dabei einen sogenannten Leerlauf-Zustand dar. Dieser zeigt lediglich auf den Startzustand (*player_idleRight*). Die Zustände *Idle* und *Walk* sind ebenso Leerlauf-Zustände. Sie sind dafür da, dass sich die jeweiligen Idle- bzw. Walk-Zustände untereinander erreichen können. Jeder Idle-Zustand kann jedoch nur zu dem zugehörigen Walk-Zustand und umgekehrt wechseln.

Die Bedingungsparameter, die für die Zustandswechsel verantwortlich sind, sind einfache *Boolean*-Werte. Andere Datentypen werden jedoch auch unterstützt, wie zum Beispiel *Integer* oder *Float*.

In der nachstehenden Tabelle 7 werden die Verknüpfungen und die erforderlichen Bedingungen anhand der Idle-Zustände dargestellt.

Zustand	Verknüpft mit	Bedingung		
Idle	<i>player_idleRight</i>	Right	→	false
	<i>player_idleLeft</i>	Left	→	false
	<i>player_idleUp</i>	Up	→	false
	<i>player_idleDown</i>	Down	→	false
<i>player_idleRight</i>	Idle	Right	→	true
	<i>player_walkRight</i>	isWalk	→	false
<i>player_idleLeft</i>	Idle	Left	→	true
	<i>player_walkLeft</i>	isWalk	→	false
<i>player_idleUp</i>	Idle	Up	→	true
	<i>player_walkUp</i>	isWalk	→	false
<i>player_idleDown</i>	Idle	Down	→	true
	<i>player_walkDown</i>	isWalk	→	false

Tabelle 7: Bedingungen und Verknüpfungen der Idle-Zustände

Um diese Bedingungen aktualisieren zu können, muss ein weiteres Script der Spielfigur hinzugefügt werden (siehe Listing 57).

```

4 public class PlayerAnimationControl : MonoBehaviour {
5
6     public Animator anim;
7
8     ↳ Verweise
9     void Start () {
10         anim = GetComponentInChildren<Animator>();
11
12     }

```

Listing 57: Klassendefinition und Start des Scripts zum Aktualisieren der Animationsparameter

Zu Beginn wird in der *Start()*-Methode eine Referenz auf die Animator-Komponente der Spielfigur gesetzt, über die im Nachfolgenden auf die Bedingungsparameter der Animationen zugegriffen wird (siehe Listing 58).

```

14 void Update () {
15
16     if (Input.GetKey(KeyCode.DownArrow))
17     {
18         anim.SetBool("Down", true);
19         anim.SetBool("Up", false);
20         anim.SetBool("Left", false);
21         anim.SetBool("Right", false);
22
23         anim.SetBool("isWalk", true);
24     }

```

Listing 58: Setzen der Bedingungsparameter

Auch hier werden die Eingaben, wie beim Bewegungssystem, abgefragt und je nachdem welche Richtungstaste gedrückt wurde, die entsprechenden Bedingungsparameter aktualisiert. Sobald keine Taste mehr gedrückt wird, wird *isWalk* auf *false* gesetzt und die *Walk*-Zustände werden verlassen (siehe Listing 59).

```

56     if (!(Input.GetKey(KeyCode.DownArrow) ||
57         Input.GetKey(KeyCode.UpArrow) ||
58         Input.GetKey(KeyCode.LeftArrow) ||
59         Input.GetKey(KeyCode.RightArrow)))
60     {
61         anim.SetBool("isWalk", false);
62     }

```

Listing 59: Setzen des Walk-Parameters bei keiner betätigten Taste

4.3.3 Audio

Wie in den Anforderungen beschrieben, soll das Spiel Hintergrundmusik und bei bestimmten Aktionen des Spielers Effektmusik abspielen. Dieser Abschnitt handelt von der Nutzung der dafür erforderlichen Komponenten und zeigt beispielhaft die Implementierung. Auf die üblichen UML-Diagramme wird an dieser Stelle verzichtet, da für die Nutzung des Audiosystems im Rahmen dieser Arbeit nur bereits integrierte Bestandteile der UnityEngine genutzt werden. Diese Bestandteile werden nachfolgend kurz erläutert.

Audio Importer

Der Audio Importer übernimmt das importieren von Audiodaten. Dabei kann der Entwickler zwischen verschiedenen Optionen zur Regulierung der Qualität wählen, wie in Abbildung 57 zu sehen ist.

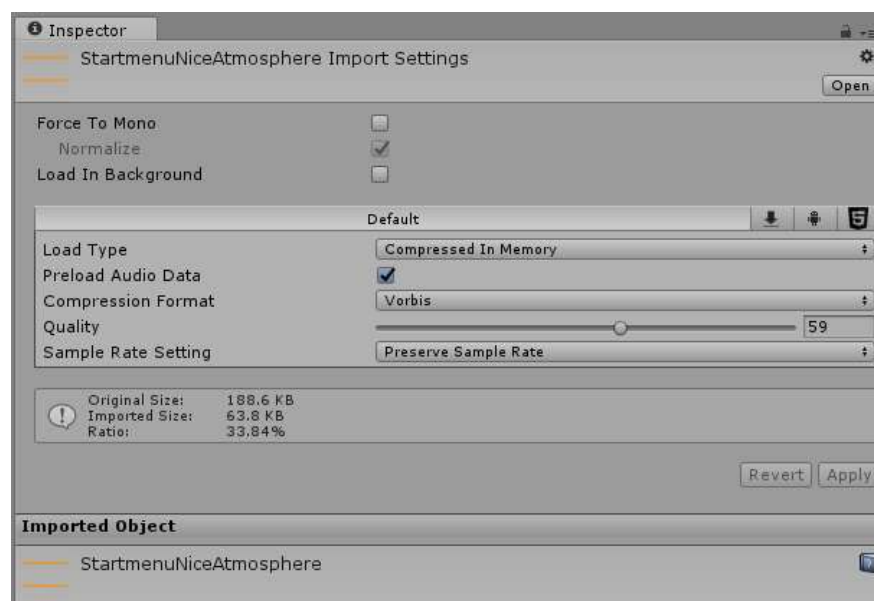


Abbildung 57: Optionen für das importieren von Audiodaten

Nachfolgend werden nur die wichtigsten Optionen erläutert.

Load Type

Mit Load Type kann der Entwickler einstellen, wie die Audiodaten geladen werden, denn die UnityEngine speichert sie komprimiert. Wenn sie genutzt werden sollen,

müssen sie dekomprimiert werden. Der Entwickler kann zwischen den einzelnen Optionen frei wählen. Für jeden Einsatzzweck einer Datei gibt es eine optimale Wahl. Es stehen drei Optionen zur Auswahl, welche folgende Möglichkeiten bieten [20]:

Decompress on Load: Hier wird die UnityEngine angewiesen, die Dateien beim Laden direkt zu dekomprimieren. Das ist ideal, wenn Speicherplatz gespart werden soll, da sie ja erst bei direkter Nutzung in den Speicher geladen werden. Das Laden dauert hier allerdings länger.

Compressed in Memory: Sobald diese Option gewählt wird, werden die Audiodaten komprimiert in den Zwischenspeicher geladen. Sobald sie genutzt werden sollen, werden die Daten dekomprimiert und sind einsatzbereit. Die Daten liegen jederzeit im Zwischenspeicher. Im Vergleich zur *Decompress on Load*-Option geht das Laden der Daten schneller vonstatten. Allerdings wird mehr Arbeitsspeicher benötigt.

Streaming: Die Streaming-Option ermöglicht, dass die Audiodaten on-the-fly dekomprimiert werden. Daraus ergibt sich eine kurze Ladezeit und wenig Speicherverbrauch, allerdings geht damit auch eine hohe Rechenlast einher.

Im Rahmen dieser Arbeit werden nur relativ kurze und damit wenig speicherintensive Daten genutzt. Daher wird für die benutzten Beispieldaten die Option *Compressed in Memory* gewählt. In einer späteren Version müssen für die verschiedenen Musikteile eventuell andere Optionen gewählt werden, da etwa mobile Geräte nicht über große Mengen Arbeitsspeicher verfügen.

Compression Format

Das *Compression Format* bestimmt das Format, mit welchem die Daten komprimiert werden. Je nach Format verändert sich die Qualität der Audiodatei.

PCM: PCM ermöglicht eine hohe Qualität, auf Kosten größerer Dateien.

ADPCM: Diese Option ist zu wählen, wenn es um Audiodaten geht, die von sich aus ein Grundrauschen mitbringen, etwa Schritte oder Schüsse. Der Vorteil dieser Kompression ist die niedrige CPU-Last.

Vorbis: Die Kompressionsrate ist einstellbar und ermöglicht somit eine individuelle Qualität.

Für die genutzten Audiodaten wird das Vorbis Kompressionsformat gewählt, welches einen guten Kompromiss zwischen Qualität und Dateigröße darstellt.

Sample Rate Setting

Diese Option gibt an, wie „fein aufgelöst die Audiodatei bezüglich des Frequenzspektrums aufgenommen wurde“ [23]. Hier kann der Entwickler eine hohe Frequenz angeben, sofern die Audiodatei mit der richtigen Profiausrüstung aufgenommen wurde. Im Rahmen dieser Arbeit ist dies nicht der Fall, daher bleibt diese Option unangetastet.

Damit wurden die wesentlichen Optionen für das Importieren von Audiodateien diskutiert. Die Optionen können entsprechend genutzt werden.

Audio Source

Mit der Audio Source-Komponente kann eine Audiodatei innerhalb der aktuellen Szene abgespielt werden. Auch hier kann der Entwickler mit diversen Optionen das Verhalten der Komponente ändern. Die Möglichkeiten sind in Abbildung 58 zu sehen.

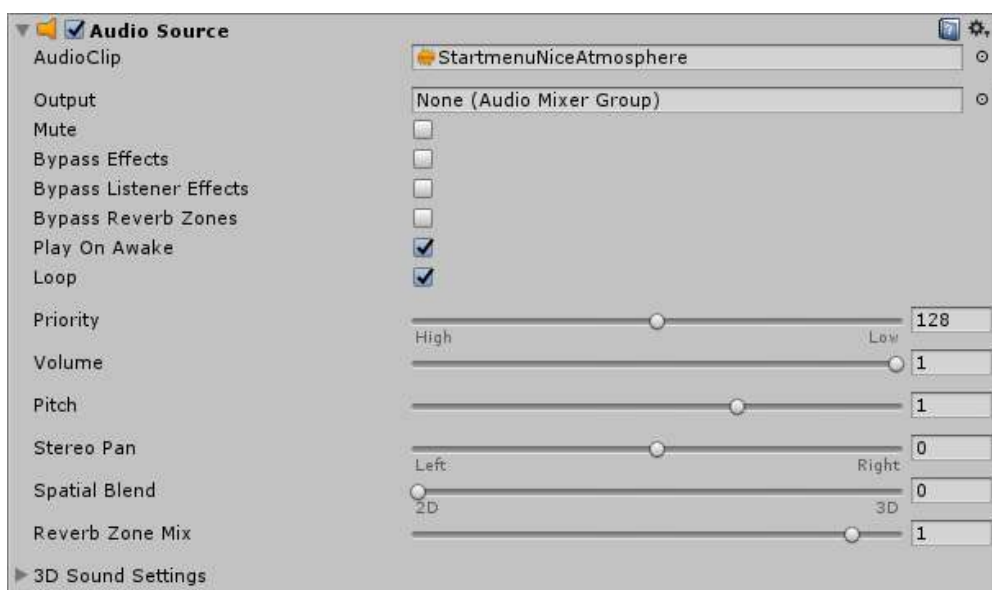


Abbildung 58: Einstellungsmöglichkeiten der Audio Source-Komponente

Auch an dieser Stelle werden wieder nur die wichtigsten und im Rahmen dieser Arbeit benötigten Optionen erläutert.

AudioClip

Hier kann der Entwickler festlegen, welche Audiodatei abgespielt werden soll. Diese Audiodatei muss vorher mit dem *Audio Importer* importiert werden.

Play On Awake

Gibt an, ob die Audiodatei gestartet werden soll, sobald das Spielobjekt, an welchem die *Audio Source*-Komponente angeheftet wurde, aktiv wird.

Loop

Sofern die *Loop*-Option aktiv ist, wird die gewählte Audiodatei unendlich oft abgespielt, wenn der Entwickler die *Audio Source*-Komponente nicht zerstört. Für Hintergrundmusik wird die Option aktiv geschaltet, bei einzelnen Effekten wäre dies allerdings hinderlich und damit wird die Wiederholung dort abgeschaltet.

Spatial Blend

Die *Spatial Blend*-Option bestimmt, wie sehr die 3D Engine die Audiodatei beeinflusst. In einer 3D Welt müssen bestimmte Effekte räumlich wahrnehmbar sein. Da die in dieser Arbeit erstellte Welt allerdings in 2D entwickelt wird, wird die Option auf 2D geschaltet.

Damit sind alle benötigten Komponenten und Optionen erläutert und bereit zur Benutzung. Der nachfolgende Abschnitt beschreibt schließlich die Nutzung für das Spiel und die Implementierung.

Hintergrundmusik und Effekte

Für die Hintergrundmusik des Spiels wird die *Audio Source*-Komponente genutzt. Diese wird an ein leeres Spielobjekt angehängt und die Audiodatei wird spezifiziert. Da die Hintergrundmusik dauerhaft laufen soll, wird die Option **Loop** aktiv geschaltet. Somit ist die Komponente fertig gestellt. Für die Effekte, welche bei bestimmten Aktionen des Spielers abgespielt werden sollen, hat der Entwickler diverse Optionen.

Zum einen kann eine weitere *Audio Source*-Komponente erstellt werden, bei welcher die Option **Play On Awake** deaktiviert ist. Anschließend kann das Event, bei welchem die Datei abgespielt werden soll, die Audiodatei starten. Eine weitere Möglichkeit besteht darin, ohne eine separate *Audio Source*-Komponente zu arbeiten. Dabei wird eine Audiodatei über den Quellcode abgespielt, wie Listing 60 zeigt:

```
349     if (navigateInUI)
350     {
351         AudioSource.PlayClipAtPoint(
352             Resources.Load<AudioClip>("Audio/ApplePickup-01Green_2"),
353             transform.position);
354         tokenWithFocus.elementWithFocus.submit();
355     }
```

Listing 60: Abspielen einer Audiodatei mittels Code

Wie ab Zeile 351 zu sehen ist, wird die *AudioSource*-Klasse angewiesen, eine Audiodatei abzuspielen. Diese Audiodatei wird aus den Ressourcen des Projektes geladen und dann in der Szene platziert. In diesem Beispiel wird die Datei abgespielt, wenn der Spieler die „Akzeptieren“-Taste betätigt. Somit gibt es für jede bestätigte Aktion des Spielers ein akustisches Feedback. Hiermit ist die beispielhafte Implementation der Audio-Komponenten fertig gestellt.

4.3.4 Zwischenstand des Spiels – die Spielwelt

Mit den vorherig erläuterten Komponenten kann die Spielwelt geformt und visualisiert werden. Dazu wird, wie in Abbildung 59 zu sehen ist, mittels des Level Designers zum Start ein leerer Untergrund aus grünen Blöcken erstellt. Diese Blöcke stellen den Boden dar, auf dem sich der Spieler bewegt.



Abbildung 59: Untergrund des Beispiellevels

Nachdem der Untergrund erstellt wurde, können diverse Elemente auf dem Spielfeld platziert werden. Beispielhaft wird an dieser Stelle ein Block platziert, welcher eine Art Mauer für den Spieler darstellt. Die Spielfigur kann sich nur um diesen Block herumbewegen. Zusätzlich wird ein Baum platziert, um zu zeigen, wie sich Objekte mit einer Höhe von mehreren Tiles auf das Level auswirken. Abbildung 60 zeigt, wie der Baum im Level-Designer ausgewählt wird.

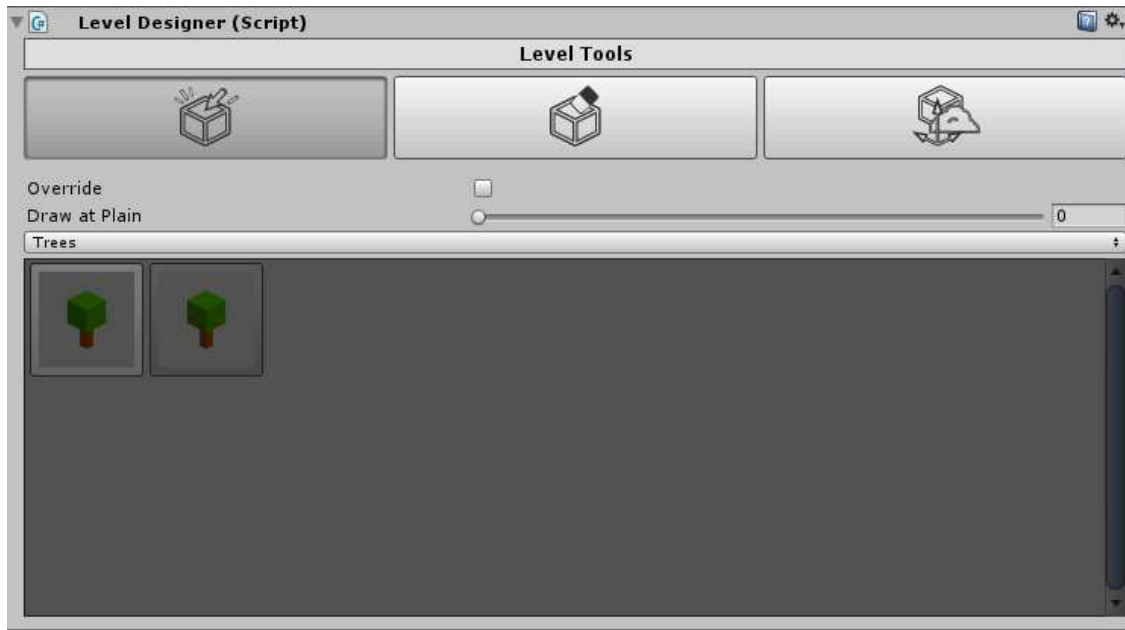


Abbildung 60: Auswählen eines Baums im Level Designer

In obiger Abbildung ist zusätzlich zu sehen, dass der Baum auf Ebene Null des Spielfeldes platziert werden soll. Die *Override*-Option ist deaktiviert, also werden eventuell schon vorhandene Objekte nicht überschrieben. Sobald der Baum ausgewählt ist, kann er auf der Karte platziert werden (siehe Abbildung 61).

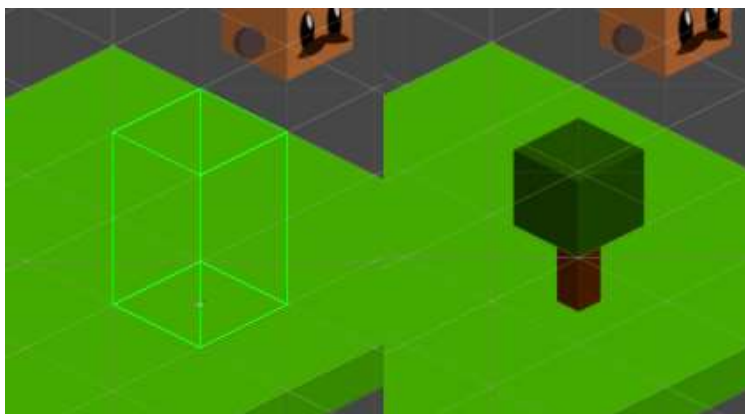


Abbildung 61: Platzieren des Baumes

Wie bereits erwähnt hat der Baum eine Höhe von 2, andere Objekte könnten also verdeckt werden, wenn sie sich hinter dem Baum befinden. Um diese Mechanik zu verdeutlichen, wurde beispielhaft ein blauer Block so positioniert, dass dieser zumindest teilweise verdeckt ist, wie in Abbildung 62 zu sehen ist.

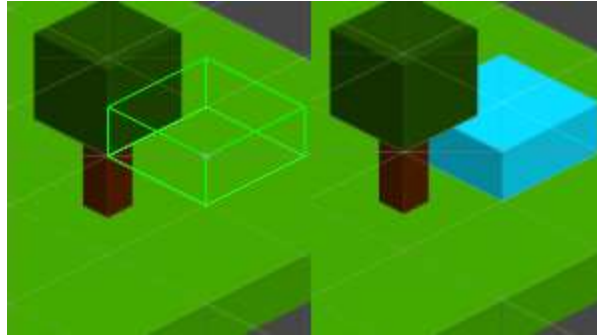


Abbildung 62: Platzieren eines (teilweise) verdeckten Objektes

Dieser Block ist ein unüberwindbares Hindernis für den Spieler. Auch um den Baum muss der Spieler herumlaufen. Der Spieler kann also keines der beiden Objekte überqueren. Abbildung 63 zeigt das Beispiellevel während der Ausführung, mitsamt der Spielfigur, die vom Spieler gesteuert wird.

