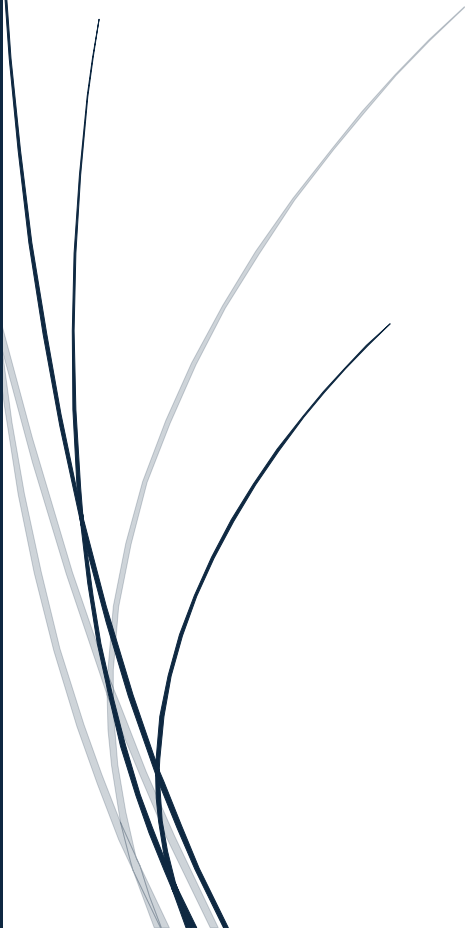


Mittwoch, 18. Dezember
2024

Kenopsia

Report Anfängerpraktikum



Skrobanek, Timo
Schäfer, Christian
Costeniuc, Andrei

Inhalt

1 - Einführung und Grundlagen	3
2 – Dokumentation / Architektur	4
2.1 – Storyboard.....	4
2.2 – Use Case.....	5
2.3 – Use Cases Beschreibung	6
2.4 – State Machine – Diagramm	8
2.5 – Anforderungsanalyse.....	12
2.6 - Doxygen.....	15
3 – Implementierung.....	15
3.1 - Allgemeines	15
3.2 - Zuhause	16
3.3 - Im Büro.....	16
3.4 - Im Traum	17
3.5 - Zurück im Büro	17
3.6 - Hauptmenü.....	17
4 – Development Process.....	18
4.1 - Anfangsprozess.....	18
4.1.1 - Meetings	18
4.1.2 - Prototype Development.....	19
4.1.3 - Split Task	19
4.1.4 - Detailed Requirement Analysis for T	20
4.1.5 - Tests for T	20
4.1.6 - Code and Documentation for T	21
4.1.7 - Test T and Code Review for T	21
4.2 - Prozessaktualisierung.....	22
4.2.1 - Meetings	22
4.2.2 - Prototype Development.....	23
4.2.3 - Split Task	23
4.2.4 - Detailed Requirement Analysis for T	23
4.2.5 - Tests for T	24
4.2.6 - Code and Documentation for T	24
4.2.7 - Test and Code Review for T	25
4.2.8 - Grafik des aktualisierten Prozesses	26
5 - Fazit	27

5.1 – Selbsteinschätzung	27
5.2 – Zeitplan.....	27
5.3 - Learnings.....	28
6 – Appendix	29
Requirements Document	29
Storyboard	29
Klassendiagramm.....	29
Usability Test.....	29
Doxygen Dokumentation	29

1 - Einführung und Grundlagen

“The goal of the internship is to learn agile development processes similar to SCRUM and to further optimize the own process in order to achieve a 0-error level.”¹

[Spiel downloaden](#)

Für das Projekt wurde die Spiel-Engine Unity benutzt.

Die wichtigsten Grundbegriffe in Unity sind **Szenen**, **GameObjects**, **Komponenten** und **Scripts**. Ein Spiel in Unity wird in Szenen unterteilt, welche verschiedene Teile des Spiels darstellen, so zum Beispiel ein Level, ein Hauptmenü oder eine Benutzeroberfläche. In unserem Spiel stellen die Szenen Spielbereiche oder Menüs dar. Jede Szene kann eine Vielzahl von GameObjects enthalten, weshalb eine Szene auch gleichzeitig ein Level oder Menü sein kann.

GameObjects sind grundlegende Bausteine der Engine. Ein GameObject kann alles Mögliche darstellen, wie ein Charakter, ein Modell, ein Licht oder auch ein UI-Element (beispielsweise ein Button oder Text). Diese GameObjects sind leer, bis ihnen Komponenten hinzugefügt werden, die ihr Verhalten bestimmen.

Komponenten sind spezialisierte Module, die einem GameObject hinzugefügt werden, um es funktional zu machen. Eine Transform-Komponente bestimmt z.B. die Position eines GameObjects in der Szene, während eine Renderer-Komponente dafür sorgt, dass es sichtbar wird. Andere wichtige Komponenten umfassen sogenannte Collider, die zur physischen Interaktion dienen und Rigidbody, die für die physikbasierte Bewegungen sorgen.

Scripts sind in Unity erstellte C#-Dateien, die das Verhalten der GameObjects steuern. Sie werden oft verwendet, um benutzerdefinierte Logik zu implementieren, wie etwa die Steuerung eines Spielers oder das Auslösen einer Aktion, wenn auf einen Button geklickt wird.

Der **Szenenbaum** fasst alle GameObjects der Szene zusammen. Das heißt, der Szenenbaum ist eine hierarchische Darstellung aller Objekte, die in einer Szene vorhanden sind. In Unity wird dies als **Hierarchy**-Fenster dargestellt. Dort sieht man die Struktur der GameObjects und ihre Parent-Child-Beziehung.

Unity bietet allgemein eine Vielzahl von **vorgefertigten Komponenten** und **Klassen**, die für das Projekt verwendet wurden. Dazu gehören unter anderem:

- **GameObject, Transform, Rigidbody, Collider** (für die grundlegenden Strukturen und Physik-Interaktionen)
- **UI-Elemente** wie **Button, Text, Image** (zur Erstellung von Benutzeroberflächen)
- **Animator, AudioSource** und **Camera** (für Animationen, Audio und Kamera-Steuerung).

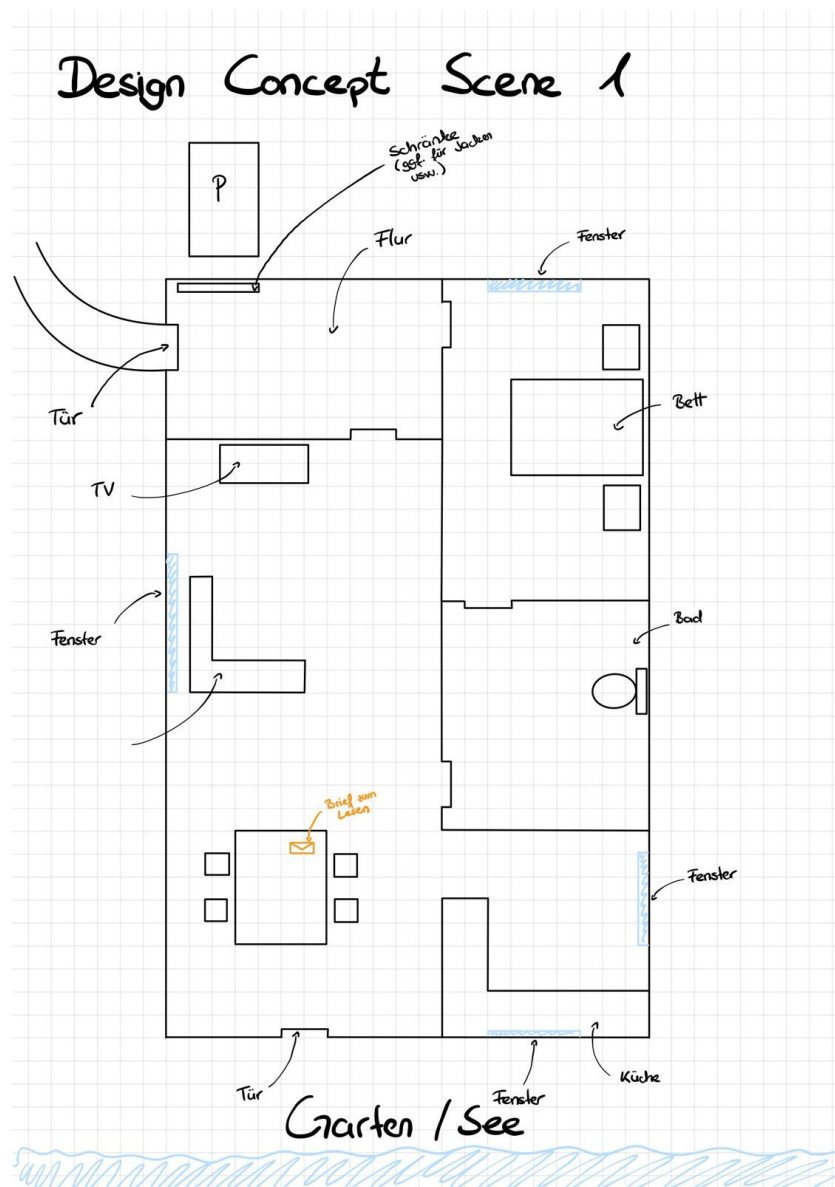
Darüber hinaus verwendet Unity **Packages**, die als zusätzliche Frameworks oder Erweiterungen dienen können. Zum Beispiel bietet das **Cinemachine-Package** erweiterte Kamerasteuerungsfunktionen oder das **Post-Processing Stack** für erweiterte graphische Effekte. Für unser Projekt wurde jedoch kein spezifisches Package oder Framework verwendet.

2 – Dokumentation / Architektur

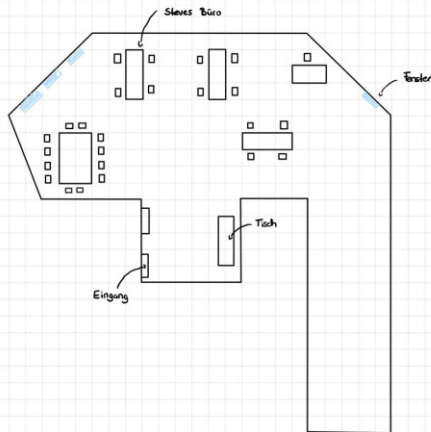
2.1 – Storyboard

Eine anfängliche Darstellung des Spiels ist im Storyboard festgehalten. Dies beinhaltet die einzelnen Szenen des Spiels, das heißt, die gewünschte Oberfläche sowie spezifische Spielmechaniken, die später implementiert werden sollten.

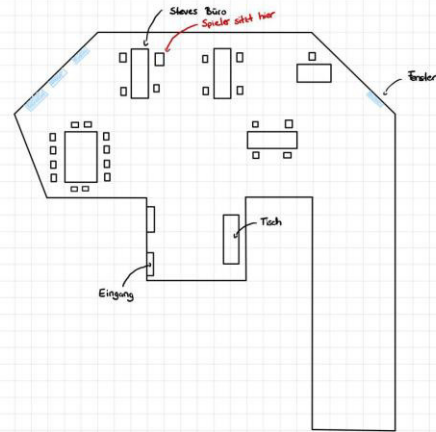
In diesem Storyboard werden die verschiedenen Szenen des Spiels detailliert dargestellt. Jede Szene entspricht einem Abschnitt des Spiels und enthält Informationen zur Atmosphäre, Spielmechanik und optionalen Elementen. Zum Beispiel startet die erste Szene im Zuhause des Spielers, wo er unterschiedliche Aufgaben erledigen muss, bevor er das Haus verlassen kann. Die Szenenübergänge sind klar definiert und beinhalten wichtige Details, wie die Wechsel von Aufgaben im Haus zu Szenen außerhalb.



Design Concept Scene 2



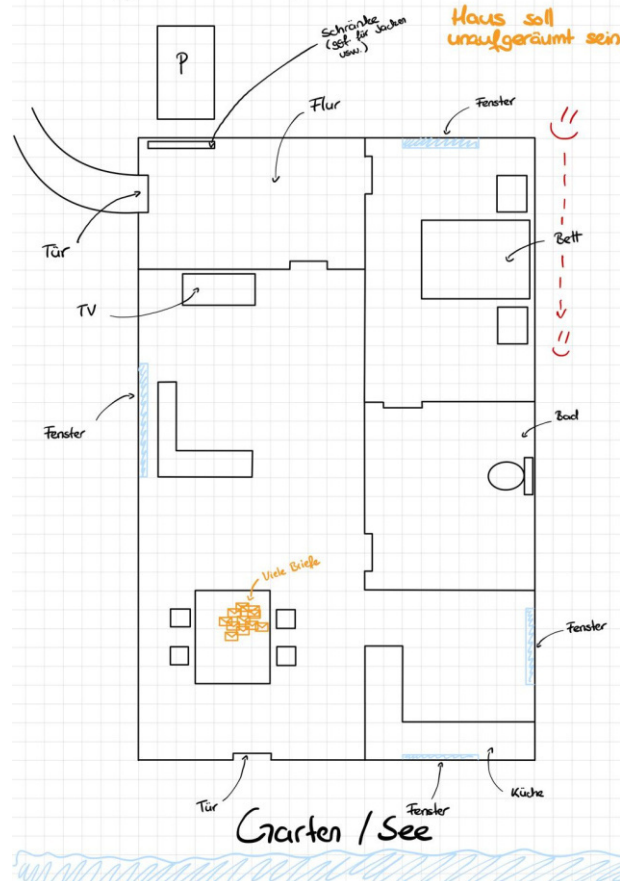
Design Concept Scene 3



2.2 – Use

Nachdem das wurde, diente es Ableitung der Use Cases werden Interaktionen des Spiel zu Darüber hinaus den Use Cases die Features ableiten, enthalten soll, und

Design Concept Scene 4



Case

Storyboard erstellt als Grundlage zur Cases. Diese Use verwendet, um die Spielers mit dem definieren. lassen sich aus Funktionen und die das Spiel diese können in

der Anforderungsanalyse weiter spezifiziert werden.

Um sicherzustellen, dass die Use Cases diesen Anforderungen entsprechen und ihren Zweck erfüllen, wurden Use Case Diagramme für die jeweiligen Szenen erstellt. Der Akteur in den Use Case Diagrammen ist der Nutzer des Systems, beziehungsweise des Spiels. Die Use Cases sind die Interaktionsszenarien zwischen Akteur und System. Außerdem können sie Beziehungen untereinander haben, welche durch Assoziationen dargestellt sind.

2.3 – Use Cases Beschreibung

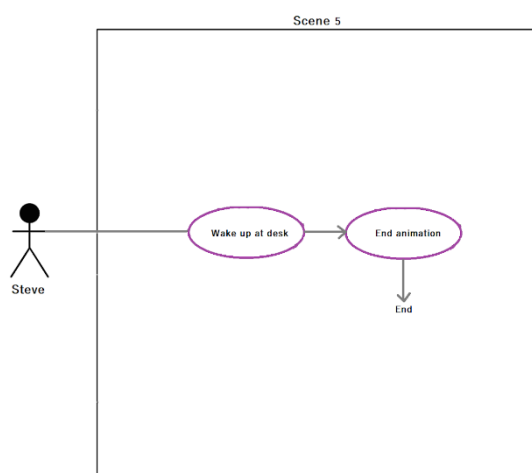
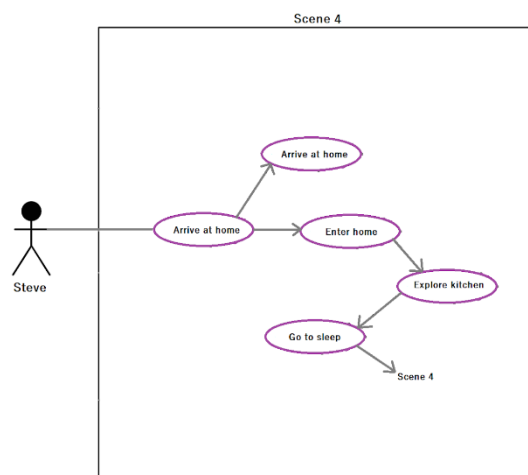
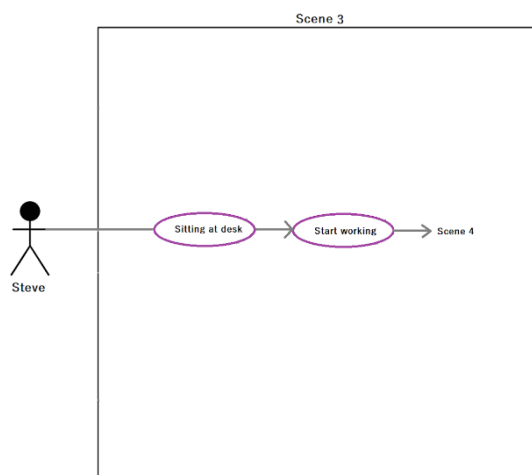
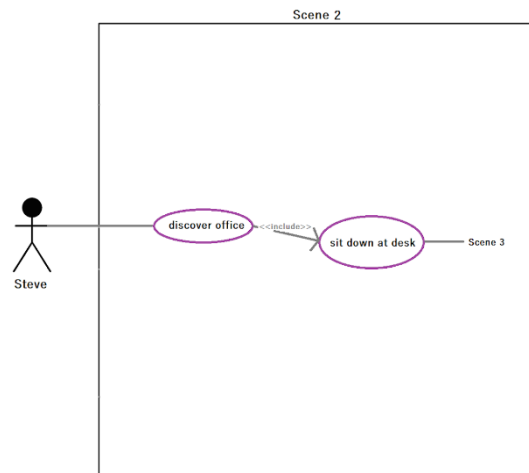
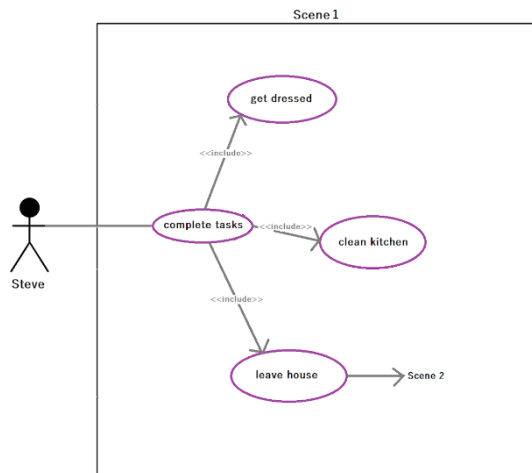
In diesem psychologischen Horrorspiel beginnt die Hauptfigur Steve, ein geschiedener Mann, der mit innerer Zerrissenheit zu kämpfen hat, seinen Tag mit dem Aufwachen auf seinem Sofa. Der Spieler, der die Welt durch Steves Augen sieht, muss alltägliche Aufgaben erledigen, wie sich anziehen, den Müll in seinem Haus aufräumen oder die Küche putzen, bevor er das Haus verlassen und zur Arbeit fahren kann.

Die zweite Szene wechselt in Steves Büro, wo der Spieler das Büro erkunden kann oder sich an seinen Schreibtisch setzen kann, wodurch die Handlung fortgesetzt wird.

In der dritten Szene sitzt der Spieler am Schreibtisch und arbeitet. Durch Klicken auf den Bildschirm schläft die Spielfigur ein und tritt in eine surreale Traumwelt ein.

In der vierten Szene kehrt der Spieler in Steves Haus zurück, aber jetzt wird die Atmosphäre noch unheimlicher. Der Spieler wird mit unheimlichen Gestalten bereits außerhalb des Hauses konfrontiert. Wie z.B. ein dunkler lachender Smiley. Dann muss er das Haus erkunden, in dem viele optische Illusionen auftreten, wie z.B. viele Kameras, die an den Wänden hängen oder Tische und Stühle, die an der Decke kleben. Beim Erkunden der Küche erscheint eine Figur, die die geschiedene Frau des Spielcharakters Steve darstellen soll. Der Spieler muss als nächstes ins Schlafzimmer gehen und sich dort ins Bett legen, um wieder aus der Traumwelt zu entkommen.

Das Spiel endet mit einer fünften und letzten Szene: einer Zwischensequenz, in der Steve an seinem Schreibtisch aufwacht und von seinem Chef angemotzt wird. Eine Kameraanimation soll darstellen, dass Steve gerade wieder aufgewacht ist. Die Zwischensequenz wird schwarz und beendet das Spiel mit einer eindringlichen Note.

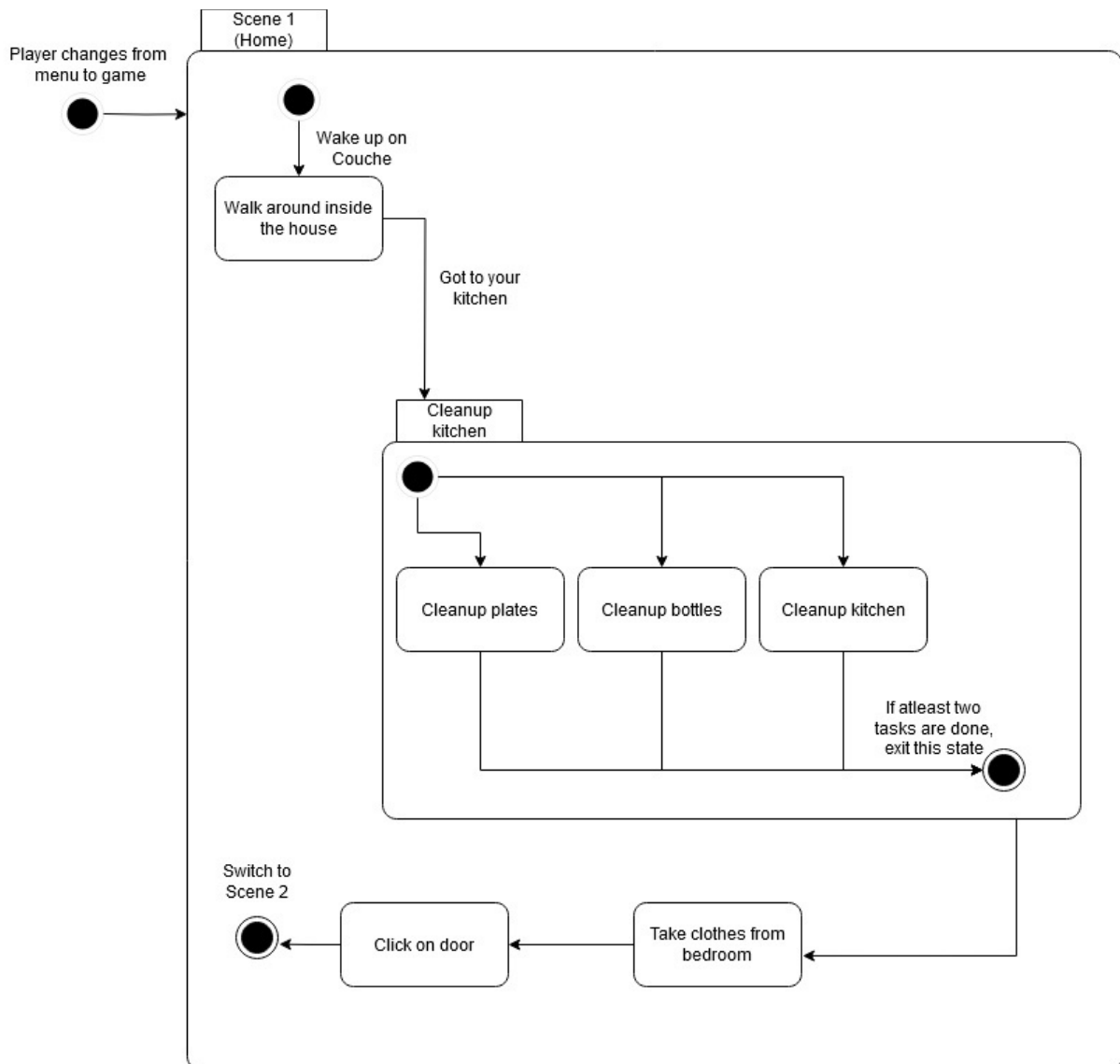


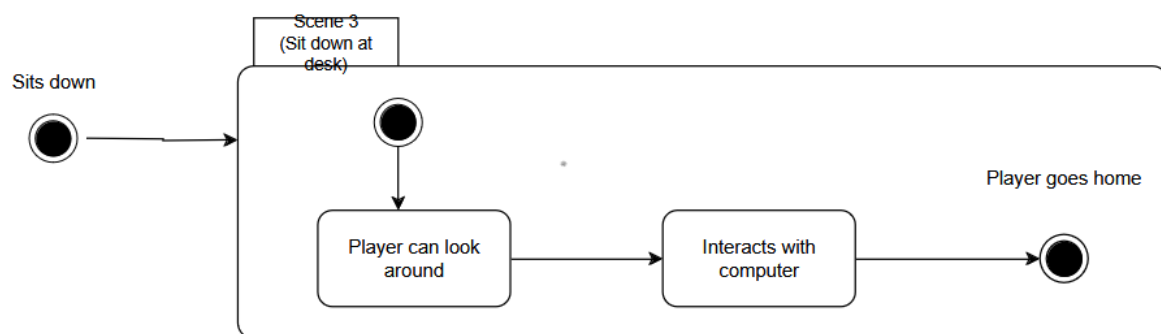
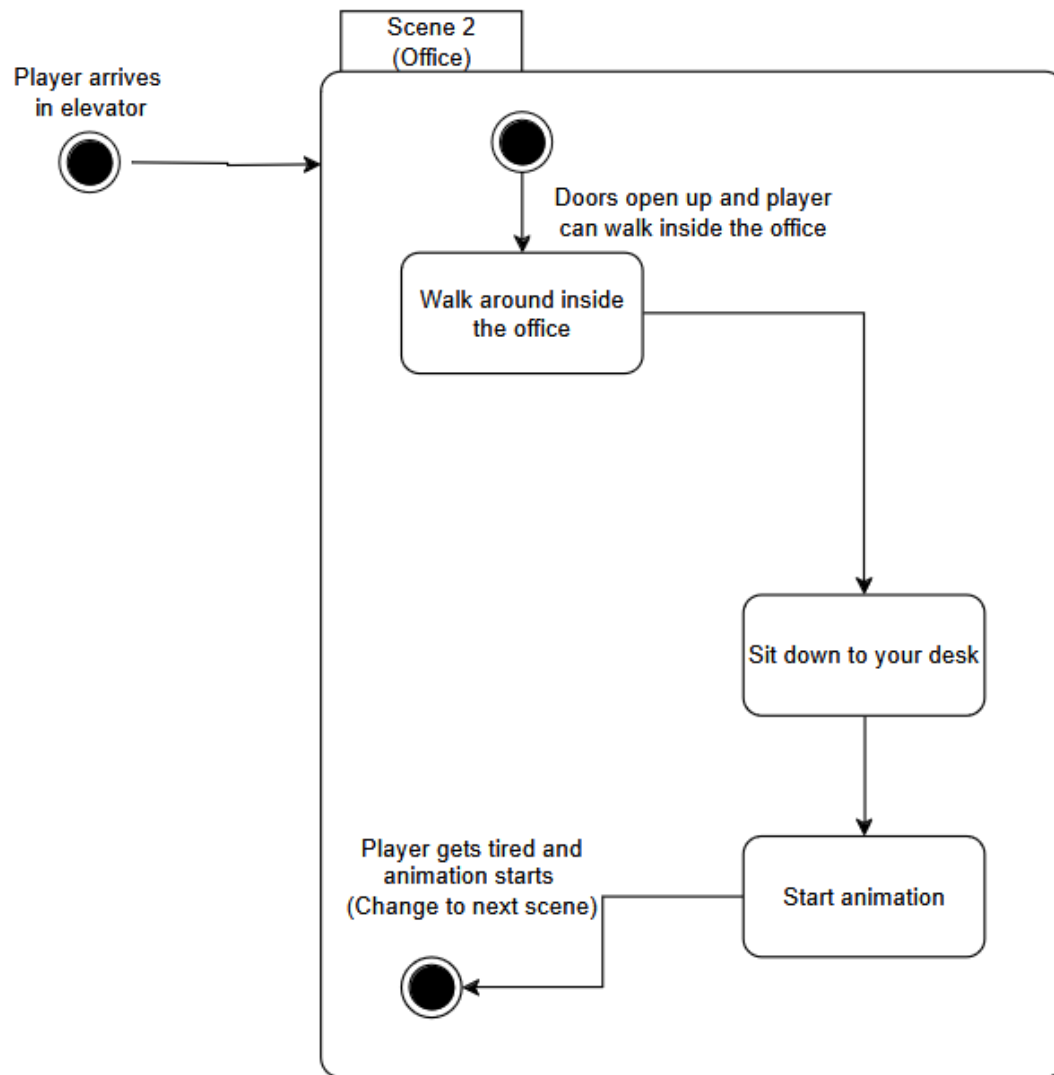
2.4 – State Machine – Diagramm

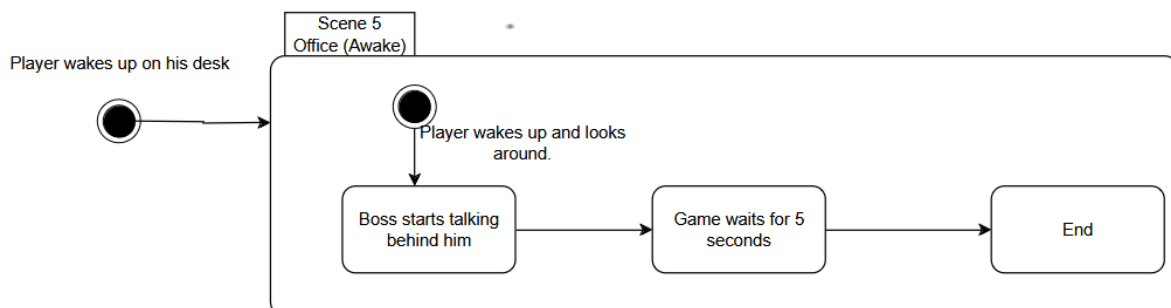
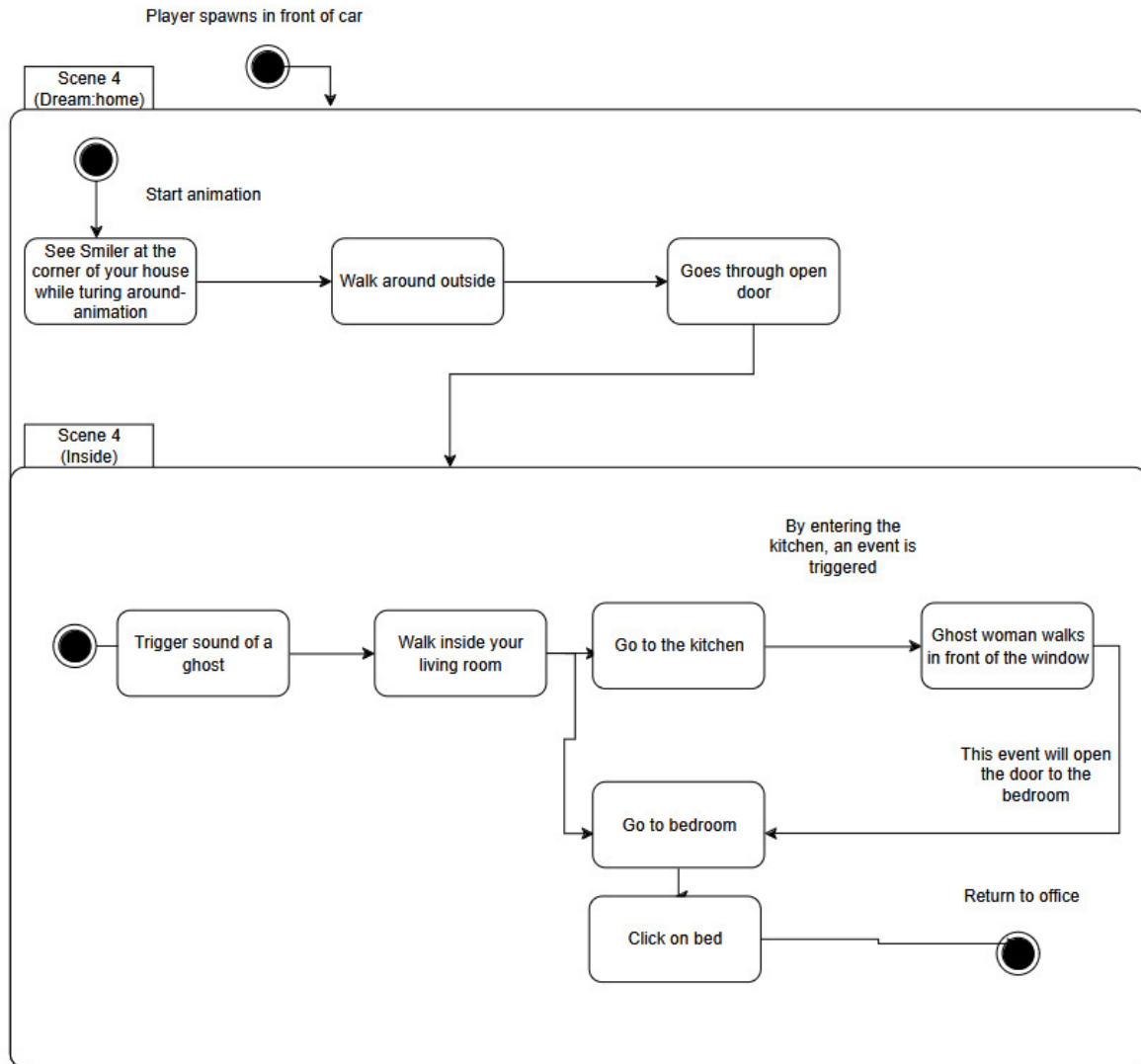
Für unser Spiel haben wir State Machine Diagramme verwendet, um den Ablauf und die Interaktionen in den einzelnen Szenen zu steuern. Eine State Machine ist ein Modell, das den aktuellen Zustand eines Objekts oder Systems beschreibt und darstellt, wie es auf verschiedene Ergebnisse oder Aktionen reagiert. Zustandsübergänge werden durch bestimmte Bedingungen oder Ereignisse ausgelöst, die im Spiel auftreten, wie zum Beispiel das Lösen von Aufgaben oder das Interagieren mit der Umgebung.