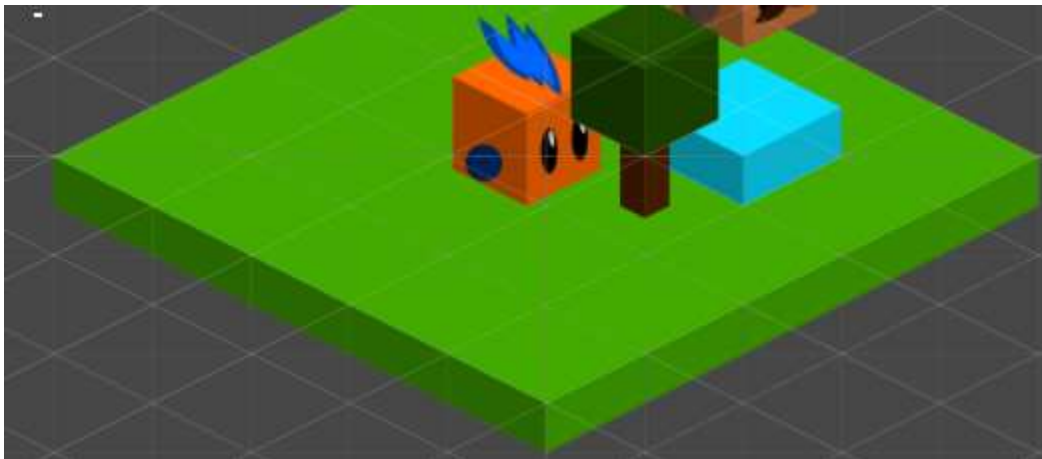


Abbildung 63: Ausführen des Beispiellevels

Die Spielfigur kann sich nicht, wie bereits erwähnt, durch oder über die platzierten Objekte bewegen. Sie wird daher durch den implementierten Collider beziehungsweise von der Physik-Engine von Unity ein Stück zurückgeworfen. Die Implementierung und Visualisierung des Levels ist hiermit abgeschlossen.

4.4 UI-Systeme

4.4.1 Aufgabensystem

Das Aufgabensystem soll den Spieler durch die Spielwelt leiten. Den Anforderungen nach soll ein Aufgabensystem inklusive GUI-Oberfläche implementiert werden, wo etwa die offenen und erledigten Aufgaben sowie deren Details zu sehen sind.

Anwendungsfalldiagramm

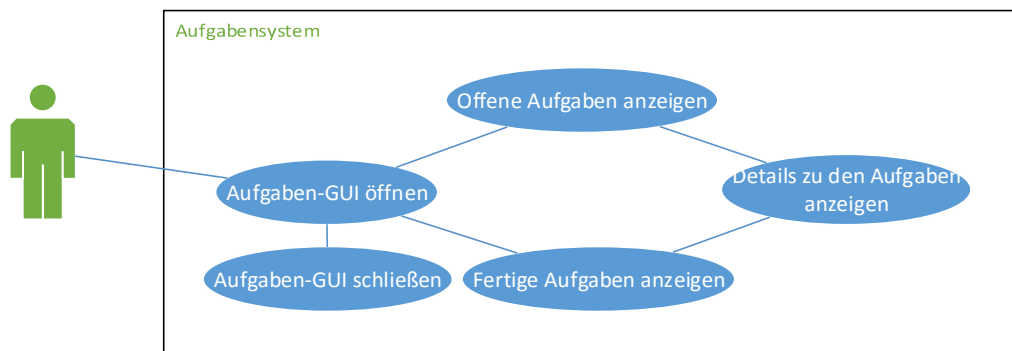


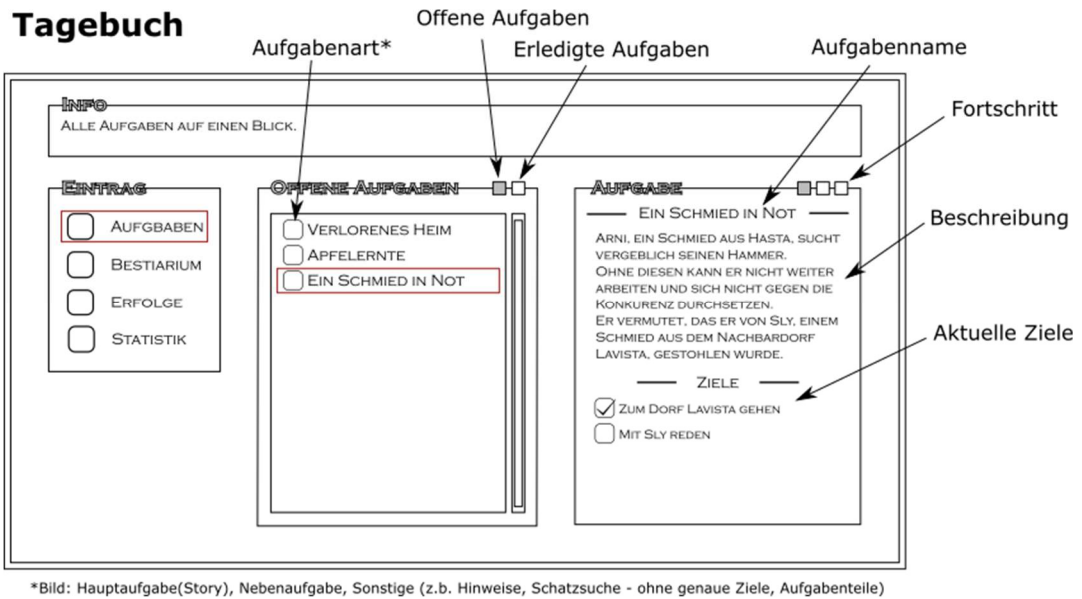
Abbildung 64: Anwendungsfälle des Aufgabensystems

Wie im Anwendungsfalldiagramm in Abbildung 64 zu sehen ist, soll der Spieler das Aufgaben-GUI öffnen und von dort aus weitere Aktionen ausführen können. Er kann etwa zwischen offenen und fertigen Aufgaben wechseln, sich Details zu einzelnen Aufgaben (etwa Beschreibung und Ziele) anzeigen lassen oder die Oberfläche wieder schließen. Aus diesen Anwendungsfällen wurde der folgende UI-Entwurf erstellt.

UI-Entwurf

Der UI-Entwurf in Abbildung 65 zeigt, welche Daten dem Spieler im UI angezeigt werden. Dabei wurden die Anforderungen an das Aufgabensystem berücksichtigt und eingearbeitet.

So kann der Spieler zwischen den Aufgaben wechseln oder die Details zu einer Aufgabe einsehen.

Abbildung 65: Entwurf des Aufgaben-UIs⁵

Zusätzlich wurden weitere Menüpunkte eingearbeitet, welche im Falle der Weiterentwicklung des Spiels implementiert werden können.

Komponentendiagramm

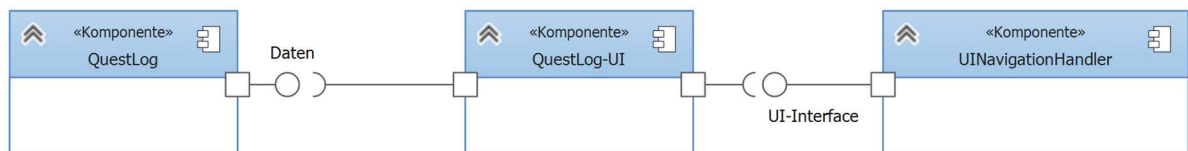


Abbildung 66: Komponenten des Aufgabensystems

Das Aufgabensystem nutzt dabei die *UINavigationController*-Komponente, damit der User mit dem GUI interagieren kann. Die Daten, welche durch das Aufgabensystem-UI (*QuestLog-UI*) angezeigt werden, kommen von der *QuestLog*-Komponente des Spielers.

⁵ Ein Bestiarium ist ein Katalog von Tieren oder Geschöpfen. Im Zusammenhang mit diesem Spiel sind natürlich statt der Tiere Mozo im Bestiarium enthalten.

Klassendiagramm

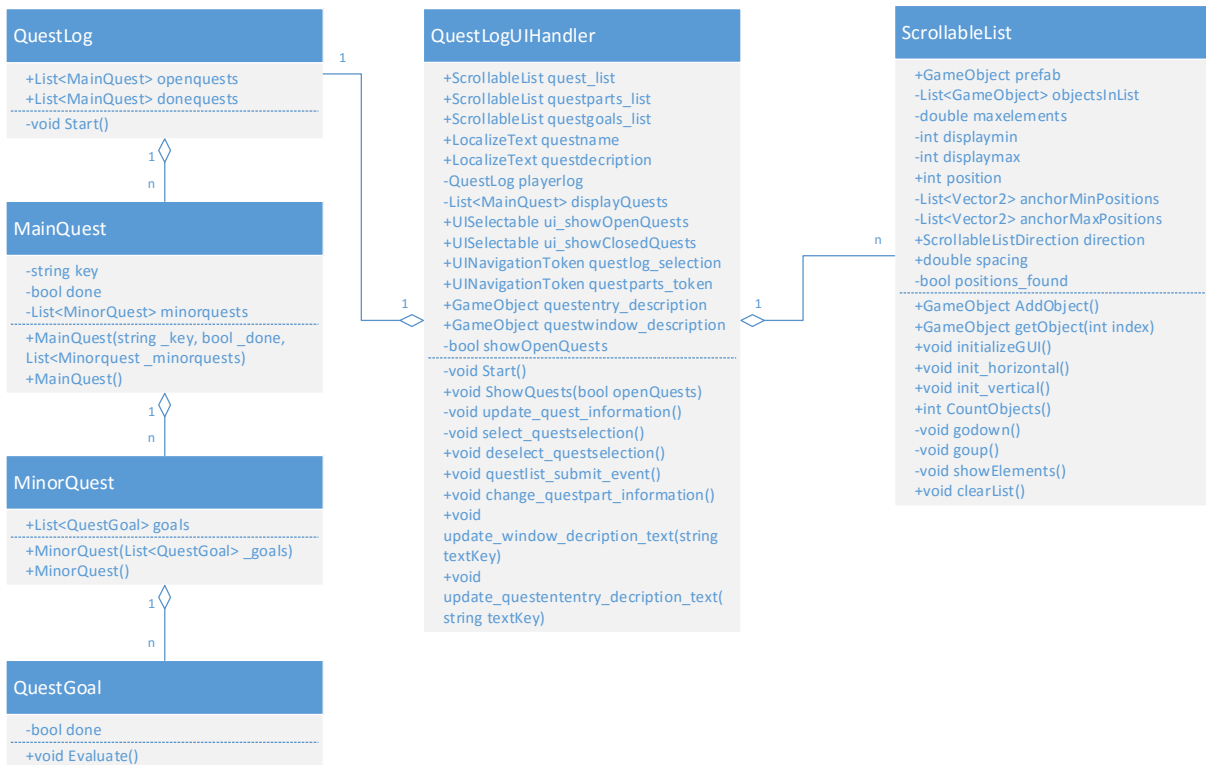


Abbildung 67: Klassen des Aufgabensystems und ihre Zusammenhänge

Das Klassendiagramm in Abbildung 67 zeigt, wie die entworfenen Klassen des Aufgabensystems zusammenhängen. Die *QuestLog*-Klasse beispielsweise setzt sich aus mehreren *MainQuests* (Hauptaufgaben) zusammen. Die *MainQuests* ihrerseits bestehen aus mehreren *MinorQuests* (Aufgabenteilen). Diese Aufgabenteile haben dann eine Liste von *QuestGoals* (Zielen). Der *QuestLogUIHandler* bezieht Daten aus dem *QuestLog* des Spielers und zeigt sie an. Wie im UI-Entwurf auf Seite 78 zu sehen ist, werden für die grafische Umsetzung mehrere Listen benötigt. Der Spieler soll mit diesen Listen interagieren können. Er soll Objekte auswählen und Aktionen durchführen können. Dazu wurde die Klasse *ScrollableList* entworfen, welche eine solche Funktionalität bereitstellt. Nachfolgend wird die Implementation der einzelnen Klassen beschrieben, ausgehend vom kleinsten Glied der Kette – dem Quest-Ziel.

Implementierung

QuestGoal / Quest-Ziel

Das Quest-Ziel ist eine der Grundklassen des Aufgabensystems. Dabei ist die eigentliche Klasse nur eine Vorlage: Es gibt kein allgemeines Quest-Ziel, da der Spieler viele unterschiedliche Typen von Zielen verfolgen soll. So soll er bei einer Aufgabe etwa mit einem NPC reden, bei einer anderen soll er einen bestimmten Gegenstand im Inventar haben. Dazu implementiert die Klasse die virtuelle Funktion *Evaluate()*. Diese Funktion muss von Klassen, die von *QuestGoal* erben, implementiert beziehungsweise überschrieben werden. Dabei hat die überschriebene Version der Methode Vorrang vor der eigentlichen Implementation. Wie der Name bereits andeutet, kümmert sich diese Funktion um die Überprüfung des Status der Aufgabe. Soll der Spieler einen Gegenstand sammeln, so prüft die *Evaluate()*-Methode einer fiktiven Klasse *ItemGoal*, ob sich dieser Gegenstand im Inventar des Spielers befindet. So kann eine breite Varianz an Zielen bereitgestellt werden, ohne dass eine Klasse, die die Quest-Statistiken aktualisiert, wissen muss, welche unterschiedlichen Zielarten existieren.

MinorQuest – MainQuest – QuestLog

Der *MinorQuest* – also ein Aufgabenteil – besteht aus einer Liste von Zielen, die der Spieler erledigen muss. Der Haupt-Quest besteht aus mehreren *MinorQuests* und der *QuestLog* seinerseits aus mehreren Haupt/Main-Quests. Diese Klassen werden trivial implementiert. Daher wird an dieser Stelle auf ein Listing verzichtet. Ein Beispiel für eine Aufgabe könnte folgendermaßen aussehen:

- Der Haupt-Quest mit dem Namen „Ein Schmied in Not“ hat zwei Aufgabenteile.
- Im ersten Aufgabenteil soll der Spieler mit einem NPC reden. Diese Aufgabe hat zwei Ziele:
 - o Zu einem Dorf gehen, um den NPC zu finden
 - o Mit dem NPC reden
- Der zweite Aufgabenteil könnte sein, einen bestimmten Gegenstand zu suchen.

Der *QuestLog* baut auf diesem System auf. Er hält eine Liste mit aktiven und fertigen Aufgaben vor, so dass das UI-System diese Aufgaben anzeigen kann.

ScrollableList

Die zweite Hauptkomponente, welche von dem UI-System des Aufgabensystems genutzt wird, ist die *ScrollableList*-Klasse. Wie bereits beim Klassendiagramm beschrieben, soll diese Klasse eine Liste bereitstellen, welche selektierbare Elemente enthält und es erlaubt, zwischen diesen Elementen zu wechseln.

Dazu bietet die Klasse unter anderem folgende Funktionalitäten an:

- Der Liste ein Objekt hinzufügen (öffentlich),
- die Liste initialisieren (öffentlich),
- die Elemente (neu) zeichnen (nicht öffentlich),
- alle Listenobjekte entfernen (öffentlich).

Um der Liste ein Objekt hinzuzufügen, wurde die Methode *addObject()* implementiert. Das hinzugefügte Objekt hat keinen speziellen Typ, sondern wird aus einem Prefab geladen. Das Prefab muss vor dem Benutzen der Liste angegeben werden. Nachdem alle Objekte zur Liste hinzugefügt wurden, kann sie initialisiert werden.

Die Liste initialisieren

Die öffentliche Methode zum Initialisieren liest nur aus, welche Richtung für die Liste angegeben ist. Ist die *ScrollableList* beispielsweise als eine vertikale Liste initialisiert, wird die interne Methode zum Initialisieren einer vertikalen Liste aufgerufen. In analoger Weise wird bei einer horizontalen Liste verfahren. Die beiden Methoden unterscheiden sich nur in der Berechnung der Positionen.

In diesem Abschnitt wird auf die Implementierung der vertikalen *ScrollableList* eingegangen. An erster Stelle wird die maximale Anzahl von Elementen berechnet, die die Liste gleichzeitig anzeigen kann. Dazu wird die Höhe des ersten Prefabs der Liste (Objekt 1) genutzt (siehe Abbildung 68).

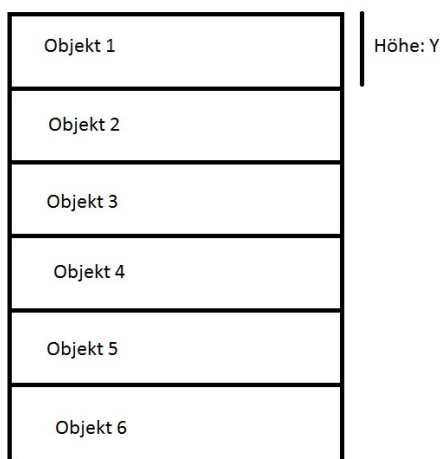


Abbildung 68: Beispielhafte Liste mit Objekten

Folgende Formel wurde zur Berechnung der maximalen Elemente genutzt:

$$\text{Maximale Elemente} = \frac{1}{1 - \text{Abstand des ersten Elements von der Grundlinie}}$$

Formel 4: Berechnung der maximalen Anzahl der Elemente

Wenn in der UnityEngine Elemente ausgerichtet werden, so erhalten sie immer eine Höhe in Relation zur Grundlinie. Die Grundlinie der Y-Achse liegt immer bei $Y = 1$. Ein Element mit einer Höhe von 0,15 würde also von Position 0,85 anfangen und bei $Y = 1$ enden. Ausgehend von diesem Beispiel passen sechs Elemente in die Liste. Danach wird geprüft, ob es überhaupt genug Objekte gibt, um die Liste komplett zu füllen. Sind etwa sieben Objekte in der Objektliste, aber die Liste kann nur sechs Objekte anzeigen, dann werden maximal sechs Elemente angezeigt. Sind allerdings nur fünf Objekte in der Objektliste und sechs könnten angezeigt werden, werden dennoch nur fünf angezeigt.

Nachdem die Rahmenbedingungen bekannt sind, werden die Detailberechnungen vorgenommen. Unter anderem wird jedem Listenobjekt eine *UISelectable*-Komponente (siehe Abschnitt 4.2.2) angehängt. So kann der Spieler Elemente in der Liste markieren. Des Weiteren werden Listener hinzugefügt, die ausgelöst werden, wenn der Spieler in der Liste ein Element nach unten beziehungsweise nach oben geht. Schließlich werden die Positionen, an denen später die UI-Objekte in der Liste platziert werden, berechnet.

Die Implementierung der Berechnung ist in Listing 61 zu sehen:

```

165 |         anchorMinPositions.Add (new Vector2(0,(1 - (float)spacing - ((i + 1)
166 |             * (1 - objectsInList[0].GetComponent<RectTransform>().anchorMin.y
167 |             ))));
168 |         anchorMaxPositions.Add
169 |             (new Vector2(1, anchorMinPositions[i - 1].y - (float)spacing));

```

Listing 61: Berechnung der Positionen

Dabei berechnet sich die Position eines Elements immer ausgehend von der Position des letzten Elementes. Bei der Y-Minimalposition (Endposition) wird n Mal die Höhe des ersten Elements herangezogen. Die X-Minimalposition ist bei einer horizontalen Liste immer 0. Die Maximalposition ist dagegen immer 1. Die Y-Max.-Position (Startposition) ist die Endposition des letzten Elementes. Abbildung 69 zeigt die Positionen und deren Bezeichnungen.

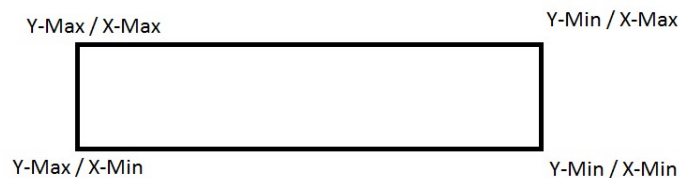


Abbildung 69: Positionen bei der Anordnung eines Elementes

Die Maximalpositionen sind die Startpositionen der jeweiligen Achsen, die Minimalpositionen die Endpositionen. Damit sind die Positionen, an denen die Elemente gezeichnet werden sollen, berechnet.

Die Elemente (neu) zeichnen

Es werden nur die Elemente neu gezeichnet, die auch in der Liste sichtbar sind. Dazu werden die Attribute *displaymin* und *displaymax* genutzt, die beim erstmaligen Start beide Null sind. Wenn der Spieler später in der Liste scrollt, werden diese Variablen geändert. Eine Beschreibung der Funktionsweise folgt auf der nächsten Seite. Listing 62 zeigt, wie nur die Elemente gezeichnet werden, die die Liste darstellen kann.

Die Elemente, welche sich im Auswahlbereich befinden (etwa Element null bis Element sechs) werden aktiviert, wie in Zeile 343 zu sehen ist.


```

339     int z = 0;
340
341     for (int i = displaymin; i < displaymax; i++)
342     {
343         objectsInList[i].SetActive(true);
344
345         //DE: Positionierung der Anker.
346         if (i > 0)
347         {
348             objectsInList[i].GetComponent<RectTransform>().anchorMin
349                 = anchorMinPositions[z];
350             objectsInList[i].GetComponent<RectTransform>().anchorMax
351                 = anchorMaxPositions[z];
352         }
353
354         //DE: Positionierung des Objektes.
355         objectsInList[i].GetComponent<RectTransform>().offsetMin = new Vector2(0, 0);
356         objectsInList[i].GetComponent<RectTransform>().offsetMax = new Vector2(0, 0);
357         z = z + 1;
358     }

```

Listing 62: Zeichnen der Elemente

Die nicht aktivierten Elemente werden nicht gezeichnet. Danach werden die vorher gespeicherten Positionen an die jeweiligen Elemente vergeben. So wird in diesem Beispiel Position Null an Element Null vergeben, wie in Zeile 348 bis 351 zu sehen ist.

Wenn der Spieler in der Liste die Elemente wechselt, werden die jeweiligen Methoden (nach-oben- oder nach-unten-Methode) ausgeführt. Diese Methoden berechnen abhängig von der aktuellen Position und der Anzahl der Elemente in der Liste, ob die Parameter der Liste neu berechnet werden müssen. Listing 63 zeigt diese Berechnung.

Zuerst wird geprüft, ob es ein Element nach der aktuellen Position gibt. Sollte dies der Fall sein, wird die aktuelle Position erhöht.

```

268     private void godown()
269     {
270         //DE: Wenn es noch ein Element nach der aktuellen Position gibt,
271         //dann erhöhe die Position
272         if (position + 1 < objectsInList.Count)
273         {
274             position = position + 1;
275         }
276
277         //DE: Wenn es nicht das letzte Objekt der Liste ist
278         //und die Liste verschoben werden muss,
279         //dann berechne die minimalen und maximalen Grenzen neu
280         if ((objectsInList.Count - 1 > position))
281         {
282             if (position.Equals(displaymax - 1))
283             {
284                 displaymax = displaymax + 1;
285                 displaymin = displaymin + 1;
286
287                 //DE: Elemente werden neu gezeichnet
288                 showElements();
289             }
290         }
291     }
292 }

```

Listing 63: Neuberechnung der Parameter

Wenn die Position dann noch niedriger ist als die maximale Anzahl an Elementen in der Liste, dann werden die Parameter neu berechnet. Zum besseren Verständnis werden an dieser Stelle zwei Beispiele angeführt:

Beispiel 1 – Das aktuelle Element ist das letzte Element in der Liste:

- In der Objektliste sind fünf Elemente.
- Die aktuelle Position ist fünf.

Wenn der Spieler jetzt nach unten geht, also ein Element nach dem aktuellen anzeigen möchte, passiert nichts, da kein weiteres Element vorhanden ist.

Beispiel 2 – Das aktuelle Element ist nicht das letzte Element in der Liste:

- In der Objektliste sind zehn Elemente.
- Die aktuelle Position ist fünf.

Wenn dieser Fall eintritt, wird zusätzlich geprüft, ob die aktuelle Position dem Ende der dargestellten Liste entspricht. Werden etwa aktuell Element null bis sechs angezeigt, dann entspricht das dem Ende der dargestellten Liste. Eigentlich wäre theoretisch noch ein Element Platz, allerdings scrollt es sich angenehmer, wenn schon ein Element vor dem letzten das nächste angezeigt wird. Ist das Ende der Liste also erreicht, so werden die Parameter der Liste *displaymin* und *displaymax* erhöht. Für Beispiel 2 wären die neuen Parameter also *displaymin* = 1 und *displaymax* = 7. Das erste Element der Objektliste wird jetzt nicht mehr angezeigt, dafür ist das Siebte hinzugekommen. Für das nach-oben-Scrollen ist diese Funktionalität dieselbe, nur, dass die Parameter hier um 1 verringert werden, solange der „Platz“ da ist.

Zusätzlich zu diesen Funktionen wurden Methoden implementiert, um auf die Liste zuzugreifen, beispielsweise eine *Count()*-Methode zum Zählen der Elemente oder eine *Clear()*-Methode zum Löschen jener. Damit ist die *ScrollableList* hinsichtlich der Funktionalität fertig und kann genutzt werden. Wie zu sehen ist, nimmt sie einen Großteil der Implementierungsarbeit ein, erspart aber bei Wiederverwendung viel Arbeit, da auch im restlichen Spiel des Öfftens Listen eingesetzt werden.

QuestLogUIHandler

Das UI-System des Aufgabensystems nutzt die eingeführten Komponenten. Die *ScrollableList* beispielsweise wird für die Anzeige der Quest-Liste, der einzelnen Quest-Teile und der Quest-Ziele genutzt. Eine erste vorläufige UI-Implementation in Unity ergibt das in Abbildung 70 zu sehende Erscheinungsbild: Dabei wurden erneut Panels, Labels und Toggles zum Aufbau des UI-Systems genutzt. Für die Lokalisation der Texte wurde erneut die Lokalisierungs-Komponente aus Abschnitt 4.2.4 genutzt. Das UI-System wurde so angepasst, dass es, genau wie alle nachfolgenden UIs, immer 20% Abstand zum Bildschirmrand aufweist. Dadurch kann das UI mit hohen oder geringen Auflösungen umgehen.

Das UI-Grundgerüst enthält noch keine Aufgaben. Diese werden erst durch den UI-Handler ergänzt, wie nachfolgend zu lesen ist.

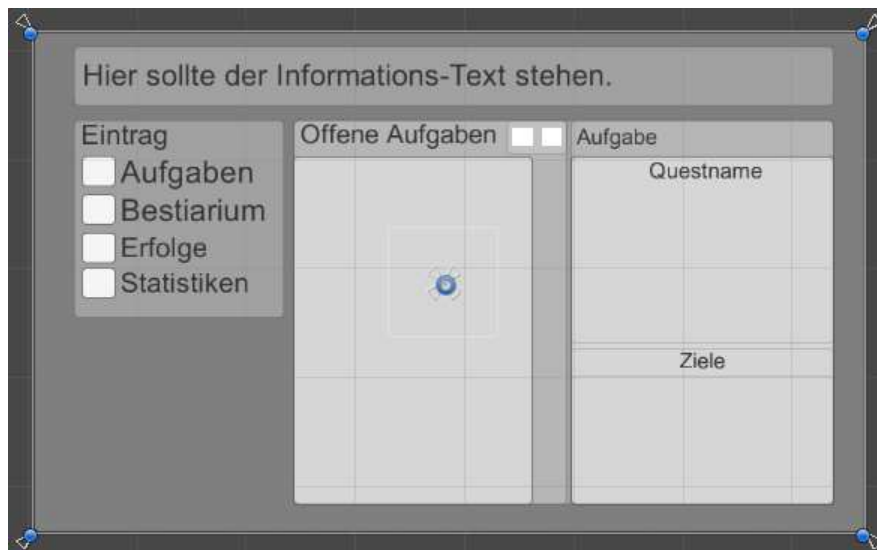


Abbildung 70: UI-Anordnung des Aufgabensystems

Die Hauptarbeit des UI-Handlers besteht darin, die einzelnen Quest-UI-Elemente zu erzeugen und diese mit zusätzlichen Listenern zu versehen. Die Aufgabenobjekte in der Aufgabenliste müssen die Details und die Anzahl der Aufgabenteile aktualisieren. Die Auswahl der Aufgabenteile muss ebenfalls wieder die Details eines Quest aktualisieren. Zum Start des Handlers werden die *QuestLog*-Komponente des Spielers und damit auch seine Aufgaben geladen. Danach wird die initiale Quest-Liste angezeigt. Der Ablauf gestaltet sich dabei folgendermaßen:

1. Prüfen, ob der Spieler Aufgaben hat
2. Der Aufgabenanzahl entsprechend neue Objekte in der *ScrollableList* der Aufgabenliste erstellen
3. *ScrollableList* initialisieren
4. Listener an die UI-Objekte anhängen
5. Aufgabendetails aktualisieren

Dieser Ablauf verhält sich abgeändert auch für die beiden anderen Listen (Aufgabenteile, Zielliste) ähnlich, so dass an dieser Stelle nur die Implementierung der Aufgabenliste aufgezeigt wird.

1. Prüfen, ob der Spieler Aufgaben hat

Listing 64 zeigt, wie diese Prüfung abläuft. Wenn der Parameter *openQuests*, welcher der UI-Methode übergeben wurde, den Wert „*true*“ annimmt, dann sollen die offenen Aufgaben gezeigt werden, wenn der Wert „*false*“ ist, entsprechend die fertigen Aufgaben.

```

98      //DE: Es wird geprüft, welche Quests angezeigt werden sollen.
99      if (openQuests)
100     {
101         displayQuests = playerlog.Openquests;
102         update_questentry_description_text("QuestEntrypanelDescription");
103     }
104     else if(!openQuests)
105     {
106         displayQuests = playerlog.Donequests;
107         update_questentry_description_text("QuestEntrypanelDescriptionClosed");
108     }

```

Listing 64: Prüfen des QuestLogs des Spielers

Dabei wird zusätzlich noch die Beschreibung, welche Kategorie aktuell gesetzt ist, aktualisiert. Zu sehen ist das in den Zeilen 102 beziehungsweise 107.

2. Der Aufgabenanzahl entsprechend neue Objekte in der *ScrollableList* der Aufgabenliste erstellen

Nachdem herausgefunden wurde, welche Aufgaben angezeigt werden sollen, können die UI-Objekte für die Aufgabenliste erzeugt werden. Diese Objekte werden, wie von der *ScrollableList* verlangt, aus einem Prefab geladen. Das Prefab besteht aus einer Checkbox und einem Namen.

3. *ScrollableList* initialisieren

Nachdem die *ScrollableList*-Komponente alle Objekte entgegengenommen hat, muss sie initialisiert werden. Das ist nötig, da die *ScrollableList* selbst ebenfalls Listener an die einzelnen Listenobjekte anhängt. Die Listener, die der UI-Handler anhängt, sollen aber weniger hoch priorisiert sein, die eigenen Listener der Liste. Der Grund hierfür ist, dass es sonst aufgrund der vertauschten Aufruffreihenfolge zu Komplikationen kommen kann.

4. Listener an die UI-Objekte anhängen

Der UI-Handler muss folgende zusätzliche Listener anhängen:

- Beim Wechsel des Elementes nach unten oder oben müssen die entsprechenden Details für die Aufgabe aktualisiert werden.
- Bei einem Druck auf die Akzeptieren-Taste soll der Spieler zur Auswahl der Teilaufgaben geleitet werden.

Die Listener werden hinzugefügt, indem die Objekt-Liste der *ScrollableList* noch einmal durchlaufen wird. In diesem Durchlauf werden dann jedem Objekt die Listener angehängt, wie in Listing 65 zu sehen ist.

```

129     for (int b = 0; b < displayQuests.Count; b++)
130     {
131         UISelectable addedObject = prefab_liste.GetObject(b).
132             GetComponent<UISelectable>();
133         addedObject.ExecuteOnDown.
134             AddListener(delegate { update_quest_information(); });
135         addedObject.ExecuteOnUp.
136             AddListener(delegate { update_quest_information(); });
137
138         //Fügt die Submit-Events hinzu.
139         EventTrigger trigger = addedObject.gameObject.AddComponent<EventTrigger>();
140         EventTrigger.Entry entry = new EventTrigger.Entry();
141         entry.eventID = EventTriggerType.Submit;
142         entry.callback.AddListener(delegate { questpart_token.register(); });
143         trigger.triggers.Add(entry);
144     }

```

Listing 65: Anhängen der Listener an die UI-Objekte

Dabei wird in Zeile 131 zunächst das entsprechende Objekt aus der Liste geholt und seine *UISelectable*-Komponente geladen. Diese Komponente ist für das Vorhalten von Listenern zuständig, die bei Tastendrücken ausgelöst werden.

Danach werden den zugehörigen Events wie *ExecuteOnDown* Listener angehängt. Diese Listener erwarten eine Methode, die aufgerufen wird, wenn das Event ausgelöst wird. Beim Wechsel der Elemente nach oben oder unten wird die Methode *update_quest_information()* angehängt, welche die Aufgabendetails aktualisiert. Für das Bestätigen eines Elementes und den damit zusammenhängenden Wechsel zu den Aufgabenteilen wurde Unitys Event System genutzt, wie in den Zeilen 139 bis 143 zu sehen ist. Im Grunde wurde auch hier aber wieder nur eine Methode angehängt.

5. Aufgabedetails aktualisieren

Die Aufgabedetails, welche im UI-Entwurf in Abbildung 65 am rechten Rand zu sehen sind, enthalten die Auswahl der Aufgabenteile, eine Beschreibung für jeden Aufgabenteil und eine Liste von Zielen. Um diese Daten anzuzeigen, wurde wieder die *QuestLog*-Komponente des Spielers genutzt und die entsprechenden Aufgabedaten ausgelesen. Auch an dieser Stelle wurden erneut *ScrollableList*-Komponenten genutzt – der Ablauf ist also in etwa derselbe wie für die Anzeige der Aufgabenliste (siehe Listing 66).

```

163 void update_quest_information()
164 {
165     //DE: Entfernt alle eventuell vorhandenen Wechsel-UI-Elemente für die Teilquests
166     questparts_ui_list.clearlist();
167
168     //DE: Fügt die Wechsel-UI-Elemente für die Teilquests hinzu
169     for (int a = 0; (displayQuests.Count >= 1) &&
170         (a < displayQuests[prefab_liste.Position].Minorquests.Count) && (a <= 5); a++)
171     {
172         questparts_ui_list.AddObject();
173     }
174     questparts_ui_list.initializeGUI();
175
176     //DE: Fügt den Aufgabenteilen-UI-Elementen Listener hinzu.
177     for (int b = 0; b < questparts_ui_list.CountObjects(); b++)
178     {
179         UISelectable sel_object = questparts_ui_list.GetObject(b).
180             GetComponent<UISelectable>();
181         sel_object.ExecuteOnLeft.
182             AddListener(delegate { change_questpart_information(); });
183         sel_object.ExecuteOnRight.
184             AddListener(delegate { change_questpart_information(); });
185     }
186
187     //DE: Wenn es mindestens eine Teilaufgabe gibt,
188     //dann aktualisiere die Beschreibungen für diese.
189     if (displayQuests.Count >= 1) change_questpart_information();
190 }

```

Listing 66: Aktualisieren der Aufgaben-Details

Die UI-Objekte für die Auswahl der Teilaufgaben werden demzufolge erzeugt, indem der entsprechenden *ScrollableList* neue Objekte erstellt werden. Auch hier werden wieder Listener an die Objekte geheftet, um die Nutzereingaben zu verarbeiten. Bei der Auswahl der Teilaufgabe kann der Spieler dann mit einem Druck auf die Links- oder Rechts-Taste die aktuell ausgewählte Teilaufgabe ändern. Diese Aufgabe übernimmt die *change_questpart_information()*-Methode. In dieser Methode werden schließlich die Aufgabedetails angezeigt.

Exemplarisch für die anderen Daten zeigt Listing 67, wie die Beschreibung einer Aufgabe geändert wird.

```

201     questdescription.textKey = "QuestPartDescription"
202     + displayQuests[prefab_liste.Position].Key
203     + questparts_ui_list.Position;
204     questdescription.Start();

```

Listing 67: Anzeigen der Aufgabenbeschreibung

Alle Texte, die im Spiel vorkommen, werden in einer lokalen Übersetzungsdatei gespeichert. Wie dieses System funktioniert, kann gegebenenfalls in Abschnitt 4.2.4 nachgelesen werden. Der Schlüssel zur jeweiligen Übersetzung muss dynamisch anhand verschiedener Faktoren generiert werden. Feste Schlüssel wie „Aufgabename“ wären an dieser Stelle unnützlich, da es verschiedene Aufgaben gibt und jede einen eigenen Namen, eigene Beschreibungen und eigene Zieltexte besitzt. Der Schlüssel für die Beschreibung und auch für die anderen Aufgabendetails setzt sich dann folgendermaßen zusammen:

1. Aufgabendetail, welches angezeigt werden soll.
2. Schlüssel der Aufgabe.
3. Schlüssel der Teilaufgabe.

Ein valider Schlüssel ist etwa der folgende:

QuestPartDescriptionABlacksmithInNeeds0

Abbildung 71: Valider Aufgabenschlüssel

Dieser Schlüssel gehört zur Beschreibung der ersten Teilaufgabe der Aufgabe „Ein Schmied in Not“. Diesen Aufbau nutzen auch die anderen Komponenten zum dynamischen Anzeigen von Textinhalten.

Die Implementierung des Aufgabensystems ist hiermit abgeschlossen. Wie das Aufgabensystem während der Ausführung dargestellt wird, kann im Zwischenstands-Abschnitt 4.4.4 nachgelesen werden.

4.4.2 Inventar

Das Inventar dient zur Verwaltung von Gegenständen des Spielers. Dieser kann über das Inventar-UI, wie in den Anforderungen beschrieben, Gegenstände (*Items*) auswählen, sortieren, ablegen und natürlich auch benutzen. Ein Item im Inventar kann für den Spieler unterschiedliche Nutzungsmöglichkeiten bieten. Es soll zwei Grundkategorien geben:

- *Mozo-Items*, welche bestimmte Effekte auf Mozos haben. Hier gibt es die Unterkategorien „Heilung“, „Support“, „Upgrade“, „Offensiv“ und „Sonstiges“.
- *Quest-Items* werden für bestimmte Aufgaben benötigt, etwa wenn der Spieler einem anderen Charakter einen Gegenstand bringen muss.

Neben diesen Grundkategorien werden zusätzlich Unterkategorien benötigt, mit welchen sich die Items nach einem bestimmten Schema sortieren lassen. Im Rahmen dieser Arbeit werden nur Mozo-Items in Unterkategorien unterteilt, da diese essenziell für den Spielfortschritt und die Dynamik sind. Folgende Unterkategorien kann ein Mozo-Item annehmen:

- „Offensiv“ – ein Item, welches bestimmte Offensivattribute eines Mozo ändert,
- „Support“ – zur Verbesserung der Defensivattribute,
- „Heilung“ – um etwa ein Mozo wiederzubeleben oder verlorengegangene Lebenspunkte zu regenerieren,
- „Upgrade“ – um einem Mozo ein Plus an Erfahrung zu geben,
- „Sonstige“ – alle sonstigen Gegenstände.

Zunächst soll es zwei Schemata geben. Wenn das Kampfschema gewählt wird, werden die Gegenstände im Inventar folgendermaßen sortiert:

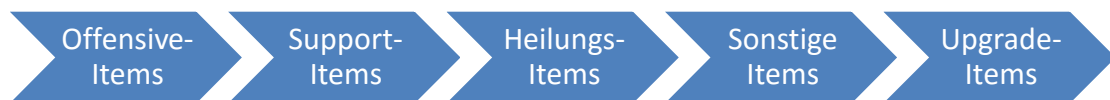


Abbildung 72: Kampfschema

Wie in Abbildung 72 zu sehen ist, werden die Offensiv-Gegenstände zuerst angezeigt, darauf folgen die Support-, Heilungs-, Sonstige- und zum Schluss die Upgrade-Gegenstände.

Wenn dagegen das Normalschema genutzt werden soll, ändert sich die Sortierreihenfolge.

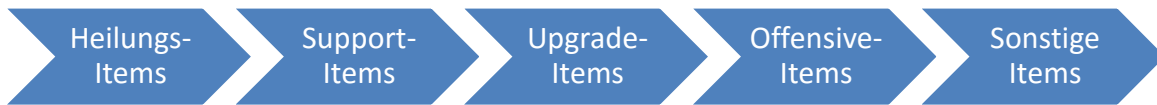


Abbildung 73: Normalschema

Abbildung 73 zeigt das Normalschema. Im Gegensatz zum Kampfschema werden hier die Gegenstände aus der Unterkategorie Heilung priorisiert dargestellt. Darauf folgen die Support-, Upgrade-, Offensiv-, und Sonstige-Gegenstände. Innerhalb der einzelnen Unterkategorien werden die Items dann alphabetisch von A bis Z sortiert.