Wir haben uns dazu entschieden, State Machines zu verwenden, da es uns geholfen hat, die Interaktionslogik des Spiels zu strukturieren und sicherzustellen, dass das Verhalten des Spiels konsistent und nachvollziehbar bleibt. Dadurch konnten wir sehr einfach den Fortschritt des Spielers durch verschiedene Zustände (wie „Aufwachen“, „Küche aufräumen“, „Haus verlassen“) modellieren und zu kontrollieren, wann und wie der Spieler von einem Zustand in den nächsten übergeht.

Die Entwicklung der State Machines basierte direkt auf den Use Case Diagrammen des Spiels, die die einzelnen Szenen und Interaktionen beschreiben. Für jede Szene gibt es spezifische Aufgaben und Zustände, die den Fortschritt des Spielers steuern. Zum Beispiel in Szene 1, wo der Spieler in seinem Zuhause Aufgaben erledigen muss, gibt es Zustände wie „Anziehen“ oder „Küche aufräumen“. Sobald diese Aufgaben abgeschlossen sind, wird der Zustand „Haus verlassen“ freigeschaltet, der den Übergang in die nächste Szene ermöglicht.







2.5 – Anforderungsanalyse

Zusammenfassung der Funktionalen Anforderungen

Das Spiel basiert auf einem interaktiven Erlebnis, bei dem der Spieler in einer First-Person-Perspektive durch verschiedene Umgebungen navigiert und Aufgaben löst. Der Spieler hat die Möglichkeit, „Objekte“ zu untersuchen und mit ihnen zu interagieren, indem bestimmte Tasten oder die Maus verwendet werden. Jede Interaktion im Spiel löst spezifische Ereignisse aus, die den Fortschritt des Spiels beeinflussen.

Die Spielmechanik basiert auf einer konstanten Bewegung innerhalb der Spielwelt, bei der der Spieler frei durch die Umgebungen navigieren kann. Der Fortschritt des Spiels wird in Form von Zuständen innerhalb des Arbeitsspeichers gespeichert.

Eine besondere Anforderung ist die Implementierung eines funktionalen SaveGameManagers, der dafür sorgt, dass der Fortschritt des Spielers in regelmäßigen Abständen gesichert wird.

Nun folgt eine detaillierte Beschreibung der funktionalen Anforderungen und deren Umsetzung in den verschiedenen Bereichen des Spiels.

Generelles

Perspektive und Spielmechanik

Das Spiel ist in 3D designed, verwendet perspektivische Darstellungen und hat dementsprechend auch die Bewegungsrichtungen des Spielers zu beachten.

Der Spieler kann sich durch die Verwendung der Tasten W, A, S, D in alle Richtungen bewegen, während bestimmte Aktionen, wie das Aufnehmen von Gegenständen oder das Interagieren mit Objekten durch Mausklicks oder Tastendruck ausgelöst werden.

Springen wurde nicht implementiert, jedoch wird eine Bewegung durch z.B. unebenes Terrain auch in y-Richtung möglich.

Interaktion mit der Umgebung

Das Spiel bietet eine Vielzahl an interaktiven Elementen, die der Spieler erkunden kann. Von alltäglichen Aufgaben, wie das Aufräumen der Küche oder das Einsammeln von Briefen, bis hin zu entscheidenden Momenten, in denen der Spieler innerhalb der Geschichte voranschreitet, entwickelt sich die Geschichte durch die Aktionen, die der Spieler ausführt. Jede Interaktion hat das Potential, die Umgebung zu verändern oder den Fortschritt in der Geschichte zu beeinflussen.

Szenen und Zustandsübergänge

Das Spiel ist in verschiedenen Szenen unterteilt, die der Spieler nacheinander durchläuft. In jeder Szene gibt es spezifische Aufgaben, die abgeschlossen werden müssen, um den Übergang zur nächsten Szene zu ermöglichen. Die Zustände des Spiels werden durch eine State-Machine verwaltet, die den Fortschritt des Spielers in Abhängigkeit von seinen Aktionen verfolgt.

Bildschirmanforderungen

Das Spiel wird nur im Querformat unterstützt und kann im Vollbildmodus gespielt werden.

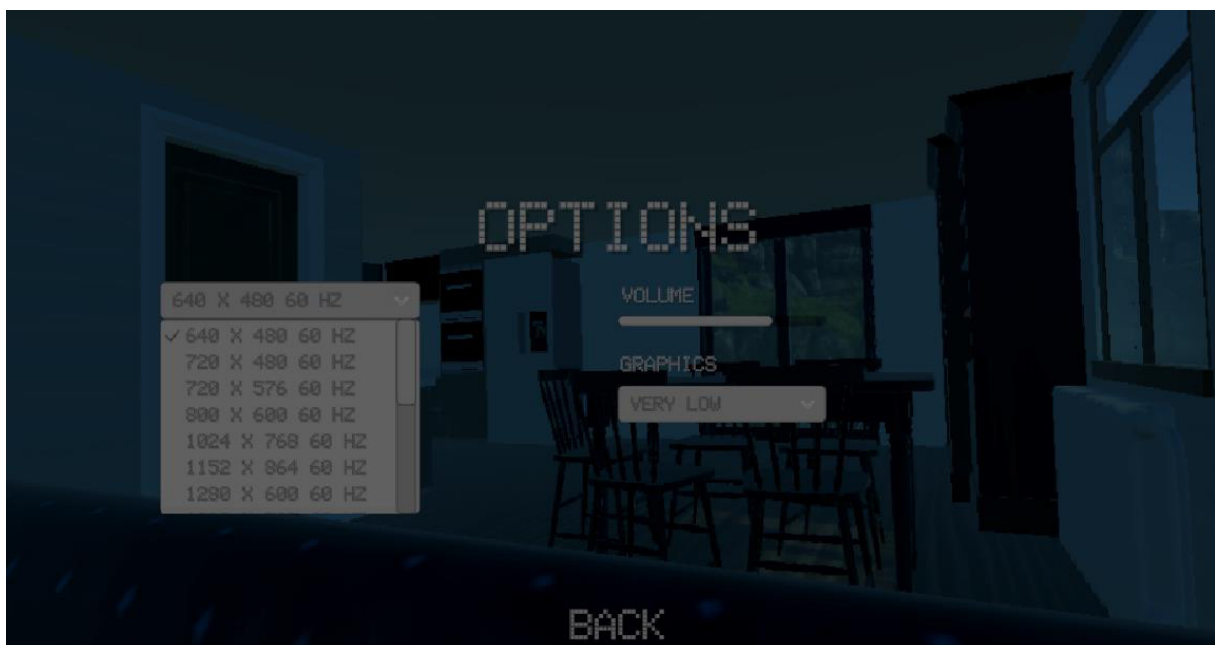
Willkommen-Screen / Hauptmenü

Das Programm startet mit einem Willkommen-Screen, beziehungsweise mit dem Hauptmenü. Das Hauptmenü beinhaltet Buttons für den Spielstart, Optionen und fürs Beenden des Programms.

Optionen

In den Optionen kann die Lautstärke des Spiels angepasst werden, und diese Änderungen werden sofort übernommen. Die Einstellungen bleiben über die gesamte Spieldauer hinweg erhalten.

Darüber hinaus verfügt das Spiel über eine automatische Erkennung der angeschlossenen Monitore. Beim Start des Spiels wird die maximale Auflösung jedes Bildschirms ermittelt, sodass der Spieler die höchste, von seinem Monitor unterstützte Auflösung auswählen kann.



Spielanforderungen

Hauptmenü

Beim Start des Spiels erscheint unser Hauptmenü (s. oben für Beschreibung). Sobald auf „Play“ gedrückt wird, geht die erste Szene mit einer Animation des Spielcharakters los.

Esc-Menü

Während des Spiels kann man jederzeit mit der Taste „T“ ein Zwischenmenü erreichen, was das Spiel pausiert.

Szenenauswahl

Der Spieler durchläuft mehrere Szenen, die jeweils eine eigenständige Umgebung und Aufgaben beinhalten. Jede Szene hat spezifische Aufgaben, die der Spieler abschließen muss, um zur nächsten Szene überzugehen.

Interaktion mit der Umgebung

Der Spieler kann Objekte in der Umgebung durch Mausklicks oder die „E“-Taste auf der Tastatur untersuchen und mit ihnen interagieren. Bestimmte Objekte lösen Ereignisse aus, die den Fortschritt im Spiel beeinflussen.

Objekt-Interaktionen und Hinweise

Verschiedene Objekte und Details in der Umgebung bieten Hinweise, die den Spieler tiefer in die Geschichte eintauchen lassen. So können beispielsweise verstreute Gegenstände Hinweise auf die Vergangenheit des Hauptcharakters geben.

Aufgaben- und Zustandsverwaltung

Jede Szene enthält spezifische Aufgaben, die der Spieler erledigen muss. Durch eine State-Machine wird der Spielfortschritt erfasst und auf die Aktionen des Spielers reagiert. Aufgaben, wie das Aufräumen der Küche oder das Einsammeln von Briefen führen zu neuen Zuständen, die den Übergang zur nächsten Szene ermöglichen.

Eingabe

Der Spieler bewegt sich mit den Tasten W, A, S, D durch die von den Entwicklern festgelegten Umgebungen. Die Geschwindigkeit kann mit der Shift-Taste geändert werden. Hierbei wird zwischen Gehen und Rennen gewechselt.

Bezug zur restlichen Dokumentation

Bezug zum Storyboard

Das Storyboard dient nicht nur als Grundlage für die Anforderungsanalyse, sondern beeinflusst sie auch maßgeblich. Während des Planungsprozesses für die einzelnen Szenen des Spiels wurden viele wichtige Aspekte deutlich, die in den Anforderungen integriert werden mussten. Die visuelle Darstellung des Spielablaufs im Storyboard ermöglichte es uns, die benötigten Funktionen und Features abzuleiten, um das Spielerlebnis zu optimieren.

Bezug zum Use Case Diagramm

Die Use Case Diagramme waren ein zentraler Bestandteil der Anforderungsanalyse, da es uns ermöglichte, die verschiedenen Interaktionen und Funktionen des Spiels klar zu definieren. Während der Erstellung des Diagramms haben wir unterschiedliche Aspekte angesprochen, die für das Spiel erforderlich sind. Die Use Case Diagramme halfen uns, die Hauptfunktionen und Interaktionen des Spiels zu verstehen und sicherzustellen, dass alle relevanten Anwendungsfälle berücksichtigt wurden.

Bezug zum Klassendiagramm

Die Anforderungsanalyse half bei der Erstellung des Klassendiagramms, da darauf Informationen abgeleitet werden konnten.

Entwicklung

Da die Anforderungsanalyse ein dynamischer Prozess ist, der sich im Laufe der Entwicklung eines Computerspiels ständig weiterentwickelt, gab es über die Zeit auch Veränderungen in unserer Anforderungsanalyse. Im Folgenden sind die wichtigsten Anpassungen aufgeführt, die wir vorgenommen haben:

Wechsel von Unreal Engine 5 zu Unity

Ursprünglich haben wir uns vorgenommen, das Spiel in Unreal Engine 5 zu entwickeln. Aufgrund von Herausforderungen, wie unklarer Dokumentation und Schwierigkeiten mit dem Blueprint-System, haben wir uns entschieden, auf Unity umzusteigen. Unreal Engine 5 entsprach nicht unseren Anforderungen, Unity hingegen bot uns eine stabilere und benutzerfreundlichere Entwicklungsumgebung, eine größere Community und Zugriff auf zahlreiche hilfreiche Ressourcen, was den Entwicklungsprozess deutlich verbesserte.

Anpassung des Storyboards

Während der Entwicklung stellten wir fest, dass der ursprüngliche Umfang des Spiels ehrgeizig war, daher reduzierten wir das Storyboard und fokussierten uns auf die wesentlichen Elemente der Geschichte. Diese Entscheidung ermöglichte uns, die wichtigsten Spielinhalte effizienter zu entwickeln.

2.6 - Doxygen

Einführung der Doxygen-Dokumentation

Um den Überblick über den Code zu verbessern, führten wir Doxygen für die Dokumentation ein. Zuvor hatten wir Schwierigkeiten, den Zusammenhang zwischen verschiedenen Code-Teilen schnell zu verstehen, was die Weiterentwicklung erschwerte. Durch Doxygen konnten wir den Code effizienter nachvollziehen und neue Methoden nahtloser integrieren.

Siehe Appendix.

3 – Implementierung

Unser Spiel ist im Allgemeinen ein Zustandsautomat. Nach diesem Prinzip haben wir auch die Inhalte der einzelnen Szenen, ausgenommen dem Menü, implementiert. Jeder Zustand steuert Elemente, die im Spiel eine zeitliche Abhängigkeit haben (Bsp. Intro-Animationen).

3.1 - Allgemeines

Für unseren Zustandsautomaten haben wir folgende Zustände definiert.

- SCENE 1
- SCENE1_INTRO_ANIMATION

- SCENE1_CLEANUP_DONE
- SCENE1_NOT_DRESSED
- SCENE1_COMPLETED
- SCENE2
- SCENE2_COMPLETED
- SCENE3
- SCENE3_OUTRO
- SCENE3_OUTRO_DONE
- SCENE4_INTRO
- SCENE4
- SCENE4_GHOST_APPEAR
- SCENES

Diese werden innerhalb der Klasse State, der mit StateManager zusammenarbeitet, verwaltet.

Eine weitere Komponente, für unser Spiel bildet die Interaktion mit verschiedenen Spielobjekten - in Unity GameObject genannt – welche wir durch eine Klasse möglich machen, die uns Grundfunktionalitäten dafür bietet.

Zusätzlich gibt es die Logik für die Navigation im Spiel, welche die Spielerbewegung, Kamerarotation und auch die Kollision mit Objekten beinhaltet.

Außerhalb des eigentlichen Spiels gibt es noch Skripte zum Navigieren im Menü, sowie dem Vornehmen von Einstellungen oder auch das einfache Anzeigen von Text im und außerhalb vom Spiel.

3.2 - Zuhause

Das Spiel startet mit der Szene 1 “Zuhause” und beginnt direkt mit einer Animation, welche dem Spieler signalisiert, dass der Charakter gerade aufgestanden ist. Nachdem diese fertig ist, wird dem Spieler die Steuerung freigegeben und dadurch vom Zustand SCENE1_INTRO_ANIMATION zu SCENE1 gewechselt. Nun kann sich der Spieler frei in dem vorgegebenen Bereich bewegen und mit verschiedenen Gegenständen interagieren.

Räumt der Spieler seine Wohnung auf, wechselt das Spiel in Zustand SCENE1_CLEANUP_DONE woraufhin, der Spieler eine weitere Aufgabe visuell in Form eines Textes erhält.

Hat der Spieler alle Aufgaben erledigt, wechselt das Spiel in den Zustand SCENE1_COMPLETED. Dadurch werden Animationen gestartet und die nächste Szene geladen.

3.3 - Im Büro

In der Szene vom Büro gibt es zwei Zustände. SCENE2 ist der Zustand, welcher dem Spieler erlaubt sich frei zu bewegen und mit Gegenständen zu interagieren. SCENE2_COMPLETED wechselt in die dritte Szene, sobald der Spieler mit seinem Arbeitsplatz interagiert.

Diese Szene ist von der Implementierung sehr ähnlich zur ersten Szene und hat keine neuen Herausforderungen in der Implementierung mit sich gebracht.

3.4 - Im Traum

Nun kommen wir zu der Szene mit dem größten Implementierungsaufwand. In dieser Szene ist der Spieler in der Traumwelt zuhause.

Die Anforderungen an die Szene haben verschiedene Elemente in die Programmierung einfließen lassen, die in vorherigen Szenen nicht implementiert wurden, weshalb unter anderem das Interaktionssystem zu diesem Zeitpunkt grundlegend geändert werden musste, um erweiterte Funktionalität zu gewährleisten.

In dieser Szene gab es nämlich nicht nur Interaktionen durch Anklicken der Maus, sondern ebenfalls durch Position des Spielers und Rotation der Kamera.

Dazu mussten auch hier wieder die Zustände beachtet werden damit Animationen zur Richtigen Zeit ausgeführt werden.

3.5 - Zurück im Büro

Die letzte Szene im Spiel hat keinerlei Interaktionsmöglichkeiten mit Ausnahme der Kopfbewegung.

Die Szene behandelt die Situation, dass der Spieler aus seinem Albtraum aufgewacht ist und nun an seinem Schreibtisch sitzt, wo der Chef von hinten auf ihn einredet, dass er doch nicht bei der Arbeit schlafen kann. Wenn dieser Monolog fertig ist, beendet das Spiel indem zu den Credits gewechselt wird.

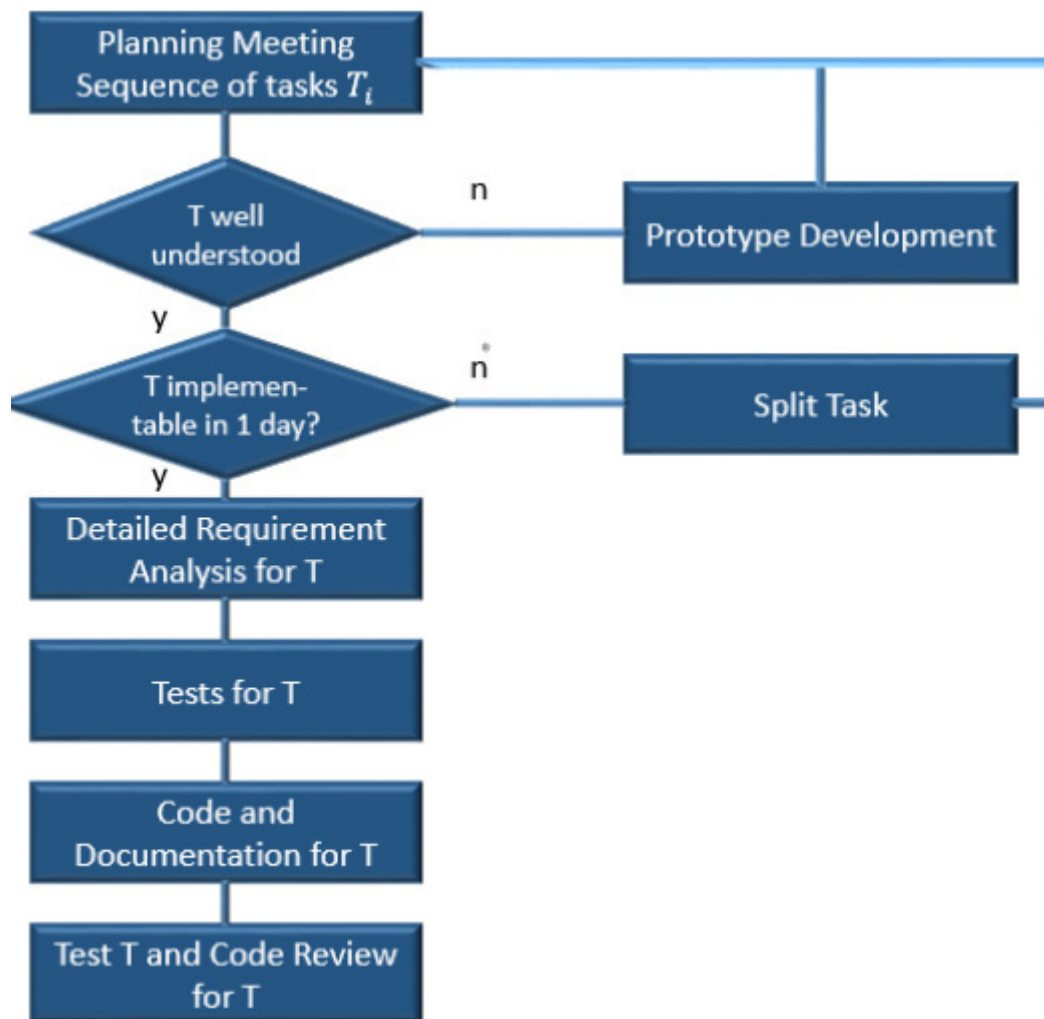
3.6 - Hauptmenü

Im Hauptmenü wurden nur die grundlegendsten Funktionen implementiert. Es soll möglich sein, das Spiel zu starten und auch Einstellungen vorzunehmen. Ebenfalls soll das Spiel beendet werden können.

Beim Starten des Spiels soll jedoch immer die erste Szene geladen werden, da wir schlussendlich auf die Implementierung von Save-Games verzichten.

Außerdem gibt es ein extra "Optionen"-Fenster, welches in der Anforderungsanalyse genauer beschrieben wurde.

4 – Development Process



4.1 - Anfangsprozess

In der Anfangsphase des Projekts sind wir streng nach dem vorgegebenen Development-Schema vorgegangen. Im Verlauf des Projekts haben wir dann Anpassungen und Aktualisierungen vorgenommen, die das Development-Schema effektiver für die Entwicklung unseres Spiels machen.

4.1.1 - Meetings

Rolle:

Ein wesentlicher Bestandteil des Entwicklungsprozesses ist die Koordination der einzelnen Aufgaben und die Vorgehensweise während des Projekts. In den Meetings werden genau diese Fragen beantwortet. Es werden eine Reihe an Tasks definiert und priorisiert, die dann nacheinander individuell abgearbeitet werden. Unklarheiten werden besprochen und es wird sichergestellt, dass das Team weiterhin zielstrebig und erwartungsgemäß arbeitet.

Umsetzung über Videokonferenz:

Durch regelmäßige wöchentliche Meetings auf Discord haben wir uns über den aktuellen Stand des Projekts austauschen können. Wir haben Aufgaben neu priorisiert und diese dann Personen zugeordnet, die diese dann abarbeiten. Mit dem Projektmanagement-Tool Trello haben wir alle erarbeiteten Entscheidungen zeitnah dokumentiert, um einen visuellen Überblick des Projekts beizubehalten. Die grundlegendsten Ziele unserer Meetings war es, sicherzustellen, dass jedes Teammitglied alle Aufgaben versteht und dessen Meinungen und Vorschläge einbringen konnte. Wir haben auch stets die Meetings dazu genutzt, unseren Prozess anhand von aufgetretenen Fehlern anzupassen und zu optimieren.

Umsetzung in Präsenz:

Jedes zweite Meeting haben wir in Präsenz abgehalten. Dafür haben wir uns zu dritt in einem Seminarraum im Mathematikon getroffen und vor Ort an komplizierten Themen gearbeitet. Wir haben den Vorteil der direkten Kommunikation meistens ausgenutzt, indem wir über aufgekommene Implementierungsfehler oder komplizierten Problemen in Präsenz, an meist nur einem Laptop, diskutiert haben.

4.1.2 - Prototype Development

Rolle:

Bei komplizierten oder unklaren Anforderungen bzw. Aufgaben können Prototypen erstellt werden, um sicherzustellen, dass die Umsetzung der Implementierung klar verstanden wurde und keine Missverständnisse auftreten. Es können außerdem auch verschiedene Lösungsansätze einfach und effektiv ausprobiert werden. Somit kann man ohne große Arbeit die beste Implementierung ausarbeiten.

Umsetzung:

Wir haben Prototypen genutzt, um die Implementierung grundlegender Funktionen zu erarbeiten. Es war uns sehr wichtig, die häufig aufkommenden Mechaniken im Spiel effektiv programmieren zu können. Vor allem die Interaktionen mit Gegenständen, das Bewegen des Spielers oder das Eventsystem wurden mit Hilfe von Prototypen optimiert und vollkommen nachvollzogen. Dadurch konnten wir sicherstellen, dass diese Funktionen fehlerfrei und effektiv implementiert wurden.

4.1.3 - Split Task

Rolle:

Es kann auch vorkommen, dass Anforderungen zu komplex oder zu aufwendig sind, um sie innerhalb eines Tages zu implementieren. In solch einem Fall ist es empfehlenswert, sich für eine Aufteilung der Anforderung in einzelne Teile, zu entscheiden. Somit kann sichergestellt werden, dass die einzelnen Aufgaben nicht einen Arbeitstag (circa 8 Stunden) überschreiten.

Umsetzung:

Um festzustellen, ob eine Aufgabe mehr als einen Tag in Anspruch nimmt, haben wir die Aufgabe im Team nochmals genauer angeschaut. Wir haben aufgezählt, was alles für die Realisierung

getan werden muss. Die Einschätzung des zeitlichen Aufwands haben wir basierend auf vergangenen Erfahrungen getroffen.

4.1.4 - Detailed Requirement Analysis for T

Rolle:

Nachdem klargestellt wurde, dass eine Aufgabe innerhalb eines Tages implementierbar ist, wird bei der Detailed Requirement Analysis überprüft, ob alle Details in den Anforderungen der Aufgabe vorhanden sind. Das muss noch vor dem Beginn der Implementierung geschehen, damit dann später im Verlauf keine Missverständnisse entstehen. Denn weitere Details während der Implementierung auszuarbeiten ist ineffizient.

Umsetzung:

Wir haben uns alle Informationen zu der jeweiligen Aufgabe aus den Anforderungen angeschaut und hinterfragt, ob man allein damit die Aufgabe so implementieren kann, wie wir uns das auch vorgestellt haben. Außerdem haben wir überprüft, ob die Anforderungen eindeutig und ohne mögliche Missverständnisse formuliert sind. Falls es Einwände gab, haben wir diese umgehend in den Anforderungen ergänzt bzw. konkretisiert.

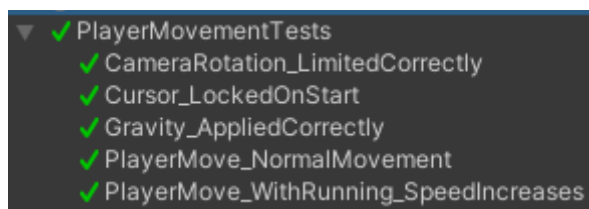
4.1.5 - Tests for T

Rolle:

Mit der Entwicklung von Tests kann die Funktionalität der Tasks garantiert werden. Es muss vor der Implementierung noch hinterfragt werden, was für Anforderungen die Task erfüllen muss und wie diese dann getestet werden. Ohne Tests hätte man keine Garantie, dass die geplante Umsetzung der Task auch tatsächlich allen Anforderungen gerecht wird.

Umsetzung:

Zum Testen der Funktionen haben wir uns in erster Linie für Unit-Tests entschieden. Wir haben uns zu einer bestimmten Funktion Test Cases überlegt und eine passende Testumgebung aufgebaut. Innerhalb dieser Testumgebung haben wir die im Spiel verwendete Funktion ausgeführt und im Nachhinein überprüft, ob die Testwerte den erwarteten entsprechen.



*Testergebnisse der Unittests in Unity

Für ein besseres User Interface haben wir für die ersten beiden Szenen Usability-tests erstellt, die wir bei bekannten Testpersonen durchgeführt haben. Mit Hilfe dieser Benutzererfahrungen konnten wir überprüfen, ob die Non-Functional-Requirements entsprechend erfüllt wurden. Der Usability-test wurde mit einer einfachen Google-

Forms Umfrage durchgeführt und eine Person aus dem Entwicklerteam ist mit der Testperson in Präsenz durch das Spiel gegangen.

4.1.6 - Code and Documentation for T

Rolle:

Nachdem Anforderungen entsprechend formuliert wurden und Tests für die Task erstellt wurden, folgt die Implementierung der Task. Das ist der eigentliche produktive Schritt, der im Prozess am Ende das Produkt darstellt. Hierbei geht es darum, dass das aktuelle Projekt um eine bestimmte Funktion iterativ erweitert wird, bis das Endprodukt entstanden ist.

Die Dokumentation des Codes ist wichtig, um die Nachvollziehbarkeit und Wartbarkeit des Codes zu verbessern. Es sollte ohne große Umstände möglich sein, dass noch in ferner Zukunft der Code auch von externen Entwicklern nachvollzogen oder abgeändert werden kann. Dabei spielt eine gründliche Dokumentation eine wichtige Rolle. Programme, wie Doxygen oder einfache Diagramme, wie z.B. Klassendiagramme helfen bei der Verständlichkeit des Codes.

Umsetzung:

Die Implementierung haben wir entweder allein oder als Paar gemacht, abhängig von der Komplexität der Aufgabe. Bei Meetings in Präsenz hat sich das Pair-Programming angeboten, da es deutlich leichter ist als über Videokonferenzen. Wir haben Unity als Engine benutzt und in C# mit Visual Studio Code als Entwicklungsumgebung programmiert.

Für die allgemeine Dokumentation haben wir Doxygen verwendet und die daraus resultierende index.html Datei auf einem Server gehostet, um immer einen aktualisierten Zugriff auf die Dokumentation von überall aus zu haben. Wir haben zu jeder Funktion in C# einen Doxygen-Kommentar hinzugefügt und die config-Datei unseren Bedürfnissen entsprechend angepasst. Konkret bei der Programmierung haben wir hauptsächlich mit den Diagrammen gearbeitet, da diese sehr einfach den Sachverhalt dargestellt haben und die Klassen-basierte Vorstellung der Diagramme in C# leicht zu übernehmen war.