Anwendungsfalldiagramm

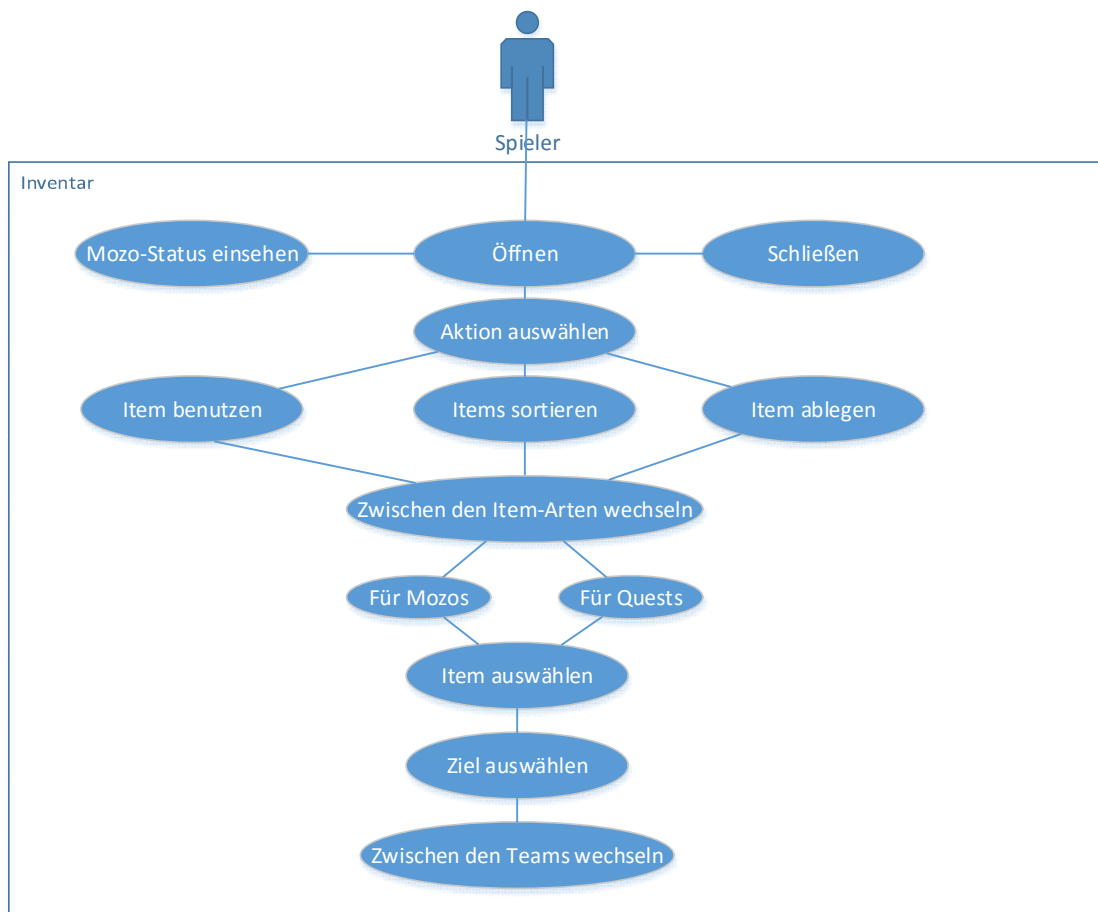


Abbildung 74: Anwendungsfalldiagramm des Inventars

Wie in Abbildung 74 zu sehen ist, verfügt der Spieler im Inventar-UI über weitreichende Handlungsmöglichkeiten. Zum einen können Aktionen ausgeführt werden, welche nicht von einem Item abhängig sind - etwa den Status der Mozos

einsehen oder das Inventar schließen. Zum anderen kann mit den Items auch interagiert werden. Der Spieler kann:

- Ein Item zum Benutzen auswählen.
- Die Items nach verschiedenen Kriterien sortieren. Dazu gehören die Sortierschemata „Manuell“, „Alphabetisch“, „Normal“ und „Kampf“.
- Ein Item zum Ablegen auswählen.

Nachdem der Spieler die Aktion ausgewählt hat, muss er das Item, welches er benutzen möchte, auswählen.



Abbildung 75: Ablauf für Aktionen im Inventar

Hier kann er zwischen den drei Item-Kategorien Mozo, Quest und Trophäen wählen. Hat der Spieler ein Item ausgewählt, muss er noch das Ziel für die gewählte Aktion bestimmen. Möchte er ein Item nutzen, dann muss er ein Mozo als Ziel auswählen.

Komponentendiagramm



Abbildung 76: Komponentendiagramm des Inventars

In Abbildung 76 ist das Komponentendiagramm des Inventars zu sehen. Es besteht aus drei Komponenten: Zum einen natürlich das Inventar, welches alle Daten über gesammelte Gegenstände des Spielers beinhaltet. Diese Daten werden mittels der Schnittstellen bereitgestellt, über die die Komponenten Item-Liste und Teams verfügen. Die Item-Liste stellt Daten zu den Items bereit, über welche der Spieler verfügt. Die Team-Komponente gibt Aufschluss über die Mozos des Spielers (für die Definition siehe Abschnitt 4.2.3).

Klassendiagramm der Daten-Komponenten

Das folgende Klassendiagramm zeigt jene Klassen, welche für die Vorhaltung und Speicherung der Daten notwendig sind. Die Klassen zum Anzeigen des Inventar-UIs werden separat im nächsten Abschnitt definiert.

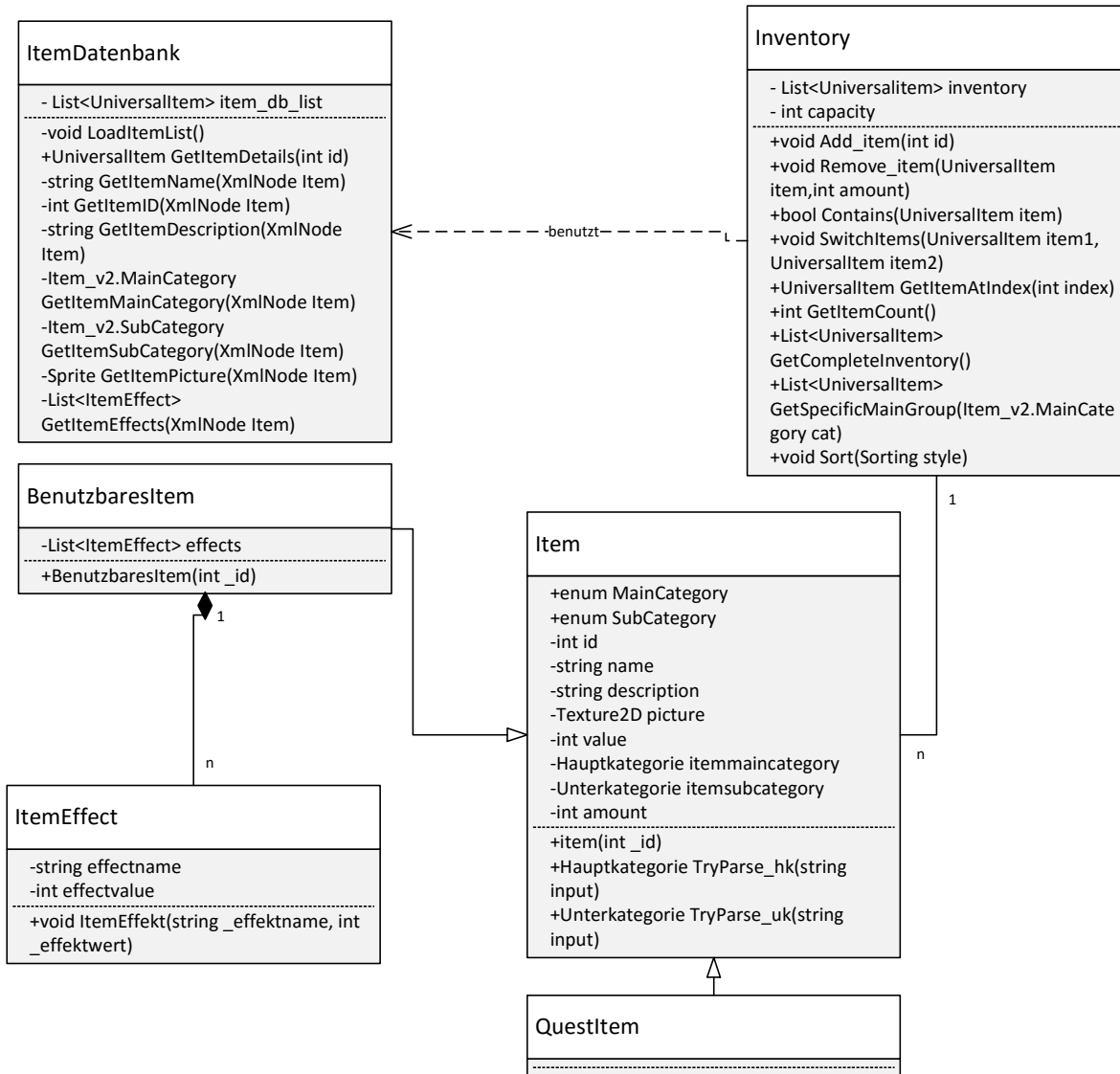


Abbildung 77: Klassendiagramm des Inventars

Abbildung 77 zeigt die Unterteilung des Inventars in die Klassen. Dabei dreht sich alles um die Basisklasse *Item*, von welcher die Klassen *BenutzbaresItem* und *QuestItem* erben. Das *Inventory* hält die Logik zum Hinzufügen beziehungsweise Entfernen des Inventars vor. Zusätzlich implementiert die Klasse weitere Methoden, die zum Verwalten des Inventars hilfreich sind.

Die Grundklasse *Item*, welche einen Gegenstand im Spiel repräsentiert, definiert sich durch folgende Attribute:

- ID: Die Identifikationsnummer des Items. Hierüber wird das Item im Item-Verzeichnis gefunden und die Details können geladen werden.
- Name: Der Anzeigename des Gegenstandes. Dieser Name wird dem Spieler im Inventar angezeigt.
- Description: Hier werden der Gegenstand und die Wirkung in wenigen Worten beschrieben.
- Picture: Ein Bild des Gegenstandes, welches im Inventar angezeigt wird.
- Value: Wenn der Spieler einen Gegenstand bei einem Händler verkaufen möchte, wird dieser Grundwert herangezogen.
- MainCategory/SubCategory: Die Kategorien des Gegenstandes.
- Amount: Die Anzahl des Gegenstandes im Inventar.

Diese Attribute können durch ihre öffentlich erreichbaren *Pendants* bearbeitet beziehungsweise ausgelesen werden. Das interne Feld „*id*“ kann durch das öffentlich erreichbare Feld „*id*“ bearbeitet werden. Natürlich bietet die Klasse einen Konstruktor, welcher ein neues Item initialisiert. Die Klasse implementiert das Interface *UniversalItem*, welches erwartet, dass bestimmte Methoden implementiert werden. Das Arbeiten mit den verschiedenen Item-Ablegern kann entsprechend vereinfacht werden, da alle Ableger über dieselben Methoden zum Auslesen einzelner Felder verfügen.

Das benutzbare Item (*UsableItem*) erbt von der Basisklasse *Item* und erweitert sie um eine Liste von Effekten. Diese Effekte haben Auswirkungen auf das Ziel, indem sie beispielsweise die Stärke des Spielers um 10 erhöhen. Jeder Effekt definiert sich durch einen Effektnamen, der mit jenem Attributnamen gleichzusetzen ist, welchen der Effekt verändert. Für „Stärke“ wäre der Effektnamen also wieder „Stärke“. Der Effektwert definiert die Höhe der Auswirkung, zum Beispiel 10. Jedes Item kann n verschiedene Effekte haben. Durch die Separierung von Item und benutzbarem Item können im weiteren Verlauf der Entwicklung noch weitere Kategorien von Items hinzugefügt werden, etwa nicht benutzbare Items wie Trophäen oder einzigartige Gegenstände. Die Klasse *QuestItem* implementiert keine neuen Felder. Sie dient

lediglich als Gegenpart zu den benutzbaren Items. QuestItems werden genutzt, wenn der Spieler für eine Aufgabe einen Gegenstand finden soll.

Die Klasse *Inventory* speichert die Gegenstände, die ein Spieler während seines Abenteuers sammelt. Dazu stellt sie Methoden zum Hinzufügen und Entfernen von Gegenständen bereit. Es wird auch eine Methode bereitgestellt, welche anzeigt, ob ein spezieller Gegenstand im Inventar vorhanden ist oder nicht. Intern arbeitet die Klasse mit einer Liste von Items und einem Wert, der anzeigt, wie viele Items der Spieler bei sich tragen kann. Jedes Inventar kann m verschiedene Items beinhalten.

UI-Entwurf

Basierend auf den vorgestellten Überlegungen wird der erste Entwurf für das User-Interface erstellt:

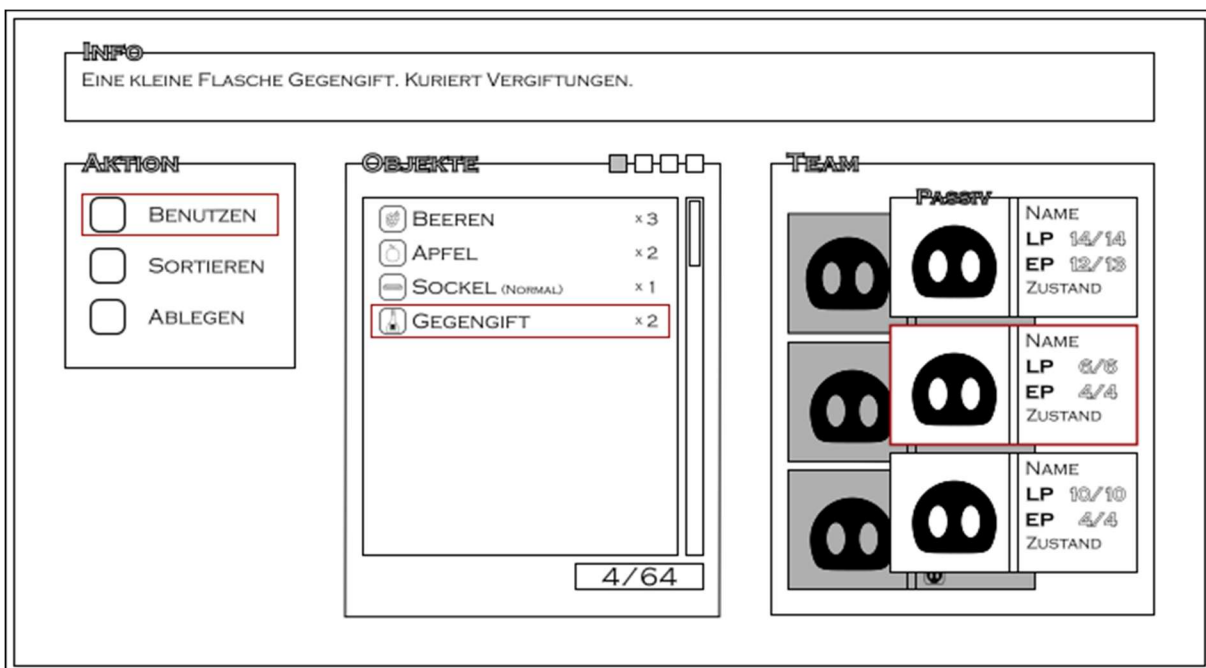


Abbildung 78: UI-Entwurf des Inventars

Wie in Abbildung 78 zu sehen ist, werden die verschiedenen Komponenten in einzelnen Panels dargestellt. Das Infopanel zieht sich über die gesamte Breite des Inventarfensters und zeigt die Beschreibung des ausgewählten Gegenstandes an. Unterhalb des Infopanel sind das Aktions-, das Objekte- und das Teampanel angeordnet. Im Aktionspanel kann der Spieler einen Gegenstand benutzen, das Inventar sortieren oder einen Gegenstand ablegen.

Das Objektpanel gibt Aufschluss über die Gegenstände im Inventar des Spielers. Diese Gegenstände werden mit Bild, Name und Anzahl untereinander aufgelistet. Unterhalb dieser Liste wird aufgeführt, wie viel Platz der Spieler in seinem Inventar belegt hat. Im Teampanel werden Kennwerte der Mozos des Spielers angezeigt. Es werden der Name, die Lebenspunkte (Lifepoints – LP), die Energiepunkte (Energypoints – EP) und der Zustand dargestellt. Des Weiteren kann im Teampanel zwischen den Teams gewechselt werden.

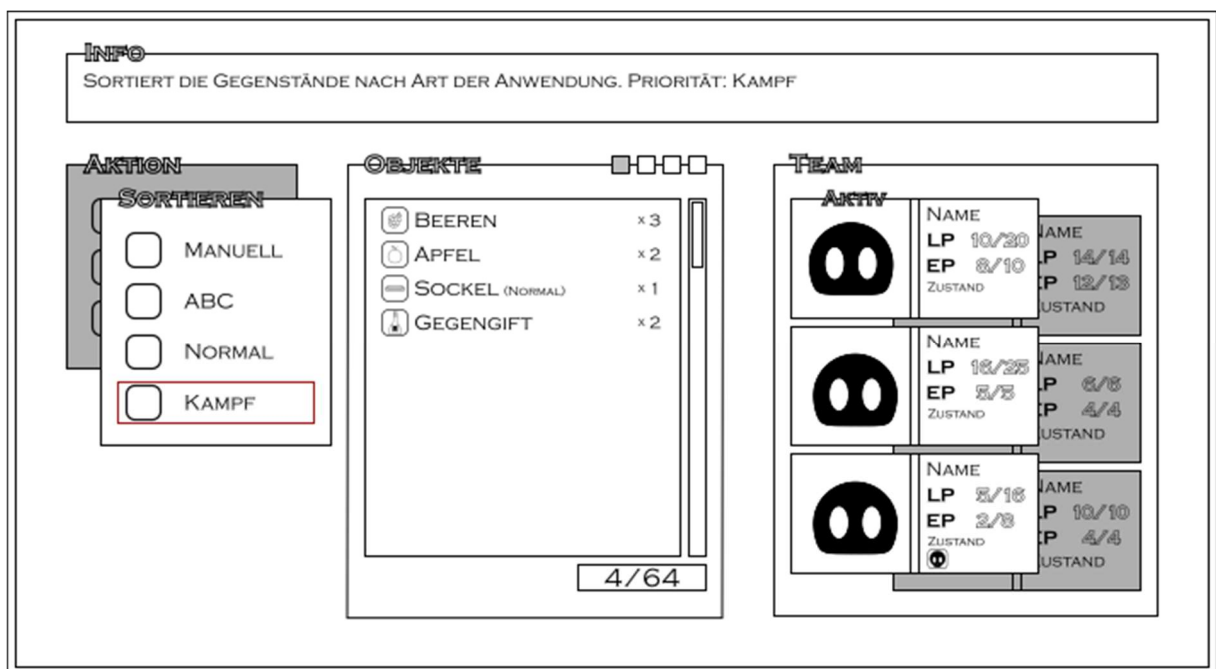


Abbildung 79: Inventar beim Sortieren

Abbildung 79 zeigt schließlich die Sortierschemata, die der Spieler auswählen kann.

Implementierung der Klassen

Item

Die Klasse Item ist eine der Basisklassen des Inventars. Alle benötigten Felder wurden gekapselt, so dass auf sie von außen nur über definierte Schnittstellen zugegriffen werden kann. Initialisiert wird diese Klasse durch einen Konstruktor, welcher nur die ID des Items als zwingenden Parameter erwartet.

Benutzbares Item

Das benutzbare Item stellt eine spezielle Art von Items dar. Wie der Name bereits suggeriert, kann dieses Item vom Spieler benutzt werden. Das Item besitzt dazu eine Liste von Effekten, welche die zu verändernden Attribute gespeichert hat. Die Klasse *ItemEffekt* wird trivial definiert - inklusive Konstruktoren und Zugriffsmethoden, daher wird die Implementierung hier nicht erläutert.

Die Grundklassen, auf denen das Inventar basiert sind somit fertiggestellt. Als nächstes muss die Item-Datenbank definiert werden.

Item-Datenbank

Die Item-Datenbank hält Daten zu den im Spiel vorkommenden Items bereit. Sie bietet eine Schnittstelle an, welche zu eingehenden Item-IDs weiterführende Daten, etwa die Beschreibung oder den Wert des Items, ausgibt. Dazu müssen die Items allerdings erst geladen werden.

Um eine möglichst einfache Erweiterbarkeit zu garantieren, werden die Items per XML-Datei (Extensible Markup Language) auf der Festplatte gespeichert. Diese Dateien werden dann durch die Item-Datenbank ausgelesen und im Spiel gespeichert. Das XML-Format ist ein „text-basiertes Format für den Austausch strukturierter Informationen“ [24]. Die Daten werden in folgender Anordnung gespeichert (entsprechende Beispieldaten sind bereits eingetragen):

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <Item>
3      <id>1</id>
4      <name_key>ItemSimpleHealingPotionName</name_key>
5      <beschreibung_key>ItemSimpleHealingPotionDesc</beschreibung_key>
6      <bild>pic_Trank</bild>
7      <wert>10</wert>
8      <Hauptkategorie>consumable</Hauptkategorie>
9      <Unterkategorie>healing</Unterkategorie>
10 <effekt>
11     <effektname>Lebenspunkte</effektname>
12     <effektwert>20</effektwert>
13 </effekt>
14 </Item>

```

Listing 68: Beispielitem im XML-Format

Wie in Listing 68 zu sehen ist, wird in XML mit einzelnen Nodes(Knoten) gearbeitet, welche die Werte beinhalten. Ein solcher Knoten wäre im gezeigten Beispiel etwa „Item“, welcher weitere untergeordnete Knoten besitzt. Um also ein Item aus einer

Datei auszulesen, müssen die einzelnen Nodes ausgelesen und in ein Item-Objekt gespeichert werden. Im ersten Schritt werden alle im Item-Verzeichnis des Projektes gespeicherten Items geladen.

Listing 69 zeigt das Laden der XML-Dokumente. Dazu werden zunächst alle Dateien aus dem Item-Ordner geladen.

```

63 | //Lade alle Items im Item-Ordner der Ressourcen
64 | Object[] unkonv_obj = Resources.LoadAll("Items");
65 | List<TextAsset> konv_obj = new List<TextAsset>();
66 |
67 | //Konvertiere die Objekte zu Text-Assets
68 | for (int i = 0; i < unkonv_obj.Length; i++)
69 | {
70 |     if (unkonv_obj[i].name.Contains("data"))
71 |         {konv_obj.Add(unkonv_obj[i] as TextAsset);}
72 | }

```

Listing 69: Laden der XML-Dateien

Da diese jedoch von der *LoadAll()*-Methode als *Object* zurückgegeben werden, müssen sie noch in ein lesbares Format konvertiert werden. Des Weiteren ist zu beachten, dass im Item-Ordner nicht nur die Quelldateien für die Item-Datenbank liegen, sondern auch Bilder für die jeweiligen Items. Um nur die Itemdaten auszulesen, wurden die XML-Dateien mit dem Tag *data* versehen. Beim Konvertieren wird nun geprüft, ob der Dateiname den Tag *data* enthält – wenn dies der Fall ist, wird das Objekt konvertiert und der Liste von konvertierten Objekten hinzugefügt. Nachdem der reine Quelltext geladen ist, kann der XML-Text als solches behandelt werden. Dazu wird der Typ *XmlDocument* genutzt, welcher zum Auslesen und Bearbeiten von XML-Quellcode gedacht ist.

Der Typ bietet Methoden zum Finden und Anzeigen einzelner Nodes und zum Auslesen der darin gespeicherten Attribute:

```

83 | XmlNodeList xnList = xml.SelectNodes("/Item");

```

Listing 70: Auswahl des Itemknotens

Dabei wird, wie in Listing 70 zu sehen ist, zuerst der Grundknoten „*Item*“ selektiert. In der Variable *xnList* sind nun alle Knoten vom Typ *Item* gespeichert. Dies ist nützlich, wenn mehrere Items in einer Datei gespeichert werden. Der Übersichtlichkeit und einfachen Erweiterbarkeit halber wird jedoch pro XML-Datei nur ein Item gespeichert.

Nachdem ein einzelnes Item eingelesen ist, können die einzelnen Nodes mit den Attributen eingelesen werden.

Da das Inventar verschiedene Item-Arten, wie benutzbare oder aufgabenbezogene Items, beinhaltet, muss das zu ladende Item direkt passend konvertiert werden. Dazu wird die Hauptkategorie des Gegenstandes ausgelesen und je nachdem, wie das Ergebnis ist, entweder ein neues *UsableItem* oder ein neues *QuestItem* erstellt. Listing 71 zeigt die Implementierung für den Fall des *QuestItems*.

```

87         Item_v2.MainCategory item_category = GetItemMainCategory(xn);
88
89         if (item_category.Equals(Item_v2.MainCategory.quest))
90         {
91             //DE: Definition + ID
92             QuestItem temporary_QuestItem = new QuestItem(GetItemID(xn));

```

Listing 71: Unterscheidung der Kategorien

Wenn die Hauptkategorie der Questkategorie entspricht, wird ein neues Item von diesem Typ instanziiert. Nachdem die Itemkategorie festgestellt wurde, können die Eigenschaften des Items ausgelesen werden. In Zeile 87/Listing 71 ist zu sehen, wie die ID-Eigenschaft des Items abgefragt wird. Die Methoden-Implementierung zeigt Listing 72:

```

172     private int GetItemID(XmlNode Item)
173     {
174         return int.Parse(Item["id"].InnerText);
175     }

```

Listing 72: Laden einer Item-Eigenschaft

Dazu wird die Eigenschaft *InnerText* eines Knotens abgefragt (Zeile 159). Diese Eigenschaft enthält den Text eines einzelnen Attributes. Nach und nach werden alle Details über das Item ausgelesen und in einem temporären Item gespeichert. Das temporäre Item wird zum Abschluss des Prozesses einer Liste hinzugefügt. Diese Liste speichert Objekte vom Typ *UniversalItem*. Dieser Typ ist allerdings ein reines Interface. Damit ist es möglich, Objekte von unterschiedlichen Typen zu speichern, solange sie das Interface implementieren. Da die Grundklasse *Item* das Interface implementiert, tun es auch die vererbten Klassen *Usable-* und *QuestItem*. Die Item-Liste und das Inventar des Spielers können so die unterschiedlichen Item-Arten in einer einzigen Liste speichern, da die speziellen Eigenschaften eines Objektes dabei nicht verloren gehen. Damit hat die Item-Datenbank nun die Item-Daten, welche sie nach außen geben soll. Um die Klasse zu komplettieren, wird jetzt noch die Methode

zur Herausgabe dieser Daten benötigt. Jene Methode gibt die Item-Details des Items zurück, dessen ID als Parameter übergeben wird, wie in Listing 73 zu sehen ist.

Die Liste wird nach der entsprechenden ID durchsucht. Sobald das Item gefunden wurde, wird es zurückgegeben.

```

150 public UniversalItem GetItemDetails(int _id)
151 {
152     return item_db_list.Find(x => x.Id.Equals(_id));
153 }

```

Listing 73: Rückgabe der Item-Details

Alle Klassen, welche vom Inventar genutzt werden, sind nun fertig gestellt. Jetzt muss die Inventarklasse selbst implementiert werden.

Inventar

Das Inventar fügt die vorhergegangenen Klassen zusammen und nutzt sie. Dazu wird ein Feld *Inventory_complete* deklariert, welches eine Liste von Items ist. Auf diesem Feld baut das restliche Inventar auf, indem Items hinzugefügt oder entfernt werden beziehungsweise geprüft wird, ob sich ein bestimmtes Item im Inventar des Spielers befindet. Diese Funktion wird beispielsweise beim auch Aufgabensystem benutzt, etwa wenn der Spieler einen bestimmten Gegenstand suchen muss. Exemplarisch für die Implementation der Methoden wird in Listing 74 der Quellcode der Methode zum Hinzufügen eines neuen Items erläutert:

```

53     if (Contains(tmp_item))
54     {
55         //DE: Index des Items wird gefunden
56         int index = inventory_complete.IndexOf(tmp_item);
57
58         //DE: Anzahl wird hochgezählt
59         inventory_complete[index].Amount = inventory_complete[index].Amount + 1;
60     }
61     else
62     {
63         //DE: Kapazitäts-Abfrage.
64         if (Capacity >= inventory_complete.Count)
65         {
66             //DE: Item wird dem Inventar hinzugefügt.
67             inventory_complete.Add(tmp_item);
68         }
69     }

```

Listing 74: Hinzufügen eines neuen Items

Zuerst wird geprüft, ob sich ein Item bereits im Inventar des Spielers befindet. Dazu wird die **öffentliche** Methode *Contains()* implementiert – diese Methode prüft mittels der *Contains()*-Methode der Listenklasse, ob sich ein Objekt in der **privaten** Liste befindet. Sollte diese Methode zurückgeben, dass der Spieler bereits über dieses Item verfügt, dann wird die Anzahl des Items im Inventar des Spielers um eins hochgezählt. Das Item wird im Inventar gefunden, indem die ID an die Item-Datenbank übergeben wird. Diese gibt dann das konkrete Item zurück. Danach wird mittels der *IndexOf()*-Methode der Platz des Items im Inventar ermittelt. Anschließend wird die Anzahl um eins erhöht.

Sollte das Item noch nicht im Inventar vorhanden sein, wird es der Liste der Items hinzugefügt. Auch hier werden die konkreten Item-Details wieder von der Item-Datenbank abgefragt.

Wie in den Anforderungen zum Inventar zu lesen ist, soll der Spieler die Items und Kategorien auch nach bestimmten Mustern sortieren können. Dazu stehen ihm folgende Sortierschemata zur Auswahl:

- „Manuell“,
- „A bis Z“,
- „Normal“,
- „Kampf“.

Das manuelle Sortierschema erlaubt dem Spieler, sich eine eigene Reihenfolge der Items im Inventar zu erstellen. Dazu braucht es eine Methode, welche zwei Items im Inventar des Spielers tauscht:

```

122  public void SwitchItems(UniversalItem item1,UniversalItem item2)
123  {
124      //DE: Die Indizes der beiden Items werden gespeichert,
125      //danach werden die Items getauscht.
126      int index_item1 = inventory_complete.IndexOf(item1);
127      int index_item2 = inventory_complete.IndexOf(item2);
128
129      inventory_complete[index_item1] = item2;
130      inventory_complete[index_item2] = item1;

```

Listing 75: Tauschen zweier Items des Inventars

In Listing 75 ist zu sehen, dass zur Realisierung des Itemtauschs einfach der Index der zu tauschenden Items gespeichert und danach die Items mit dem jeweiligen anderen Item überschrieben werden. Diese Methode erlaubt dem Spieler eine manuelle Sortierung.

Beim A bis Z-Sortierschema werden die Items im Inventar alphabetisch sortiert. Dafür muss keine eigene Methode implementiert werden. Es wird einfach die *Sort()*-Methode der *List*-Klasse genutzt. Diese erwartet als Parameter einen neuen Vergleich, also einen Ausdruck der angibt, wie eine Eigenschaft des Typs in der Liste verglichen werden soll.

```

226     case Sorting.AZ:
227         inventory_complete.Sort((x, y) =>
228             string.Compare
229             (Localizer.Instance.GetText(x.Name), Localizer.Instance.GetText(y.Name)));
230         break;

```

Listing 76: Implementierung der A bis Z-Sortierung

Für einen neuen Vergleich werden dabei zwei Einträge aus der Liste benötigt, welche verglichen werden sollen. Den eigentlichen Vergleich übernimmt in diesem Fall die *String*-Klasse, auf welcher der Name des Items basiert. Die *Sort()*-Methode vergleicht nun alle Einträge der Liste und sortiert sie dabei. Der genutzte Sortieralgorithmus der Methode unterscheidet sich je nach Größe der Liste [25]:

- Bei kleinen Listen wird der Insertion-Algorithmus verwendet.
- Bei mittleren Listen nutzt die Methode den Quicksort-Algorithmus.
- Bei großen Listen wird der Heapsort-Algorithmus genutzt.

Beim Inventar wird in der Regel entweder der Insertion- oder der Quicksort-Algorithmus eingesetzt, da die Größe des Inventars beschränkt ist.

Die Sortierschemata „Normal“ und „Kampf“ sortieren die Items je nach ihrer Unterkategorie – nachzulesen auf den Seiten 125 und 126. Hier kann nicht einfach auf eine *Compare()*-Methode eines zu Grunde liegenden Typs zurückgegriffen werden, da kein Vergleich für eine Enumeration existiert. Damit die Items dennoch sortiert werden können, gibt es zwei verschiedene Möglichkeiten:

- ein *Dictionary* erstellen oder
- einen eigenen Vergleich implementieren.

Zum Vergleich werden beiden Möglichkeiten nachfolgend implementiert. Auch wird evaluiert, welche von beiden die bessere Variante darstellt. Das Hauptaugenmerk liegt hierbei auf der Effizienz des Algorithmus. Je schneller die Sortierung von statten

geht, desto wertvoller ist sie für ein Spiel, denn die Rechenarbeit kann auch an anderen Stellen genutzt werden.

Ein Dictionary erstellen

Das *Dictionary* speichert zu Einträgen bestimmte Werte - wie in einem Wörterbuch, wo zu einem Wort eine Erklärung vorhanden ist. Das Wort ist dabei ein Key und die Erklärung der Wert(Value). Zusammen ergeben Key und Value einen Eintrag im *Dictionary*. Andere Programmteile erhalten jetzt zu einem Key den dazugehörigen Value. Daraus kann eine Sortierreihenfolge implementiert werden, indem jeder *Enum*-Eintrag eine bestimmte Priorität bekommt. Diese Priorität steht als zweiter Parameter im Methodenaufruf der *Add()*-Methode, für „Offensiv“ also die „0“.

```

215 Dictionary<Item_v2.SubCategory, int> sort;
216 sort = new Dictionary<Item_v2.SubCategory, int>();
217 sort.Add(Item_v2.SubCategory.offensive, 0);
218 sort.Add(Item_v2.SubCategory.support, 1);
219 sort.Add(Item_v2.SubCategory.healing, 2);
220 sort.Add(Item_v2.SubCategory.other, 3);
221 sort.Add(Item_v2.SubCategory.upgrade, 4);

```

Listing 77: Füllen des Dictionarys

Listing 77 zeigt, wie ein neues *Dictionary* angelegt und zu jeder Unterkategorie der Items eine Priorität gespeichert wird. Mit dieser Priorisierung kann nun die *OrderBy()*-Methode gefüllt werden, welche die Items nach der Unterkategorie sortiert:

```

224 inventory_complete = inventory_complete.OrderBy(m => sort[m.ItemSubCategory]).
225     .ThenBy(m => m.Name).ToList();

```

Listing 78: Sortierung des Inventars mit OrderBy

Die Items werden innerhalb gleicher Unterkategorien zusätzlich noch alphabetisch sortiert. Eine Messung der benötigten Zeit mit 7 Items im Inventar des Spielers zeigt folgende Werte:

Messung	Benötigte Zeit zum Sortieren
1	3,20 Millisekunden
2	3,91 Millisekunden
3	3,24 Millisekunden
4	3,18 Millisekunden
5	3,41 Millisekunden

Tabelle 8: Benötigte Zeit zum Sortieren mit OrderBy

Diese benötigte Zeit kann natürlich je nach Reihenfolge der Items variieren – jedoch werden bei den Messungen im praktischen Test immer mindestens drei Millisekunden benötigt. Dieser Wert erklärt sich durch die Funktionsweise der *OrderBy()*-Funktion [26]: Zuerst wird die gesamte Liste sortiert, indem jedes Element in der Liste durchgegangen wird, danach muss diese sortierte Sammlung noch in eine Liste konvertiert werden (erkennbar am *ToList()* am Ende der Funktion in Listing 78). Dabei wird im Speicher eine temporäre Liste angelegt, die sich vergrößert, bis alle Elemente aus der sortierten Liste hineinpassen. Dadurch wird Speicher reserviert – welcher unter Umständen an anderen Stellen fehlen kann. Des Weiteren muss zu jedem Item erst die Priorität im Wörterbuch nachgeschlagen werden, was sich wieder negativ in der Performance bemerkbar macht.

Einen eigenen Vergleich bereitstellen

Eine andere Möglichkeit besteht darin, einen eigenen Vergleich bereitzustellen, anhand dessen die *Sort*-Methode der Klasse *List* die Items sortiert. Dazu wird eine Helferklasse erstellt, welche vom Typen *IComparer* erbt. Dieser Typ stellt ein Interface zur Implementation einer Vergleichsfunktion bereit. Die Klasse muss eine Methode bereitstellen, die zwei Items erwartet und einen Integer-Wert zurückgibt, welcher angibt, wie die beiden Items zueinanderstehen:

- „-1“, wenn Item 1 vor Item 2 stehen müsste,
- „1“, wenn Item 2 vor Item 1 stehen müsste,
- „0“, wenn beide Items dieselbe Unterkategorie haben.

Die Vergleichsfunktion muss nun mit Vergleichen gefüllt werden, welche die Reihenfolge repräsentieren, in der die Items sortiert werden sollen:

```
39         //DE: offensive kommt vor support
40         if (a.ItemSubCategory.Equals(Item_v2.SubCategory.offensive)
41             && b.ItemSubCategory.Equals(Item_v2.SubCategory.support))
42         {
43             return -1;
44         }
45         if (a.ItemSubCategory.Equals(Item_v2.SubCategory.support)
46             && b.ItemSubCategory.Equals(Item_v2.SubCategory.offensive))
47         {
48             return 1;
49         }
```

Listing 79: Vergleich zweier Unterkategorien

Listing 79 zeigt beispielhaft für die restlichen Unterkategorien, wie der Vergleich aufgebaut ist. Wenn Item 1 ein Item mit der Unterkategorie „Offensiv“ ist und Item 2 die Unterkategorie „Support“ hat, dann stimmt die Reihenfolge. Wenn allerdings das erste Item ein Support-Item und das zweite ein Offensiv-Item ist, dann müssen beide Items getauscht werden. Nach diesem Muster werden dann alle restlichen Vergleiche aufgebaut:

- Offensiv > Support, Heilung, Sonstiges, Upgrade,
- Support > Heilung, Sonstiges, Upgrade,
- Heilung > Sonstiges, Upgrade,
- Sonstiges > Upgrade.

Nachdem alle Vergleiche innerhalb Unterkategorie abgeschlossen wurden, wird geprüft, ob beide Items dieselbe Unterkategorie besitzen:

```

151         //DE: Innerhalb der Gruppen wird von A bis Z sortiert.
152         if (a.ItemSubCategory.Equals(b.ItemSubCategory))
153         {
154             return string.Compare(a.Name, b.Name);
155         }
156         else
157         {
158             return 0;
159         }

```

Listing 80: Prüfung auf dieselbe Unterkategorie

Wenn dies der Fall ist, werden beide Items mittels des String-Vergleichers noch alphabetisch sortiert – zu sehen in Listing 80. Anschließend sortiert die *Sort()*-Methode das Inventar. Um die Performance beurteilen zu können, wurden wieder fünf Messungen, bei denen jeweils sieben Items sortiert werden sollen, vollzogen:

Nummer Messung	der Benötigte Zeit zum Sortieren
1	1,07 Millisekunden
2	0,95 Millisekunden
3	1,05 Millisekunden
4	1,07 Millisekunden
5	1,04 Millisekunden

Tabelle 9: Benötigte Zeit zum Sortieren mit einem eigenen Vergleich

Es ist festzustellen, dass die Performance im Vergleich zum Sortieren mittels Dictionary wesentlich besser ist, im Durchschnitt wird nur rund $\frac{1}{3}$ der Zeit benötigt. Diese Dauer kann sich je nach den Umständen ändern, da sich die Werte bei vielen Items oder anderen Anordnungen erhöhen oder verringern können – jedoch bietet das Inventar des Spielers nur Platz für verhältnismäßig wenig Items (<128).

Nachdem die Implementation der Kernfunktionen abgeschlossen ist, muss noch die Schnittstelle zwischen Spieler und Spiel geschaffen werden, das User-Interface.

GUI-Implementierung

Im ersten Schritt wird der Entwurf des User-Interfaces mittels der in Unity implementierten UI-Elemente realisiert. Abbildung 80 zeigt, wie das UI umgesetzt wurde:

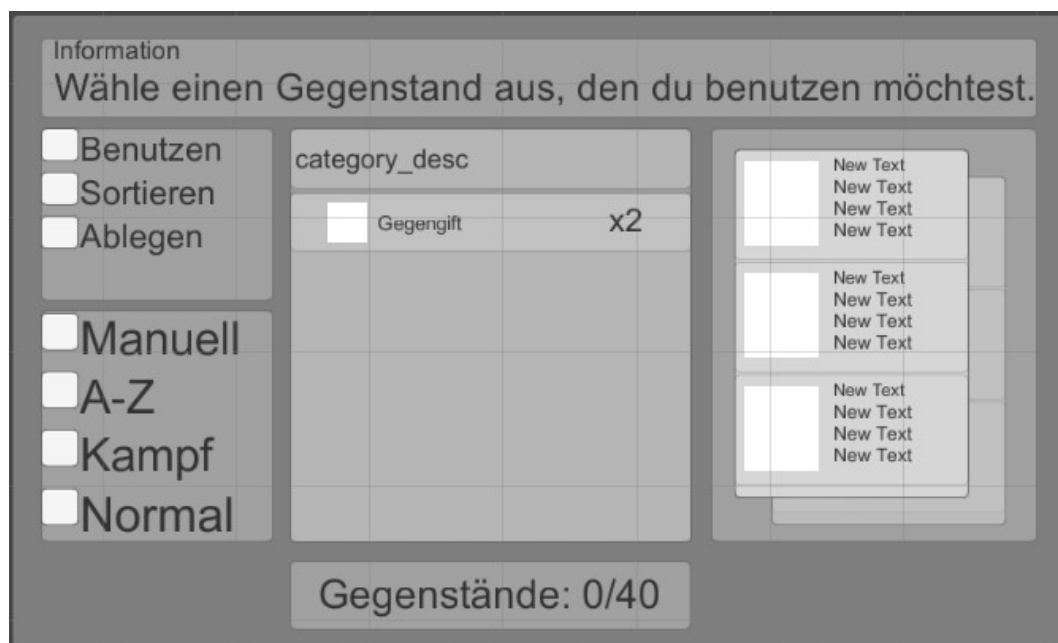


Abbildung 80: Grafische Umsetzung des Inventars

Damit wurde der UI-Entwurf, welcher auf Seite 130 zu sehen ist, umgesetzt.

Realisiert wird das grafische Grundgerüst unter anderem mit Panels. Diese Panels beinhalten weitere Komponenten, unter anderem Labels für die Texte und Toggle-Elemente zum Auswählen von Optionen. Die Teamanzeige besteht aus einem Bild und mehreren Labels.

Zusätzlich zu den Standard-Komponenten wurde die in Abschnitt 4.4.1 eingeführte Komponente *ScrollableList* genutzt, um eine Liste von Gegenständen anzuzeigen. Um die Funktionalität der einzelnen Teile des Inventars kümmern sich die UI-Handler.

UI-Handler	Beschreibung
Aktions-UI-Handler	Damit zur Item-Liste gesprungen oder das Sortierungs-GUI dargestellt wird.
Sortieren-UI-Handler	Damit das Inventar und damit auch die Item-Liste nach einem bestimmten Muster sortiert wird.
Info-UI-Handler	Der Info-Handler zeigt Beschreibungen zu Aktionen oder Gegenständen an.
Objekte-UI-Handler	Zum Anzeigen der Gegenstände und der Kapazität.
Team-UI-Handler	Zur Anzeige der vorhandenen Mozos.

Tabelle 10: UI-Handler des Inventars

Aktions-UI-Handler

Der Aktions-UI-Handler stellt Funktionen bereit, damit der Spieler mit dem Inventar interagieren kann. Der Spieler kann entweder ein Item auswählen beziehungsweise ablegen, oder das Inventar nach einem bestimmten Schema neu sortieren lassen. Zum Wechseln zwischen den Toggle-Elementen wurde das in Abschnitt 4.2.24.2.2 eingeführte Navigationssystem und die *UISelectables* genutzt. Damit Aktionen ausgeführt werden, sobald der Spieler ein bestimmtes Toggle-Element bestätigt, wurde jedem Toggle ein Event Trigger hinzugefügt. Dieser Trigger wurde so konfiguriert, dass er eine als Parameter übergebene Methode aufruft.

Abbildung 81 zeigt den Event Trigger für das „Benutzen“-Toggle.

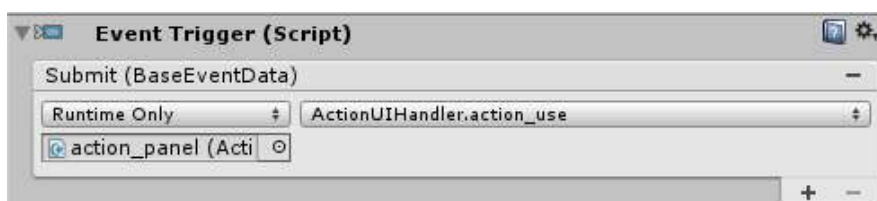


Abbildung 81: Event Trigger des "Benutzen"-Toggles

In diesem Trigger wurde konfiguriert, dass die `action_use()`-Methode des Aktions-UI-Handlers aufgerufen wird. Diese Methode registriert ein UI-Navigation-Token, welches am Panel der Item-Liste angeheftet wurde.