4.1.7 - Test T and Code Review for T

Rolle:

Nach einem gesamten Durchlauf des Development-Process für eine Task T wird am Ende nochmal eine Überprüfung der Umsetzung durchgeführt. Hierbei wird sichergestellt, dass der Code auch, den Anforderungen entsprechend, erstellt wurde. Außerdem wird geschaut, dass die Tests auch alle Testfälle abdecken, die Regeln des Clean Codes eingehalten wurden und die Dokumentation vollständig ist. Wenn Fehler oder Probleme gefunden wurden, muss im nächsten Schritt überlegt werden, an welcher Stelle im Prozess dieser Fehler vorgekommen ist und wie man solche Fehler in Zukunft verhindern kann.

Umsetzung:

Wir haben beim Code Review die simple Regel angewandt, dass derjenige, der den Code geschrieben hat, selbst nochmal im Anschluss drüber schaut und hinterfragt, ob der Code alle

Anforderungen erfüllt. Nur bei Zweifeln oder Unsicherheiten wurde das Team gefragt. Bei dem Testreview haben wir als gesamtes Team die Tests erstellt und konnten somit auch alle zusammen sicherstellen, dass wir alle Testfälle abgedeckt haben. Für die Dokumentation des Codes haben wir auch immer alle über das resultierende Doxygen Dokument drüber geschaut.

4.2 - Prozessaktualisierung

Nach dem Anfangsprozess haben wir revidiert, was für Erfolge und Probleme beim Befolgen des Development-Schemas auftraten. Anhand dessen haben wir das Schema erweitert und für die effektivste und fehlerfreiste Entwicklung gesorgt.

4.2.1 - Meetings

Erfolge:

Durch die wöchentlichen Meetings konnte man sich im Team immer genug austauschen und aufgekommene Probleme zusammen lösen. Wir konnten regelmäßig sicherstellen, dass alle auf dem aktuellen Stand sind und die weitere Vorgehensweise verstanden haben. Durch die Benutzung von Trello konnten wir einzelne Aufgaben übersichtlich anordnen oder Teammitgliedern zuordnen und somit diese nach Prioritäten/Themen abarbeiten.

Besonders zu erwähnen sind die Meetings in Präsenz. Wir haben klare Vorteile feststellen können, wenn man physisch miteinander interagieren konnte. Das Arbeiten zu zweit an einem Laptop, wie z.B. beim Pair-Programming hat uns absolut überzeugt und deshalb haben wir das immer bei komplizierteren Aufgaben praktiziert.

Probleme:

Es gab allerdings auch manchmal Meetings, in denen nicht viel erarbeitet werden konnte. Bei Meinungsunterschieden oder Missverständnissen musste viel Zeit und Energie aufgebracht werden, um diese Probleme zu lösen. Vor allem bei Fragen in der Implementierung oder technischen Problemen war die Kommunikation sehr ineffektiv. Trotz Bildschirmübertragung oder langen Erklärungen über Videokonferenzen konnten Probleme nicht so leicht gelöst werden, wie in Präsenz. Ein weiteres Problem war die falsche Einschätzung von Relevanz einer Anforderung. Wir haben teilweise zu detailliert an einer Anforderung gearbeitet, obwohl diese Entscheidungen zu diesem Zeitpunkt noch absolut irrelevant waren. Die ausgearbeiteten Details mussten im Laufe der Entwicklung verworfen werden, da diese nicht mehr mit dem Gesamtkonzept übereingestimmt haben.

Prozessaktualisierungen:

Wir haben uns gegen die Verwendung von festen Zeitlimits für die Diskussion über ein bestimmtes Thema entschieden, da manche Anforderungen tatsächlich mehr Zeit benötigt haben und trotzdem noch effektiv ausgearbeitet wurden. Da wir mit drei Mitgliedern kein großes Team waren, haben wir einfach alle stets hinterfragt, wie relevant die aktuelle Diskussion auch für die nächsten Aufgaben ist. Bei dem Verdacht, dass es zu sehr ins Detail geht, wurde das ganz klar kommuniziert. Außerdem haben wir in Trello die Aufgaben nach Priorität sortiert und somit wussten wir zu jeder Zeit, auf was wir uns konzentrieren müssen. Um umständliche Erklärungen über Videokonferenzen zu vermeiden, haben wir im Voraus bei komplizierten Funktionen oder ähnliches eine Konferenz in Präsenz geplant, damit wir dann vor Ort das Problem effektiv angehen können.

4.2.2 - Prototype Development

Erfolge:

Das Erstellen der Prototypen hat uns nicht nur einen Vorteil in der Implementierung gebracht, sondern hat es uns überhaupt erst ermöglicht, die Funktionsweisen der Mechaniken zu verstehen. Als größtenteils Anfänger in der Spieleentwicklung konnten wir uns die Umsetzung z.B. eines Eventsystems nicht einmal vorstellen. Um dieser Überforderung entgegenzuwirken, haben wir uns strikt daran gehalten, Prototypen zu erstellen. Und mit der Zeit haben wir uns dann im Kleinen eine Vorstellung der Umsetzung machen können. Das war jedes Mal eine riesige Erleichterung, mit wenig Aufwand und so simpel, wie möglich, die Funktionsweisen zu verstehen.

Probleme:

Manche Prototypen waren nicht umfassend genug. Wir haben versucht, diese so simpel, wie möglich, zu gestalten, was aber in der tatsächlichen Implementierung des Spiels nicht möglich war. Dort wurde es viel umfangreicher und somit auch komplizierter. Es entstanden neue unvorhergesehene Probleme, die den Arbeitsfluss aufgehalten haben.

Prozessaktualisierungen:

Diesen Prozess zu optimieren, war einer der schwersten, da hier unvorbereitete Probleme entstanden, bei einem Thema, wo wir uns nicht auskannten. Wir konnten also nur das Risiko minimieren, indem wir uns intensiver mit der Umsetzung der Mechaniken im Großen auseinandergesetzt haben. Das haben wir meistens mittels Youtube-Tutorials gemacht, da dort anhand von Beispielen manche Mechaniken erklärt wurden.

4.2.3 - Split Task

Erfolge:

Das Aufsplitten der Aufgaben hat immer gut funktioniert. Wir haben uns im Team am Anfang jeder Sitzung Zeit genommen und erst mal analysiert, was als nächsten Schritt abgearbeitet werden muss. Beim Aufzählen der einzelnen Schritte, konnte schon ziemlich offensichtlich erkannt werden, ob die Aufgabe an einem Tag machbar war oder nicht. Mit der Zeit wurden die Einschätzungen umso präziser, da wir diese auf Basis von vergangenen Erfahrungen getroffen haben.

4.2.4 - Detailed Requirement Analysis for T

Erfolge:

Indem wir zusammen als Team die Anforderungen überprüft haben, konnten wir uns effektiv verschiedene Ansichten der Implementierung überlegen und konnten diese potenziellen Missverständnisse durch klarere Formulierungen verhindern.

Probleme:

Wie auch unten in Kapitel 5 bei Learnings erwähnt, hatten wir hier große Probleme. Wir haben uns sehr schwergetan, sich in einen außenstehenden Entwickler hineinzusetzen. Es gab unklare Aussagen, die nach unserem Erachten eindeutig waren. Manche Missverständnisse haben wir erst bei der Implementierung bemerkt und dann mussten wir mitten in der Entwicklung nochmal Steps zurückgehen und die Details konkretisieren, was sehr viel Zeit und Aufwand gekostet hat.

Prozessaktualisierungen:

Nach den ersten Rückschlägen haben wir uns dazu entschieden, manche Anforderungen nochmal zu überarbeiten mit starkem Fokus auf unklare Aussagen. Wir haben versucht, uns die extremen Missverständnisse auszudenken, um diese dann aus dem Weg zu räumen.

4.2.5 - Tests for T

Erfolge:

Durch das Erstellen der Tests haben wir einen deutlich besseren Einblick bekommen, ob die Anforderungen für die Task klar formuliert wurden, da die Tests auf den Anforderungen basiert haben. Nur mit verständlichen Anforderungen konnten wir passende Tests schreiben.

Probleme:

Zu Beginn war es noch sehr unsicher, für welche Tasks wir Tests erstellen sollen und falls ja, wie intensiv wir die Tests gestalten sollen. Ohne Erfahrung von den weiteren Schritten, die später erst folgen, wie z.B. die Implementierung, war es sehr schwer einzuschätzen, ob uns die erstellten Tests tatsächlich später helfen werden.

Prozessaktualisierungen:

Bevor wir die eigentlichen Tests implementiert haben, haben wir erst mal eine allgemeine Liste aller benötigten Tests verfasst, um einen Überblick zu bekommen. Das hat Struktur in die Abdeckung der Testfälle gebracht und somit konnten wir viel leichter abschätzen, ob wir alle Fälle abgedeckt haben. Erst danach haben wir mit der eigentlichen Implementierung begonnen.

4.2.6 - Code and Documentation for T

Erfolge:

Speziell beim Interaktionssystem haben wir große Erfolge gemacht, da wir direkt von Anfang an eine funktionale Grundlage erstellt haben, die Interaktion jeglicher Art mit Objekten realisiert. Diese konnten wir nutzen, um spezifischere Interaktionen effektiv und schnell zu programmieren. Das hat einwandfrei funktioniert.

Ein weiterer Erfolg war das Übertragen von Prototypen in das fertige Spiel. Bei oft genutzten Funktionen, wie Interaktion oder PlayerMovement haben wir die Prototypen sehr detailliert und weitreichend erstellt. Somit konnten wir ohne großen Aufwand die Prototypen in den Produktiv-Code integrieren und die finale Realisierung war nahezu problemlos.

Die ausführlichen Diagramme, die wir im Voraus erstellt hatten, haben sehr stark geholfen, den Überblick während der Implementierung beizubehalten. Doxygen hat teilweise wie eine

Überprüfung gedient. Wir haben unsere erstellten Klassendiagramme mit den generierten aus Doxygen verglichen und konnten so leicht überprüfen, ob die Programmierung auch unseren ursprünglichen Vorstellungen entsprach. Das war immer eine Befriedigung, wenn das generierte Diagramm dem zuvor geplanten entsprach.

Probleme:

Wie bereits in 2.5 Anforderungsanalyse beschrieben, hatten wir zu Beginn der Programmierung sehr große Probleme mit Unreal Engine 5, da die neuste Version noch keine klare Dokumentation hatte und noch nicht viele Projekte im Internet damit erstellt wurden. Das hat es uns erschwert, als Anfänger in diese Funktionalitäten einzusteigen. Hinzu kommt, dass das Blueprint-System nicht passend für unser programmier-basiertes Projekt war und wir somit von Unreal Engine 5 ablassen mussten.

Was die Dokumentation mit Doxygen angeht, haben wir ungefähr 3 Wochen zu spät angefangen. Wir hatten bereits den ersten Produktiv-Code geschrieben, allerdings ohne Doxygen-Kommentare. Als wir diese dann nachholen wollten, mussten wir zuerst nochmal den früheren Code nachvollziehen und uns zurückerinnern. Das hat sich als sehr ineffizient rausgestellt.

Prozessaktualisierungen:

Nach mehreren Wochen, in denen wir versucht haben, mit Unreal Engine 5 klarzukommen, haben wir uns dazu entschieden, zu Unity zu wechseln, da dort die Dokumentation ausführlicher war und es als einsteigerfreundlicher beschrieben wurde. Dieser verspätete Umschwung erwies sich im Nachhinein als richtige Entscheidung, da wir uns sehr leicht in Unity einarbeiten konnten und damit sehr schnell Erfolge machen konnten.

Sobald wir dann die Doxygen Dokumentation nachbearbeitet hatten, haben wir immer parallel zur Implementierung auch die Doxygen-Kommentare verfasst. Das wurde ein fester und relevanter Punkt in der Implementierung, damit wir immer auf dem aktuellen Stand sind und zu jeder Zeit den aktuellen Code nachvollziehen können.

4.2.7 - Test and Code Review for T

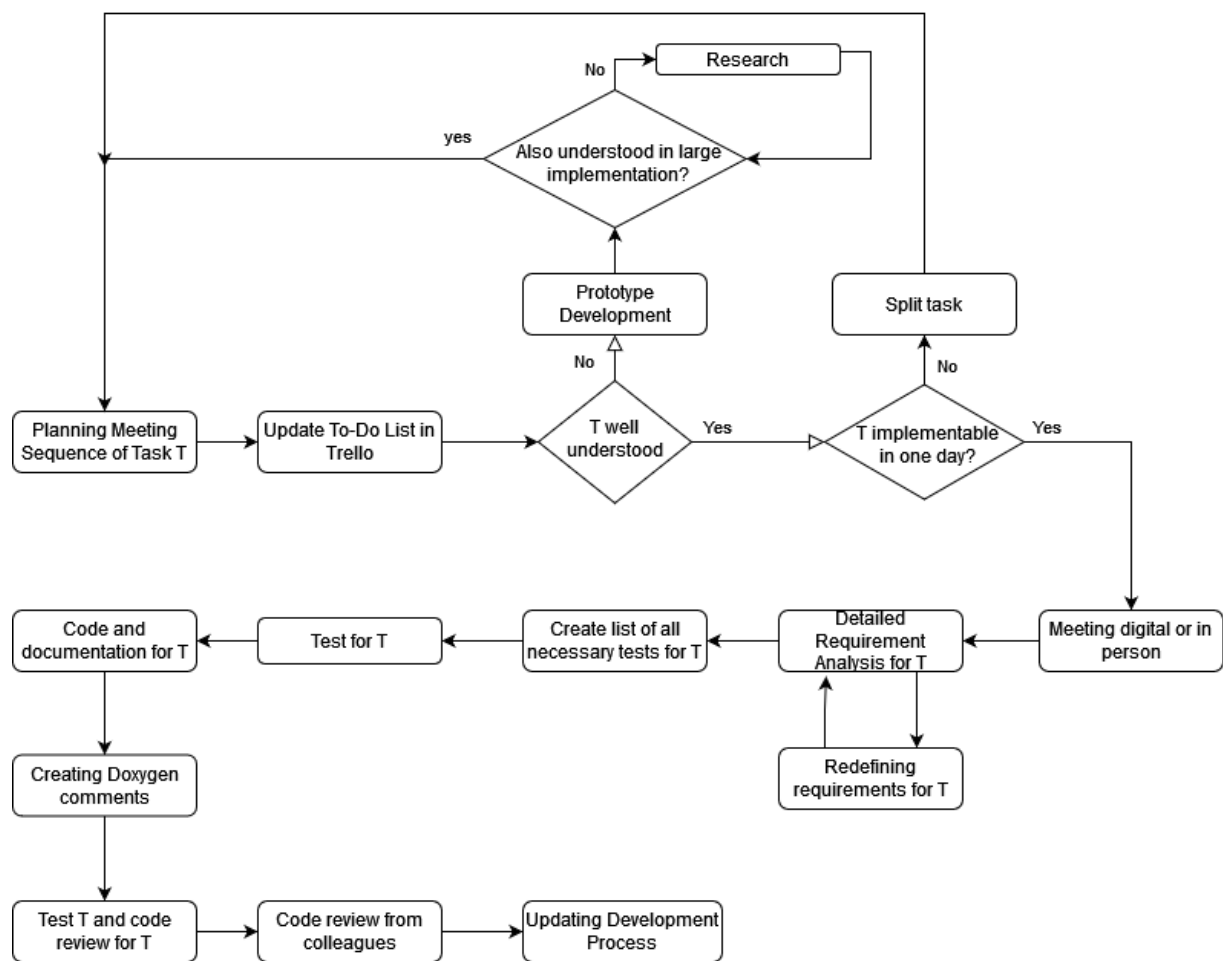
Erfolge:

Da die Überprüfung des Codes von derselben Person durchgeführt wurde, die kurz davor den Code geschrieben hatte, war es sehr effizient, da diese Person sich nicht erst noch einarbeiten musste. Außerdem hatten wir fast jedes Mal alle Testfälle abgedeckt, da wir als gesamtes Team dort sehr gründlich durchgegangen sind.

Probleme und Prozessaktualisierungen:

Es hat sich herausgestellt, dass die Überprüfung des selbstgeschriebenen Codes nicht besonders hilfreich ist. Eine weitere Person war hierbei viel wertvoller, da diese neutral auf den Code schaut und besser den Sinn oder die Code-Struktur hinterfragen konnte. Nachdem dann der Code von anderen angeschaut wurde, haben wir schnell festgestellt, dass die anderen Teammitglieder ebenfalls den Code regelmäßig bewerten sollten. Das haben wir dann auch als festen Bestandteil im Development-Process eingefügt.

4.2.8 - Grafik des aktualisierten Prozesses



5 - Fazit

5.1 – Selbsteinschätzung

Im Rückblick auf das Projekt ist uns klar geworden, dass eine präzisere Definition der Anforderungen essenziell gewesen wäre. Die Anforderungen an das Spiel wurden von uns zwar recht ausführlich definiert, allerdings führte das trotzdem dazu, dass wir während der Entwicklung häufig auf unerwartete Herausforderungen stießen, insbesondere in Bezug auf den Umfang der Aufgaben und die benötigte Zeit für deren Umsetzung. In einigen Fällen hatten wir das Gefühl, dass wir Aufgaben „on the fly“ anpassen mussten, was zusätzlichen Aufwand und zeitliche Verzögerung verursachte.

Für zukünftige Projekte ist unser Ziel, die Anforderungen noch detaillierter und realistischer zu definieren, um eine präzisere Planung und bessere Ressourcenzuteilung zu ermöglichen. Ein strukturierter Anforderungsprozess würde es uns erlauben, potenzielle Herausforderungen frühzeitig zu identifizieren und zeitaufwendige Umplanungen während der Entwicklung zu vermeiden. Außerdem möchten wir lernen, den Umfang von Aufgaben kritischer zu bewerten und realistischere Zeitpuffer einzuplanen, um flexibel auf unvorhergesehene Probleme reagieren zu können.

Dieser Prozess der Selbstreflexion hat uns gezeigt, wie wichtig es eigentlich ist, einen klar definierten Rahmen für die Projektanforderungen zu haben, um effizienter und gezielter arbeiten zu können. Unser Ziel ist es, aus diesen Erfahrungen zu lernen und sie in zukünftigen Projekten anzuwenden, um die Qualität und die Effizienz unserer Arbeit zu steigern.

5.2 – Zeitplan

Insgesamt kann man festhalten, dass unsere ursprünglichen Zeiteinschätzungen definitiv zu optimistisch waren, da wir kaum erkannt hatten, wie umfangreich und anspruchsvoll die meisten der uns zugewiesenen Aufgaben in Bezug auf das Zeitmanagement sein würden. Obwohl wir gegen Ende des Praktikums besser in der Lage waren, unsere Zeiteinschätzungen anzupassen, waren sie erfahrungsgemäß noch immer zu optimistisch, da wir am Ende unserer Arbeitszeiten ziemlich oft immer noch eine Stunde über der ursprünglich geplanten Zeit lagen, und manchmal sogar mehr.

Ein besonderes Beispiel war hier die Modellierung unserer Welt, was sich als zeitintensiver herausstellte als wir ursprünglich gedacht hatten. Wir haben begonnen, unsere Spielwelt, einschließlich des Hauses zu modellieren. Das führte dazu, dass der Aufwand für das Erstellen und Gestalten der Umgebung deutlich größer war, als wir erwartet hatten, und es sah auch nicht besonders schön aus. Die umfangreiche Modellierung hat viele Ressourcen beansprucht, darunter vor allem sehr viel Zeit, was unseren Zeitplan stark belastete. Rückblickend darauf wäre es sinnvoll gewesen, entweder mehr Zeit für diesen Aspekt einzuplanen oder die Modellierungsaufgaben so aufzuteilen, dass der zeitliche Rahmen besser eingehalten wird.

5.3 - Learnings

Aus diesem Anfängerpraktikum haben wir einige wichtige Erkenntnisse gewonnen, die uns in zukünftigen Arbeiten helfen wird, effizienter und zielgerichteter zu arbeiten:

Realistische Einschätzung des Arbeitsumfangs und Zeitmanagement

Zu Beginn des Projekts haben wir ehrgeizige Ziele gesetzt und den Zeitaufwand für viele Aufgaben unterschätzt. Während wir in den frühen Phasen den Zeitplan noch einhalten konnten, wurden unsere Zeitvorgaben gegen Ende zunehmend schwammiger. Das hat uns verdeutlicht, wie wichtig es ist, den Arbeitsaufwand realistischer abzuschätzen und ausreichend Pufferzeit für unvorhergesehene Schwierigkeiten einzuplanen.

Effiziente Organisation von Speicherorten für Dokumente

Im Verlauf des Projekts haben wir festgestellt, dass das Speichern von Dokumenten an zu vielen unterschiedlichen Orten schnell zu Verwirrung führen kann. Wichtige Informationen und Unterlagen gingen verloren und waren schwer auffindbar, was zu unnötigen Verzögerungen führte. Zukünftig möchten wir klare und zentrale Speicherorte festlegen, um die Suche nach Dokumenten zu vereinfachen und wertvolle Zeit zu sparen.

Flexibilität in der Anfangsplanung

Ein weiteres Learning war, dass eine zu detailreiche Ausarbeitung der Funktionalitäten am Anfang des Projekts nicht immer sinnvoll ist. Während der Entwicklung haben sich einige Anforderungen und Funktionen geändert, was zur Folge hatte, dass viele Details aus der Anfangsphase nochmal überarbeitet werden mussten. In zukünftigen Projekten werden wir darauf achten, flexibler zu bleiben und den Detailgrad der Planung an die jeweilige Projektphase anzupassen, um unnötige Arbeit zu vermeiden.

Ein besseres Verständnis für die Perspektive des Entwicklers

Ein weiteres Learning aus diesem Projekt ist die Erkenntnis, wie wichtig es eigentlich ist, Anforderungen so zu formulieren, sodass sie für alle Beteiligten klar und eindeutig verständlich sind (insbesondere für Entwickler, die das Hintergrundwissen vom Spiel bzw. vom Code haben). Da wir unsere Anforderungen selbst erstellt und diese direkt umgesetzt haben, traten bei uns keine Verständnisprobleme auf. Allerdings hätten wir stärker darauf achten sollen, die Anforderungen so zu formulieren, sodass sie auch für außenstehende Entwickler, die nur die Anforderungen ohne zusätzliche Erklärungen erhalten, unmissverständlich sind. In zukünftigen Projekten müssen wir daher mehr Aufwand darin investieren, uns in die Perspektive der Entwickler bzw. Programmierer hineinzuversetzen, dass alle Anforderungen klar, präzise und für andere gut nachvollziehbar sind.

6 – Appendix

Requirements Document

Storyboard

Klassendiagramm

Usability Test

Doxygen Dokumentation

Requirements analysis

1. Introduction

The story of Steve is a story-driven psychological horror game that is as close to reality as possible, putting the player in the role of Steve - a man in the midst of a personal crisis. Accompanied through the day and his nightmarish experiences, the player delves deep into Steve's reality and his fragile psyche.

The working name of the game is "**GetOut**"

The official name is "**Kenopsia**"

The points that haven't been mentioned yet can't be filled out at this given point of time in the development process.

1.1. Purpose of the system

Creating an immersive, terrifying experience for the player. The system encompasses various components, each serving a specific purpose to enhance the horror elements.

This game is based on the principle of a walking simulator where the main purpose is to explore the world by simply walking around.

1.2. Objectives and success criteria of the project

Create a game using an engine and documenting the process of development.

In the end all participants should be able to understand the steps of game-specific software development processes and be able to organize themselves as a team.

1.3.1 Definitions, acronyms, and abbreviations

Steve: the working name of the main character who is played in first person

Boss: Steve's boss who will occur in the fifth scene of the game, at work. He will be a side character.

Wife/Horror lady: divorced wife of Steve, she will only occur in the fourth scene. She is the main antagonist and will be included in the main part of the story.

Smiler: a grinning face which looks very scary that will occur as an emoji in the fourth scene at home. This is a part of the scary illusions that the player will face during the game.

Game Objects: The following list contains all items that are defined as objects in the game.

- 2 Bottles laying on the couch
- Letter laying on the table
- Pan in the kitchen
- Chocolate Chunks Package in kitchen

- Plate in the kitchen
- Wardrobe in living room
- Front door
- Desk in office
- Computer on desk
- Smiler
- Security cameras
- Horror lady
- Hitbox in scene 4 main entrance and kitchen
- bed scene 4

Definition of Hitboxes:

Throughout the game there are hitboxes where when the player enters, an event will occur such as starting an audio file or moving another object.

1.3.2 Technical definitions

Feedback message:

Should be a small text at the top left of the screen. The messages ensure an unproblematic gameplay and give advice for the player to have a better user experience.

StateManager:

The class that defines the different stages and saving points throughout the game.

Interactable:

The interface that defines the function for objects to be interacted with. It provides features like distance for interaction and a “checkup” method to prove if an object is in range.

PlayerInteractionComponent:

The class that handles the interaction of the player with other objects having an Interactable-Interface.

Defined interaction range:

public float interactRange = 5f is the actual range at which the player is able to interact with the predefined object. The range is based on the player position.

Player Movement:

The class that implements the ability for the player to move by using the keyword.

Christian Schäfer
Timo Skrobanek
Andrei Costeniuc

UI Manager:

The class that updates the Graphic Interface by analyzing the data from the StateManager.

Animation Manager:

The class for all general animations throughout the game.

Menu:

Composition of classes that create the main menu in the beginning when starting the game.

Escape-Menu:

A menu that can be accessed from every point throughout the game. This is used to pause the game.

1.4 References

[Trello](#)

[GitHub](#)

[Game-Release](#)

Storyboard (see appendix)

Usability (see appendix)

2. Proposed system**2.1 Overview**

- Game should be able to run on Windows/MAC/Linux
- All developers are using Windows
- Game will be programmed and designed with Unity

Proposed input devices:

- Mouse
- Keyboard

Proposed mouse and Keyboard layout:

The game should be programmed for the standard German QWERTZ keyboard layout.

The game should be programmed for a typical computer mouse only taking left click, right click and movement of the mouse into consideration.

Proposed keys in usage:

WASD: moving of the player

E: interact

T: entering Esc-Menu

Shift: sprint

2.2 Functional requirements

General gameplay:

- Player wakes up on the couch, then he has to accomplish assignments eg. tidying up the kitchen
- The player has to dress up in the bedroom so that he can go to work
- At work he sits on his desk and falls asleep
- Inside of his dream the set is way darker and the player should get the feeling of being watched
- While exploring the house the player notices strange anomalies and finally wakes up at work

ID: FR1	Title: Interact with surroundings
Description	User interacts with objects by using the left mouse button or "E" on the keyboard
Acceptance Criterion	<ul style="list-style-type: none">- User triggers event with mouse or keyboard clicking followed by a specified action
Notes	Will be called in update function throughout the game (List of objects can be found in 1.4 Definitions) (Type of keyboard layout and mouse can be found in 2.1 Overview)

ID: FR2	Title: Move in World
Description	User can move by using W,A,S,D
Acceptance Criterion	<ul style="list-style-type: none">- User must be able to move in all directions except for jumping- Movement is developed for FP (First person)- Movement speed is constant and can change between walking and running speed using shift
Notes	Player can only move within the defined areas of the scenes (Type of keyboard layout can be found in 2.1 Overview)

ID: FR3	Title: Enter Gameplay
Description	User must be able to load gameplay
Acceptance Criterion	<ul style="list-style-type: none">- User must be able to start the gameplay from the main menu
Notes	Loading the gameplay at a different point in the game is not possible

ID: FR4	Title: SaveGame Manager
Description	User is able to save at any given point throughout the game but only the beginning of the current scene is being saved.
Acceptance Criterion	<ul style="list-style-type: none">- After saving the game, there is a file containing the current state of the story
Notes	As an addition: the data will be encrypted with SHA256 or AES256 (see chapter "Encryption Process")

2.3 Nonfunctional Requirements

ID: NFR1	Title: Performance
Description	Make sure that game runs smoothly at all times
Acceptance Criterion	1. The game should have a smooth gameplay, even during complex scenes and effects, therefore the game should have different graphic settings (low, medium, high)
Notes	For each graphic setting, we apply shadow and light in different resolutions.

ID: NFR2	Title: Load time
Description	Make sure that user doesn't have to wait too long to load into the game
Acceptance Criterion	1. The game should load within 10 seconds when starting a new session
Notes	

ID: NFR3	Title: Usability
Description	The game should have understandable instructions in general
Acceptance Criterion	1. Every button should have a suitable description or name and suitable design 2. The user is getting introduced to the game without a direct tutorial (Indirect explanation) 3. At all time throughout the game the current assignment should be clear to the player
Notes	Will be tested with suitable Usability Tests. Usability description can be found in the appendix.

ID: NFR4	Title: Reliability
Description	Title: The game must not crash or freeze during gameplay session
Acceptance Criterion	<ol style="list-style-type: none">1. The game must not crash more than once per hour under normal conditions2. The game must not crash more than once every half hours under stress conditions (e.g. maximum settings, switching between scenes)3. The game must auto-save progress at regular intervals4. Auto-save points should be created at key moments (Change of scenes)
Notes	All crashes and critical errors must be logged with detail information in order to debug

ID: NFR5	Title: Visual Quality
Description	The game should have a decent resolution
Acceptance Criterion	<ol style="list-style-type: none">1. Game should work at low and high quality settings
Notes	

2.4 Documentation

Documentation is done using Doxygen.

Can be found in the appendix.

2.5 Error handling and extreme conditions

Changing of StoryBoard

Problem:

During the development process, we realized that the initial scope of the game was too ambitious. The original storyboard included too many features and elements, making it impossible to implement everything within our time and resource constraints.

Solution:

To stay on track, we made the decision to reduce the scope of the storyboard, focusing only on the core aspects that are essential to the game. This allowed us to

Christian Schäfer
Timo Skrobanek
Andrei Costeniuc

streamline development and concentrate our efforts on completing the necessary requirements within the available timeline (timeline between meetings).

Starting too late with Doxygen Documentation

Problem:

After a few developing and implementation steps, it took us more and more time to relate the past ideas and methods we created. Whether we were modifying completed features or unfinished ones, we always had to invest a significant amount of time to fully understand and immerse ourselves in the code before making any changes.

Solution:

With the use of doxygen documentation we were able to improve our workflow and relate our written methods.

Wrong origin of ray-tracing

Problem:

Initially, the ray used for ray-intersection was originating from the center of the player model instead of the player's head, which caused difficulties in picking up objects and interacting with the environment since the angle between view and interaction ray was different.

Solution:

To fix this, we recalibrated the ray's origin point, moving it from the center of the model of the player's head. This adjustment greatly improved the precision of item interactions and overall gameplay experience.

2.6 System modifications

At the beginning, we developed a prototype to validate the core functionality of the system and ensure that our intended features worked as expected in the game environment. This prototype served as a proof of concept and allowed us to quickly test ideas without focusing on code quality.

Once the prototype was successful, we transitioned to a more structured development phase. The code was refactored in order to follow clean code principles, such as elimination of redundancies, modularization and improved naming conventions. We ensured that methods and classes were kept concise and focused on single responsibilities. Furthermore, we incorporated proper

Christian Schäfer
Timo Skrobanek
Andrei Costeniuc

documentation and comments to enhance maintainability (especially by using Doxygen).