```

48 public void action_use()
49 {
50     category_selection_token.register();
51 }

```

Listing 81: Registrierung des Item-List-Tokens

Mit dem Registrieren wechselt der Fokus automatisch zum neuen Token. Für die anderen Toggle-Elemente und deren Event Trigger wurden lediglich die Ziel-Tokens geändert, welche beim Bestätigen des Toggles aktiviert werden.

Sortieren-UI-Handler

Genau wie der Aktions- nutzt der Sortieren-UI-Handler das Navigationssystem zum Wechseln zwischen den Elementen und Event Trigger zum Ausführen von Methoden, sobald der Spieler eine Aktion bestätigt. Listing 82 zeigt exemplarisch die Implementation für den Fall, dass der Spieler das A-Z-Schema auswählt:

```

31 public void gui_sort_az()
32 {
33     //Setzt die Sortierung neu (und sortiert damit auch das Inventar)
34     playerinventory.Inventory_sorting = Inventory.Sorting.AZ;
35
36     //Der GUI-Container des Inventars wird aktualisiert.
37     objectListUI.showItems(true);
38 }

```

Listing 82: Ausführen eines Sortierschemas

Dabei wird in Zeile 34 das Sortierschema im Inventar des Spielers neu gesetzt. Das Inventar sortiert daraufhin die Gegenstände im Inventar neu. Abschließend wird in Zeile 37 die UI-Liste, welche die Gegenstände grafisch darstellt, neu geladen.

Info-UI-Handler

Das Beschreibungs-GUI-Script aktualisiert die Beschreibung. Dazu nutzt es das Lokalisierungssystem, um in der jeweiligen Sprache eine Beschreibung anzuzeigen. Sobald eine neue Beschreibung angezeigt werden soll, etwa wenn ein neuer

Gegenstand ausgewählt wird, wird das *textKey*-Attribut der *LocalizeText*-Komponente überschrieben und die Lokalisierung gestartet.

Objekte-UI-Handler

Das Herzstück des Inventars ist der Objekte-UI-Handler, welcher das Anzeigen der Gegenstände und Verwalten der UI-Liste übernimmt. Zu diesem Zweck wurde ein UI-Prefab (zu sehen in Abbildung 82) erstellt, welcher dann vom Objekte-UI-Handler zum Anzeigen genutzt wird.



Abbildung 82: UI-Prototyp eines Gegenstandes

Die *ScrollableList*, die das Zeichnen und alle Interaktionen mit der Liste verwaltet, wurde bereits in Abschnitt 4.4.1 erläutert.

```

113     //DE: Fügt der Quest-Liste die neuen Quest-UI-Objekte hinzu.
114     for (int i = 0; i < displayItems.Count; i++)
115     {
116         //DE: Name
117         GameObject addedObject = prefab_liste.AddObject();
118         LocalizeText addedObjectLocalizeText =
119             addedObject.GetComponentInChildren<LocalizeText>();
120         addedObjectLocalizeText.textKey = displayItems[i].Name;
121         addedObjectLocalizeText.Start();
122
123         //DE: Menge
124         Text addedObjectAmountText = addedObject.GetComponentInChildren<Text>()[1];
125         addedObjectAmountText.text = "x" + displayItems[i].Amount;
126
127         //DE: Bild
128         addedObject.GetComponentInChildren<Image>()[1].sprite =
129             displayItems[i].Picture;

```

Listing 83: Hinzufügen von neuen Objekten in die Gegenstandsliste

Listing 83 zeigt, wie das Hinzufügen von neuen Objekten in die Liste von statten geht. Des Weiteren werden die UI-Objekte mit Daten befüllt. Dazu wird in Zeile 117 zunächst mittels der *ScrollableList*-Komponente ein neues Objekt hinzugefügt. In den Zeilen 118 bis 121 wird dann das *textKey*-Attribut der *LocalizeText*-Komponente geändert und die Lokalisierung gestartet, damit der Name des Gegenstandes in der eingestellten Sprache visualisiert wird. Die Zeilen 124/125 und 128/129 zeigen dann, wie die aktuelle Menge und das Gegenstandsbild eingestellt wird.

Team-UI-Handler

Der Team-UI-Handler bildet die Kennwerte jener Mozos ab, welche der Spieler aktuell in seinem Team aufgestellt hat. Zum Darstellen dieser Werte lädt der Team-UI-Handler das Team des Spielers und liest dort die gewünschten Attribute wie Lebenspunkte oder Energiepunkte aus. Zusätzlich müssen *UINavigationTokens* hinzugefügt werden, damit der Spieler im Teampanel navigieren kann.

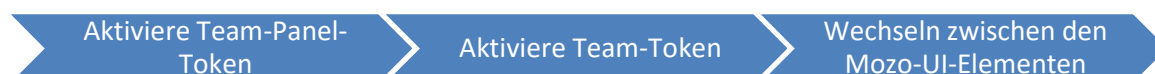


Abbildung 83: Reihenfolge der Navigation zum Selektieren eines Mozo

Dabei zeigt Abbildung 83, in welcher Reihenfolge die Navigation im Teampanel abläuft, um ein Mozo zu selektieren. An erster Stelle wechselt der Spieler zur Teamauswahl, um entweder das aktive oder das passive Team auszuwählen. Sobald diese Auswahl getätigt wurde, wird das Token des selektierten Teams aktiviert und der Spieler kann zwischen den Mozos wechseln, um ein Ziel für einen Gegenstand auszuwählen. Wird dieses Mozo bestätigt, dann werden die Item-Effekte angewendet, wie in Listing 84 zu sehen ist.

```

134         //DE: Wende den Item-Effekt an
135         if (!targetPlayerMozo.applyItemEffect(applyItem.Effects))
136         {
137             //Item-Effekt konnte nicht angewendet werden
138             new Notification("NotificationNotAllItemEffectsApplied4s", 4);
139         }
  
```

Listing 84: Anwenden des Item-Effektes

Die *ApplyItemEffect()*-Methode nimmt genau eine Liste von Effekten entgegen. Wenn die Methode ein *false* zurückgibt, dann konnten nicht alle Effekt angewandt werden, etwa, weil dem Mozo das Zielattribut fehlt. An dieser Stelle könnte bei der zukünftigen Weiterentwicklung des Spiels eine Fehlermeldung an den Spieler ausgegeben werden. Nachdem der Gegenstand auf ein Mozo angewendet wurde, wird es entweder aus dem Inventar entfernt oder die Anzahl des Gegenstandes wird um eins reduziert. Ebenso wird die Navigation im Teampanel beendet und der Spieler wird zurück zur Gegenstandsliste geleitet. Damit ist auch das Inventar fertig implementiert und kann vom Spieler genutzt werden. Im späteren Abschnitt 4.4.4 wird das Inventar während der Ausführung gezeigt.

4.4.3 Dialogsystem

Wie in den Anforderungen definiert, dient das Dialogsystem zur Interaktion mit verschiedenen Charakteren in der Spielwelt. Es können Fragen und Aussagen definiert werden.

Anwendungsfalldiagramm

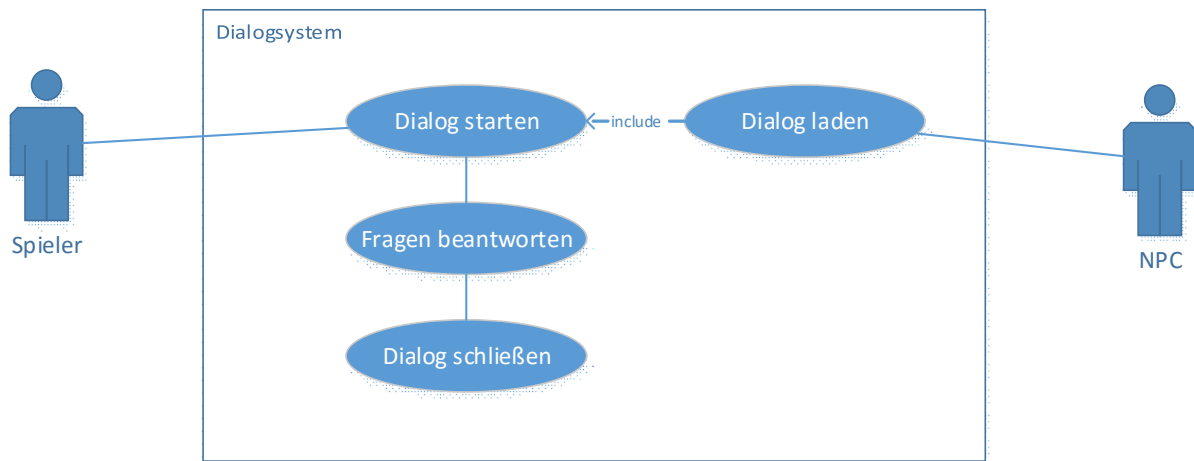


Abbildung 84: Anwendungsfalldiagramm des Dialogsystems

Abbildung 84 zeigt die Anwendungsfälle des Dialogsystems. Sobald der Spieler beispielsweise einen NPC anspricht, der über einen Dialog verfügt, so wird dieser Dialog geladen. Danach können Aussagen getätigt und Fragen beantwortet werden. Am Ende wird der Dialog geschlossen.

Komponentendiagramm

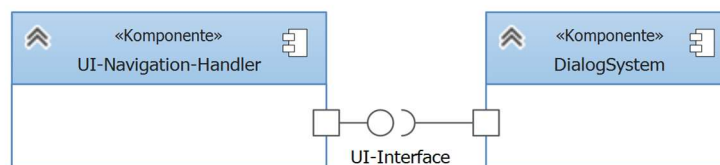


Abbildung 85: Komponenten des Dialogsystems

Das Dialogsystem benötigt dabei nur eine Komponente, und zwar den UI-Navigation-Handler. Bei diesem registriert sich der Dialog, da der Spieler zwischen den einzelnen Antwortoptionen wechseln kann. Das Dialogsystem wurde nicht noch einmal in mehrere Komponenten unterteilt, da es ein relativ kompaktes System ist und keine Schnittstellen für andere Komponenten bietet.

Klassendiagramm

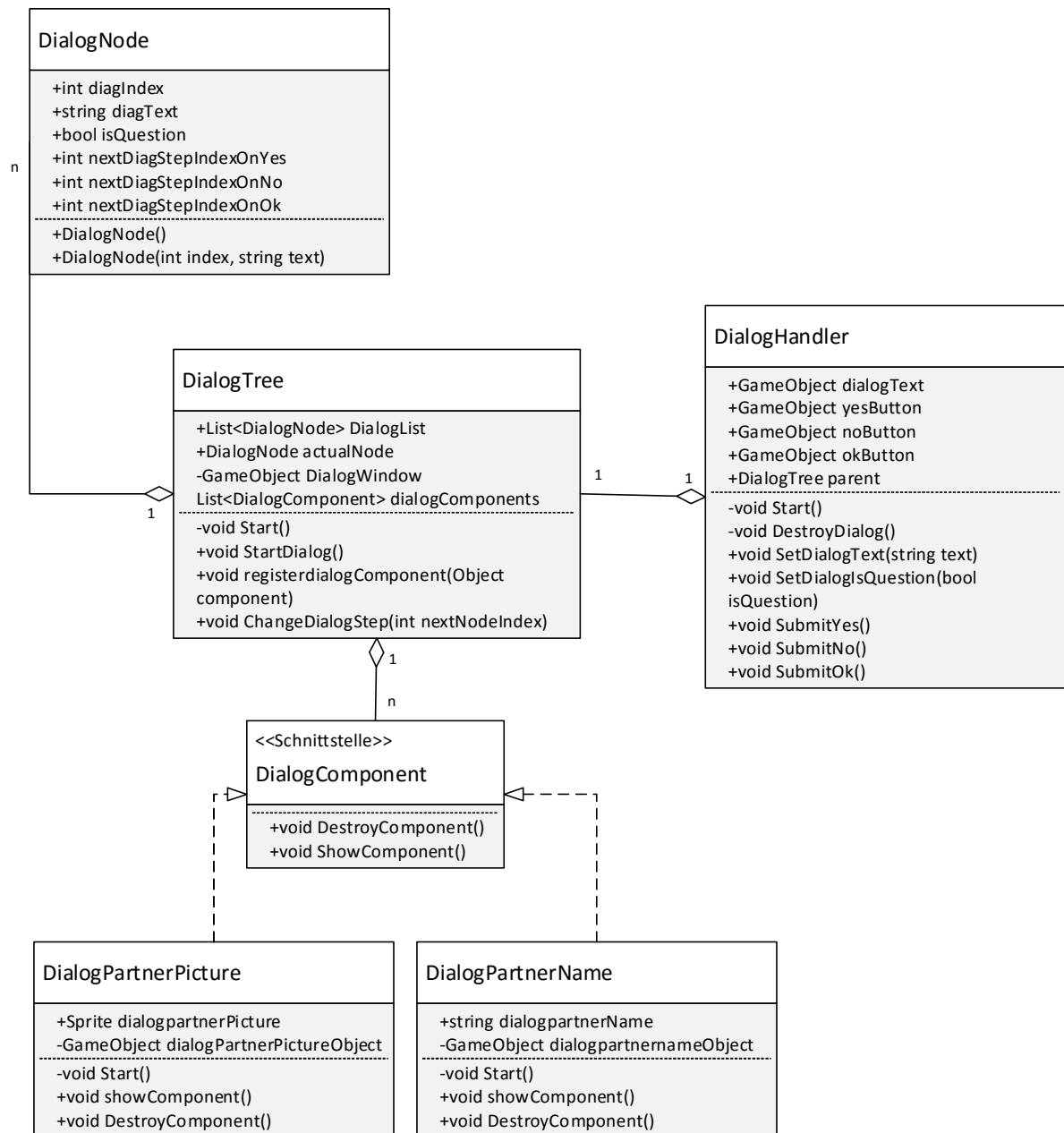


Abbildung 86: Klassen des Dialogsystems

Nachfolgend werden die in Abbildung 86 gezeigten Klassen und ihre Funktionsweise genauer beschrieben.

Implementierung

DialogTree

Wie in Abbildung 86 zu sehen, ist die *DialogTree*-Klasse die bedeutsamste Klasse. Hier wird der eigentliche Dialog gespeichert. Jeder Dialog besteht aus einer Liste von *DialogNodes*. In diesen Nodes wird ein Dialogschritt gespeichert, welcher unter anderem aus dem Text, einem Index und den nächsten Schritten besteht. Zusätzlich wurden Konstruktoren implementiert. Der *DialogTree* verwaltet die einzelnen Schritte. So werden etwa der aktuell angezeigte Schritt und das dazu gehörende Spielobjekt gespeichert und geändert. Dazu nutzt die *DialogTree* die *ChangeDialogStep()*-Methode, welche in Listing 85 zu sehen ist:

```

63 public void ChangeDialogStep(int nextNodeIndex)
64 {
65     //DE: Wenn die nächste Node "-1" ist, dann ist der Dialog zuende.
66     if (!(nextNodeIndex.Equals(-1)))
67     {
68         actualNode = DialogList[nextNodeIndex];
69
70         //DE: Neues Dialog-Fenster wird aus dem Prefab erzeugt.
71         DialogWindow = (GameObject)Instantiate
72             (Resources.Load("Prefabs/DialogPanelv1.4.6"));
73         DialogWindow.GetComponent<DialogHandler>().
74             setDialogIsQuestion(actualNode.isQuestion);
75         DialogWindow.GetComponent<DialogHandler>().
76             setDialogTextKey(actualNode.diagTextKey);
77         DialogWindow.GetComponent<DialogHandler>().
78             Parent = this;
79     }
80     else
81     {
82         StopDialog();
83     }
84 }

```

Listing 85: Wechseln des Dialogschrittes

Als Parameter muss der Index des nächsten Knotens(*Node*) übergeben werden. Sollte dieser Index „-1“ sein, so bedeutet das, dass der Dialog an dieser Stelle zu Ende ist. Danach wird das Dialogfenster geschlossen. Wenn ein gültiger Index übergeben wurde, so wird, wie in Zeile 68 zu sehen, zunächst die aktuelle *Node* gespeichert und danach ein neues Dialogfenster erzeugt. Nach dem Erzeugen werden Fragestatus und Text des Fensters gesetzt.

Neben dem Anzeigen von Dialogschritten übernimmt die *DialogTree*-Klasse das Hinzufügen von zusätzlichen Komponenten wie Bild oder Name des Dialogpartners. Diese Komponenten werden entgegengenommen und in einer Liste gespeichert.

Damit kann der Dialog angezeigt werden. Diesen Schritt übernimmt die *StartDialog()*-Methode (siehe Listing 86).

Beim Laden des Dialogs wird der erste Dialogschritt, welcher gespeichert ist, angezeigt.