2.7 Physical environment

Desktop

For all operating systems, the Unity Player is supported on workstations, laptop or tablet form factors, running without emulation, container or compatibility layer.

Operating system	Operating system version	CPU	Graphics API	Additional requirements
Windows	Windows 10 version 21H1 (build 19043) or newer	x86, x64 architecture with SSE2 instruction set support, ARM64	DX10, DX11, DX12 or Vulkan capable GPUs	Hardware vendor officially supported drivers For development: IL2CPP scripting backend requires Visual Studio 2019 with C++ Tools component or later and Windows SDK version 10.0.19041.0 or newer
Universal Windows Platform	Windows 10 version 21H1 (build 19043) or newer; Xbox One, Xbox Series X and Series S, HoloLens	x86, x64 architecture with SSE2 instruction set support, ARM, ARM64	DX10, DX11, DX12 capable GPUs	Hardware vendor officially supported drivers. For development: Visual Studio 2019 with C++ Tools component or later and Windows SDK version 10.0.19041.0 or newer.
macOS	Big Sur 11 or newer	Apple Silicon, x64 architecture with SSE2	Metal capable Intel and AMD GPUs	Apple officially supported drivers. For development: IL2CPP scripting backend requires Xcode.
Linux	Ubuntu 22.04, Ubuntu 24.04	x64 architecture with SSE2 instruction set support Note: Desktop Linux supports only 64-bit architecture.	OpenGL 3.2+, Vulkan capable GPUs	Gnome desktop environment running on top of X11 windowing system Other configuration and user environment as provided stock with the supported distribution (such as Kernel or Compositor) Nvidia and AMD GPUs using Nvidia official proprietary graphics driver or AMD Mesa graphics driver

[Source](#)

2.8 Resource issues

Encryption Process (Andrei Costeniuc)

Overview

The save data is protected by encrypting it using the AES-256 encryption algorithm. This ensures that even if someone gains access to the save file, they cannot easily read or modify its contents without the encryption key. Additionally, a SHA.256 hash is used to detect tampering with the encrypted data.

Serializing the Save Data

Before encryption, the game's current state is serialized into a byte array using the BinaryFormatter. Serialization converts the SaveData object into a sequence of bytes that can be stored in a file:

```
BinaryFormatter formatter = new BinaryFormatter();
using (MemoryStream memoryStream = new MemoryStream())
{
    // Serialisieren der aktuellen Spieldaten
    formatter.Serialize(memoryStream, currentSave);
    byte[] serializedData = memoryStream.ToArray();
}
```

Encrypting the Serialized Data

The serialized byte array is encrypted using AES-256. AES (Advanced Encryption Standard) is a symmetric encryption algorithm that requires a secret key and an initialization vector to encrypt and decrypt data.

- The Key is a 32-character string, padded or trimmed to ensure the correct length.

- The Initialization Vector (IV) is a 16-byte array initialized to all zeroes. We initialized it to zero for simplicity, though a random IV can enhance security.

The Encrypt function takes the serialized data and:

1. Initializes the AES encryption engine
2. Creates an encryptor using the key and IV
3. Encrypts the byte array using TransformFinalBlock

```
using (Aes aes = Aes.Create())
{
    aes.Key = Encoding.UTF8.GetBytes(encryptionKey.PadRight(32).Substring(0, 32)); // 32 Byte Schlüssel
    aes.IV = new byte[16]; // Gleicher IV wie beim Verschlüsseln

    using (ICryptoTransform decryptor = aes.CreateDecryptor(aes.Key, aes.IV))
    {
        return decryptor.TransformFinalBlock(encryptedData, 0, encryptedData.Length);
    }
}
```

The result is a fully encrypted version of the serialized save data, which is unreadable without the correct key and IV.

Creating a Hash for Tamper Detection

To ensure that the save file has not been tampered with, a SHA-256 hash of the encrypted data is created. This hash acts as a unique “fingerprint” of the encrypted data. When loading the save file, this hash is recalculated and compared with the stored hash to verify data integrity.

```
// Überprüfe den Hash, um Manipulationen zu erkennen
byte[] calculatedHash = HashHelper.ComputeSHA256Hash(encryptedData);
```

The ComputeSHA256Hash function creates a new SHA256 instance, computes the hash of the encrypted data and returns the 32-byte hash.

Combining the Hash and Encrypted Data

The hash and the encrypted data are combined into a single byte array - the first 32 bytes store the hash, the remaining bytes store the encrypted data. This combined array is then written to the save file.

```
// Kombiniere die Hash-Werte und die verschlüsselten Daten (erst der Hash, dann die verschlüsselten Daten)
byte[] combinedData = new byte[hash.Length + encryptedData.Length];
System.Buffer.BlockCopy(hash, 0, combinedData, 0, hash.Length);
System.Buffer.BlockCopy(encryptedData, 0, combinedData, hash.Length, encryptedData.Length);

File.WriteAllBytes(saveFilePath, combinedData);
```

Save File Validation During Loading

When loading the save file, the first 32 bytes are extracted as the hash, the remaining bytes are extracted as the encrypted data and a new hash is computed

from the encrypted data and compared with the extracted hash. If the hashes do not match, it means the file has been tampered with and the load process is aborted.

```
// Überprüfe den Hash, um Manipulationen zu erkennen
byte[] calculatedHash = HashHelper.ComputeSHA256Hash(encryptedData);
if (!HashHelper.CompareHashes(hash, calculatedHash))
{
    Debug.LogError("Save file has been tampered with!");
    return;
}
```

Decrypting the Encrypted Data

If the hashes match, the encrypted data is decrypted using the same AES-256 algorithm with the same key and IV. The decrypted data is then deserialized back into a SaveData object.

```
using (Aes aes = Aes.Create())
{
    aes.Key = Encoding.UTF8.GetBytes(encryptionKey.PadRight(32).Substring(0, 32)); // 32 Byte Schlüssel
    aes.IV = new byte[16]; // Gleicher IV wie beim Verschlüsseln

    using (ICryptoTransform decryptor = aes.CreateDecryptor(aes.Key, aes.IV))
    {
        return decryptor.TransformFinalBlock(encryptedData, 0, encryptedData.Length);
    }
}
```

2.9 Pseudo requirements

ID: FR_SCENE1	Title: Scene 1 "Home"
Description	The user has an environment simulating his home. He can freely walk through the house and interact with objects.
Acceptance Criterion	<ul style="list-style-type: none">- Alpha Walls cover the complete house, so the player can't go outside- The player can interact with a letter, bottles and objects in the kitchen as well as the wardrobe and entrance door
Notes	

ID: FR_SCENE2	Title: Scene 2 "Office"
Description	An American style decorated office with views of other buildings in a city.
Acceptance Criterion	<ul style="list-style-type: none">- Alpha Walls cover the complete house, so the player

	can't go outside - One computer is turned on and lightened up by external light sources.
Notes	

ID: FR_SCENE3	Title: Scene 3 "Office sitting"
Description	The user is at his desk (still in office) and can only rotate the camera.
Acceptance Criterion	<ul style="list-style-type: none">- The player has a full 360 degree view at his desk- When interacting (clicking) with his desk, the player is sent to next scene
Notes	

ID: FR_SCENE4	Title: Scene 4 "Home - nightmare"
Description	The user has an environment simulating his home. He can freely walk through the house. It's dark and music is playing in background
Acceptance Criterion	<ul style="list-style-type: none">- Music is playing in background- environment is kept dark- Alpha Walls are placed, such that the player can't exit the property of his house-
Notes	

ID: FR_MENU	Title: Scene "Main Menu"
Description	The user can start the game, setup graphical settings and quit the game.
Acceptance Criterion	<ul style="list-style-type: none">- Start button to change to scene 1. This one will always start the game from beginning. Meaning => Scene 1

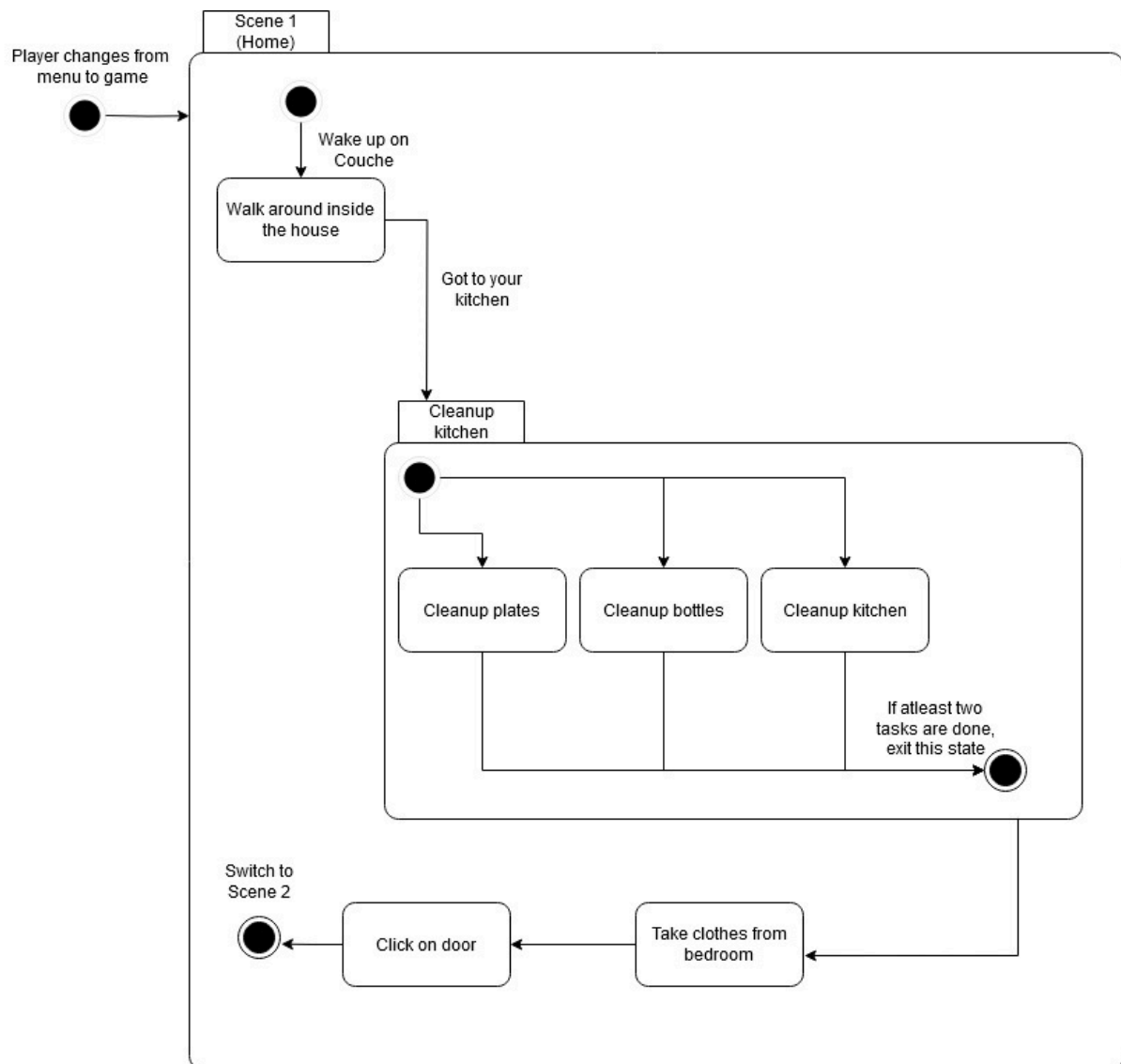
	<ul style="list-style-type: none">- Settings button will change to graphical settings. The settings will be stored during the game runtimeQuit button will stop the game
Notes	

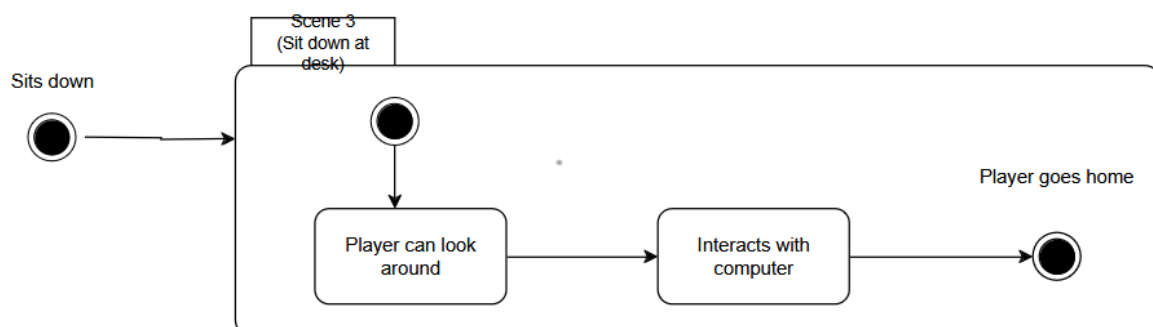
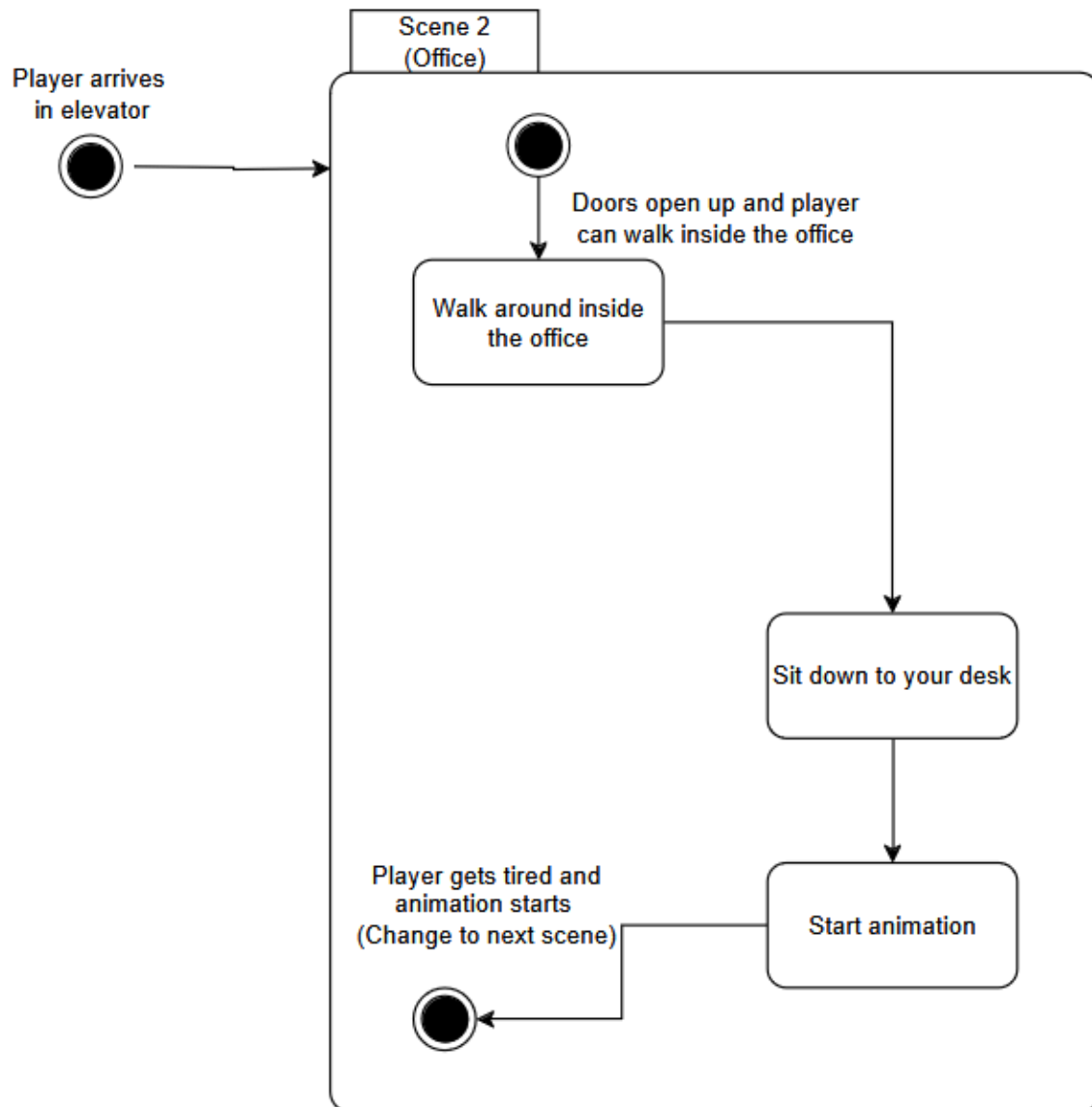
ID: FR_EscMENU	Title: Scene "Esc-Menu"
Description	The user can pause the game at any time and enter the Esc-Menu.
Acceptance Criterion	<ul style="list-style-type: none">- User can always enter Esc-Menu using "T"- Esc-Menu cannot be accessed from the Main-Menu- User can resume to game or leave to Main-Menu
Notes	

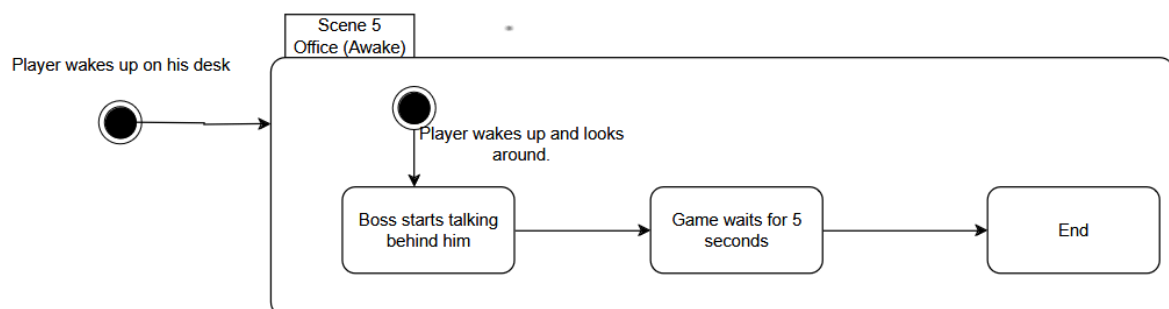
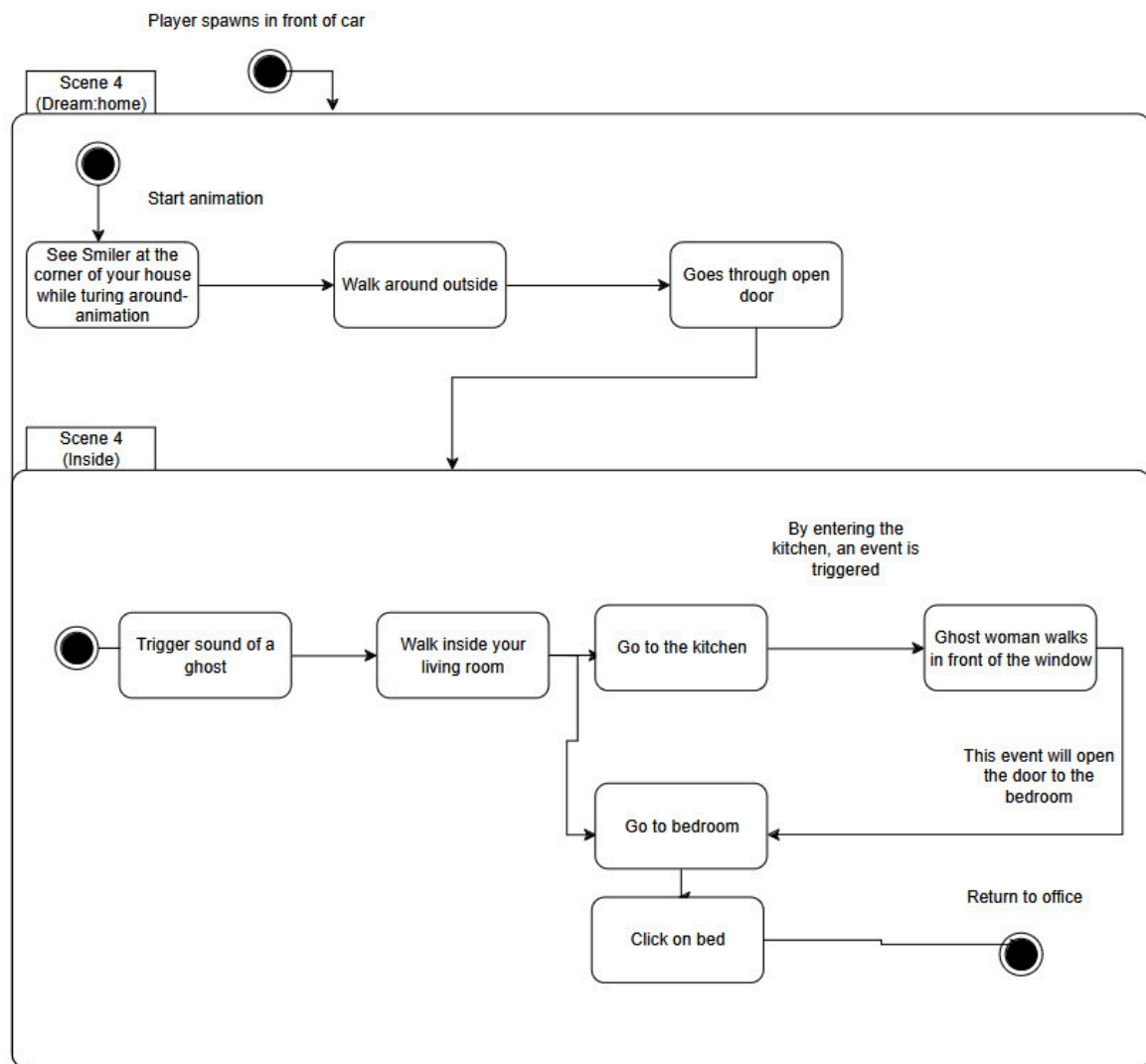
2.10 System models

2.10.1 Scenarios

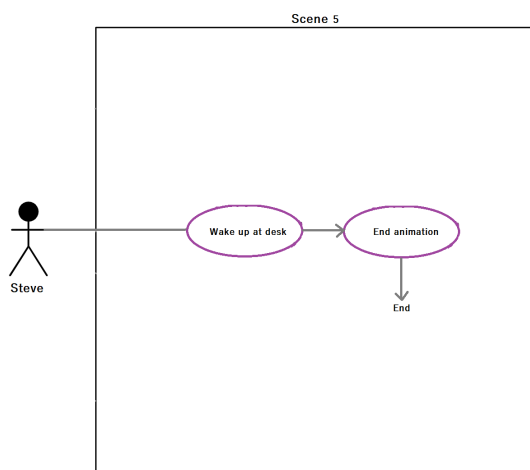
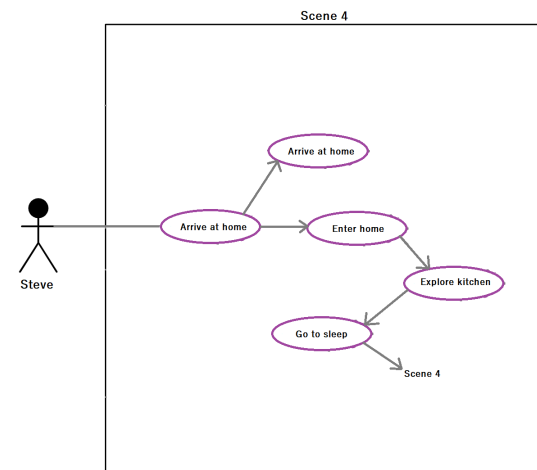
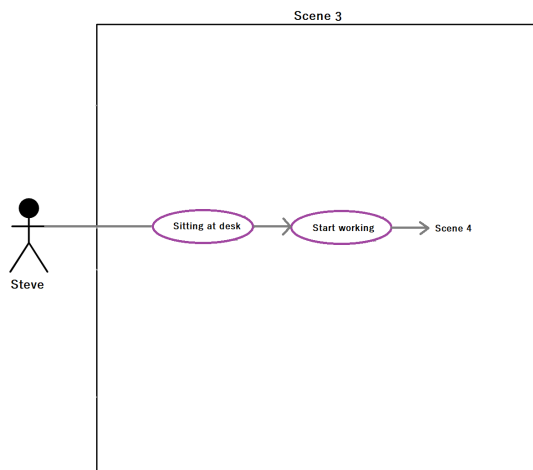
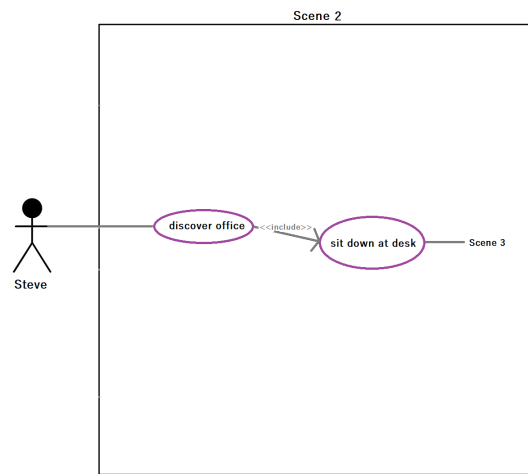
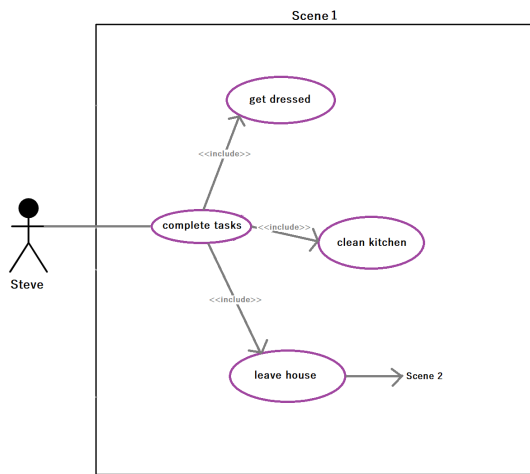
In order to easily describe the scenarios of the game we created StateMachineDiagrams that visually shows the possible interactions that the player can make and which interactions lead to the following scene.







2.10.2 Use case model



UseCase Description:

The main character is called Steve and he's a divorced man who has psychological problems.

In the first scene he wakes up on his sofa and the player can interact in first person with items inside the house.

The player has to accomplish tasks before being able to continue with the story. He has to get dressed, clean the kitchen, pick up a letter laying on the table and finally leave the house and drive to the city to work by clicking on the house door.

For the second scene the player will be teleported into the office and is able to sit down at his desk.

As the third scene starts the player is now at his desk where he can start to work by clicking on the computer. That's also where Steve will fall asleep.

In the fourth scene the player first has to enter his house and follow a mostly strict order of tasks which trigger new events.

First he has to enter the house and then explore the kitchen where he will see various optical illusions and figures moving around the house.

He will then have to go into the bedroom where he goes to bed and wakes up at his desk again. This leads to scene 5.

The fifth scene only consists of a cut scene where it is shown that Steve wakes up on his desk while his boss is yelling at him. The cut scene ends in Credits and the game is finished.

2.11 User-interface

The requirements for our user-interface were focused on being simple. Every function should be easily understandable without much detail or information.

So we plan to do a Main-Menu and Esc-Menu with only the most necessary functions:





For the user-interface while playing the game we created Usability-Tests:

In order to properly improve our user interface and user-friendliness we created a Usability-Test for scene 1 and 2, where we let other people play the game and ask whether there were difficulties or not.

In the following are the most important arguments of the test subjects:

Complaint: The text description in the upper corner didn't stay long enough to read.

Solution: We increased the duration of appearance of the text description.

Complaint: It was not clear whether I did enough tasks to get out of the house or not.

Solution: We mentioned in the text description that it is now able to leave the house.

Complaint: I couldn't see the key suggestion when I looked at an interactable object.

Solution: We made sure that every time the user looks at an interactable object a key suggestion appears in a size and color that can't be overseen.

3.0 Sources

Tutorials for programming:

- [1] Official documentation of Unity “First Steps”. [Online]. Available:
<https://unity.com/de/learn/get-started>
- [2] System requirements for Unity 6. [Online]. Available:
<https://docs.unity3d.com/6000.0/Documentation/Manual/system-requirements.html>
- [3] Documentation for documentation with Doxygen. [Online] Available:
<https://opensource.com/article/22/5/document-source-code-doxygen-linux>
- [4] Make Human [Online]. Available:
<http://www.makehumancommunity.org/content/downloads.html>
- [5] Sketchfab [Online]. Available:
<https://sketchfab.com/feed>
- [6] draw.io [Online]. Available:
<https://draw.io>
- [7] Poly Haven [Online]. Available:
<https://polyhaven.com/>
- [8] Soundcloud [Online]. Available:
<https://soundcloud.com/>
- [9] Blender [Online]. Available:
<https://www.blender.org/>
- [10] Unreal Engine C++ Tutorial [Online]. Available:
<https://www.youtube.com/watch?v=cPgtd4m5-EI&list=PLitYOd30OzhJBQRqYZEGBzJYemA-SH8Z>
- [11] Unreal Engine 5 Beginner Tutorial [Online]. Available:
<https://www.youtube.com/watch?v=XRmn-EYt8wI&list=PLncmXJdh4q88DFCEVuGpOY3AGQwBvoQnhp>
- [12] AI Image Generator [Online]. Available:
<https://perchance.org/ai-text-to-image-generator>

Requirements analysis old

1. Introduction

The story of Steve is a story-driven psychological horror game that is as close to reality as possible, putting the player in the role of Steve - a man in the midst of a personal crisis. Accompanied through the day and his nightmarish experiences, the player delves deep into Steve's reality and his fragile psyche.

The working name of the game is "**GetOut**"

The points that haven't been mentioned yet can't be filled out at this given point of time in the development process.

1.1. Purpose of the system

Creating an immersive, terrifying experience for the player. The system encompasses various components, each serving a specific purpose to enhance the horror elements.

1.3. Objectives and success criteria of the project

Create a game using an engine and documenting the process of development. In the end all participants should be able to understand the steps of game-specific software development processes and be able to organize themselves as a team.

1.4. Definitions, acronyms, and abbreviations

Steve: the working name of the main character who is played in first person (his using name in game will be clarified in the future)

Boss: Steve's boss who will occur in the second scene of the game, at work. He will be a side character.

Wife: divorced wife of Steve, she will only occur in the forth (last) scene. She is the main antagonist and will be included in the main part of the story.

Smiley: a grinning face which looks very scary that will occur as an emoji in the last scene at home. This is a part of the scary illusions that the player will face during the game.

1.5. References

[Trello](#)

[GitHub](#)

[Storyboard](#)

[SCRUM-Protocol](#)

Time Management (will be translated in future)

1. Phasenaufteilung

1. Konzeptionelles Design

Das Spiel wird sowohl von der Story als auch vom ersten Aussehen beschrieben. Es wird ein Storyboard erstellt und erste Statemachine Diagramme werden erstellt um den Ablauf im Spiel zu beschreiben.

Zeitaufwand: 10h

2. Technisches Design

Die Architektur sowie das genaue Aussehen vom Programmiercode werden hier beschrieben. Dabei werden Klassendiagramme, Sequenzdiagramme und ähnliche angelegt.

Zeitaufwand: 14h

3. Entwicklung

In dieser Phase wird GetOut schlussendlich umgesetzt. Mithilfe einer Engine werden nun alle Anforderungen übersetzt.