```

47 public void StartDialog()
48 {
49     if (DialogList.Count != 0) ChangeDialogStep(0);
50
51     //DE: Startet die zusätzlichen GUI-Komponenten (sollten welche vorhanden sein)
52     foreach(DialogComponent component in dialogComponents)
53     {
54         component.showComponent();
55     }
56 }

```

Listing 86: Starten des Dialoges

Zusätzlich werden alle zuvor registrierten Komponenten gestartet, wie in den Zeilen 52 bis 55 zu sehen ist.

DialogHandler

Der *DialogHandler* verbindet einen angezeigten Dialogschritt mit den Daten aus dem *DialogTree*. Zu diesem Zweck sendet der *DialogTree* über die Methoden *SetDialogText()* und *SetDialogsQuestion()* die Daten zum anzuzeigenden Dialogschritt. In der *SetDialogText()*-Methode wird der Dialogtext verarbeitet und angezeigt. In der *SetDialogsQuestion()*-Methode dagegen wird geprüft, ob der Dialog eine Frage beinhaltet: Sollte sich dies bestätigen, so wird der Ja- und der Nein-Button eingeblendet, der Ok-Button wird ausgeblendet. Wenn der Dialog keine Frage ist, werden die Fragen-Buttons ausgeblendet und der Ok-Button wird eingeblendet. Danach fügt der *DialogHandler* dem Spielobjekt einen Navigations-Token hinzu, damit der Benutzer zwischen den Buttons wechseln kann:

```

78 | gameObject.AddComponent<UINavigationToken>();

```

Listing 87: Hinzufügen einer Komponente während der Laufzeit

Da die Navigation-Token-Komponente per Code erzeugt wurde, konnte zum Start kein selektierbares Element übergeben werden, welches zuerst markiert werden soll. Für diesen Fall sucht sich die Komponente selbst ein Element, indem alle verfügbaren, selektierbaren Elemente gesucht werden. Begrenzt wird dies dadurch,

dass nur in dem Spielobjekt gesucht wird, an dem der Navigation-Token angehängt ist. In diesem Fall findet die Komponente nach dem Start entweder den Ja- oder den Ok-Button, je nachdem welche Buttons zurzeit eingeblendet sind. Damit ist der eigentliche Dialog startbereit und lauffähig.

DialogComponent

Um zusätzlich zum reinen Dialogtext und den Antwortmöglichkeiten weitere Komponenten zu ermöglichen, wurde ein Interface *DialogComponent* geschaffen. Dieses Interface sagt aus, dass Klassen, die das Interface implementieren, über die Methoden *ShowComponent()* und *DestroyComponent()* verfügen müssen.

```

5 interface DialogComponent
6 {
7     3 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    void destroyComponent();
8     3 Verweise | 0 Änderungen | 0 Autoren, 0 Änderungen
    void showComponent();
9 }

```

Listing 88: Implementierung des Interfaces für neue Dialog-Komponenten

Die Implementierung des Interfaces ist in Listing 88 zu sehen. Mit diesen Methoden ist es der Verwalterklasse des Dialogs, *DialogTree*, möglich, beliebige zusätzliche Komponenten hinzuzufügen und anzuzeigen, ohne zu wissen, um welche Komponenten es sich speziell handelt. Beispielhaft für eine neue Komponente wird an dieser Stelle das Implementieren des Namens des Dialogpartners gezeigt. Nach dem Erzeugen der Komponente meldet sie sich bei dem *DialogTree*-Objekt an, an das es angehängt wurde:

```

27 | gameObject.GetComponent<DialogTree>().registerdialogComponent(this);

```

Listing 89: Registrierung der Komponente

Danach kann die *DialogTree*-Klasse die Komponente starten und auch wieder zerstören. Wenn die Komponente gestartet wird, wird das GUI-Prefab geladen und positioniert, wie in Listing 90 zu sehen. Nachdem das Objekt positioniert ist (Zeilen 44 und 45), wird der Name des Dialogpartners gesetzt. Dieser wird vorher im Unity-Editor festgelegt.

Mithilfe des `DialogComponent`-Interfaces könnten auch weitere Komponenten erstellt werden, welche sich mit dem Dialog verknüpfen.