Zeitaufwand: ~120h

4. Tests

Nachdem GetOut implementiert wurde, gibt es mehrere große Testphasen, die möglichst viele Szenarien abdecken sollen.

Zeitaufwand: 26h

5. Fertigstellung

In dieser letzten Phase wird das Spiel für die verschiedenen Plattformen fertiggestellt und ein Report geschrieben. In diesem Report wird nochmals genauer auf die einzelnen Phasen eingegangen

Zeitaufwand: 10h

2. Proposed system

2.1. Overview

- Game should be able to run on Windows/MAC/Linux
- All developers are using Windows
- Game will be programmed and designed with Unity

2.2. Functional requirements

ID: FR1	Title: Interact with surroundings
Description	User interacts with objects by using the left mouse button
Acceptance Criterion	1. User triggers event with mouse clicking and some action will follow
Notes	Will be the main function throughout the game

ID: FR2	Title: Move in World
Description	User can move by using W,A,S,D
Acceptance Criterion	<ol style="list-style-type: none"> 1. User must be able to move in all directions except for jumping 2. Movement is developed for FP (First person) 3. Movement speed is variable 4. Movement is independent from Head up and down rotation
Notes	

ID: FR3	Title: Enter Gameplay
Description	User must be able to load gameplay at any part of the story and from the main menu
Acceptance Criterion	<ol style="list-style-type: none"> 1. User must be able to start or resume gameplay 2. If a checkpoint is given, user should be able to spawn there
Notes	Requires functional memory management

ID: FR4	Title: Save Gameplay at any point
Description	User is able to save at any given point throughout the game
Acceptance Criterion	<ol style="list-style-type: none"> 1. After saving the game, there is a file containing the current state of the story.
Notes	Requires functional memory management

2.3 Nonfunctional Requirements

ID: NFR1	Title: Performance
Description	Make sure that game runs smoothly at all times
Acceptance Criterion	<ol style="list-style-type: none"> 1. The game should have a smooth gameplay, even during complex scenes and effects

Notes	
-------	--

ID: NFR2	Title: Load time
Description	Make sure that user doesn't have to wait too long to load into the game
Acceptance Criterion	<ol style="list-style-type: none"> 1. The game should load within 10 seconds when starting a new session
Notes	

ID: NFR3	Title: Usability
Description	The game should have understandable instructions in general
Acceptance Criterion	<ol style="list-style-type: none"> 1. Every button should have a suitable description or name and suitable design 2. The user is getting introduced to the game without a direct tutorial (Indirect explanation)
Notes	

ID: NFR4	Title: Reliability
Description	The game must not crash or freeze during gameplay session
Acceptance Criterion	<ol style="list-style-type: none"> 1. The game must not crash more than once per hour under normal conditions 2. The game must not crash more than once every half hours under stress conditions (e.g. maximum settings, switching between scenes) 3. The game must auto-save progress at regular intervals 4. Auto-save points should be created at key moments (Change of scenes, interaction with specific objects)
Notes	All crashes and critical errors must be logged with detail information in order to debug

ID: NFR5	Title: Audio-Visual Quality
Description	The game should have a decent resolution and a pleasant sound quality
Acceptance Criterion	<ol style="list-style-type: none"> 1. Game should work at low and high quality settings 2. Using a quality microphone

2.3.2 Documentation

Documentation is done using Doxygen

Short guide on how to use Doxygen:

apt install Doxygen

Nach dem Programmieren muss man ein neues Verzeichnis machen namens docs und dort drin dann doxygen -g machen, das generiert ein DoxygenFile wo die ganzen configs drin sind

2.3.5 Error handling and extreme conditions

2.3.6. Quality issues

2.3.7. System modifications

2.3.8. Physical environment

2.3.9. Security issues

2.3.10. Resource issues

2.4. Pseudo requirements

Using Unity instead of Unreal Engine 5:

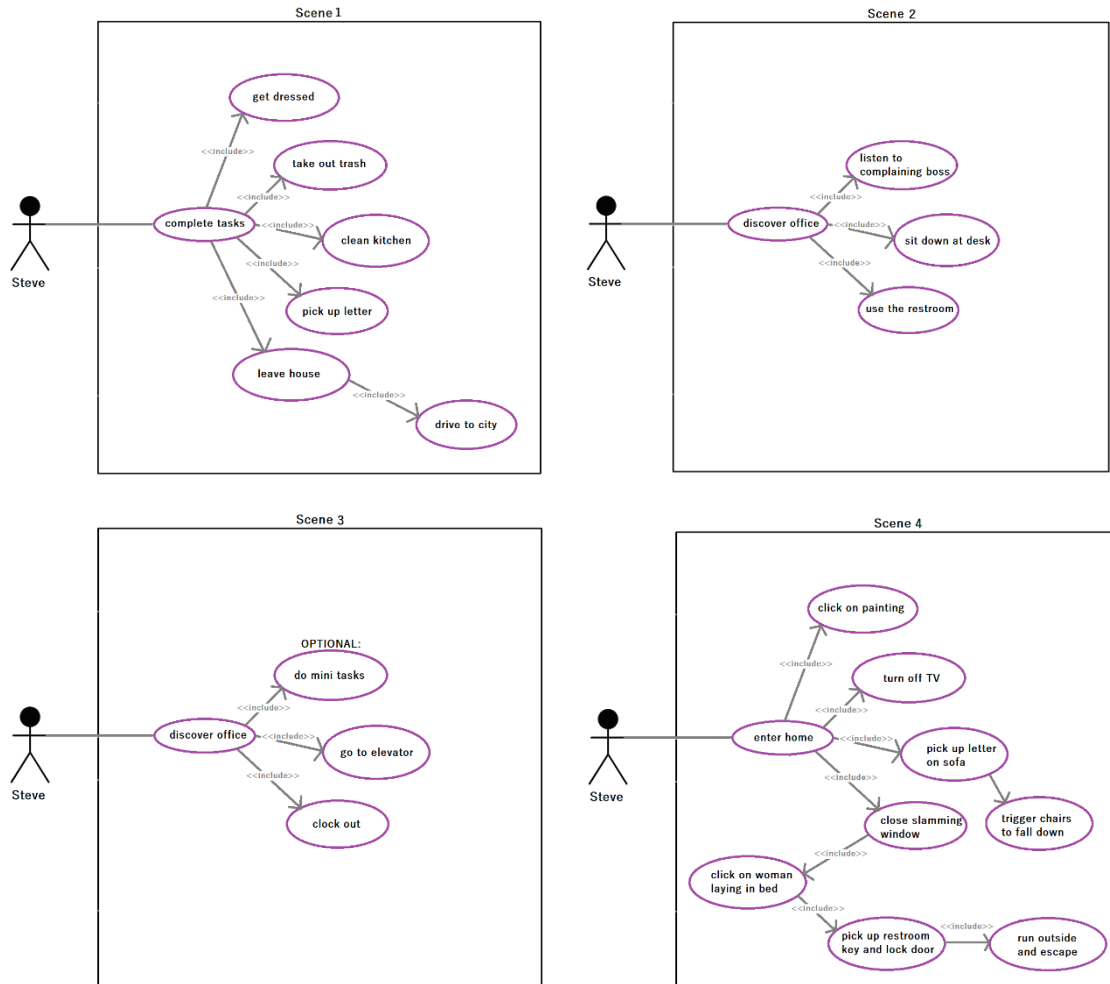
We had many problems with using Unreal Engine. There were a lot of questions about how to use it and we were not able to get answers from the internet which forced us to change the engine.

In the long run we totally benefited from it because we understand the functional processes much better.

2.5. System models

2.5.1. Scenarios

2.5.2. Use case model



The main character is called Steve and he's a divorced man who has psychological problems.

In the first scene he wakes up on his sofa and the player can interact in first person with items inside the house.

The player has to accomplish tasks before being able to continue with the story. He has to get dressed, take out the trash, clean the kitchen, pick up a letter laying on the floor and finally leave the house and drive to the city to work by clicking on the car.

For the second scene the player will be teleported into the office and again is able to accomplish optional tasks such as listening to complaining boss by stepping in front of the boss's door, using the restroom or sitting down at his desk, which allows the story to continue.

As the third scene starts the player is now in the dream world but still in his office where he again has to do some mini

tasks that have not been specified yet, but definitely clocking out and leaving the office via the elevator.

In the fourth scene the player first has to enter his house and follow a mostly strict order of tasks which trigger new events.

First he has to click on a painting hanging on the wall which will turn on the TV. The TV has to be turned off and after that the player has to pick up a letter laying on the sofa which triggers chairs, that have been hanging on the ceiling all the time, to fall down and a window to start slamming loudly. The player has to close the slamming window and is somehow led to the bedroom where a women is laying in bed. He has to click on the women which will trigger an event where a women is slamming on the window and the player has to escape into the restroom and pick up the restroom key and lock the door.

After that he has to run outside of the house which will lead to scene 5.

The fifth scene only consists of a cut scene where it is shown that Steve wakes up on his desk while his boss is yelling at him. The cut scene slowly blacks out and the game is finished.

2.5.3. Object model

2.5.3.1. Data dictionary

2.5.3.2. Class diagrams

Backend for Getout

This document will describe how the game is designed in code.



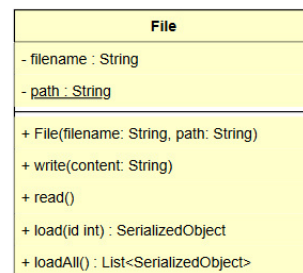
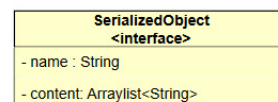
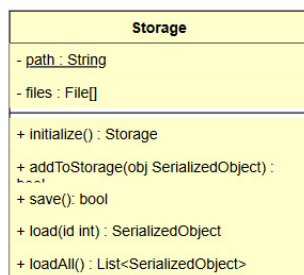
Storage/Backend-System

Storage related class

Logic related class

object related class

implemented as
ngleton



2.5.4. Dynamic models

2.5.5. User-interface

3. Glossary

Storyboard GetOut

Idee für Spielnamen: Kenopsia

Kenopsia beschreibt die unheimliche, einsame Atmosphäre eines Ortes, die normalerweise voller Menschen ist/war, und jetzt jedoch völlig verlassen ist.

Szene 0: Menü

Beim Starten des Spiels gelangt man zuerst ins Menü. Es ist eine Startseite mit verschiedenen Knöpfen, wie:

- Spiel starten
- Optionen (Umleitung in ein anderes Menü)
- Spiel beenden

Szene 1: Zuhause

Atmosphäre: Mittags, Sonne, Frühling

Spielmechanik:

Spieler wacht auf der Couch auf. Er kann durchs Haus frei rumlaufen, das Haus darf er nicht verlassen, da er erstmal verschiedene Aufgaben im Haus erledigen muss. Sobald alle fertig sind, darf er das Haus verlassen.

Der Spieler befindet sich zuhause (aufgeräumt, dreckig) in seiner Hütte am Straßenrand zu einem Wald/See. Hütte besteht aus einem Stockwerk, 3 Zimmern. Toilette, Wohnzimmer(mit Küche) und Schlafzimmer.

Es wird ruhige atmosphärische Musik gespielt.

Muss kleine Aufgaben erledigen, wie z.B.:

- **Klamotten** anziehen (Kleiderschrank anklicken) - kurze Animation kommt
- **Küche** aufräumen

Text links oben mit Hinweisen und Dialogen, was der Spieler sagt und was er tun soll.

Das Haus des Spielers ist eher unaufgeräumt und Alkoholflaschen liegen rum.

Spieler klickt auf **Haustür** -> wird in Szene 2 teleportiert.

Szene 2: Auf der Arbeit/Im Büro

Atmosphäre: Normale helle Atmosphäre, Büro

- Spieler startet vor der Tür des Aufzugs
- Spieler läuft durch großen Raum mit vielen mini Büros
- Links hinten ist sein **Arbeitsplatz** (ist detaillierter gemacht)

OPTIONAL:

=> Wenn auf **Arbeitsplatz** geklickt wird, gehts weiter

Szene 3: Büro (Noch wach)

Spieler sitzt am Schreibtisch

- Durch Klicken auf der Bildschirm wird zur nächsten Szene gewechselt

Szene 4: Zuhause (Im Traum)

Atmosphäre: Dunkle Nacht

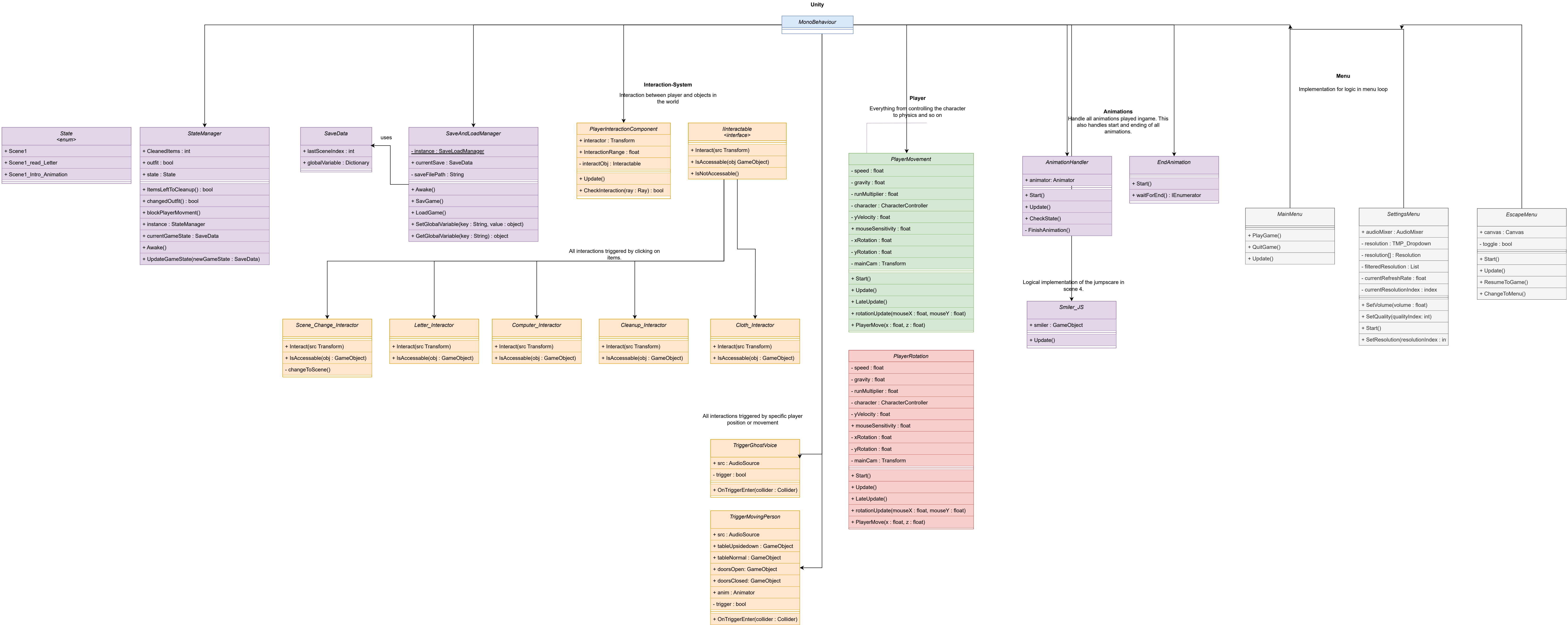
- Der Spieler spawnet vor der Fahrertür des Autos.
- Animation, dass Spieler aussteigt und sich dreht - in der Bewegung sieht man an Ecke der Hütte (die Ecke vom Schlafzimmer → Szene 1) einen **Smiler**
- Nach Animation ist Smiler wieder weg
- Haustür steht offen und im Flur hängen viele **Überwachungskameras**
- Durchs Eintreten wird ein Event getriggert, wodurch ein unheimliches Geräusch abgespielt wird
- Schlafzimmertür ist zu und Tür zum Wohnzimmer ist offen
- Man sieht verschiedene optische Effekte im Wohnzimmer (**Alkoholflaschen** sind groß und auf dem Tisch liegen ganz viele **Briefe**)
- Wenn Spieler in der Küche das Fenster anschaut, läuft eine **weibliche Figur** am Fenster entlang und der **Tisch** ist auf einmal an der Decke
- Die Haustür ist dann geschlossen und die Tür zum Schlafzimmer wurde geöffnet
- Im Schlafzimmer kann man noch auf das **Bett** klicken (Übergang zu Szene 5)

Szene 5: Im Büro (Aufwachen und Monolog vom Chef)

Atmosphäre: Mittags, hell, freundlich

- Spieler wacht auf (POV Kopf Bewegungen mit Kopfschütteln)
- Oben ist ein Text, der die Story aufklärt

(Ende)



Usability-Test für Szene 1 und 2

Ziel des Tests:

Die Testpersonen sollen mögliche Schwierigkeiten in der Benutzererfahrung identifizieren, insbesondere in Bezug auf das Navigieren, Erkennen von Zielen und die intuitive Interaktion mit der Spielumgebung.

Anweisungen für die Testperson:

Bitte folge den Anweisungen Schritt für Schritt und beantworte die Fragen so genau wie möglich.

Abschnitt 1: Navigation und erste Szene

1. Spielstart

- **Aufgabe:** Starte das Spiel und navigiere zur ersten Spielszene (Zuhause).
- **Fragen:**
 - Gab es Schwierigkeiten beim Finden oder Starten der Szene?
[Ja/Nein]

2. Interaktion in der ersten Szene

- **Aufgabe:** Räume alle Items auf, die auf der Couch oder in der Küche liegen.
- **Fragen:**
 - Konntest du alle Items leicht finden? [Ja/Nein]
 - Ist es offensichtlich, ob genügend Items aufgeräumt wurden?
[Ja/Nein]

Abschnitt 2: Interaktion mit Objekten

1. Anziehen der Kleidung

- **Aufgabe:** Ziehe deine Klamotten an und gehe zur Arbeit.
- **Fragen:**
 - War der Kleiderschrank leicht zu finden?
[Ja/Nein]

2. Finden des Arbeitsplatzes in Szene 2

- **Aufgabe:** Im Büro, finde deinen Arbeitsplatz und setze dich hin.
- **Fragen:**
 - War der Arbeitsplatz eindeutig und leicht zu finden?
[Ja/Nein]

Bewertung der Benutzererfahrung

1. Allgemeine Benutzererfahrung

- **Frage:** Auf einer Skala von 1 bis 10, wie würdest du die Benutzererfahrung bewerten (Wie einfach ist es, sich im Spiel zurechtzufinden)?

[1 = sehr schwierig, 10 = sehr einfach]

2. Verbesserungsvorschläge

- **Frage:** Was würdest du am Spiel ändern, um die Benutzererfahrung zu verbessern?

[Bitte erläutere]

Hinweise für die Durchführung

- Der Test wird per Google Forms durchgeführt und die Auswertung wird als PDF angehängt.

Siehe [Google Forms](#)

Kenopsia

1.0.0

Generated by Doxygen 1.12.0

1 Hierarchical Index	1
1.1 Class Hierarchy	1
2 Class Index	3
2.1 Class List	3
3 Class Documentation	5
3.1 AnimationHandler Class Reference	5
3.1.1 Detailed Description	5
3.1.2 Member Function Documentation	6
3.1.2.1 CheckState()	6
3.1.2.2 Start()	6
3.1.2.3 Update()	6
3.1.3 Member Data Documentation	6
3.1.3.1 animator	6
3.2 Bed_Interactor Class Reference	6
3.2.1 Member Function Documentation	7
3.2.1.1 Interact()	7
3.2.1.2 IsAccessable()	7
3.3 Cleanup_Interactor Class Reference	7
3.3.1 Detailed Description	8
3.3.2 Member Function Documentation	8
3.3.2.1 Interact()	8
3.3.2.2 IsAccessable()	9
3.4 Cloth_Interactor Class Reference	9
3.4.1 Detailed Description	10
3.4.2 Member Function Documentation	10
3.4.2.1 Interact()	10
3.4.2.2 IsAccessable()	11
3.5 Computer_Interactor Class Reference	11
3.5.1 Detailed Description	12
3.5.2 Member Function Documentation	12
3.5.2.1 Interact()	12
3.5.2.2 IsAccessable()	13
3.6 EndAnimation Class Reference	13
3.6.1 Detailed Description	13
3.6.2 Member Function Documentation	13
3.6.2.1 Start()	13
3.7 EscapeMenu Class Reference	14
3.8 FileEncryptionTests Class Reference	14
3.9 Interactable Class Reference	14
3.9.1 Detailed Description	15
3.10 InteractionComponent Class Reference	15

3.10.1 Member Function Documentation	16
3.10.1.1 CheckInteraction()	16
3.11 InteractionTest Class Reference	16
3.12 Letter_Interactor Class Reference	16
3.12.1 Detailed Description	17
3.12.2 Member Function Documentation	17
3.12.2.1 Interact()	17
3.12.2.2 IsAccessible()	18
3.13 MainMenu Class Reference	18
3.14 SettingsMenuTest.MockSceneManager Class Reference	18
3.15 PlayerMovement Class Reference	19
3.15.1 Detailed Description	19
3.15.2 Member Function Documentation	20
3.15.2.1 LateUpdate()	20
3.15.2.2 PlayerMove()	20
3.15.2.3 rotationUpdate()	21
3.15.2.4 Start()	21
3.15.2.5 Update()	21
3.16 PlayerMovementTests Class Reference	22
3.17 PlayerRotation Class Reference	22
3.17.1 Detailed Description	23
3.17.2 Member Function Documentation	23
3.17.2.1 LateUpdate()	23
3.17.2.2 rotationUpdate()	23
3.18 SaveData Class Reference	24
3.19 SaveLoadManager Class Reference	24
3.20 Scene1_Change_Interactor Class Reference	24
3.20.1 Detailed Description	25
3.21 SettingsMenu Class Reference	26
3.22 SettingsMenuTest Class Reference	26
3.23 Sleep_Interactor Class Reference	26
3.23.1 Member Function Documentation	27
3.23.1.1 Interact()	27
3.23.1.2 IsAccessible()	27
3.24 Smiler_JS Class Reference	27
3.24.1 Detailed Description	28
3.24.2 Member Data Documentation	28
3.24.2.1 smiler	28
3.25 Startbutton Class Reference	28
3.26 StateManager Class Reference	28
3.27 StateManagerTest Class Reference	29
3.28 Test Class Reference	30

3.29 TriggerGhostVoice Class Reference	30
3.29.1 Detailed Description	30
3.29.2 Member Data Documentation	30
3.29.2.1 src	30
3.30 TriggerMovingPerson Class Reference	31
3.30.1 Detailed Description	31
3.30.2 Member Data Documentation	31
3.30.2.1 anim	31
3.30.2.2 doorsClosed	31
3.30.2.3 doorsOpen	32
3.30.2.4 src	32
3.30.2.5 table	32
3.30.2.6 tableUpsideDown	32
3.31 UI_Updater Class Reference	32
Index	33

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

FileEncryptionTests	14
Interactable	14
Bed_ Interactor	6
Cleanup_ Interactor	7
Cloth_ Interactor	9
Computer_ Interactor	11
Letter_ Interactor	16
Scene1_ Change_ Interactor	24
Sleep_ Interactor	26
InteractionTest	16
SettingsMenuTest.MockSceneManager	18
MonoBehaviour	
AnimationHandler	5
Bed_ Interactor	6
Cleanup_ Interactor	7
Cloth_ Interactor	9
Computer_ Interactor	11
EndAnimation	13
EscapeMenu	14
InteractionComponent	15
Letter_ Interactor	16
MainMenu	18
PlayerMovement	19
PlayerRotation	22
SaveLoadManager	24
Scene1_ Change_ Interactor	24
SettingsMenu	26
Sleep_ Interactor	26
Smiler_ JS	27
Startbutton	28
StateManager	28
StateManagerTest	29
TriggerGhostVoice	30
TriggerMovingPerson	31
UI_ Updater	32

PlayerMovementTests	22
SaveData	24
SettingsMenuTest	26
Test	30

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

AnimationHandler	
Behandelt die Animationen aller Szenen im Spiel und steuert diese abhängig vom aktuellen Spielstatus	5
Bed_Interaction	6
Cleanup_Interaction	
Handles the interaction logic for cleaning up or disabling objects in the game	7
Cloth_Interaction	
Handles the interaction between player and cloths laying around	9
Computer_Interaction	
Handles the interaction between player his computer in office	11
EndAnimation	
Handles the delay before transitioning to the credits scene after the end animation	13
EscapeMenu	14
FileEncryptionTests	14
Interactable	
Handles the interaction between player and gameobjects	14
InteractionComponent	15
InteractionTest	16
Letter_Interaction	
Handles the interaction logic for readable items	16
MainMenu	18
SettingsMenuTest.MockSceneManager	18
PlayerMovement	
Handles the movement logic for the player character in Unity	19
PlayerMovementTests	22
PlayerRotation	
Handles the rotation logic for the player character in Unity	22
SaveData	24
SaveLoadManager	24
Scene1_Change_Interaction	
Handles the change between scenes	24
SettingsMenu	26
SettingsMenuTest	26
Sleep_Interaction	26
Smiler_JS	
Controls the visibility of the "smiler" GameObject based on game state	27

Startbutton	28
StateManager	28
StateManagerTest	29
Test	30
TriggerGhostVoice Plays a ghost voice sound effect when the player enters a specified trigger area	30
TriggerMovingPerson Handles triggering animations, sounds, and object states when the player enters a specific area	31
UI_Updater	32

Chapter 3

Class Documentation

3.1 AnimationHandler Class Reference

Behandelt die Animationen aller Szenen im Spiel und steuert diese abhängig vom aktuellen Spielstatus.

Public Member Functions

- void **Start** ()
Initialisiert die Animationen beim Start des Spiels.
- void **Update** ()
Aktualisiert die Animationen in jeder Frame-Iteration.
- void **CheckState** ()
Überprüft den aktuellen Spielstatus und startet die passende Animation.

Public Attributes

- Animator **animator**

3.1.1 Detailed Description

Behandelt die Animationen aller Szenen im Spiel und steuert diese abhängig vom aktuellen Spielstatus.

Diese Klasse überprüft regelmäßig den Zustand des Spiels und startet entsprechende Animationen für verschiedene Szenen. Die Klasse verwendet einen Animator, um Übergänge und Animationen basierend auf den Zuständen im **StateManager** (p.28) zu steuern.

Author

Timo Skrobanek

Date

28.10.2024

Version

2.0

3.1.2 Member Function Documentation

3.1.2.1 CheckState()

```
void AnimationHandler.CheckState ()
```

Überprüft den aktuellen Spielstatus und startet die passende Animation.

Diese Methode wechselt zwischen den verschiedenen Spielzuständen und spielt die jeweilige Animation basierend auf dem Zustand im **StateManager** (p. 28).

3.1.2.2 Start()

```
void AnimationHandler.Start ()
```

Initialisiert die Animationen beim Start des Spiels.

Diese Methode überprüft den Spielstatus beim Start und startet ggf. die passende Animation.

3.1.2.3 Update()

```
void AnimationHandler.Update ()
```

Aktualisiert die Animationen in jeder Frame-Iteration.

Diese Methode wird in jedem Frame aufgerufen und prüft, ob Animationen abgeschlossen sind, um entsprechende Maßnahmen zu ergreifen.

3.1.3 Member Data Documentation

3.1.3.1 animator

```
Animator AnimationHandler.animator
```

Animator, der die verschiedenen Szenen-Animationen steuert.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Animations/AnimationHandler.cs

3.2 Bed_ Interactor Class Reference

Public Member Functions

- void **Interact** (GameObject obj)
Executes the interaction logic when the player interacts with this object.
- void **IsAccessible** (GameObject obj)
Indicates whether the object is accessible for interaction.
- void **IsNotAccessible** ()

Public Member Functions inherited from Interactable

- void **Interact** (GameObject obj)
- void **IsAccessible** (GameObject obj)
- void **IsNotAccessible** ()

3.2.1 Member Function Documentation

3.2.1.1 Interact()

```
void Bed_Interactor.Interact (
    GameObject obj)
```

Executes the interaction logic when the player interacts with this object.

Parameters

<i>obj</i>	The GameObject that is being interacted with.
------------	---

This method checks if the interaction key (E) is pressed, and if so, it disables the object, effectively removing it from the scene. A debug message is logged to confirm the action.

3.2.1.2 IsAccessible()

```
void Bed_Interactor.IsAccessible (
    GameObject obj)
```

Indicates whether the object is accessible for interaction.

This method is intended to provide feedback or state information about whether the object can be interacted with. Currently, it is not implemented.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Interaction/Click_Interaction/Bed_Interactor.cs

3.3 Cleanup_Interactor Class Reference

Handles the interaction logic for cleaning up or disabling objects in the game.

Public Member Functions

- void **Interact** (GameObject obj)
Executes the interaction logic when the player interacts with this object.
- void **IsAccessible** (GameObject obj)
Indicates whether the object is accessible for interaction.
- void **IsNotAccessible** ()

Public Member Functions inherited from `Interactable`

- void **Interact** (`GameObject obj`)
- void **IsAccessible** (`GameObject obj`)
- void **IsNotAccessible** ()

Public Attributes

- `GameObject` **key**
- `AudioSource` **src**

3.3.1 Detailed Description

Handles the interaction logic for cleaning up or disabling objects in the game.

This component is designed to be attached to objects that should be disabled or "cleaned up" when the player interacts with them. It implements the **Interactable** (p. 14) interface.

Features:

- Disable gameobjects when clicked

Requirements:

- Check each frame for interaction
- Make gameobjects invisible that got cleaned up

How to Use: Give this component to any gameobject you'd like to cleanup or make disappear when clicked.

Author

Timo Skrobanek

Date

18.8.2024

Version

1.0

3.3.2 Member Function Documentation

3.3.2.1 `Interact()`

```
void Cleanup_Interactor.Interact (  
    GameObject obj)
```

Executes the interaction logic when the player interacts with this object.

Parameters

<i>obj</i>	The GameObject that is being interacted with.
------------	---

This method checks if the interaction key (E) is pressed, and if so, it disables the object, effectively removing it from the scene. A debug message is logged to confirm the action.

3.3.2.2 IsAccessible()

```
void Cleanup_Interactor.IsAccessible (  
    GameObject obj)
```

Indicates whether the object is accessible for interaction.

This method is intended to provide feedback or state information about whether the object can be interacted with. Currently, it is not implemented.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Interaction/Click_Interaction/Cleanup_Interactor.cs

3.4 Cloth_Interactor Class Reference

Handles the interaction between player and cloths laying around.

Public Member Functions

- void **Interact** (GameObject obj)
Executes the interaction logic when the player interacts with this object.
- void **IsAccessible** (GameObject obj)
Indicates whether the object is accessible for interaction.
- void **IsNotAccessible** ()

Public Member Functions inherited from Interactable

- void **Interact** (GameObject obj)
- void **IsAccessible** (GameObject obj)
- void **IsNotAccessible** ()

Public Attributes

- AudioSource **src**

3.4.1 Detailed Description

Handles the interaction between player and cloths laying around.

Features:

- Trigger for interaction
- Receive the hit object
- Check if the object is in a specific range for interaction

Requirements:

- Check each frame for interaction

How to Use: Give this component to the player in unity and add another class using @Interactable to any gameobject you'd like to interact with.

Author

Timo Skrobanek

Date

18.8.2024

Version

1.0

3.4.2 Member Function Documentation

3.4.2.1 Interact()

```
void Cloth_Interactor.Interact (  
    GameObject obj)
```

Executes the interaction logic when the player interacts with this object.

Parameters

<i>obj</i>	The GameObject that is being interacted with.
------------	---

This method checks if the interaction key (E) is pressed, and if so, it triggers the logic to change the player's outfit. The actual animation implementation is pending.

3.4.2.2 IsAccessible()

```
void Cloth_Interactor.IsAccessible (
    GameObject obj)
```

Indicates whether the object is accessible for interaction.

This method is intended to provide feedback or state information about whether the object can be interacted with. Currently, it is not implemented.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Interaction/Click_Interaction/Cloth_Interactor.cs

3.5 Computer_Interactor Class Reference

Handles the interaction between player his computer in office.

Public Member Functions

- void **Interact** (GameObject obj)
Executes the interaction logic when the player interacts with this object.
- void **IsAccessible** (GameObject obj)
Indicates whether the object is accessible for interaction.
- void **IsNotAccessible** ()

Public Member Functions inherited from Interactable

- void **Interact** (GameObject obj)
- void **IsAccessible** (GameObject obj)
- void **IsNotAccessible** ()

Public Attributes

- GameObject **key**

3.5.1 Detailed Description

Handles the interaction between player his computer in office.

Features:

- Trigger for interaction
- Receive the hit object
- Check if the object is in a specific range for interaction

Requirements:

- Check each frame for interaction

How to Use: Give this component to the player in unity and add another class using @Interactable to any gameobject you'd like to interact with.

Author

Timo Skrobanek

Date

1.9.2024

Version

1.0

3.5.2 Member Function Documentation

3.5.2.1 Interact()

```
void Computer_Interactor.Interact (  
    GameObject obj)
```

Executes the interaction logic when the player interacts with this object.

Parameters

<i>obj</i>	The GameObject that is being interacted with.
------------	---

This method checks if the interaction key (E) is pressed, and if so, it triggers the logic to change the player's outfit. The actual animation implementation is pending.

3.5.2.2 IsAccessible()

```
void Computer_Interactor.IsAccessible (
    GameObject obj)
```

Indicates whether the object is accessible for interaction.

This method is intended to provide feedback or state information about whether the object can be interacted with. Currently, it is not implemented.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Interaction/Click_Interaction/Computer_Interactor.cs

3.6 EndAnimation Class Reference

Handles the delay before transitioning to the credits scene after the end animation.

Public Member Functions

- void **Start** ()
Starts the coroutine that waits before loading the next scene.

3.6.1 Detailed Description

Handles the delay before transitioning to the credits scene after the end animation.

This class waits for a specified duration and then loads the "Credits" scene. Useful for creating a timed transition after a concluding animation.

3.6.2 Member Function Documentation

3.6.2.1 Start()

```
void EndAnimation.Start ()
```

Starts the coroutine that waits before loading the next scene.

This method initiates the `waitForEnd` coroutine to handle the delay.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Animations/EndAnimation.cs

3.7 EscapeMenu Class Reference

Public Member Functions

- void **Start** ()
- void **Update** ()
- void **ResumeToGame** ()
- void **ChangeToMenu** ()

Public Attributes

- Canvas **canvas**

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Menu/EscapeMenu.cs

3.8 FileEncryptionTests Class Reference

Classes

- class **EncryptionHelper**

Public Member Functions

- void **SetUp** ()
- void **TearDown** ()
- void **TestFileCreationAndEncryption** ()
- void **TestSimpleFileCreation** ()
- void **TestMissingFile_CreatesNewFile** ()
- void **TestEmptyFile_HandlesGracefully** ()
- void **TestCorruptedFile_HandlesGracefully** ()
- void **TestLargeFile_SaveAndLoad** ()

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Test/UnitTests/SaveAndLoadTest.cs

3.9 Interactable Class Reference

Handles the interact-action between player and gameobjects.

Public Member Functions

- void **Interact** (GameObject obj)
- void **IsAccessible** (GameObject obj)
- void **IsNotAccessible** ()

3.9.1 Detailed Description

Handles the interact-action between player and gameobjects.

Using this interface a gameobject, has an interaction ability

Features:

- Trigger for interaction
- Receive the hit object
- Check if the object is in a specific range for interaction

Requirements:

- When the object including this interface is hit by the ray sent from the user `@Interact()` is executed
- When the object including this interface is in a predefined distance from the interactor `@IsAccessable()` is executed

How to Use: Extend a class by this interface and give it to the gameobject as component. Finally give the player `@PlayerInteractionComponent` resulting in a working system.

Author

Timo Skrobanek

Date

18.8.2024

Version

1.0

The documentation for this class was generated from the following file:

- `GetOut/Assets/Scripts/Interaction/Interactable.cs`

3.10 InteractionComponent Class Reference

Public Member Functions

- bool **CheckInteraction** (Ray directedRay)
Checks if the directed ray intersects with an interactable object.
- Transform **GetInteractor** ()
- void **SetInteractor** (Transform value)
- float **GetInteractRange** ()
- void **SetInteractRange** (float value)

Public Attributes

- Transform **interactor**
The Transform of the camera or the head where the raycast should start.
- float **interactRange** = 5f
The maximum range within which the player can interact with objects.

3.10.1 Member Function Documentation

3.10.1.1 CheckInteraction()

```
bool InteractionComponent.CheckInteraction (
    Ray directedRay)
```

Checks if the directed ray intersects with an interactable object.

Parameters

<i>directedRay</i>	The ray cast from the interactor to check for interactions.
--------------------	---

This method uses a raycast to detect if any objects within the interaction range are interactable. If an interactable object is detected, the interaction method on that object is called.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Interaction/PlayerInteractionComponent.cs

3.11 InteractionTest Class Reference

Public Member Functions

- void **Setup** ()
- void **Teardown** ()
- void **CheckInteractionFailed** ()
- void **CheckInteractionSuccessful** ()

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Test/UnitTests/InteractionTest.cs

3.12 Letter_Interactor Class Reference

Handles the interaction logic for readable items.

Public Member Functions

- void **Interact** (GameObject obj)
Executes the interaction logic when the player interacts with this object.
- void **IsAccessible** (GameObject obj)
Indicates whether the object is accessible for interaction.
- void **IsNotAccessible** ()

Public Member Functions inherited from Interactable

- void **Interact** (GameObject obj)
- void **IsAccessible** (GameObject obj)
- void **IsNotAccessible** ()

Public Attributes

- GameObject **letter**
- GameObject **stateManager**

3.12.1 Detailed Description

Handles the interaction logic for readable items.

This component is designed to be attached to gameobject. Providing an id you'll be able to read items.

Features:

- Show text when clicking on an object containing this component

Requirements:

- Check if the item is clicked with "E"
-

How to Use: Give this component to any gameobject you'd like to cleanup or make disappear when clicked.

Author

Timo Skrobanek

Date

18.8.2024

Version

1.0

3.12.2 Member Function Documentation

3.12.2.1 Interact()

```
void Letter_Interactor.Interact (  
    GameObject obj)
```

Executes the interaction logic when the player interacts with this object.

Parameters

<i>obj</i>	The GameObject that is being interacted with.
------------	---

This method checks if the interaction key (E) is pressed, and if so, it disables the object, effectively removing it from the scene. A debug message is logged to confirm the action.

3.12.2.2 IsAccessible()

```
void Letter_Interactor.IsAccessible (  
    GameObject obj)
```

Indicates whether the object is accessible for interaction.

This method is intended to provide feedback or state information about whether the object can be interacted with. Currently, it is not implemented.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Interaction/Click_Interaction/Letter_Interactor.cs

3.13 MainMenu Class Reference

Public Member Functions

- void **PlayGame** ()
- void **QuitGame** ()

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Menu/MainMenu.cs

3.14 SettingsMenuTest.MockSceneManager Class Reference

Public Member Functions

- void **LoadScene** (int index)

Properties

- int **ActiveSceneIndex** = 0 [get]

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Test/UnitTests/SettingsMenuTest.cs

3.15 PlayerMovement Class Reference

Handles the movement logic for the player character in Unity.

Public Member Functions

- void **Start** ()
*Initializes the **PlayerMovement** (p. 19) script.*
- void **Update** ()
Handles player input for movement.
- void **LateUpdate** ()
Handles camera rotation based on mouse input.
- void **rotationUpdate** (float mouseX, float mouseY)
Updates the camera rotation based on mouse movement.
- Vector3 **PlayerMove** (float x, float z)
Moves the player based on input.
- void **SetGravity** (float value)
- void **SetRunMultiplier** (float value)
- void **SetMouseSensitivity** (float value)
- float **GetSpeed** ()
- float **GetGravity** ()
- float **GetRunMultiplier** ()
- float **GetMouseSensitivity** ()
- object **GetVelocity** ()

Public Attributes

- CharacterController **charController**
Reference to the CharacterController component.
- float **yRotation** = -90f
Starting yaw (y-axis) rotation.
- float **mouseSensitivity**
Sensitivity of mouse movement for camera rotation.
- Transform **mainCam**
Reference to the main camera transform for rotating the camera.

3.15.1 Detailed Description

Handles the movement logic for the player character in Unity.

This script allows the player to move in various directions using keyboard input, and it manages both walking and running speeds. The script is designed to work with Unity's Rigidbody component for physics-based movement.

Features:

- Supports basic movement (forward, backward, left, right) using WASD or arrow keys.
- Allows the player to sprint by holding down a designated key (e.g., Left Shift).
- Implements smooth movement by interpolating the player's velocity.

- Adjusts movement speed based on whether the player is walking or running.
- Includes optional jump functionality if the player is grounded.

Requirements:

- This script should be attached to a GameObject with a Rigidbody component.
- The GameObject should also have a Collider component for proper collision detection.

How to Use:

- Attach the **PlayerMovement** (p. 19) script to your player GameObject.
- Configure movement speed, running speed, and jump force in the Unity Inspector.
- Ensure the Rigidbody component is set to "Interpolate" for smooth movement.
- Set the correct input axes in the Unity Input Manager (default is WASD or arrow keys).

Author

Timo Skrobanek

Date

18.8.2024

Version

1.0

3.15.2 Member Function Documentation

3.15.2.1 LateUpdate()

```
void PlayerMovement.LateUpdate ()
```

Handles camera rotation based on mouse input.

This method is called after all Update functions have been called. It captures mouse movement input and calls **rotationUpdate()** (p. 21) to update the camera rotation, provided motion is not disabled.

3.15.2.2 PlayerMove()

```
Vector3 PlayerMovement.PlayerMove (  
    float x,  
    float z)
```

Moves the player based on input.

Parameters

<i>x</i>	Horizontal movement input.
<i>z</i>	Vertical movement input.

This method calculates the player's movement vector based on the input axes and the player's orientation relative to the camera. It also applies gravity to keep the player grounded and checks if the player is running to adjust the movement speed.

3.15.2.3 rotationUpdate()

```
void PlayerMovement.rotationUpdate (
    float mouseX,
    float mouseY)
```

Updates the camera rotation based on mouse movement.

Parameters

<i>mouseX</i>	Horizontal mouse movement.
<i>mouseY</i>	Vertical mouse movement.

This method updates the player's camera rotation by applying pitch and yaw rotations. The camera's pitch rotation (looking up and down) is clamped to prevent extreme angles.

3.15.2.4 Start()

```
void PlayerMovement.Start ()
```

Initializes the **PlayerMovement** (p. 19) script.

This method is called on the first frame when the script is enabled. It sets up the CharacterController component, locks the cursor to the center of the screen, and plays the introductory animation.

3.15.2.5 Update()

```
void PlayerMovement.Update ()
```

Handles player input for movement.

This method is called once per frame. It checks if the introductory animation has finished, then captures player input for movement and calls the **PlayerMove()** (p. 20) method to move the player accordingly.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Movement/PlayerMovement.cs

3.16 PlayerMovementTests Class Reference

Public Member Functions

- void **Setup** ()
- void **Teardown** ()
- void **PlayerMove_NormalMovement** ()
- void **PlayerMove_WithRunning_SpeedIncreases** ()
- void **Gravity_AppliedCorrectly** ()
- void **CameraRotation_LimitedCorrectly** ()
- IEnumerator **Cursor_LockedOnStart** ()

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Test/UnitTests/PlayerMovementTest.cs

3.17 PlayerRotation Class Reference

Handles the rotation logic for the player character in Unity.

Public Member Functions

- void **Start** ()
- void **LateUpdate** ()
Handles camera rotation based on mouse input.
- void **rotationUpdate** (float mouseX, float mouseY)
Updates the camera rotation based on mouse movement.

Public Attributes

- float **mouseSensitivity**
Sensitivity of mouse movement for camera rotation.
- Transform **mainCam**
Reference to the main camera transform for rotating the camera.

3.17.1 Detailed Description

Handles the rotation logic for the player character in Unity.

This script allows the player to rotate the camera inside a predefined angle. In Scene 3 of the game, the player is only allowed to move his head. With a few more requirements it's more efficient to have an disjoint script for head rotation.

Features:

- Player rotation in x and y axes inside predefined angles

Requirements:

- This script should be attached to a GameObject with a Rigidbody component.
- The GameObject should also have a Collider component for proper collision detection.

How to Use:

- Attach the **PlayerMovement** (p. 19) script to your player GameObject.
- Configure rotation speed and limits.
- Set the correct input axes in the Unity Input Manager (default is WASD or arrow keys).

Author

Timo Skrobanek

Date

28.10.2024

Version

2.0

3.17.2 Member Function Documentation

3.17.2.1 LateUpdate()

```
void PlayerRotation.LateUpdate ()
```

Handles camera rotation based on mouse input.

This method is called after all Update functions have been called. It captures mouse movement input and calls **rotationUpdate()** (p. 23) to update the camera rotation, provided motion is not disabled.

3.17.2.2 rotationUpdate()

```
void PlayerRotation.rotationUpdate (  
    float mouseX,  
    float mouseY)
```

Updates the camera rotation based on mouse movement.

Parameters

<i>mouseX</i>	Horizontal mouse movement.
<i>mouseY</i>	Vertical mouse movement.

This method updates the player's camera rotation by applying pitch and yaw rotations. The camera's pitch rotation (looking up and down) is clamped to prevent extreme angles.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Movement/PlayerRotation.cs

3.18 SaveData Class Reference

Public Attributes

- int **lastSceneIndex**
- Dictionary< string, object > **globalVariables** = new Dictionary<string, object>()

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/SaveAndLoad/SaveAndLoadManager.cs

3.19 SaveLoadManager Class Reference

Public Member Functions

- void **SaveGame** ()
- void **LoadGame** ()
- void **SetGlobalVariable** (string key, object value)
- object **GetGlobalVariable** (string key)

Properties

- static **SaveLoadManager Instance** [get]

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/SaveAndLoad/SaveAndLoadManager.cs

3.20 Scene1_Change_Interactor Class Reference

Handles the change between scenes.

Public Member Functions

- void **Interact** (GameObject obj)
- void **IsAccessible** (GameObject obj)
- void **IsNotAccessible** ()

Public Member Functions inherited from Interactable

- void **Interact** (GameObject obj)
- void **IsAccessible** (GameObject obj)
- void **IsNotAccessible** ()

3.20.1 Detailed Description

Handles the change between scenes.

Features:

- Trigger for interaction
- Receive the hit object
- Check if the object is in a specific range for interaction

Requirements:

- Check each frame for interaction

How to Use: Give this component to the player in unity and add another class using @Interactable to any gameobject you'd like to interact with.

Author

Timo Skrobanek

Date

2.9.2024

Version

1.0

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Interaction/Scene_Change_Interactor.cs

3.21 SettingsMenu Class Reference

Public Member Functions

- void **SetVolume** (float volume)
- void **SetQuality** (int qualityIndex)
- Resolution **SetResolution** (int resolutionIndex)

Public Attributes

- AudioMixer **audioMixer**

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Menu/SettingsMenu.cs

3.22 SettingsMenuTest Class Reference

Classes

- class **MockSceneManager**

Public Member Functions

- void **Setup** ()
- void **Teardown** ()
- void **SceneExistsInBuildSettings** ()
- void **QuitGame_LogsQuitMessage** ()
- void **PlayGame_CallsLoadScene** ()

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Test/UnitTests/SettingsMenuTest.cs

3.23 Sleep_Interactor Class Reference

Public Member Functions

- void **Interact** (GameObject obj)
Executes the interaction logic when the player interacts with this object.
- void **IsAccessible** (GameObject obj)
Indicates whether the object is accessible for interaction.
- void **IsNotAccessible** ()

Public Member Functions inherited from Interactable

- void **Interact** (GameObject obj)
- void **IsAccessable** (GameObject obj)
- void **IsNotAccessable** ()

Public Attributes

- **AnimationHandler** handler

3.23.1 Member Function Documentation

3.23.1.1 Interact()

```
void Sleep_Interactor.Interact (
    GameObject obj)
```

Executes the interaction logic when the player interacts with this object.

Parameters

<i>obj</i>	The GameObject that is being interacted with.
------------	---

This method checks if the interaction key (E) is pressed, and if so, it disables the object, effectively removing it from the scene. A debug message is logged to confirm the action.

3.23.1.2 IsAccessable()

```
void Sleep_Interactor.IsAccessable (
    GameObject obj)
```

Indicates whether the object is accessible for interaction.

This method is intended to provide feedback or state information about whether the object can be interacted with. Currently, it is not implemented.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Interaction/Click_Interaction/Sleep_Interactor.cs

3.24 Smiler_JS Class Reference

Controls the visibility of the "smiler" GameObject based on game state.

Public Attributes

- GameObject **smiler**

3.24.1 Detailed Description

Controls the visibility of the "smiler" GameObject based on game state.

This class disables the MeshRenderer of the "smiler" object when the game enters SCENE4, making it invisible.

3.24.2 Member Data Documentation

3.24.2.1 smiler

`GameObject Smiler_JS.smiler`

The GameObject representing the smiler character.

The documentation for this class was generated from the following file:

- `GetOut/Assets/Scripts/Animations/Smiler_JS.cs`

3.25 Startbutton Class Reference

Public Member Functions

- void **startGame** ()

Public Attributes

- Button **yourButton**

The documentation for this class was generated from the following file:

- `GetOut/Assets/Scripts/Menu/Startbutton.cs`

3.26 StateManager Class Reference

Public Types

- enum **State** {
SCENE1 = 0 , **SCENE1_INTRO_ANIMATION** = 1 , **SCENE1_CLEANUP_DONE** = 2 , **SCENE1__NOT_↵**
DRESSED = 3 ,
SCENE1_COMPLETED = 4 , **SCENE2** = 5 , **SCENE2_COMPLETED** = 7 , **SCENE3** = 6 ,
SCENE3_OUTRO = 8 , **SCENE3_OUTRO_DONE** = 9 , **SCENE4_INTRO** = 10 , **SCENE4** = 11 ,
SCENE4_GHOST_APPEAR = 12 , **SCENE5** , **ESCAPE_MENU** = 13 }

Public Member Functions

- void **Start** ()
- void **UpdateGameState** (**SaveData** newGameState)

Static Public Member Functions

- static void **startScene1** ()
- static bool **ItemsLeftToCleanup** ()
- static void **cleanUp** ()
- static void **changeOutfit** ()
- static bool **outfitChanged** ()
- static bool **scene1Complete** ()
- static void **stopIntroAnimation** ()
- static void **StopIntroAnimationScene4** ()

Public Attributes

- **SaveData** currentGameState

Static Public Attributes

- static State **state** = State.SCENE1_INTRO_ANIMATION

Properties

- static **StateManager Instance** [get]
- static int **CleanedItems** = cleanedItems [get]
- static bool **Outfit** = outfit [get]
- static bool **IntroToggle** = introToggle [get]
- static bool **Intro4Toggle** = intro4Toggle [get]

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/StoryStateManager/StateManager.cs

3.27 StateManagerTest Class Reference

Public Member Functions

- void **SetUp** ()
- void **TearDown** ()
- void **TestStartScene1** ()
- void **TestChangeOutfit** ()
- void **TestScene1Complete** ()
- void **TestStopIntroAnimation** ()

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Test/UnitTests/StateManagerTest.cs

3.28 Test Class Reference

Public Member Functions

- void **TestSimplePasses** ()
- IEnumerator **TestWithEnumeratorPasses** ()

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Test/UnitTests/EditModeTest/Test.cs

3.29 TriggerGhostVoice Class Reference

Plays a ghost voice sound effect when the player enters a specified trigger area.

Public Attributes

- AudioSource **src**

3.29.1 Detailed Description

Plays a ghost voice sound effect when the player enters a specified trigger area.

This class activates an audio source when the player enters the trigger zone, playing a sound effect once and preventing further activation.

Author

Timo Skrobanek

Date

28.10.2024

Version

1.0

3.29.2 Member Data Documentation

3.29.2.1 src

AudioSource `TriggerGhostVoice.src`

Audio source that plays the ghost voice sound effect.

The documentation for this class was generated from the following file:

- GetOut/Assets/Scripts/Interaction/Movement_Interaction/TriggerGhostVoice.cs

3.30 TriggerMovingPerson Class Reference

Handles triggering animations, sounds, and object states when the player enters a specific area.

Public Attributes

- AudioSource **src**
- GameObject **tableUpsideDown**
- GameObject **table**
- Animator **anim**
- GameObject **doorsOpen**
- GameObject **doorsClosed**

3.30.1 Detailed Description

Handles triggering animations, sounds, and object states when the player enters a specific area.

This class initiates a sequence of actions when the player enters the trigger zone: it updates the game state, plays an audio clip, enables an animation, and toggles visibility of objects (e.g., tables and doors).

The script ensures that the sequence is only triggered once upon the first entry of the player into the trigger zone.

Author

Timo Skrobanek

Date

10.10.2024

Version

1.0

3.30.2 Member Data Documentation

3.30.2.1 anim

`Animator TriggerMovingPerson.anim`

Animator component that controls the movement of a character or object.

3.30.2.2 doorsClosed

`GameObject TriggerMovingPerson.doorsClosed`

GameObject representing closed doors that will be toggled on.

3.30.2.3 doorsOpen

`GameObject TriggerMovingPerson.doorsOpen`

GameObject representing open doors that will be toggled off.

3.30.2.4 src

`AudioSource TriggerMovingPerson.src`

Audio source for playing a sound effect upon triggering.

3.30.2.5 table

`GameObject TriggerMovingPerson.table`

GameObject representing the table in its original position.

3.30.2.6 tableUpsideDown

`GameObject TriggerMovingPerson.tableUpsideDown`

GameObject representing the table in an upside-down state.

The documentation for this class was generated from the following file:

- `GetOut/Assets/Scripts/Interaction/Movement_Interaction/TriggerMovingPerson.cs`

3.31 UI_Updater Class Reference

Public Member Functions

- void **Start** ()
- void **Update** ()

Public Attributes

- TMP_Text **task**

The documentation for this class was generated from the following file:

- `GetOut/Assets/Scripts/StoryStateManager/UI_Updater.cs`

Index

- anim
 - TriggerMovingPerson, 31
- AnimationHandler, 5
 - animator, 6
 - CheckState, 6
 - Start, 6
 - Update, 6
- animator
 - AnimationHandler, 6
- Bed_Interactor, 6
 - Interact, 7
 - IsAccessible, 7
- CheckInteraction
 - InteractionComponent, 16
- CheckState
 - AnimationHandler, 6
- Cleanup_Interactor, 7
 - Interact, 8
 - IsAccessible, 9
- Cloth_Interactor, 9
 - Interact, 10
 - IsAccessible, 10
- Computer_Interactor, 11
 - Interact, 12
 - IsAccessible, 12
- doorsClosed
 - TriggerMovingPerson, 31
- doorsOpen
 - TriggerMovingPerson, 31
- EndAnimation, 13
 - Start, 13
- EscapeMenu, 14
- FileEncryptionTests, 14
- Interact
 - Bed_Interactor, 7
 - Cleanup_Interactor, 8
 - Cloth_Interactor, 10
 - Computer_Interactor, 12
 - Letter_Interactor, 17
 - Sleep_Interactor, 27
- Interactable, 14
- InteractionComponent, 15
 - CheckInteraction, 16
- InteractionTest, 16
- IsAccessible
 - Bed_Interactor, 7
 - Cleanup_Interactor, 9
 - Cloth_Interactor, 10
 - Computer_Interactor, 12
 - Letter_Interactor, 18
 - Sleep_Interactor, 27
- LateUpdate
 - PlayerMovement, 20
 - PlayerRotation, 23
- Letter_Interactor, 16
 - Interact, 17
 - IsAccessible, 18
- MainMenu, 18
- PlayerMove
 - PlayerMovement, 20
- PlayerMovement, 19
 - LateUpdate, 20
 - PlayerMove, 20
 - rotationUpdate, 21
 - Start, 21
 - Update, 21
- PlayerMovementTests, 22
- PlayerRotation, 22
 - LateUpdate, 23
 - rotationUpdate, 23
- rotationUpdate
 - PlayerMovement, 21
 - PlayerRotation, 23
- SaveData, 24
- SaveLoadManager, 24
- Scene1_Change_Interactor, 24
- SettingsMenu, 26
- SettingsMenuTest, 26
- SettingsMenuTest.MockSceneManager, 18
- Sleep_Interactor, 26
 - Interact, 27
 - IsAccessible, 27
- smiler
 - Smiler_JS, 28
- Smiler_JS, 27
 - smiler, 28
- src
 - TriggerGhostVoice, 30
 - TriggerMovingPerson, 32
- Start
 - AnimationHandler, 6

- EndAnimation, 13
- PlayerMovement, 21
- Startbutton, 28
- StateManager, 28
- StateManagerTest, 29
- table
 - TriggerMovingPerson, 32
- tableUpsideDown
 - TriggerMovingPerson, 32
- Test, 30
- TriggerGhostVoice, 30
 - src, 30
- TriggerMovingPerson, 31
 - anim, 31
 - doorsClosed, 31
 - doorsOpen, 31
 - src, 32
 - table, 32
 - tableUpsideDown, 32
- UI_Updater, 32
- Update
 - AnimationHandler, 6
 - PlayerMovement, 21

Code Dokumentation

Datei: ../../Animations/AnimationHandler.cs

```
/**

 * @class AnimationHandler

 * @brief Behandelt die Animationen aller Szenen im Spiel und steuert diese abhängig vom
aktuellen Spielstatus.

 *

 * Diese Klasse überprüft regelmäßig den Zustand des Spiels und startet entsprechende
Animationen für

 * verschiedene Szenen. Die Klasse verwendet einen Animator, um Übergänge und
Animationen basierend

 * auf den Zuständen im StateManager zu steuern.

 *

 * @author Timo Skrobanek

 * @date 28.10.2024

 * @version 2.0

 */

using Unity.VisualScripting;

using UnityEngine;

using UnityEngine.SceneManagement;

public class AnimationHandler : MonoBehaviour {

    /** Animator, der die verschiedenen Szenen-Animationen steuert. */
```

```

public Animator animator;

/**
 * @brief Initialisiert die Animationen beim Start des Spiels.
 *
 * Diese Methode überprüft den Spielstatus beim Start und startet ggf. die passende
Animation.
 */

public void Start() {

    CheckState();

    Debug.Log("Checking state for animation");
}

/**
 * @brief Aktualisiert die Animationen in jeder Frame-Iteration.
 *
 * Diese Methode wird in jedem Frame aufgerufen und prüft, ob Animationen
abgeschlossen sind,
 * um entsprechende Maßnahmen zu ergreifen.
 */

public void Update() {

    FinishAnimation();
}

/**
 * @brief Überprüft den aktuellen Spielstatus und startet die passende Animation.
 *

```

```

    * Diese Methode wechselt zwischen den verschiedenen Spielzuständen und spielt die
    jeweilige Animation

    * basierend auf dem Zustand im StateManager.

    */

    public void CheckState() {

        switch (StateManager.state) {

            case StateManager.State.SCENE1_INTRO_ANIMATION:

                // Startet die Intro-Animation für Szene 1

                animator.Play("Scenel_Intro");

                break;

            case StateManager.State.SCENE3_OUTRO:

                // Startet die Outro-Animation für Szene 3

                animator.Play("Scene3_Outro");

                break;

            case StateManager.State.SCENE4_INTRO:

                // Startet die Intro-Animation für Szene 4

                animator.Play("Scene4_Intro");

                break;

        }

    }

    /**

    * @brief Beendet die Animation und führt Spielaktionen nach Abschluss der Animation
    durch.

    *

```

* Diese Methode prüft, ob die laufende Animation beendet ist und schaltet dann den Animator ab.

* Bei bestimmten Zuständen wird der Spielstatus aktualisiert und es wird ggf. eine neue Szene geladen.

*/

```
private void FinishAnimation() {

    if (animator.GetCurrentAnimatorStateInfo(0).IsName("Intro_Done")) {

        animator.enabled = false;

        StateManager.stopIntroAnimation();

    }

    else if (animator.GetCurrentAnimatorStateInfo(0).IsName("Scene3_Outro")) {

        Debug.Log("Loading Scene 4");

        animator.enabled = false;

        StateManager.state = StateManager.State.SCENE4_INTRO;

        SceneManager.LoadScene("Scene4");

    }

    else if (animator.GetCurrentAnimatorStateInfo(0).IsName("Scene4_Done")) {

        animator.enabled = false;

        StateManager.StopIntroAnimationScene4();

    }

    else if (animator.GetCurrentAnimatorStateInfo(0).IsName("Ghost_Appeared")) {

        animator.enabled = false;

        StateManager.state = StateManager.State.SCENE4;

    }

}
```

Date: ..\..\Animations\EndAnimation.cs

```
/**

 * @class EndAnimation

 * @brief Handles the delay before transitioning to the credits scene after the end
animation.

 *

 * This class waits for a specified duration and then loads the "Credits" scene.

 * Useful for creating a timed transition after a concluding animation.

 */

using System.Collections;

using UnityEngine;

using UnityEngine.SceneManagement;

public class EndAnimation : MonoBehaviour

{

    /**

    * @brief Starts the coroutine that waits before loading the next scene.

    *

    * This method initiates the waitForEnd coroutine to handle the delay.

    */

    public void Start() {

        StartCoroutine(waitForEnd());

    }

    /**
```

```

    * @brief Coroutine that waits for a set period before loading the credits scene.
    *
    * This coroutine waits for 7 seconds in real-time and then loads the "Credits"
scene
    * via SceneManager.
    *
    * @return IEnumerator Required for coroutine functionality.
    */
IEnumerator waitForEnd() {
    Debug.Log("Waiting for end");

    // Wait for 7 seconds in real time
    yield return new WaitForSecondsRealtime(7);

    Cursor.lockState = CursorLockMode.None;

    Cursor.visible = true;

    SceneManager.LoadScene("Credits");
}
}

```

Datei: ..\..\Animations\Smiler_JS.cs

```

/**
 * @class Smiler_JS
 *
 * @brief Controls the visibility of the "smiler" GameObject based on game state.
 *
 * This class disables the MeshRenderer of the "smiler" object when the game enters
SCENE4,

```

```

    * making it invisible.

    */

using UnityEngine;

public class Smiler_JS : MonoBehaviour

{

    /** The GameObject representing the smiler character. */

    public GameObject smiler;

    /**

    * @brief Checks the game state and updates the smiler's visibility.

    *

    * This method is called once per frame. It disables the MeshRenderer component of
the
    * smiler object when the game state is set to SCENE4, hiding the smiler in that
scene.

    */

    void Update()

    {

        if (StateManager.state == StateManager.State.SCENE4) {

            smiler.GetComponent<MeshRenderer>().enabled = false;

        }

    }

}

```

Date: ..\..\Interaction\Interactable.cs


```
/**

* @class Interactable

* @brief Handles the interact-action between player and gameobjects

*

* Using this interface a gameobject, has an interaction ability

*

* @details

* Features:

* - Trigger for interaction

* - Receive the hit object

* - Check if the object is in a specific range for interaction

*

* Requirements:

* - When the object including this interface is hit by the ray sent from the user

@Interact() is executed

* - When the object including this interface is in a predefined distance from the

interactor @IsAccessable() is executed

*

* How to Use:

* Extend a class by this interface and give it to the gameobject as component. Finally

give the player

* @PlayerInteractionComponent resulting in a working system.

*

* @author Timo Skrobanek

* @date 18.8.2024

* @version 1.0

*/
```

```

using UnityEngine;

interface Interactable

{

    /*

        @brief executed when the interaction with the given object is going on right now

        @param obj GameObject that is aimed for from users view.

    */

    public void Interact(GameObject obj);

    /*

        @brief executed when the interaction with the given object is possible from a
certain distance

    */

    public void IsAccessable(GameObject obj);

    public void IsNotAccessable();

}

```

Datei: ..\..\Interaction\PlayerInteractionComponent.cs

```

using System;

using UnityEngine;

public class InteractionComponent : MonoBehaviour

{

    /// @brief The Transform of the camera or the head where the raycast should start.