```

34 public void showComponent()
35 {
36     //DE: Prefab wird geladen
37     dialogpartnernameObject = (GameObject)
38         Instantiate(Resources.Load("Prefabs/DialogNamePanelv1.0"));
39
40     //DE: GameCanvas wird zum Parent, damit das Prefab dargestellt werden kann.
41     dialogpartnernameObject.transform.SetParent(GameObject.Find("GameCanvas").transform);
42
43     //DE: Positionierung des Objektes.
44     dialogpartnernameObject.GetComponent<RectTransform>().offsetMin = new Vector2(0, 0);
45     dialogpartnernameObject.GetComponent<RectTransform>().offsetMax = new Vector2(0, 0);
46
47     //DE: Der Name des Dialogpartners wird gesetzt.
48     dialogpartnernameObject.GetComponentInChildren<Text>().text = dialogPartnerName;
49 }

```

Listing 90: Start einer zusätzlichen Dialog-Komponente

In der `DestroyComponent()`-Methode wird dann einfach nur ein `Destroy()` aufgerufen. `Destroy()` ist eine in der UnityEngine eingebaute Methode zum Zerstören von Objekten. Damit ist die Namens-Komponente fertig zum Benutzen und sie kann an ein Spielobjekt mit vorhandenem `DialogTree` angehängt werden. Das Resultat ist in Abbildung 87 abgebildet:

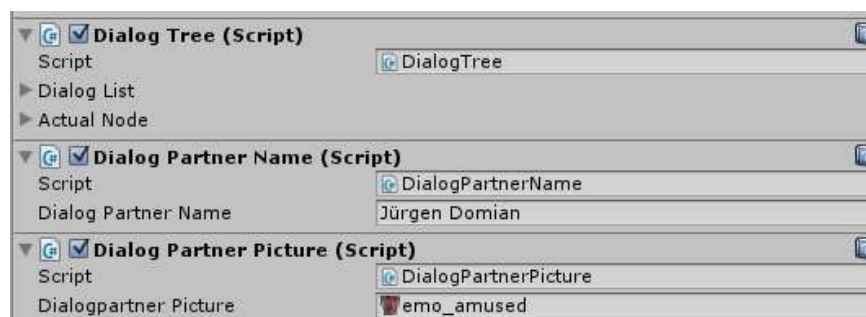


Abbildung 87: Ansicht des Dialogs im Unity-Inspector

Nach dem Anhängen der Komponenten können die Eigenschaften gesetzt werden.

GUI-Implementierung

Nachdem die benötigten Klassen für das Dialogsystem implementiert wurden, fehlen nur noch die GUI-Objekte, welche zum Anzeigen des Dialogs benötigt werden. Diesbezüglich wurden einige GUI-Elemente wie Panels und Labels zusammen-

gestellt, so dass sie rudimentär für das Anzeigen von Inhalt geeignet sind. Dieser erste Entwurf ist in Abbildung 88 zu sehen:



Abbildung 88: Erster Entwurf eines Dialogfensters

Das Dialogfenster wurde dabei so entworfen, dass zum Bildschirmrand immer 30% Platz ist. So vergrößert und verkleinert sich das UI dynamisch. Aus dem Entwurf wurde anschließend ein Prefab erzeugt, damit es per Code aufgerufen werden kann. Dasselbe Verfahren wurde für die zusätzlichen Komponenten des Dialoges angewandt. Das Resultat für einen fertigen, beispielhaften Dialog kann in Abbildung 89 betrachtet werden:

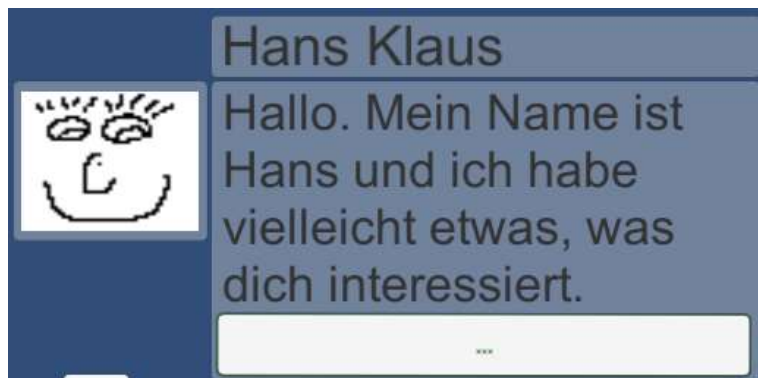


Abbildung 89: Der fertige Dialog

Zu sehen ist ein Dialogschritt mit den angehängten Komponenten für die Namens- und die Bildanzeige. Der Schritt ist keine Frage, daher wurde nur der „...“-Button eingeblendet, welcher zum nächsten Dialogschritt leitet. Ein weiteres Beispiel für einen Dialog während der Ausführung des Spiels kann im nächsten Abschnitt betrachtet werden.

4.4.4 Zwischenstand des Spiels – die UI-Systeme

Da die in Abschnitt 4.4 erläuterten Komponenten für sich genommen keine Basis für andere Komponenten, sondern Endresultate aus den vorherigen Basiselementen darstellen, werden sie an dieser Stelle nur bei der praktischen Ausführung erläutert.

Aufgabensystem

Das UI des Aufgabensystems sollte die offenen beziehungsweise geschlossenen Aufgaben des Spielers darstellen. Zusätzlich waren die Details zu jeder Aufgabe anzuzeigen. Abbildung 90 zeigt das Aufgabensystem während der Ausführung.

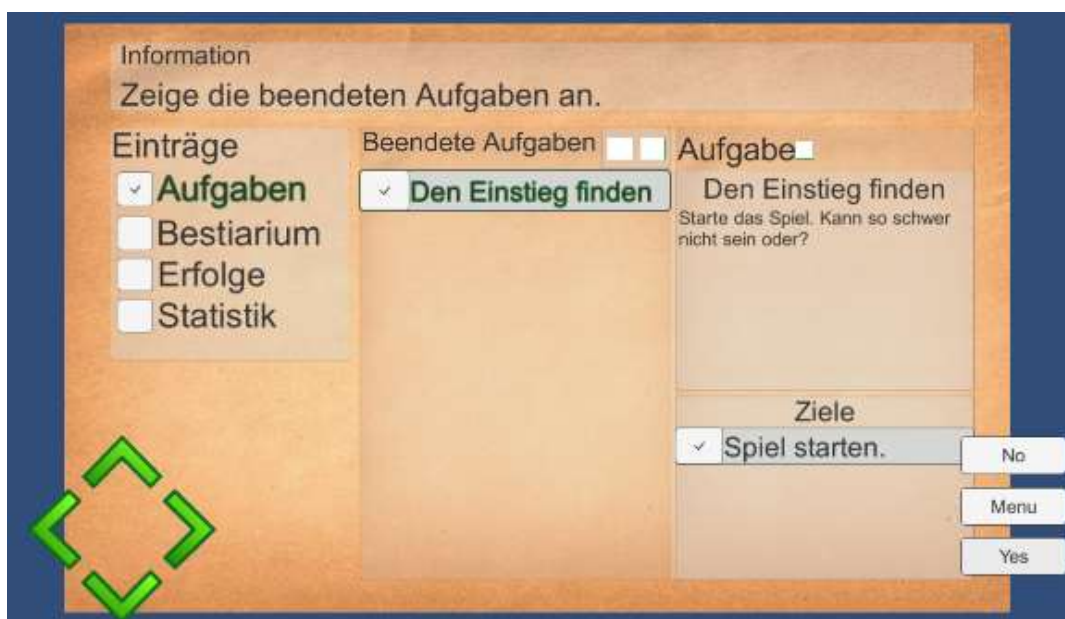


Abbildung 90 UI des Aufgabensystems während der Ausführung

Wie zu sehen ist, konnte das UI entsprechend den Anforderungen umgesetzt werden. Beispielhaft ist die Aufgabe „Den Einstieg finden“ markiert. Im Detailfenster wird die Beschreibung der (einzigen) Teilaufgabe angezeigt. Des Weiteren wird dem Spieler eine Liste von Aufgabenteilen präsentiert. Im obigen Beispiel wurde der Aufgabenteil „Spiel starten“ bereits erfolgreich ausgeführt. Da dies der einzige Aufgabenteil war, ist die Aufgabe erledigt. Sie wird bei den beendeten Aufgaben des Spielers angezeigt.

Inventar

Im UI des Inventars sollte der Spieler seine Gegenstände verwalten können.

Das fertige UI ist in Abbildung 91 während der Ausführung zusehen.



Abbildung 91 UI des Inventars während der Ausführung

Dabei ist zu erkennen, dass alle erforderlichen Funktionen erfolgreich implementiert werden konnten. Beispielhaft ist zu sehen, wie der Gegenstand „Einfacher Sockel“ markiert ist. Außerdem kann der Spieler das Inventar sortieren, Gegenstände ablegen und benutzen und die Anzahl an Gegenständen in seinem Inventar einsehen.

Dialog

Mithilfe des Dialogs sollte der Spieler in Interaktion mit seiner Umwelt kommen.

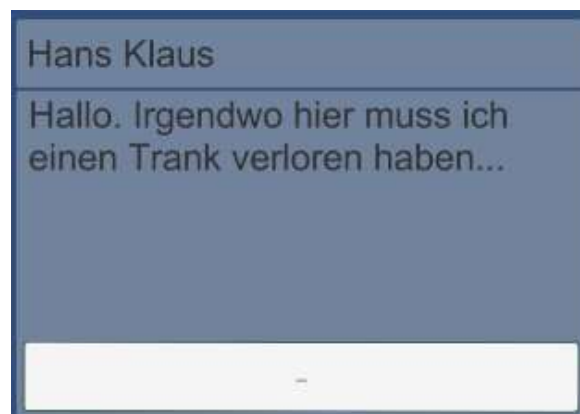


Abbildung 92 Dialog während der Ausführung

Wie in Abbildung 92 zu sehen ist, wurde auch dieses UI erfolgreich umgesetzt.

5 Evaluierung und Test der Anwendung

5.1 Usability-Test auf verschiedenen Plattformen

Zur Evaluierung des Spiels und der enthaltenen Funktionen wurde ein Usability-Test vollzogen. Um den Rahmen dieser Arbeit nicht zu übersteigen wurde dieser Test stark verkürzt. Der Test dient dazu, durch unabhängige Tester zu überprüfen, ob die Funktionen und die UI-Systeme ordnungsgemäß und gut bedienbar implementiert wurden.

Dazu haben vier Personen drei Aufgaben absolviert. Ihre Eindrücke wurden nach dem Test notiert. Dabei wurde geprüft, wie gut die Testpersonen die einzelnen Aufgaben bearbeiten konnten und welche Probleme eventuell dabei auftraten. Folgende Aufgaben wurden an die Tester gestellt:

Aufgabe 1 – Wende einen Gegenstand auf ein Mozo an

Ein zufälliger Gegenstand aus dem Inventar des Spielers soll auf ein selbst gewähltes Mozo angewendet werden.

Aufgabe 2 – NPC finden und ansprechen

Es soll ein NPC gefunden werden, mit welchem der Spieler interagieren kann. Dieser NPC soll angesprochen werden.

Aufgabe 3 – Aufgabedetails zu den offenen Aufgaben einsehen

Damit Daten zu den aktuellen Aufgaben eingesehen werden können, soll das Aufgabensystem geöffnet werden. Anschließend ist eine Aufgabe auszuwählen und die Details sind anzuzeigen.

Teilnehmer des Tests

Die Teilnehmer des Tests wurden so ausgewählt, dass ein möglichst großes Spektrum an unterschiedlichen Profilen erreicht wird.

	Geschlecht	Alter	Beruf	Spiel bekannt?	Spielerfahrung?	Plattform
1	Männlich	20	Auszubildender	Nein	Ja	Android
2	Männlich	20	Student	Nein	Ja	PC
3	Weiblich	40	Ingenieur	Nein	Nein	PC
4	Weiblich	20	Erzieherin	Nein	Ja	Android

Tabelle 11: Usability-Testteilnehmer

Bei allen Teilnehmern wurde der Test unter Aufsicht durchgeführt und anschließend in Interviewform aufgezeichnet, welche Aufgaben erfolgreich oder nicht erfolgreich gemeistert werden konnten. Auch ein persönliches Fazit und positive sowie negative Aspekte wurden notiert. Allen Teilnehmern wurde das Spiel auf einem Testgerät vorgelegt. Installation und Inbetriebnahme waren also nicht Teil dieses Tests, genau wie die Spezifikationen der Nutzergeräte.

Auswertung des Usability-Tests

In Tabelle 12 sind die Ergebnisse des Usability-Tests zu sehen. Dabei stehen die genutzten Symbole für folgende Testergebnisse:

E – Die Aufgabe wurde erfolgreich und problemlos gelöst.

V – Die Aufgaben wurde erfolgreich, aber mit einer Verzögerung gelöst.

P – Die Aufgabe wurde erfolgreich, aber nur mit erheblichen Problemen gelöst.

F – Die Aufgabe musste abgebrochen werden und gilt damit als nicht gelöst.

Teilnehmer	1	2	3	4
Aufgabe 1	V	E	E	V
Aufgabe 2	E	E	V	E
Aufgabe 3	E	E	V	E

Tabelle 12: Ergebnisse des Usability-Tests

Durch den Test konnten wertvolle Erkenntnisse gesammelt werden. Die Tabelle zeigt, dass etwa beim Inventar noch Verbesserungspotenzial herrscht. Insgesamt ist der Test aber positiv verlaufen. Er dokumentiert, dass das grundlegende Bediensystem gut konzipiert und ausgearbeitet wurde.

5.2 Performance-Test

5.2.1 Tools zur Performance-Messung in Unity

Die UnityEngine und die zugehörige Entwicklungsumgebung bringen bereits Tools mit, um die Performance von Spielen und deren Komponenten zu messen und auszuwerten. Dazu wird der integrierte **Profiler** genutzt, welcher verschiedene Daten zum aktuell ausgeführten Spiel bereithält. Abbildung 93 zeigt den Profiler und seine wichtigsten Komponenten.

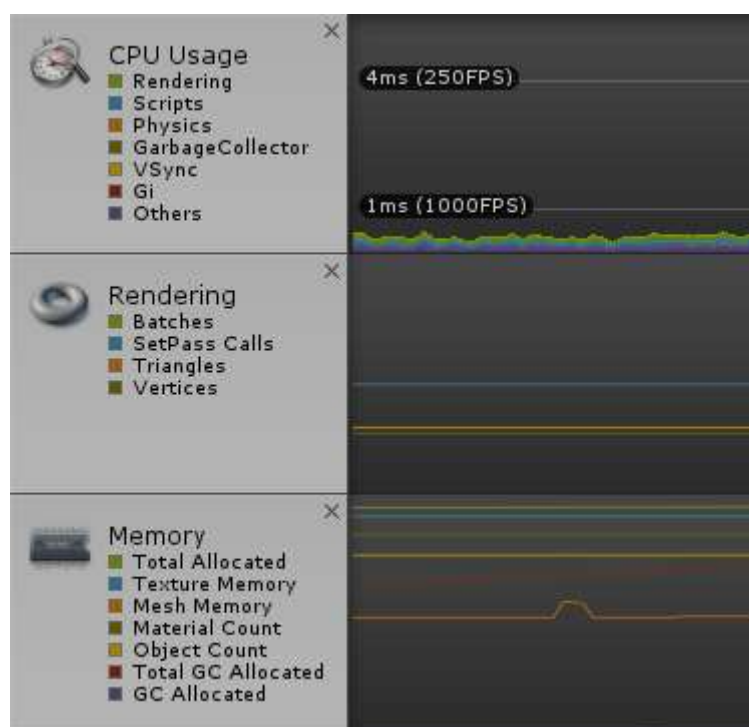


Abbildung 93: Profiler des Unity-Editors

Neben den in Abbildung 93 zu sehenden Komponenten gibt es auch weitere, etwa zum Messen der benötigten Netzwerkbandbreite. Die wichtigsten Komponenten werden nachfolgend kurz erläutert.

CPU Usage

Der *CPU Usage*-Tab zeigt Informationen zur CPU-Auslastung. Jeder Tab im Profiler zeigt Daten zu verschiedenen, integrierten Komponenten an. Im Falle der CPU-Auslastung ist das etwa die genutzte Zeit für das Rendern von Objekten oder das Ausführen von Skripten. Wie in Abbildung 93 ebenfalls zu sehen ist, kann detailliert verfolgt werden, wie viele Frames per Second (FPS) das Spiel in der aktuellen Szene

erreicht. Die FPS geben an, wie oft innerhalb einer Sekunde ein neues Bild erzeugt wird. Je höher dieser Wert ist, desto flüssiger läuft das Spiel. Der Wert sollte oberhalb der 60 FPS-Grenze liegen, damit reibungsloses Spielen garantiert ist [27].

Rendering

Der *Rendering*-Tab zeigt hauptsächlich, wie viele Objekte die UnityEngine zeichnen muss. Hier werden größere Grafiken, Spielobjekte und Texturen gezeichnet. Diese Arbeit übernimmt in der Regel die Grafikeinheit des ausführenden Systems.

Memory

Hier zeigt sich, wie viel Arbeitsspeicher das Spiel beansprucht. Auch werden die einzelnen Komponenten wieder grafisch aufgelistet.

Mithilfe dieser und weiterer im *Profiler* integrierten Komponenten können Entwickler genau nachverfolgen zu welchem Zeitpunkt das Spiel welche Auslastung auf dem Zielsystem hervorruft. Jede Komponente zur Auswertung verfügt dabei über weitere Details, auf die im nachfolgenden Abschnitt eingegangen wird. Der Verlauf der Auslastung lässt sich Frame für Frame nachverfolgen. Komponenten, welche eine höhere Auslastung hervorrufen, lassen sich so leicht identifizieren und gezielt optimieren.

5.2.2 Performance-Test

Mit dem *Profiler* bringt der Unity-Editor eine mächtige Möglichkeit mit, die Performance des Spiels zu überwachen und eventuelle Probleme zu identifizieren und zu lösen. In diesem Abschnitt wird anhand einer kurzen Beispielsequenz geklärt, ob das Spiel und die ausgeführte Funktion effizient sind. Des Weiteren wird geklärt, inwiefern bei der gezeigten Szene Verbesserungspotenzial vorhanden ist und wie dieses Potenzial ausgeschöpft werden kann.

In der auszuwertenden Beispielszene soll das Spiel gestartet, das Hauptlevel geladen und das Inventar geöffnet werden. Anschließend werden die aufgezeichneten Daten des *Profiler* ausgewertet und erläutert. Eventuelle Performanceprobleme können dabei erkannt und Lösungen präsentiert werden.

Damit der *Profiler* Daten aufzeichnen kann, muss die Szene, welche analysiert werden soll, neben dem geöffneten *Profiler* abgespielt werden. Anschließend wird das Spiel pausiert. Abbildung 94 zeigt, welche Daten bei der oben beschriebenen Beispielszene aufgezeichnet wurden.

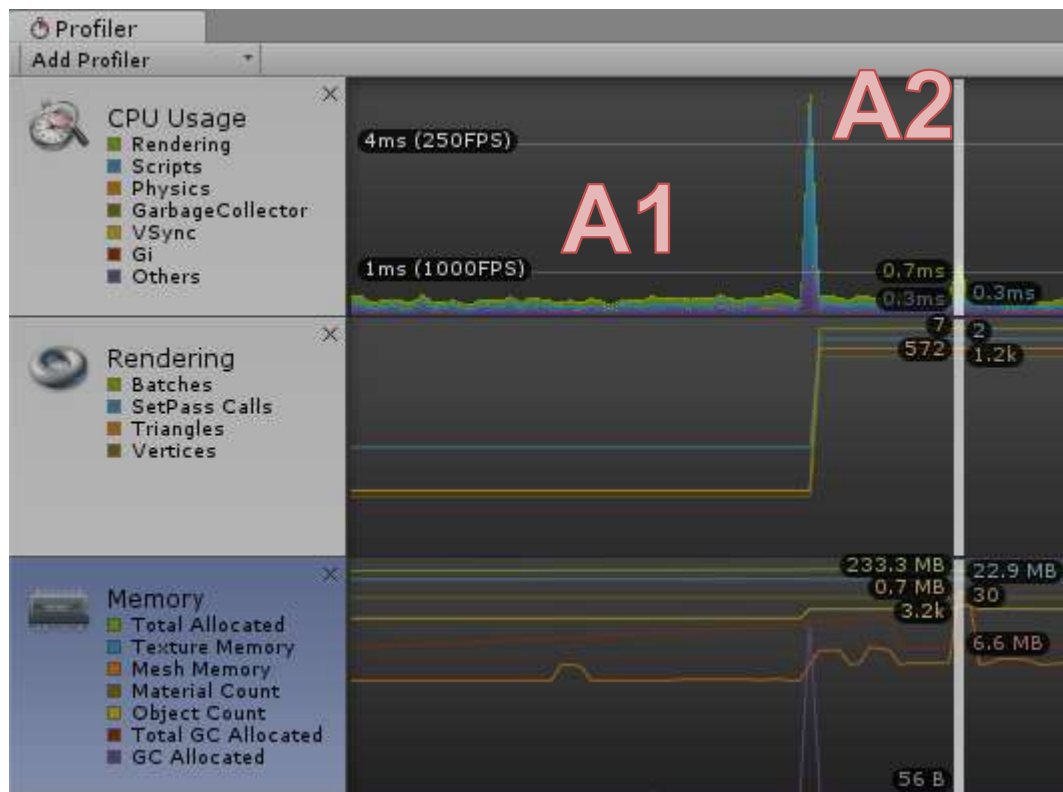


Abbildung 94: Auswertung der Profiler-Daten

Dabei ist zu sehen, dass das Spiel in Abschnitt 1 (A1) mit mehreren Tausend FPS läuft, was durchaus als flüssig bezeichnet werden kann. Der Ausschlag nach oben in einem kurzen Zeitraum (A2) zeigt, welche Auslastung durch das Öffnen des Inventars hervorgerufen wird. Diese Auslastung kann sowohl im *CPU Usage*-Tab als auch im *Rendering*-Tab beobachtet werden. Die Peaks in diesen Tabs zeigen, dass die Auslastung vor allem durch das Zeichnen von neuen Elementen bewirkt wird. Dies kann auch in den Details der *CPU Usage*-Komponente beobachtet werden. Wie in Abbildung 95 zu sehen ist, können Details zu den aufrufenden Funktionen eingesehen werden. Der Entwickler kann beispielsweise beobachten, wie viel der verursachten Auslastung durch eine spezielle Funktion erzeugt wurde.

Auch die Zeit, die die Funktion zur Ausführung benötigt hat, wird aufgelistet. Abbildung 95 zeigt die Auslastung zum dem Zeitpunkt, als das Inventar des Spielers aufgerufen wurde.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ BehaviourUpdate	60.8%	0.0%	1	196.1 KB	3.42	0.00
▼ EventSystem.Update()	60.6%	3.4%	1	196.1 KB	3.41	0.19
▶ Instantiate	51.5%	0.0%	1	195.0 KB	2.89	0.00
▶ Transform.SetParent	3.3%	0.7%	1	0 B	0.18	0.04
Graphic.OnRectTransformDimensionsChange()	1.0%	1.0%	44	0 B	0.05	0.05
CanvasRenderer.OnTransformChanged	0.9%	0.9%	245	0 B	0.05	0.05
Image.SetupCoroutine() [Coroutine: InvokeMoveNext]	0.1%	0.1%	1	144 B	0.00	0.00
UIBehaviour.OnRectTransformDimensionsChange()	0.0%	0.0%	14	0 B	0.00	0.00
Image.Start() [Coroutine: System.Collections.IEnumerat	0.0%	0.0%	1	0 B	0.00	0.00
CanvasScaler.Update()	0.0%	0.0%	2	0 B	0.00	0.00
UINavigationController.Update()	0.0%	0.0%	1	0 B	0.00	0.00
▶ Canvas.SendWillRenderCanvases()	25.9%	24.5%	1	5.5 KB	1.45	1.37
Overhead	2.7%	2.7%	1	0 B	0.15	0.15
▶ Camera.Render	2.4%	0.7%	1	0 B	0.13	0.04
▶ UINavigationToken.Start()	2.0%	0.8%	1	12.5 KB	0.11	0.04

Abbildung 95: Verteilung der Auslastung

So ist zu sehen, dass die *BehaviourUpdate()*-Funktion etwa 61% der Auslastung verursacht. Wie in der 3. Zeile der Übersicht zu erkennen ist, entfallen etwa 51% davon auf das Initialisieren des Inventars und der grafischen Komponenten des Inventars. Dadurch fallen die FPS beim Aufrufen des Inventars kurzzeitig auf unter 250. Das mag immer noch viel erscheinen, jedoch ist der Rückgang von mehreren Tausend FPS auf wenige Hundert ein wesentlicher Einbruch.

Dank dieser Daten kann der Entwickler Optimierungen einpflegen, welche Einbrüche verhindern oder zumindest reduzieren kann. In dem oben gezeigten Beispiel könnte das Inventar beispielsweise in einem separaten Ladeprozess am Anfang des Spiels im Hintergrund geladen werden. Dadurch sind die benötigten Grafikdaten bereits im Arbeitsspeicher vorhanden und der Aufruf des Inventars sollte weniger Auslastung verursachen. In diesem Stil kann auch für andere Komponenten verfahren werden, so dass das Spiel optimal auf die Hardware abstimmbare ist.

6 Zusammenfassung

Das Ziel der Arbeit war die Entwicklung des Prototyps eines Rollenspiels, welches auf verschiedenen Plattformen verfü- und spielbar ist. Zu diesem Zweck war es notwendig, eine Spiel-Engine zu verwenden, welche das Kompilieren für die einzelnen Plattformen übernimmt. Mit der UnityEngine wurde das passende Framework und die dazu gehörende Entwicklungsumgebung gewählt. Mithilfe der UnityEngine können verschiedene Plattformen wie Konsolen, Smartphones und PCs als Ziele genutzt werden. Die UnityEngine bringt auch einige Komponenten mit, welche für die Umsetzung der Anforderungen nützlich sind. Auf Basis der in der UnityEngine verfügbaren Technologien wurden diverse Basissysteme für das Spiel implementiert, beispielweise das Navigationssystem. Mithilfe dieser Komponente wurde es dem Spieler möglich gemacht, das Spiel plattformunabhängig zu steuern. Das Navigationssystem wurde in die verschiedenen UI-Systeme eingearbeitet. Dabei wurden im Laufe der Entwicklung des Spiels weitere Basis-Komponenten ausgearbeitet, welche universell in verschiedenen UI-Systemen einsetzbar sind. Die dynamische Liste zum Anzeigen von Inhalten ist nur ein Beispiel von mehreren. Sie kann relativ performant mit Listeneinträgen gefüllt werden, ohne dass vorher bekannt ist, wie der Listeneintrag UI-technisch aufgebaut ist. Auch andere, auf den Basis-Komponenten aufbauende Systeme konnten problemlos implementiert werden. Das Inventar etwa setzt sämtliche Basis-Komponenten ein. Die Lokalisierung übersetzt die UI-Texte, das Navigationssystem wird zum Steuern genutzt und die Mozo-Komponenten werden visualisiert. Auch der Karteneditor konnte erfolgreich umgesetzt werden. Dazu wurden diverse Systeme implementiert, um direkt im Unity-Editor Spielwelten erstellen und bearbeiten zu können. Anspruchsvoll war insbesondere die Umrechnung der normalen Koordinaten zu isometrischen Koordinaten. Diese Umrechnung wurde durch einige Parameter beeinflusst, welche nur durch praktische Tests ermittelt werden konnten. Schlussendlich wurden jedoch alle Funktionen implementiert. Damit konnte die Spielwelt erfolgreich realisiert werden. Die Anforderungen wurden bestmöglich umgesetzt. Einige der Komponenten verfügen nicht über den kompletten, im Vorfeld geplanten Funktionsumfang. Das hat vor allem zeitliche Gründe, denn die zusätzliche Implementierungsarbeit hätte definitiv den Rahmen dieser Arbeit überschritten. Der nach der Fertigstellung des Spiels folgende Usability-Test brauchte die Erkenntnis,

dass die Navigation im Inventar noch verbessert werden kann. Auch der auf dem PC erfolgte Performance-Test verlief erfolgreich. Dabei konnte beobachtet werden, dass der größere Bildschirm einen erheblichen Pluspunkt darstellt. Dank der größeren Darstellungsfläche lassen sich UI-Elemente besser erkennen. Auch die Performance war auf dem PC erheblich besser als auf mobilen Systemen. Verbesserungspotenzial gibt es unter anderem noch bei den Menüanimationen, die aktuell nicht vorhanden sind. Auch das Erstellen eines kompletten Dialoges könnte mit einer besseren Basis effizienter und einfacher erstellt werden. Weitere Individualisierungsmöglichkeiten der Grundfläche eines Spiellevels würde die Attraktivität des Spiels erhöhen. Die allgemeine Performance des Karteneditors ist Steigerungsfähig. Auch die Tile-Objekte könnten durch erweiterte Funktionen besser nutzbar sein, beispielsweise indem sie sich rotieren lassen. Zukünftige Versionen könnten unter anderem Verbesserungen an den Menüs mitbringen, etwa ein erneuertes UI-System. Auch komplett neue Komponenten können integriert werden, so etwa ein Erfolgs- und Statistiksistem, eine dynamische Spielwelt oder eine künstliche Intelligenz, die bei Duellen einen Computergegner steuert. Diverse Komfortsysteme werden den Weg in das Spiel finden. So soll der Spieler für das Spielen auf multiplen Plattformen nur einen Speicherstand benötigen, welchen er wenn nötig von speziellen Servern herunterladen kann.

Insgesamt wurde ein sehr gutes Rollenspiel mit Potenzial zur Weiterentwicklung realisiert.

Glossar

I

Inventar

Das Inventar dient zur Verwaltung von Gegenständen des Spielers.

M

Mozos

Mozos sind die Kreaturen, die die Spielwelt nebst den Menschen bewohnen.

P

Prefab

Als Prefab bezeichnet man Spielobjekte, welche im Vorfeld von einem Entwickler erstellt werden und dann per Code erzeugt und vervielfältigt werden können.

Q

Quest / Questlog

Ein Quest ist eine Aufgabe, welche der Spieler erledigen kann.

T

Tile

Ein Tile ist eine Komponente des Level-Designers. Es sind kleine Kacheln, aus welchen man die Spielwelt erzeugen kann.

Literaturverzeichnis

- [1] R. Koncewicz, „significant-bits.com,“ 2009. [Online]. Available: <http://www.significant-bits.com/a-laymans-guide-to-projection-in-videogames>. [Zugriff am 10 Juli 2016].
- [2] „ev111426.wordpress.com,“ 22 October 2014. [Online]. Available: <https://ev111426.wordpress.com/category/research/>. [Zugriff am 26 07 2016].
- [3] „itwissen.info,“ [Online]. Available: <http://www.itwissen.info/definition/lexikon/Physik-Engine-physics-engine.html>. [Zugriff am 19 10 2015].
- [4] „unity3d.com,“ Unity Technologies, [Online]. Available: <https://unity3d.com/>. [Zugriff am 19 10 2015].
- [5] „unrealengine.com,“ Epic Games, [Online]. Available: <https://www.unrealengine.com/what-is-unreal-engine-4>. [Zugriff am 19 10 2015].
- [6] „frostbite.com,“ Frostbite, [Online]. Available: <http://www.frostbite.com/>. [Zugriff am 19 10 2015].
- [7] „cryengine.com,“ CryTek, [Online]. Available: <http://cryengine.com/>. [Zugriff am 19 10 2015].
- [8] W. Pryjda, „winfuture.de,“ 25 04 2014. [Online]. Available: <http://winfuture.de/news,81414.html>. [Zugriff am 19 10 2015].
- [9] „winfuture.de,“ 03 03 2015. [Online]. Available: <http://winfuture.de/videos/Software/Unreal-Engine-4-ist-ab-sofort-fuer-jeden-frei-fuer-eigene-Projekte-nutzbar-14161.html>. [Zugriff am 19 10 2015].
- [10] P. Steinlechner, „golem.de,“ 20 03 2014. [Online]. Available: <http://www.golem.de/news/crytek-cryengine-kuenftig-im-10-euro-abo-1403-105260.html>. [Zugriff am 19 10 2015].
- [11] M. Maciej, „giga.de,“ GIGA, 10 10 2014. [Online]. Available: <http://www.giga.de/konsolen/playstation-4/tipps/ps4-controller-anmelden-und-ausschalten-so-geht-s/>. [Zugriff am 11 11 2015].
- [12] U. Leonidas, „3dcenter.org,“ 29 06 2015. [Online]. Available: <http://www.3dcenter.org/news/umfrage-auswertung-welche-aufloesung-und-aa-setting-wird-ueblicherweise-verwendet-2015>.

[Zugriff am 11 11 2015].

- [13] T. Heuzaroth, „Die Welt,“ Die Welt, 15 02 2015. [Online]. Available: <http://www.welt.de/wirtschaft/webwelt/article137460997/Deutsche-nicht-scharf-auf-superscharfes-Fernsehen.html>. [Zugriff am 11 11 2015].
- [14] „digitalverlegen.de,“ digitalverlegen.de, 2015. [Online]. Available: <http://digitalverlegen.de/bildschirmaufloesung/smartphone-aufloesung/>. [Zugriff am 11 11 2015].
- [15] A. Sowards, „infinigeek.com,“ 2015. [Online]. Available: <http://infinigeek.com/mobile-vs-console-vs-pc-gaming/>. [Zugriff am 11 11 2015].
- [16] „unity3d.com,“ Unity Technologies, 2015. [Online]. Available: <http://docs.unity3d.com/ScriptReference/Application-platform.html>. [Zugriff am 09 12 2015].
- [17] „unity3d.com,“ Unity Technologies, 2015. [Online]. Available: <http://docs.unity3d.com/ScriptReference/RuntimePlatform.html>. [Zugriff am 09 12 2015].
- [18] P. Hauer, „<http://www.philippbauer.de>,“ 2010. [Online]. Available: <http://www.philippbauer.de/study/se/design-pattern/singleton.php#beschreibung>. [Zugriff am 09 12 2015].
- [19] aweryn, „RPG Maker VX Resource planet,“ 18 Julie 2011. [Online]. Available: <https://vxresource.wordpress.com/category/resources/tilesets/>. [Zugriff am 06 August 2016].
- [20] U. Technologies, „Unity Documentation,“ Technologies, Unity, [Online]. Available: <https://docs.unity3d.com/Manual/class-AudioClip.html>. [Zugriff am 06 09 2016].
- [21] Bérenger, „<http://wiki.unity3d.com>,“ 2014. [Online]. Available: <http://wiki.unity3d.com/index.php/ProceduralPrimitives>. [Zugriff am 19 August 2016].
- [22] C. Seifert, „Spiele entwickeln mit Unity 5,“ Carl Hanser Verlag GmbH, 2015, p. 631.
- [23] J. Chittesh, Das Unity-Buch: 2D- und 3D-Spiele entwickeln mit Unity 5, Deutschland: dpunkt.verlag GmbH, 2015.
- [24] Selfhtml-Wiki, „Selfhtml-Wiki,“ [Online]. Available: <http://wiki.selfhtml.org/wiki/XML>. [Zugriff am 25 08 2015].
- [25] Microsoft, Microsoft, [Online]. Available: <https://msdn.microsoft.com/en->

us/library/w56d4y5z.aspx. [Zugriff am 30 08 2015].

- [26] J. Lebosquain, „stackoverflow.com,“ 16 06 2010. [Online]. Available: <http://stackoverflow.com/a/3056242/5266228>. [Zugriff am 13 09 2015].
- [27] LightSource, „<http://answers.unity3d.com>,“ 31 03 2013. [Online]. Available: <http://answers.unity3d.com/questions/428433/what-is-a-good-target-frame-rate-for-a-pc-game.html>. [Zugriff am 09 09 2016].
- [28] K. S. Tauberbischofsheim, „Kaufmännische Schule Tauberbischofsheim,“ [Online]. Available: http://www.kstbb.de/informatik/oo/03/3_1_Kapselung.html. [Zugriff am 03 09 2016].
- [29] „docs.unity3d.com,“ Technologies, Unity, 2016. [Online]. Available: <https://docs.unity3d.com/Manual/index.html>. [Zugriff am 20 August 2016].
- [30] „games4win.com,“ 27 4 2006. [Online]. Available: http://games4win.com/up/landstalker-the-treasures-of-king-nole_3s.jpg. [Zugriff am 06 08 2016].
- [31] J. Bray, „globalgeeknews.com,“ 6 Dezember 2015. [Online]. Available: <http://www.globalgeeknews.com/2015/12/06/super-mario-bros-theme-music-has-lyrics/>. [Zugriff am 06 August 2016].
- [32] R. Hardgrit, „superadventuresingaming.blogspot.de,“ 24 September 2012. [Online]. Available: <http://superadventuresingaming.blogspot.de/2012/09/the-legend-of-zelda-link-to-past-snes.html>. [Zugriff am 06 August 2016].

Bearbeiter der Abschnitte

Abschnittsname	Seiten [von-bis]	Bearbeiter
1 Einleitung	1-2	Julius Forner
2 Überblick über das Spielekonzept	3-4	Michael Müller
3.1 Anforderungen an das Spiel	5-9	Julius Forner
3.2 Wahl der grafischen Umsetzung	9-14	Michael Müller
3.3 Wahl des Entwicklungsframeworks	15-21	Julius Forner
3.4 Anpassungen für das Multiplattformkonzept	22-25	Julius Forner
4.1 Ablauf und Aufbau der Planung und Realisierung	25-26	Julius Forner
4.2.1 Steuerung	26-28	Julius Forner
4.2.2 Navigationssystem	28-36	Julius Forner
4.2.3 Mozo	37-42	Julius Forner
4.2.4 Lokalisierung	42-45	Julius Forner
4.2.5 Zwischenstand des Spiels – das Hauptmenü	46-48	Julius Forner
4.3.1 Karteneditor	49-95	Michael Müller
4.3.2 Bewegung und Animation	95-102	Michael Müller
4.3.3 Audio	103-107	Michael Müller
4.3.4 Zwischenstand des Spiels – die Spielwelt	107-109	Michael Müller
4.4.1 Aufgabensystem	110-124	Julius Forner
4.4.2 Inventar	125-145	Julius Forner
4.4.3 Dialogsystem	146-152	Julius Forner
4.4.4 Zwischenstand des Spiels – die UI-Systeme	153-154	Julius Forner
5. Evaluierung und Test der Anwendung	155-160	Julius Forner
6. Zusammenfassung	161-162	Michael Müller

Eidesstattliche Erklärung zur Masterarbeit (Julius Forner)

Ich versichere, die Masterarbeit selbstständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben.

Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Merseburg, den _____

Eidesstattliche Erklärung zur Masterarbeit (Michael Müller)

Ich versichere, die Masterarbeit selbstständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben.

Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Merseburg, den _____