```

```

public Transform interactor;

/// @brief The maximum range within which the player can interact with objects.

public float interactRange = 5f;

/// <summary>

/// @brief the last interactable object. Needed for user interface

/// </summary>

private Interactable interactObj;

/**

 * @brief Called once per frame to check for interactions.

 *

 * This method casts a ray from the interactor's (camera or head's) position
forward,

 * checking if it hits any objects within the specified interaction range.

 */

void Update()

{

    // Ray originates from the interactor (camera/head) position, going forward in
the camera's forward direction

    Ray directedRay = new Ray(interactor.position, interactor.forward);

    // Check for interactions with objects the ray hits

    CheckInteraction(directedRay);

}

```

```

/**
 * @brief Checks if the directed ray intersects with an interactable object.
 *
 * @param directedRay The ray cast from the interactor to check for interactions.
 *
 * This method uses a raycast to detect if any objects within the interaction range
 * are interactable. If an interactable object is detected, the interaction method
 * on that object is called.
 */
public bool CheckInteraction(Ray directedRay)
{
    // Check if the ray hits any objects within the interaction range
    if (Physics.Raycast(directedRay, out RaycastHit hitInfo, interactRange))
    {
        // Check if the hit object has an Interactable component
        if (hitInfo.collider.gameObject.TryGetComponent(out Interactable
interactObj))
        {
            // Optional: Show that the object is in range to interact
            interactObj.IsAccessible(hitInfo.collider.gameObject);

            this.interactObj = interactObj;

            // Execute interaction with the object
            interactObj.Interact(hitInfo.collider.gameObject);

            return true;
        }

        //if there's no hit, (try to) hide the last stored ui element

```

```

        else{

            try{

                this.interactObj.IsNotAccessable();

                return true;

            }catch (Exception)

            {

                return false;

            }

        }

        return false;

    }
}

```

// Getter und Setter

```
public Transform GetInteractor()
```

```

{

    return interactor;

}

```

```
public void SetInteractor(Transform value)
```

```

{

    interactor = value;

}

```

```
public float GetInteractRange()
```

```

{

```

```

        return interactRange;
    }

    public void SetInteractRange(float value)
    {
        interactRange = value;
    }

    internal Interactable GetInteractObj()
    {
        return interactObj;
    }

    internal void SetInteractObj(Interactable value)
    {
        interactObj = value;
    }
}

```

Datei: ..\..\Interaction\Scene_Change_Interactor.cs

```

/**
 * @class Scene1_Change_Interactor
 * @brief Handles the change between scenes
 *
 *
 * @details

```

* Features:

* - Trigger for interaction

* - Receive the hit object

* - Check if the object is in a specific range for interaction

*

* Requirements:

* - Check each frame for interaction

*

* How to Use:

* Give this component to the player in unity and add another class using @Interactable
to any gameobject you'd like to

* interact with.

*

* @author Timo Skrobanek

* @date 2.9.2024

* @version 1.0

*/

using System;

using UnityEngine;

using UnityEngine.SceneManagement;

public class Scene1_Change_Interactor : MonoBehaviour, Interactable

{

public void Interact(GameObject obj) {

if (Input.GetKeyDown(KeyCode.E)) {

if (StateManager.state == StateManager.State.SCENE1_COMPLETED) {

```

        Debug.Log("Switching to Scene 2");

        StateManager.state = StateManager.State.SCENE2;

        ChangeScene("Scene2");

    }

    else if(StateManager.state == StateManager.State.SCENE3){

        Debug.Log("Switching to Scene 2");

        ChangeScene("Scene4");

    }

    else{

        Debug.Log("Change not allowed here: " + StateManager.state);

    }

}

}

```

```

public void IsAccessible(GameObject obj){

```

```

}

```

```

/**

```

```

 * @brief Triggers the event to change the scene to @scenename

```

```

 *

```

```

 */

```

```

void ChangeScene(String name){

```

```

    SceneManager.LoadScene(name);

```

```

}

```

```

public void IsNotAccessible(){

```



```
}  
  
}
```

Date: ..\..\Interaction\Click_Interaction\Bed_Interactor.cs

```
/**  
  
 * @class Letter_Interactor  
  
 * @brief Handles the interaction logic for readable items  
  
 *  
  
 * This component is designed to be attached to gameobject. Providing an id you'll be  
  
 * able to read items.  
  
 *  
  
 * @details  
  
 * Features:  
  
 * - Show text when clicking on an object containing this component  
  
 *  
  
 * Requirements:  
  
 * - Check if the item is clicked with "E"  
  
 * -  
  
 *  
  
 * How to Use:  
  
 * Give this component to any gameobject you'd like to cleanup or make disappear when  
clicked.  
  
 *  
  
 * @author Timo Skrobanek  
  
 * @date 18.8.2024  
  
 * @version 1.0  
  
 */
```

```

using UnityEngine;

using UnityEngine.SceneManagement;

public class Bed_Interactor : MonoBehaviour, Interactable

{

    /**

        * @brief Executes the interaction logic when the player interacts with this
object.

        *

        * @param obj The GameObject that is being interacted with.

        *

        * This method checks if the interaction key (E) is pressed, and if so, it
disables the

        * object, effectively removing it from the scene. A debug message is logged to
confirm

        * the action.

        */

    public void Interact(GameObject obj)

    {

        if (Input.GetKeyDown(KeyCode.E)) {

            StateManager.state = StateManager.State.SCENE5;

            SceneManager.LoadScene("Scene5");

        }

    }

    /**

```

```

    * @brief Indicates whether the object is accessible for interaction.

    *

    * This method is intended to provide feedback or state information

        * about whether the object can be interacted with. Currently, it is not
implemented.

    */

    public void IsAccessible(GameObject obj)

    {

        // Implementation pending

    }


    public void IsNotAccessible(){

    }

}

```

Date: ..\..\Interaction\Click_Interaction\Cleanup_Interactor.cs

```

/**

    * @class Cleanup_Interactor

    * @brief Handles the interaction logic for cleaning up or disabling objects in the
game.

    *

    * This component is designed to be attached to objects that should be disabled or
"cleaned up"

    * when the player interacts with them. It implements the Interactable interface.

    *

    * @details

```

* Features:

* - Disable gameobjects when clicked

*

* Requirements:

* - Check each frame for interaction

* - Make gameobjects invisible that got cleaned up

*

* How to Use:

* Give this component to any gameobject you'd like to cleanup or make disappear when clicked.

*

* @author Timo Skrobanek

* @date 18.8.2024

* @version 1.0

*/

using UnityEngine;

public class Cleanup_Interactor : MonoBehaviour, Interactable

{

public GameObject key;

public AudioSource src;

/**

* @brief Executes the interaction logic when the player interacts with this object.

*

* @param obj The GameObject that is being interacted with.

*

```

    * This method checks if the interaction key (E) is pressed, and if so, it disables
the

    * object, effectively removing it from the scene. A debug message is logged to
confirm

    * the action.

    */

public void Interact(GameObject obj)
{
    if (Input.GetKeyDown(KeyCode.E))
    {
        src.Play();

        // On interaction, disable the object's rendering and functionality
        obj.SetActive(false);

        key.GetComponent<MeshRenderer>().enabled = false;

        Debug.Log("Object disabled");

        StateManager.cleanUp();

        if(!StateManager.ItemsLeftToCleanup()){

            StateManager.state = StateManager.State.SCENE1_CLEANUP_DONE;

            Debug.Log("Cleanup task done");

        }

    }

}

/**

    * @brief Indicates whether the object is accessible for interaction.

    *

```

```

    * This method is intended to provide feedback or state information
        * about whether the object can be interacted with. Currently, it is not
implemented.

    */

    public void IsAccessible(GameObject obj)

    {

        key.GetComponent<MeshRenderer>().enabled = true;

    }


    public void IsNotAccessible() {

        key.GetComponent<MeshRenderer>().enabled = false;

    }

}

```

Date: ..\..\Interaction\Click_Interaction\Cloth_Interactor.cs

```

/**

* @class Cloth_Interactor

* @brief Handles the interaction between player and cloths laying around.

*

*

* @details

* Features:

* - Trigger for interaction

* - Receive the hit object

* - Check if the object is in a specific range for interaction

*

* Requirements:

```

```

* - Check each frame for interaction

*

* How to Use:

* Give this component to the player in unity and add another class using @Interactable
to any gameobject you'd like to

* interact with.

*

* @author Timo Skrobanek

* @date 18.8.2024

* @version 1.0

*/

using UnityEngine;

public class Cloth_Interactor : MonoBehaviour, Interactable

{

    public AudioSource src;

    /**

    * @brief Executes the interaction logic when the player interacts with this object.

    *

    * @param obj The GameObject that is being interacted with.

    *

    * This method checks if the interaction key (E) is pressed, and if so, it triggers

    * the logic to change the player's outfit. The actual animation implementation is
pending.

    */

    public void Interact(GameObject obj)

```

```

{

    if (Input.GetKeyDown(KeyCode.E) && !StateManager.outfitChanged() &&
!StateManager.ItemsLeftToCleanup())

    {

        // TODO: Implement animation

        Debug.Log("Change outfit");

        src.Play();

        StateManager.changeOutfit();

    }

}

/**

 * @brief Indicates whether the object is accessible for interaction.

 *

 * This method is intended to provide feedback or state information

    * about whether the object can be interacted with. Currently, it is not
implemented.

 */

public void IsAccessible(GameObject obj)

{

    // Implementation pending

}

public void IsNotAccessible(){

}

}

```


Datei: ../../Interaction/Click_Interaction/Computer_Interactor.cs

```
/**

 * @class Computer_Interactor

 * @brief Handles the interactaction between player his computer in office

 *

 *

 * @details

 * Features:

 * - Trigger for interaction

 * - Receive the hit object

 * - Check if the object is in a specific range for interaction

 *

 * Requirements:

 * - Check each frame for interaction

 *

 * How to Use:

 * Give this component to the player in unity and add another class using @Interactable

to any gameobject you'd like to

 * interact with.

 *

 * @author Timo Skrobanek

 * @date 1.9.2024

 * @version 1.0

 */

using UnityEngine;

using UnityEngine.SceneManagement;
```

```

public class Computer_Interactor : MonoBehaviour, Interactable
{

    public GameObject key;

    /**
     * @brief Executes the interaction logic when the player interacts with this object.
     *
     * @param obj The GameObject that is being interacted with.
     *
     * This method checks if the interaction key (E) is pressed, and if so, it triggers
     * the logic to change the player's outfit. The actual animation implementation is
pending.

    */
    public void Interact(GameObject obj)
    {
        if (Input.GetKeyDown(KeyCode.E))
        {
            // TODO: Implement animation

            Debug.Log("interact with computer");

            key.GetComponent<MeshRenderer>().enabled = false;

            StateManager.state = StateManager.State.SCENE3;

            SceneManager.LoadScene("Scene3");
        }
    }
}

```

```

/**
 * @brief Indicates whether the object is accessible for interaction.
 *
 * This method is intended to provide feedback or state information
 * about whether the object can be interacted with. Currently, it is not
implemented.
 */

public void IsAccessible(GameObject obj)
{
    key.GetComponent<MeshRenderer>().enabled = true;
}

public void IsNotAccessible() {
    key.GetComponent<MeshRenderer>().enabled = false;
}
}

```

Datei: ..\..\Interaction\Click_Interaction\Letter_Interactor.cs

```

/**
 * @class Letter_Interactor
 *
 * @brief Handles the interaction logic for readable items
 *
 * This component is designed to be attached to gameobject. Providing an id you'll be
 * able to read items.
 *
 * @details
 * Features:

```

```

* - Show text when clicking on an object containing this component

*

* Requirements:

* - Check if the item is clicked with "E"

* -

*

* How to Use:

* Give this component to any gameobject you'd like to cleanup or make disappear when
clicked.

*

* @author Timo Skrobanek

* @date 18.8.2024

* @version 1.0

*/

using UnityEngine;

public class Letter_Interactor : MonoBehaviour, Interactable

{

    public GameObject letter;

    public GameObject stateManager;

    //added boolean because of letter bug

    private bool isLetterOpen = false;

    /**

        * @brief Executes the interaction logic when the player interacts with this
object.

        *

```

```

    * @param obj The GameObject that is being interacted with.

    *

    * This method checks if the interaction key (E) is pressed, and if so, it
disables the

    * object, effectively removing it from the scene. A debug message is logged to
confirm

    * the action.

    */

public void Interact(GameObject obj)
{
    // Move the letter back along the Z-axis to make it appear farther away

    letter.transform.position += new Vector3(0, 0, -1f); // Move the letter 1
unit farther from the camera


    if (Input.GetKeyDown(KeyCode.E) && !isLetterOpen)
    {

        // On interaction, disable the object's rendering and functionality

        letter.SetActive(true);

        Cursor.lockState = CursorLockMode.None;

        Cursor.visible = false;

        isLetterOpen = true;

    }


    // After reading letter, press escape in order to continue with the game

    if (isLetterOpen && (Input.GetKeyDown(KeyCode.R) ||
Input.GetKeyDown(KeyCode.Escape)))

```

```

    {

        // close letter and lock cursor

        letter.SetActive(false);

        Cursor.lockState = CursorLockMode.Locked; // Locks cursor

        Cursor.visible = false; // Hide cursor

        isLetterOpen = false;

    }

}

/**

 * @brief Indicates whether the object is accessible for interaction.

 *

 * This method is intended to provide feedback or state information

 * about whether the object can be interacted with. Currently, it is not

implemented.

 */

public void IsAccessable(GameObject obj)

{

    // Implementation pending

}

public void IsNotAccessable(){

}

}

```

Datei: ..\..\Interaction\Click_Interaction\Sleep_Interactor.cs

```

/**
 * @class Letter_Interactor
 *
 * @brief Handles the interaction logic for readable items
 *
 * This component is designed to be attached to gameobject. Providing an id you'll be
 * able to read items.
 *
 * @details
 *
 * Features:
 *
 * - Show text when clicking on an object containing this component
 *
 * Requirements:
 *
 * - Check if the item is clicked with "E"
 *
 * -
 *
 * How to Use:
 *
 * Give this component to any gameobject you'd like to cleanup or make disappear when
 * clicked.
 *
 * @author Timo Skrobanek
 *
 * @date 18.8.2024
 *
 * @version 1.0
 */
using UnityEngine;

public class Sleep_Interactor : MonoBehaviour, Interactable
{

```

```

public AnimationHandler handler;

/**
 * @brief Executes the interaction logic when the player interacts with this
object.
 *
 * @param obj The GameObject that is being interacted with.
 *
 * This method checks if the interaction key (E) is pressed, and if so, it
disables the
 * object, effectively removing it from the scene. A debug message is logged to
confirm
 * the action.
 */
public void Interact(GameObject obj)
{
    if (Input.GetKeyDown(KeyCode.E)) {
        StateManager.state = StateManager.State.SCENE3_OUTRO;

        handler.CheckState();
    }
}

/**
 * @brief Indicates whether the object is accessible for interaction.
 *
 * This method is intended to provide feedback or state information

```


* about whether the object can be interacted with. Currently, it is not implemented.

```
*/

public void IsAccessible(GameObject obj)

{

    // Implementation pending

}

public void IsNotAccessible(){

}

}
```

Datei: ..\..\Interaction\Movement_Interaction\TriggerGhostVoice.cs

```
/**

* @class TriggerGhostVoice

* @brief Plays a ghost voice sound effect when the player enters a specified trigger
area.

*

* This class activates an audio source when the player enters the trigger zone, playing
a

* sound effect once and preventing further activation.

*

* @author Timo Skrobanek

* @date 28.10.2024

* @version 1.0

*/
```

```
using UnityEngine;

public class TriggerGhostVoice : MonoBehaviour

{

    /** Audio source that plays the ghost voice sound effect. */

    public AudioSource src;

    /** Boolean to ensure the audio plays only once upon initial trigger. */

    private bool trigger = true;

    /**

    * @brief Plays the ghost voice audio when the player enters the trigger zone.

    *

    * This method checks if the colliding object has the "Player" tag and if the audio

    * hasn't already been triggered. If both conditions are true, it plays the audio

    * and disables further triggers.

    *

    * @param other The Collider of the object entering the trigger zone.

    */

    void OnTriggerEnter(Collider other)

    {

        if (other.CompareTag("Player") && trigger) {

            src.Play();

            Debug.Log("Player entered the trigger!");

            trigger = false;

        }

    }

}
```

```
}  
  
}
```

Datei: ../../Interaction\Movement_Interaction\TriggerMovingPerson.cs

```
/**  
  
 * @class TriggerMovingPerson  
  
 * @brief Handles triggering animations, sounds, and object states when the player  
enters a specific area.  
  
 *  
  
 * This class initiates a sequence of actions when the player enters the trigger zone:  
  
 * it updates the game state, plays an audio clip, enables an animation, and toggles  
visibility  
  
 * of objects (e.g., tables and doors).  
  
 *  
  
 * @details The script ensures that the sequence is only triggered once upon the first  
entry of  
  
 * the player into the trigger zone.  
  
 *  
  
 * @author Timo Skrobanek  
  
 * @date 10.10.2024  
  
 * @version 1.0  
  
 */  
  
using UnityEngine;  
  
public class TriggerMovingPerson : MonoBehaviour  
  
{
```

```

/** Audio source for playing a sound effect upon triggering. */

public AudioSource src;


/** GameObject representing the table in an upside-down state. */

public GameObject tableUpsideDown;


/** GameObject representing the table in its original position. */

public GameObject table;


/** Animator component that controls the movement of a character or object. */

public Animator anim;


/** GameObject representing open doors that will be toggled off. */

public GameObject doorsOpen;


/** GameObject representing closed doors that will be toggled on. */

public GameObject doorsClosed;


/** Boolean to ensure the trigger actions occur only once. */

private bool trigger = true;


/**
    * @brief Activates various effects and updates the game state when the player
enters the trigger zone.
    *
    * This method checks if the colliding object is tagged "Player" and if the trigger
hasn't already been activated.

```

```

* Upon triggering, it:

* - Sets the game state to SCENE4_GHOST_APPEAR

* - Plays the associated audio

* - Enables the assigned animation

* - Toggles table and door visibility

*

* @param other The Collider of the object entering the trigger zone.

*/

```

```

void OnTriggerEnter(Collider other)

{

    if (other.CompareTag("Player") && trigger) {

        StateManager.state = StateManager.State.SCENE4_GHOST_APPEAR;

        src.Play();

        anim.enabled = true;

        Debug.Log("Player entered the trigger!");

        trigger = false;

        tableUpsideDown.SetActive(true);

        table.SetActive(false);

        doorsClosed.SetActive(true);

        doorsOpen.SetActive(false);

    }

}

}

```

Datei: ..\..\Menu\EscapeMenu.cs

```
using UnityEngine;

using UnityEngine.SceneManagement;


public class EscapeMenu : MonoBehaviour

{

    public Canvas canvas;

    private bool toggle = true;

    // Start is called before the first frame update

    public void Start()

    {

        canvas.gameObject.SetActive(false);

    }


    // Update is called once per frame

    public void Update()

    {

        if (Input.GetKeyDown(KeyCode.T)) {

            canvas.gameObject.SetActive(toggle);


            if (toggle) {

                Cursor.lockState = CursorLockMode.None;

                Cursor.visible = true;

            }

            else{
```

```

        // Hide mouse and lock to screen center

        Cursor.lockState = CursorLockMode.Locked;

        Cursor.visible = false;

    }

    toggle = !toggle;

}

}

public void ResumeToGame() {

    canvas.gameObject.SetActive(false);

    // Hide mouse and lock to screen center

    Cursor.lockState = CursorLockMode.Locked;

    Cursor.visible = false;

}

public void ChangeToMenu() {

    StateManager.state = StateManager.State.SCENE1_INTRO_ANIMATION;

    SceneManager.LoadScene("MainMenu");

}

}

```

Datei: ../../Menu/MainMenu.cs

```
using System.Collections;
```

```
using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;


public class MainMenu : MonoBehaviour

{

    public void PlayGame() {

        //TODO correct implementation for change to other scenes beside scene 1

        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);

    }


    public void QuitGame() {

        Debug.Log("QUIT!");

        Application.Quit();

    }


    void Update() {

        // Überprüfe, ob die R-Taste gedrückt wurde

        if (Input.GetKeyDown(KeyCode.R))

        {

            // Lade die Hauptmenü-Szene

            SceneManager.LoadScene("MainMenu");

        }

    }

}
```


Date: ..\..\Menu\SettingsMenu.cs

```
using System.Collections.Generic;

using UnityEngine;

using UnityEngine.Audio;

using TMPPro;

public class SettingsMenu : MonoBehaviour {

    //note: slider goes from -80 to 0 because that's what our audio mixer does

    //for audio

    public AudioManager audioMixer;

    //for Resolution

    [SerializeField] private TMP_Dropdown resolutionDropdown;

    private Resolution[] resolutions;

    private List<Resolution> filteredResolutions;

    private float currentRefreshRate;

    private int currentResolutionIndex;

    public void SetVolume(float volume) {

        audioMixer.SetFloat("volume", volume);

    }

    // set game quality

    public void SetQuality(int qualityIndex) {
```

```

        QualitySettings.SetQualityLevel(qualityIndex);
    }

// Start is called before the first frame update

[System.Obsolete]
void Start()
{
    resolutions = Screen.resolutions;

    filteredResolutions = new List<Resolution>();

    resolutionDropdown.ClearOptions();

    currentRefreshRate = Screen.currentResolution.refreshRate;

    for(int i = 0; i< resolutions.Length; i++) {

        if(resolutions[i].refreshRate == currentRefreshRate) {

            filteredResolutions.Add(resolutions[i]);

        }

    }

    List<string> options = new List<string>();

    for(int i = 0; i < filteredResolutions.Count; i++) {

        string resolutionOption = filteredResolutions[i].width + " x " +

filteredResolutions[i].height + " " + filteredResolutions[i].refreshRate + " Hz";

        options.Add(resolutionOption);

        if(filteredResolutions[i].width == Screen.width &&

filteredResolutions[i].height == Screen.height) {

```

```

        currentResolutionIndex = 1;

    }

}

```

```

resolutionDropdown.AddOptions(options);

resolutionDropdown.value = currentResolutionIndex;

resolutionDropdown.RefreshShownValue();

```

```

//für bildschirm erkennung

//PopulateScreenDropdown();

```

```

}

```

```

public Resolution SetResolution(int resolutionIndex) {

    Resolution resolution = filteredResolutions[resolutionIndex];

    Screen.SetResolution(resolution.width, resolution.height, true);

    return resolution;

}

```

```

/*[DllImport("user32.dll")]

```

```

    static extern bool EnumDisplayDevices(string lpDevice, uint iDevNum, ref
DISPLAY_DEVICE lpDisplayDevice, uint dwFlags);

```

```

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]

```

```

public struct DISPLAY_DEVICE

```

```

{

    public int cb;

    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]

    public string DeviceName;

    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]

    public string DeviceString;

    public uint StateFlags;

    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]

    public string DeviceID;

    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]

    public string DeviceKey;

}


public TMP_Dropdown screenDropdown;


void PopulateScreenDropdown()

{

    screenDropdown.ClearOptions();

    var options = new System.Collections.Generic.List<string>();

    DISPLAY_DEVICE device = new DISPLAY_DEVICE();

    device.cb = Marshal.SizeOf(device);

    uint i = 0;

    int displayIndex = 0;

    while (EnumDisplayDevices(null, i, ref device, 0))

    {

```

```

        // Überprüfen, ob das Gerät ein echter Monitor ist

        if ((device.StateFlags & 0x00000001) != 0) // DISPLAY_DEVICE_ACTIVE flag
        {

            // Prüfen, ob das Display in Unity verfügbar ist, bevor wir es
hinzufügen

            if (displayIndex < Display.displays.Length)
            {

                string option = $"{device.DeviceString}
({Display.displays[displayIndex].systemWidth}x{Display.displays[displayIndex].systemHeight})";

                options.Add(option);

                displayIndex++;

            }

        }

        i++;

    }

    screenDropdown.AddOptions(options);

    screenDropdown.onValueChanged.AddListener(delegate {
SetScreen(screenDropdown.value); });

}

public void SetScreen(int screenIndex)
{

    if (screenIndex < Display.displays.Length)
    {

        if (!Display.displays[screenIndex].active)

```

```

        {

            Display.displays[screenIndex].Activate();

        }

        Debug.Log($"Bildschirm gewechselt zu:

{Display.displays[screenIndex].systemWidth}x{Display.displays[screenIndex].systemHeight}

");

    }

    else

    {

        Debug.LogError("Ungültiger Bildschirmindex: " + screenIndex);

    }

}*/

}

```

Datei: ..\..\Menu\Startbutton.cs

```

using UnityEngine.UI;

using UnityEngine;

using UnityEngine.SceneManagement;

public class Startbutton : MonoBehaviour

{

    public Button yourButton;

    public void startGame(){

        SceneManager.LoadScene("Scene1");
    }
}

```

```
}
```

```
}
```

Date: ..\..\Movement\PlayerMovement.cs

```
/**
```

```
 * @class PlayerMovement
```

```
 * @brief Handles the movement logic for the player character in Unity.
```

```
 *
```

```
 * This script allows the player to move in various directions using keyboard input,
```

```
 * and it manages both walking and running speeds. The script is designed to work
```

```
 * with Unity's Rigidbody component for physics-based movement.
```

```
 *
```

```
 * @details
```

```
 * Features:
```

```
 * - Supports basic movement (forward, backward, left, right) using WASD or arrow keys.
```

```
 * - Allows the player to sprint by holding down a designated key (e.g., Left Shift).
```

```
 * - Implements smooth movement by interpolating the player's velocity.
```

```
 * - Adjusts movement speed based on whether the player is walking or running.
```

```
 * - Includes optional jump functionality if the player is grounded.
```

```
 *
```

```
 * Requirements:
```

```
 * - This script should be attached to a GameObject with a Rigidbody component.
```

```
 * - The GameObject should also have a Collider component for proper collision  
detection.
```

```
 *
```

```
 * How to Use:
```

```
 * - Attach the PlayerMovement script to your player GameObject.
```

- * - Configure movement speed, running speed, and jump force in the Unity Inspector.
- * - Ensure the Rigidbody component is set to "Interpolate" for smooth movement.
- * - Set the correct input axes in the Unity Input Manager (default is WASD or arrow keys).

*

* @author Timo Skrobanek

* @date 18.8.2024

* @version 1.0

*/

using System;

using UnityEngine;

[RequireComponent(typeof(CharacterController))]

public class PlayerMovement : MonoBehaviour {

/// @brief Movement speed of the player.

[SerializeField] private float speed;

/// @brief Gravity force applied to the player.

[SerializeField] private float gravity;

/// @brief Multiplier for the player's speed when running.

[SerializeField] private float runMultiplier = 2; // Multiply movement speed by 2
when player runs

public CharacterController charController; ///< Reference to the CharacterController
component.


```

private float yVelocity; ///< Current vertical velocity due to gravity.

/// @brief Starting pitch (x-axis) rotation.

private float xRotation = 0f;

/// @brief Starting yaw (y-axis) rotation.

public float yRotation = -90f;

/// @brief Sensitivity of mouse movement for camera rotation.

[SerializeField] public float mouseSensitivity;

/// @brief Reference to the main camera transform for rotating the camera.

[SerializeField] public Transform mainCam;

/**
 * @brief Initializes the PlayerMovement script.
 *
 * This method is called on the first frame when the script is enabled. It sets up
the CharacterController component,
 * locks the cursor to the center of the screen, and plays the introductory
animation.
 */

public void Start() {

    charController = GetComponent<CharacterController>();

    // Hide mouse and lock to screen center

    Cursor.lockState = CursorLockMode.Locked;

```

```

        Cursor.visible = false;

    }

    /**
     * @brief Handles player input for movement.
     *
     * This method is called once per frame. It checks if the introductory animation has
    finished, then captures player input
     * for movement and calls the PlayerMove() method to move the player accordingly.
     */
    public void Update() {

        if(StateManager.state != StateManager.State.SCENE1_INTRO_ANIMATION    &&
    StateManager.state != StateManager.State.SCENE4_INTRO){

            float x = Input.GetAxisRaw("Horizontal");

            float z = Input.GetAxisRaw("Vertical");

            // Apply movement to player controller

            charController.Move(PlayerMove(x , z));

        }

    }

    /**
     * @brief Handles camera rotation based on mouse input.
     *
     * This method is called after all Update functions have been called. It captures
    mouse movement input and calls

```

```

    * rotationUpdate() to update the camera rotation, provided motion is not disabled.
    */

public void LateUpdate() {

    float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity * Time.deltaTime;

    float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity * Time.deltaTime;

        if(StateManager.state != StateManager.State.SCENE1_INTRO_ANIMATION &&
StateManager.state != StateManager.State.SCENE4_INTRO){

            rotationUpdate(mouseX, mouseY);

        }

    }

/**

    * @brief Updates the camera rotation based on mouse movement.

    *

    * @param mouseX Horizontal mouse movement.

    * @param mouseY Vertical mouse movement.

    *

    * This method updates the player's camera rotation by applying pitch and yaw
rotations. The camera's pitch rotation

    * (looking up and down) is clamped to prevent extreme angles.

    */

public void rotationUpdate(float mouseX, float mouseY){

    // Apply pitch rotation (looking up and down)

    xRotation -= mouseY;

    yRotation += mouseX;

    xRotation = Mathf.Clamp(xRotation, -80f, 70f); // Can't look too far up or down

```

```

        // Combine both rotations

        mainCam.localEulerAngles = new Vector3(xRotation, yRotation, 0);
    }

/**
 * @brief Moves the player based on input.
 *
 * @param x Horizontal movement input.
 * @param z Vertical movement input.
 *
 * This method calculates the player's movement vector based on the input axes and
the player's orientation relative to the camera.
 *
 * It also applies gravity to keep the player grounded and checks if the player is
running to adjust the movement speed.
 */
public Vector3 PlayerMove(float x, float z) {

    // Current position

    Vector3 move = (transform.right * x + transform.forward * z).normalized;

    // Store new position depending on speed and delta time

    move = move * speed * Time.deltaTime;

    // Check if player is running

    if (Input.GetKey(KeyCode.LeftShift))

        move *= runMultiplier;

```

```

        // Keep player grounded

        yVelocity += gravity * Time.deltaTime;

        move.y = yVelocity * Time.deltaTime;


        return move;
    }


//Getter and Setter

public void SetGravity(float value) => gravity = value;

public void SetRunMultiplier(float value) => runMultiplier = value;

public void SetMouseSensitivity(float value) => mouseSensitivity = value;


// Getter-Methoden

public float GetSpeed()

{

    return speed;

}


public float GetGravity()

{

    return gravity;

}


public float GetRunMultiplier()

{

    return runMultiplier;

}

```

```

public float GetMouseSensitivity()

{

    return mouseSensitivity;

}


public object GetVelocity()

{

    throw new NotImplementedException();

}

}

```

Date: ..\..\..\Movement\PlayerRotation.cs

```

using System;

using UnityEngine;

/**

 * @class PlayerRotation

 * @brief Handles the rotation logic for the player character in Unity.

 *

 * This script allows the player to rotate the camera inside a predefined angle.

 * In Scene 3 of the game, the player is only allowed to move his head. With a few

 * more requirements it's more efficient to have an disjoint script for head rotation.

 *

 * @details

 * Features:

 * - Player rotation in x and y axes inside predefined angles

```

*

* Requirements:

* - This script should be attached to a GameObject with a Rigidbody component.

* - The GameObject should also have a Collider component for proper collision detection.

*

* How to Use:

* - Attach the PlayerMovement script to your player GameObject.

* - Configure rotation speed and limits.

* - Set the correct input axes in the Unity Input Manager (default is WASD or arrow keys).

*

* @author Timo Skrobanek

* @date 28.10.2024

* @version 2.0

*/

```
public class PlayerRotation : MonoBehaviour {
```

```
    /// @brief Starting pitch (x-axis) rotation.
```

```
    private float xRotation = 0f;
```

```
    /// @brief Starting yaw (y-axis) rotation.
```

```
    private float yRotation = 0f;
```

```
    //updated:
```

```
    private float pitch = 0f;
```

```

/// @brief Sensitivity of mouse movement for camera rotation.

[SerializeField] public float mouseSensitivity;

/// @brief Reference to the main camera transform for rotating the camera.

[SerializeField] public Transform mainCam;

public void Start() {

    // Hide mouse and lock to screen center

    Cursor.lockState = CursorLockMode.Locked;

    Cursor.visible = false;

}

/**

* @brief Handles camera rotation based on mouse input.

*

* This method is called after all Update functions have been called. It captures
mouse movement input and calls

* rotationUpdate() to update the camera rotation, provided motion is not disabled.

*/

public void LateUpdate() {

    if (StateManager.state != StateManager.State.SCENE3_OUTRO)

    {

        float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity * Time.deltaTime;

        float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity * Time.deltaTime;

        rotationUpdate(mouseX, mouseY);

    }

```



```

}

/**
 * @brief Updates the camera rotation based on mouse movement.
 *
 * @param mouseX Horizontal mouse movement.
 * @param mouseY Vertical mouse movement.
 *
 * This method updates the player's camera rotation by applying pitch and yaw
rotations. The camera's pitch rotation
 * (looking up and down) is clamped to prevent extreme angles.
 */
public void rotationUpdate(float mouseX, float mouseY){

    // Apply pitch rotation (looking up and down)

    xRotation -= mouseY;

    yRotation += mouseX;

    xRotation = Mathf.Clamp(xRotation, -80f, 70f); // Can't look too far up or down

    // Combine both rotations

    mainCam.localEulerAngles = new Vector3(xRotation, yRotation, 0);
}
}

```

Datei: ..\..\SaveAndLoad\SaveAndLoadManager.cs

```

using System.Collections.Generic;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

```

```
using System.Security.Cryptography;

using System.Text;

using UnityEngine;

using UnityEngine.SceneManagement;


[System.Serializable]

public class SaveData

{

    public int lastSceneIndex;

    public Dictionary<string, object> globalVariables = new Dictionary<string,

object>();

}


public class SaveLoadManager : MonoBehaviour

{

    private static SaveLoadManager instance;

    private SaveData currentSave;

    private string saveFilePath;

    public static SaveLoadManager Instance => instance;


    void Awake()

    {

        if(instance == null)

        {

            instance = this;

            DontDestroyOnLoad(gameObject);

            saveFilePath = Application.persistentDataPath + "/save.dat";
```

```

        LoadGame ();

    }

    else

    {

        Destroy(gameObject);

    }

}

public void SaveGame()

{

    currentSave.lastSceneIndex = SceneManager.GetActiveScene().buildIndex;

    StateManager.Instance.UpdateGameState(currentSave);


    BinaryFormatter formatter = new BinaryFormatter();

    using (MemoryStream memoryStream = new MemoryStream())

    {

        // Serialisieren der aktuellen Spieldaten

        formatter.Serialize(memoryStream, currentSave);

        byte[] serializedData = memoryStream.ToArray();


        // Verschlüsseln der serialisierten Daten

        byte[] encryptedData = EncryptionHelper.Encrypt(serializedData);


        // Hash der verschlüsselten Daten erstellen, um Manipulation zu erkennen

        byte[] hash = HashHelper.ComputeSHA256Hash(encryptedData);


        // Kombiniere die Hash-Werte und die verschlüsselten Daten (erst der Hash,

```

dann die verschlüsselten Daten)

```
        byte[] combinedData = new byte[hash.Length + encryptedData.Length];

        System.Buffer.BlockCopy(hash, 0, combinedData, 0, hash.Length);

        System.Buffer.BlockCopy(encryptedData, 0, combinedData, hash.Length,
encryptedData.Length);

        File.WriteAllBytes(saveFilePath, combinedData);
    }
}

public void LoadGame()
{
    if (File.Exists(saveFilePath))
    {
        byte[] combinedData = File.ReadAllBytes(saveFilePath);

        // Extrahiere den Hash und die verschlüsselten Daten

        byte[] hash = new byte[32]; // SHA-256 Hash ist 32 Bytes lang

        byte[] encryptedData = new byte[combinedData.Length - 32];

        System.Buffer.BlockCopy(combinedData, 0, hash, 0, 32);

        System.Buffer.BlockCopy(combinedData, 32, encryptedData, 0,
encryptedData.Length);

        // Überprüfe den Hash, um Manipulationen zu erkennen

        byte[] calculatedHash = HashHelper.ComputeSHA256Hash(encryptedData);

        if (!HashHelper.CompareHashes(hash, calculatedHash))
        {
```

```

        Debug.LogError("Save file has been tampered with!");

        return;
    }

    // Entschlüsseln der verschlüsselten Spieldaten

    byte[] decryptedData = EncryptionHelper.Decrypt(encryptedData);

    BinaryFormatter formatter = new BinaryFormatter();

    using (MemoryStream memoryStream = new MemoryStream(decryptedData))
    {
        currentSave = (SaveData)formatter.Deserialize(memoryStream);

        StateManager.Instance.UpdateGameState(currentSave);

        SceneManager.LoadScene(currentSave.lastSceneIndex);
    }
}

else
{
    currentSave = new SaveData();

    StateManager.Instance.UpdateGameState(currentSave);
}

}

public void SetGlobalVariable(string key, object value)
{
    currentSave.globalVariables[key] = value;
}

```

```

public object GetGlobalVariable(string key)

{

                                return    currentSave.globalVariables.ContainsKey(key)    ?

currentSave.globalVariables[key] : null;

}

}

public static class EncryptionHelper

{

    private static readonly string encryptionKey = "DeinSuperSichererSchlüssel123!"; //
Geheimer Schlüssel, 32 Zeichen (für AES-256)

    public static byte[] Encrypt(byte[] data)

    {

        using (Aes aes = Aes.Create())

        {

            aes.Key = Encoding.UTF8.GetBytes(encryptionKey.PadRight(32).Substring(0,
32)); // 32 Byte Schlüssel für AES-256

            aes.IV = new byte[16]; // Initialisierungsvektor, 16 Bytes für AES-128 /
AES-256

            using (ICryptoTransform encryptor = aes.CreateEncryptor(aes.Key, aes.IV))

            {

                return encryptor.TransformFinalBlock(data, 0, data.Length);

            }

        }

    }

}

```

```

public static byte[] Decrypt(byte[] encryptedData)
{
    using (Aes aes = Aes.Create())
    {
        aes.Key = Encoding.UTF8.GetBytes(encryptionKey.PadRight(32).Substring(0,
32)); // 32 Byte Schlüssel

        aes.IV = new byte[16]; // Gleicher IV wie beim Verschlüsseln

        using (ICryptoTransform decryptor = aes.CreateDecryptor(aes.Key, aes.IV))
        {
            return decryptor.TransformFinalBlock(encryptedData, 0,
encryptedData.Length);
        }
    }
}

```

```

public static class HashHelper
{
    public static byte[] ComputeSHA256Hash(byte[] data)
    {
        using (SHA256 sha256 = SHA256.Create())
        {
            return sha256.ComputeHash(data);
        }
    }
}

```

```

public static bool CompareHashes(byte[] hash1, byte[] hash2)
{
    if (hash1.Length != hash2.Length)
        return false;

    for (int i = 0; i < hash1.Length; i++)
    {
        if (hash1[i] != hash2[i])
            return false;
    }

    return true;
}
}

```

Date: ..\..\StoryStateManager\StateManager.cs

```

using UnityEngine;

using UnityEngine.SceneManagement;

public class StateManager : MonoBehaviour {

    //Achievements and their states

    private static int cleanedItems = 0;

    private static bool outfit = false;

    private static bool introToggle = false, intro4Toggle = false;

    public static State state = State.SCENE1_INTRO_ANIMATION; //Has to be changed
    for debugging when starting from a different state!

```



```

public void Start() {

    Debug.Log("State of game loading: " + state);

    switch(SceneManager.GetActiveScene().buildIndex) {

        case 1:

            state = State.SCENE1_INTRO_ANIMATION;

            break;

        case 2:

            state = State.SCENE2;

            Debug.Log("Loading Scene 2");

            break;

        case 3:

            state = State.SCENE3;

            break;

        case 4:

            state = State.SCENE4_INTRO;

            break;

        case 5:

            state = State.SCENE5;

            break;

    }

}

```

```

public static void startScene1() {

    state = State.SCENE1_INTRO_ANIMATION;

    CleanedItems = 0;

    outfit = false;

```

```
        introToggle = false;
    }

    public static bool ItemsLeftToCleanup() {

        return CleanedItems < 3;

    }

    public static void cleanUp() {

        CleanedItems++;

    }

    public static void changeOutfit() {

        outfit = true;

        state = State.SCENE1_COMPLETED;

    }

    public static bool outfitChanged() {

        return outfit;

    }

    public static bool scene1Complete() {

        return !ItemsLeftToCleanup() && outfitChanged();

    }

    public static void stopIntroAnimation() {

        if(!introToggle){

            state = State.SCENE1;

            introToggle = true;

        }

    }

}
```

```
    }  
}
```

```
public static void StopIntroAnimationScene4() {  
  
    if(!intro4Toggle){  
  
        state = State.SCENE4;  
  
        intro4Toggle = true;  
  
    }  
}
```

```
public enum State {  
  
    SCENE1 = 0,  
  
    SCENE1_INTRO_ANIMATION = 1,  
  
    SCENE1_CLEANUP_DONE = 2,  
  
    SCENE1__NOT_DRESSED = 3,  
  
    SCENE1_COMPLETED = 4,  
  
  
    SCENE2 = 5,  
  
    SCENE2_COMPLETED = 7,  
  
    SCENE3 = 6,  
  
  
    SCENE3_OUTRO = 8,  
  
    SCENE3_OUTRO_DONE = 9,  
  
    SCENE4_INTRO = 10,  
  
    SCENE4 = 11,  
  
    SCENE4_GHOST_APPEAR = 12,
```

SCENE5,

ESCAPE_MENU = 13

}

// from here on for Save and Load

public static StateManager Instance {get; private set;}

public SaveData currentGameState;

void Awake()

{

if(Instance == null)

{

Instance = this;

DontDestroyOnLoad(gameObject);

}

else

{

Destroy(gameObject);

}

}

public void UpdateGameState(SaveData newGameState)

{

currentGameState = newGameState;

```

    }

    public static int CleanedItems { get; private set; } = cleanedItems;

    public static bool Outfit { get; private set; } = outfit;

    public static bool IntroToggle { get; private set; } = introToggle;

    public static bool Intro4Toggle { get; private set; } = intro4Toggle;

}

```

Datei: ..\..\StoryStateManager\UI_Updater.cs

```

using TMPro;

using UnityEngine;

public class UI_Updater : MonoBehaviour

{

    public TMP_Text task;


    public void Start() {

        Debug.Log("UI started");

    }

    public void Update() {

        UpdateUI_Task();

    }

}

```

```
private void UpdateUI_Task() {

    switch(StateManager.state) {

        case StateManager.State.SCENE1_INTRO_ANIMATION:

            task.SetText("");

            break;

        case StateManager.State.SCENE1:

            task.SetText("Ah shit. It's already late. I have to go to work ...*sigh*
but I have to cleanup this place first");

            break;

        case StateManager.State.SCENE1_CLEANUP_DONE:

            task.SetText("Since I live alone in this house, everything is messed up
but I'm ok with it... Now I have to get dressed before I can go to work");

            break;

        case StateManager.State.SCENE1_COMPLETED:

            task.SetText("Finally I can go to my office now");

            break;

        case StateManager.State.SCENE2:

            task.SetText("I'm way too late, my boss is gonna kill me.");

            break;

        case StateManager.State.SCENE4_INTRO:

            task.SetText("");

            break;

        case StateManager.State.SCENE4:

            task.SetText("Ok what the actual fuck was that? Maybe I should go to the
kitchen and drink something");

            break;

        case StateManager.State.SCENE4_GHOST_APPEAR:
```

```

        task.SetText("What was that again? Thats enough... Maybe sleeping is a
good idea");

        break;

        case StateManager.State.SCENE5:

            task.SetText("[Chef]: Why are you sleeping at work? I know you had a
hard time with your wife and the house but keep focused otherwise I cant keep you
employed here");

            break;

        }

    }

}

```

Date: ..\..\Test\UnitTests\InteractionTest.cs

```

using NUnit.Framework;

using UnityEngine;

public class InteractionTest
{
    private GameObject player;

    private InteractionComponent interactionComponent;

    [SetUp]

    public void Setup()
    {
        // Setup a new GameObject with the InteractionComponent

        player = new GameObject("Player");

        interactionComponent = player.AddComponent<InteractionComponent>();
    }
}

```

```

}

[Test]

public void Teardown()

{
    // Clean up after tests

    GameObject.Destroy(player);
}

[Test]

public void CheckInteractionFailed()

{
    // Arrange

    GameObject myObject = new GameObject("Interactor");

    Transform myTransform = myObject.transform;

    myTransform.position = new Vector3(1, 1, 1);

    myTransform.rotation = Quaternion.Euler(0f, 90f, 0f); // Rotated by 90° so no
interaction

    interactionComponent.SetInteractor(myTransform);

    // Act

    Ray directedRay = new Ray(interactionComponent.GetInteractor().position,
interactionComponent.GetInteractor().forward);

    bool result = interactionComponent.CheckInteraction(directedRay);

    // Assert

```



```

        Assert.IsFalse(result);
    }

[Test]
public void CheckInteractionSuccessful()
{
    // Arrange

    // Create the interactor GameObject
    GameObject interactorObject = new GameObject("Interactor");

    Transform interactorTransform = interactorObject.transform;

    interactorTransform.position = Vector3.zero; // Position at (0, 0, 0)
    interactorTransform.rotation = Quaternion.identity; // Default rotation

    // Create the target interactable object
    GameObject targetObject = GameObject.CreatePrimitive(PrimitiveType.Cube); //
Cube with a collider

    targetObject.transform.position = new Vector3(0, 0, 5); // Place directly in
front of the interactor

    targetObject.AddComponent<Computer_Interactor>(); // Add the Interactable
component

    // Ensure the interaction component is properly initialized
    interactionComponent.SetInteractor(interactorTransform);

    interactionComponent.interactRange = 10f; // Set a range that includes the
target

    // Act

```

```

        // Create a ray from the interactor, pointing forward

        Ray directedRay = new Ray(interactorObject.transform.position,
interactorObject.transform.forward);

        bool result = interactionComponent.CheckInteraction(directedRay);


        // Assert

        Assert.IsTrue(result);


        // Cleanup

        Object.DestroyImmediate(interactorObject);

        Object.DestroyImmediate(targetObject);
    }

}

```

Datei: ..\..\Test\UnitTests\PlayerMovementTest.cs

```

using System.Collections;

using System.Runtime.CompilerServices;

using NUnit.Framework;

using UnityEngine;

using UnityEngine.TestTools;

public class PlayerMovementTests
{
    private GameObject playerGameObject;

    private PlayerMovement playerMovement;

    private CharacterController characterController;

```

[SetUp]

```
public void Setup()

{

    // Create a GameObject to attach the PlayerMovement script

    playerGameObject = new GameObject("Player");

    characterController = playerGameObject.AddComponent<CharacterController>();

    playerMovement = playerGameObject.AddComponent<PlayerMovement>();


    // Set default values for serialized fields

    playerMovement.SetGravity(-9.81f);

    playerMovement.SetRunMultiplier(2f);

    playerMovement.SetMouseSensitivity(100f);


    // Create a mock camera Transform

    GameObject cameraObject = new GameObject("MainCamera");

    playerMovement.mainCam = cameraObject.transform;


    // Position character controller for the test

    characterController.center = new Vector3(0, 1, 0);

}
```

[TearDown]

```
public void Teardown()

{
```

```

        Object.Destroy(playerGameObject);
    }

[Test]

public void PlayerMove_NormalMovement()
{
    // Arrange

    float inputX = 1f; // Simulate "D" key for right movement

    float inputZ = 0f; // No forward movement

    // Act

    Vector3 calcMove = playerMovement.PlayerMove(inputX, inputZ);

    // Assert

    Vector3 expectedMovement = new Vector3(inputX, 0, 0).normalized *
playerMovement.GetSpeed() * Time.deltaTime;

    Assert.AreEqual(expectedMovement.x, calcMove.x, 0.01f, "Player should move
right.");
}

[Test]

public void PlayerMove_WithRunning_SpeedIncreases()
{
    // Arrange

    float inputX = 0f; // No horizontal movement

    float inputZ = 1f; // Simulate "W" key for forward movement

```

```

// Act

Vector3 calcMove = playerMovement.PlayerMove(inputX, inputZ);

// Assert

Vector3 expectedMovement = new Vector3(0, 0, 1).normalized *
playerMovement.GetSpeed() * playerMovement.GetRunMultiplier() * Time.deltaTime;

Assert.AreEqual(expectedMovement.z, calcMove.z, 0.01f, "Player should move
faster while running.");
}

```

[Test]

```
public void Gravity_AppliedCorrectly()
```

```
{
```

```
    // Arrange input
```

```
    float inputX = 5f;
```

```
    float inputZ = 0f;
```

```
    //temporally store previous z-component
```

```
    float z = playerMovement.transform.position.z;
```

```
    // Calculate new position
```

```
    Vector3 move = playerMovement.PlayerMove(inputX, inputZ);
```

```
    Assert.IsTrue(move.z == z);
```

```
}
```

[Test]

```
public void CameraRotation_LimitedCorrectly()
```

```

{

    // Arrange

    float mouseX = 10f;

    float mouseY = -20f;


    // Act

    playerMovement.rotationUpdate(mouseX, mouseY);


    // Assert

    Assert.IsTrue(playerMovement.mainCam.localEulerAngles.x >= -80f &&
playerMovement.mainCam.localEulerAngles.x <= 70f,

        "Camera pitch should be clamped between -80 and 70 degrees.");
}


[UnityTest]
public IEnumerator Cursor_LockedOnStart()
{

    // Arrange: Setze den Cursor-Zustand explizit

    Cursor.lockState = CursorLockMode.Locked;

    Cursor.visible = false;


    yield return null; // Ein Frame warten


    // Assert

    Assert.AreEqual(CursorLockMode.Locked, Cursor.lockState, "Cursor should be locked on
start.");

    Assert.IsFalse(Cursor.visible, "Cursor should be invisible on start.");
}

```

```
}
```

```
}
```

Datei: ..\..\Test\UnitTests\SaveAndLoadTest.cs

```
using System.IO;
```

```
using NUnit.Framework;
```

```
using UnityEngine;
```

```
using System.Security.Cryptography;
```

```
using System.Text;
```

```
public class FileEncryptionTests
```

```
{
```

```
    private string filePath;
```

```
    public static class EncryptionHelper
```

```
    {
```

```
        private static readonly string encryptionKey = "TestEncryptionKey1234"; // Muss 16,  
24 oder 32 Zeichen lang sein (für AES)
```

```
        public static byte[] Encrypt(byte[] data)
```

```
        {
```

```
            using (Aes aes = Aes.Create())
```

```
            {
```

```
                aes.Key = Encoding.UTF8.GetBytes(encryptionKey.PadRight(32).Substring(0,  
32)); // 32 Byte Schlüssel
```

```
                aes.IV = new byte[16]; // Initialisierungsvektor (Standard 16 Bytes)
```

```

        using (ICryptoTransform encryptor = aes.CreateEncryptor(aes.Key, aes.IV))
        {
            return encryptor.TransformFinalBlock(data, 0, data.Length);
        }
    }
}
}

```

[SetUp]

```
public void SetUp()
```

```

{
    // Pfad für die Testdatei

    filePath = Application.persistentDataPath + "/test_encrypted_file.txt";

    // Sicherstellen, dass keine alte Datei existiert

    if (File.Exists(filePath))
    {
        File.Delete(filePath);
    }
}

```

[TearDown]

```
public void TearDown()
```

```

{
    // Entferne die Testdatei nach jedem Test

    if (File.Exists(filePath))
    {

```



```

        File.Delete(filePath);
    }
}

[Test]

public void TestFileCreationAndEncryption()
{
    // Beispiel-Daten zum Speichern

    string originalData = "Das ist ein Test zur Verschlüsselung";

    byte[] dataToEncrypt = System.Text.Encoding.UTF8.GetBytes(originalData);

    // Verschlüsselte Daten erstellen

    byte[] encryptedData = EncryptionHelper.Encrypt(dataToEncrypt);

    // Datei speichern

    File.WriteAllBytes(filePath, encryptedData);

    // Überprüfen, ob die Datei erstellt wurde

    Assert.IsTrue(File.Exists(filePath), "Datei wurde nicht erstellt");

    // Datei-Inhalt auslesen und prüfen, ob sie verschlüsselt ist

    byte[] readData = File.ReadAllBytes(filePath);

    Assert.AreNotEqual(dataToEncrypt, readData, "Datei ist nicht verschlüsselt");

    Debug.Log($"Datei erfolgreich erstellt und verschlüsselt: {filePath}");

    Debug.Log($"Datei befindet sich im Ordner: {Application.persistentDataPath}");
}

```

```
[Test]

public void TestSimpleFileCreation()

{

    // Schreibe einfache Daten (ohne Verschlüsselung) in die Datei

    string testContent = "Testinhalt für einfache Datei";

    File.WriteAllText(filePath, testContent);


    // Überprüfen, ob die Datei erstellt wurde

    Assert.IsTrue(File.Exists(filePath), "Datei wurde nicht erstellt");

    Debug.Log($"Datei erstellt unter: {filePath}");


    // Überprüfen, ob die Datei Daten enthält

    string readContent = File.ReadAllText(filePath);

    Assert.AreEqual(testContent, readContent, "Dateiinhalt stimmt nicht überein");

}
```

```
[Test]

public void TestMissingFile_CreatesNewFile()

{

    // Lösche die Datei, falls vorhanden

    if (File.Exists(filePath))

        File.Delete(filePath);


    // Versuche, eine neue Datei zu erstellen

    string newContent = "Das ist eine neue Datei";

    File.WriteAllText(filePath, newContent);

}
```

```
// Überprüfen, ob die Datei erstellt wurde

Assert.IsTrue(File.Exists(filePath), "Datei wurde nicht erstellt");

string readContent = File.ReadAllText(filePath);

Assert.AreEqual(newContent, readContent, "Dateiinhalte stimmen nicht überein");

}
```

```
[Test]
```

```
public void TestEmptyFile_HandlesGracefully()
```

```
{
```

```
// Erstelle eine leere Datei
```

```
File.WriteAllBytes(filePath, new byte[0]);
```

```
// Datei-Inhalt prüfen
```

```
byte[] readData = File.ReadAllBytes(filePath);
```

```
Assert.IsNotNull(readData, "Datei-Inhalt sollte nicht null sein");
```

```
Assert.AreEqual(0, readData.Length, "Datei sollte leer sein");
```

```
}
```

```
[Test]
```

```
public void TestCorruptedFile_HandlesGracefully()
```

```
{
```

```
// Erstelle eine beschädigte Datei
```

```
File.WriteAllBytes(filePath, new byte[] { 0xFF, 0xAA, 0xBB });
```

```
// Datei lesen und überprüfen
```

```
byte[] readData = File.ReadAllBytes(filePath);
```

```

Assert.IsNotNull(readData, "Datei-Inhalt sollte nicht null sein");

Assert.AreEqual(3, readData.Length, "Datei-Inhalt hat unerwartete Größe");

}

[Test]

public void TestLargeFile_SaveAndLoad()

{

    string largeData = new string('A', 1000000); // 1 Million Zeichen

    byte[] dataToEncrypt = Encoding.UTF8.GetBytes(largeData);

    // Verschlüsselte Daten erstellen und speichern

    byte[] encryptedData = EncryptionHelper.Encrypt(dataToEncrypt);

    File.WriteAllBytes(filePath, encryptedData);

    // Datei lesen und entschlüsseln

    byte[] readData = File.ReadAllBytes(filePath);

    Assert.AreEqual(encryptedData.Length, readData.Length, "Dateigröße stimmt nicht
    überein");

}

}

```

Datei: ..\..\Test\UnitTests\SettingsMenuTest.cs

```

using NUnit.Framework;

using UnityEngine;

using UnityEngine.SceneManagement;

```

```
using UnityEngine.TestTools;

public class SettingsMenuTest
{
    private GameObject _mainMenuObject;

    private MainMenu _mainMenuScript;

    //für PlayGame_CallsLoadScene()

    public class MockSceneManager
    {
        public int ActiveSceneIndex { get; private set; } = 0;

        public void LoadScene(int index)
        {
            ActiveSceneIndex = index;
        }

    }

    [SetUp]
    public void Setup()
    {
        // Erstelle ein leeres GameObject und füge das MainMenu-Skript hinzu

        _mainMenuObject = new GameObject();

        _mainMenuScript = _mainMenuObject.AddComponent<MainMenu>();
    }
}
```

[TearDown]

```
public void Teardown()
```

```
{
```

```
    // Lösche das GameObject nach jedem Test, um die Testumgebung sauber zu halten
```

```
    Object.DestroyImmediate(_mainMenuObject);
```

```
}
```

[Test]

```
public void SceneExistsInBuildSettings()
```

```
{
```

```
    // Arrange
```

```
    string sceneNameToTest = "MainMenu"; // Ersetze mit dem Namen deiner Szene
```

```
    bool sceneExists = false;
```

```
    // Prüfe, ob die Szene in den Build-Einstellungen enthalten ist
```

```
    for (int i = 0; i < SceneManager.sceneCountInBuildSettings; i++)
```

```
{
```

```
        string scenePath = SceneUtility.GetScenePathByBuildIndex(i);
```

```
        if (scenePath.Contains(sceneNameToTest))
```

```
{
```

```
            sceneExists = true;
```

```
            break;
```

```
}
```

```
}
```

```
    // Assert: Szene existiert in den Build-Einstellungen
```

```
    Assert.IsTrue(sceneExists, $"Die Szene '{sceneNameToTest}' ist nicht in den
```

```
Build-Einstellungen enthalten.");
```

```
}
```

```
[Test]
```

```
public void QuitGame_LogsQuitMessage()
```

```
{
```

```
    // Arrange
```

```
    LogAssert.Expect(LogType.Log, "QUIT!");
```

```
    // Act
```

```
    _mainMenuScript.QuitGame();
```

```
    // Assert
```

```
    // LogAssert prüft, ob der erwartete Log-Eintrag gemacht wurde
```

```
}
```

```
// Beispiel-Test
```

```
[Test]
```

```
public void PlayGame_CallsLoadScene()
```

```
{
```

```
    var mockSceneManager = new MockSceneManager();
```

```
    mockSceneManager.LoadScene(1);
```

```
    Assert.AreEqual(1, mockSceneManager.ActiveSceneIndex, "Die Szene wurde nicht korrekt  
geladen.");
```

```
}
```

```
}
```

Datei: ..\..\Test\UnitTests\StateManagerTest.cs

```
using NUnit.Framework;
```

```
using UnityEngine;
```

```
public class StateManagerTest : MonoBehaviour
```

```
{
```

```
    private StateManager stateManager;
```

```
    [SetUp]
```

```
    public void Setup()
```

```
    {
```

```
        // Reset static variables before each test
```

```
        stateManager = new GameObject("StateManager").AddComponent<StateManager>();
```

```
    }
```

```
    [TearDown]
```

```
    public void TearDown()
```

```
    {
```

```
        // Clean up after tests
```

```
        GameObject.Destroy(stateManager.gameObject);
```

```
    }
```

```
    [Test]
```

```
    public void TestStartScene1()
```

```
    {
```



```
// Arrange

StateManager.startScene1();

// Assert initial values

Assert.AreEqual(StateManager.state, StateManager.State.SCENE1_INTRO_ANIMATION);

Assert.AreEqual(StateManager.CleanedItems, 0);

Assert.IsFalse(StateManager.Outfit);

Assert.IsFalse(StateManager.IntroToggle);

}
```

```
[Test]

public void TestChangeOutfit()

{

    // Act

    StateManager.changeOutfit();

    // Assert

    Assert.IsTrue(StateManager.Outfit, "Outfit should be true after calling changeOutfit.");

    Assert.AreEqual(StateManager.State.SCENE1_COMPLETED, StateManager.state, "State should change to SCENE1_COMPLETED after calling changeOutfit.");

}
```

```
[Test]

public void TestScene1Complete()

{
```

```

        // Act

        bool result = StateManager.scene1Complete();

        // Assert

        Assert.IsTrue(result && StateManager.CleanedItems == 3 && StateManager.Outfit);

// scene1Complete should return true when CleanedItems == 3 and outfit == true

    }

[Test]

public void TestStopIntroAnimation()

{

    // Act

    StateManager.stopIntroAnimation();

    // Assert

    Assert.AreEqual(StateManager.State.SCENE1, StateManager.state, "State should change
to SCENE1 after calling stopIntroAnimation.");

    Assert.IsTrue(StateManager.IntroToggle, "IntroToggle should be true after calling
stopIntroAnimation.");

}

}

```

Datei: ..\..\Test\UnitTests\EditModeTest\Test.cs

```

using System.Collections;

using System.Collections.Generic;

```

```
using NUnit.Framework;

using UnityEngine;

using UnityEngine.TestTools;

public class Test

{

    // A Test behaves as an ordinary method

    [Test]

    public void TestSimplePasses()

    {

        // Use the Assert class to test conditions

    }


    // A UnityTest behaves like a coroutine in Play Mode. In Edit Mode you can use
    // `yield return null;` to skip a frame.

    [UnityTest]

    public IEnumerator TestWithEnumeratorPasses()

    {

        // Use the Assert class to test conditions.

        // Use yield to skip a frame.

        yield return null;

    }

}
```