Entwicklerfreundliche Open-Source-Lösung für das Speichern und Laden in der Spieleentwicklung

Abschlussarbeit

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

an der

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Internationale Medieninformatik*

1. Gutachter_in: Prof. Dr. Thomas Jung
2. Gutachter_in: Christoph Holtmann

Eingereicht von Robin Simeon Jaspers

25. September 2024

Zusammenfassung

Das Speichern und Laden von Spieldaten stellt eine zentrale Herausforderung in der Entwicklung von Videospielen dar. Die verbreitete Game-Engine Unity bietet jedoch keine native Funktionalität für diese Aufgabe, was bei der Entwicklung komplexer Spiele eine Hürde darstellt. Ziel dieser Arbeit ist die Entwicklung eines entwicklerfreundlichen und flexiblen Speichersystems, das sowohl für einfache als auch für komplexe Spiele geeignet ist. Die zentrale Forschungsfrage lautet: Wie muss ein Speichersystem in der Unity-Engine konzipiert werden, um eine entwicklerfreundliche und flexible Lösung zu bieten, die für einfache und komplexe Spiele gleichermaßen geeignet ist?

Im theoretischen Teil der Arbeit werden grundlegende Konzepte und Systeme dargestellt. Auf der Grundlage von Experteninterviews werden spezifische Anforderungen an ein Speichersystem formuliert. Diese Anforderungen bilden die Grundlage für eine Analyse von Best und Worst Practices bestehender Frameworks. Darauf aufbauend wird ein Speichersystem implementiert, das die ermittelten Anforderungen erfüllt. Dieses System wird anschließend mittels User-Tests und anhand der definierten Anforderungen evaluiert.

Die Ergebnisse der Arbeit zeigen, dass durch die Kombination von attribut- und komponentenbasiertem Speichern sowie durch die Unterstützung des Speicherns von Referenzen ein entwicklerfreundliches und flexibles Speichersystem realisierbar ist, das sowohl für einfache als auch für komplexe Spiele geeignet ist. Die vorgeschlagene Lösung adressiert wesentliche Herausforderungen bei der Speicherung von Spieldaten und bietet wertvolle Empfehlungen zur Implementierung effizienterer und robusterer Speichersysteme in der Spieleentwicklung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Videospiele haben sich in den letzten Jahrzehnten sowohl technologisch als auch in Bezug auf die Komplexität und den Umfang der Spielwelten erheblich weiterentwickelt. In den frühen Phasen der Videospielgeschichte bestand für Spieler in der Regel keine Möglichkeit, den Spielfortschritt zu speichern. Die Spiele wurden so konzipiert, dass sie in einem einzigen Durchlauf abgeschlossen werden mussten, wobei ein Scheitern der Spieler zum vollständigen Verlust des Fortschritts führte. Lediglich die Speicherung von Highscores war teilweise implementiert. Mit dem technologischen Fortschritt und der Zunahme der Rechenleistung wurden auch die Spielwelten immer komplexer. Open-World-Spiele und Rollenspiele, in denen der Spieler auf eine Vielzahl von Interaktionen und dynamischen Systemen stößt, erforderten neue Mechanismen, um den Spielstand dauerhaft zu speichern. Deswegen ist die Möglichkeit, den Spielstand zu speichern und zu laden, zu einem unverzichtbaren Bestandteil moderner Spiele geworden. In Open-World- und Rollenspielen ermöglicht das Speichern nicht nur die Dokumentation des Spielfortschritts, sondern erlaubt es auch, Fehler rückgängig zu machen und alternative Handlungsstränge zu erkunden. Diese Flexibilität ist besonders wichtig in Spielen, die eine Vielzahl von Handlungsoptionen bieten und in denen die Spielwelt stark von den Entscheidungen des Spielers beeinflusst wird.

1.1 Motivation

Die zunehmende Komplexität moderner Videospiele erfordert flexible, skalierbare und leistungsfähige Speichersysteme. Diese müssen sowohl für einfache als auch komplexe Spiele anwendbar sein und gleichzeitig eine zuverlässige, performante sowie leicht integrierbare und modifizierbare Architektur bieten. Entwickler stehen vor der Wahl, solche Systeme selbst zu entwickeln oder auf bestehende Frameworks zurückzugreifen. Dabei kann ein Framework den Entwicklungsprozess erheblich beschleunigen, indem es den Aufwand für die Implementierung und das Testen von Speicherfunktionen reduziert, was zu einer effizienteren Ressourcennutzung führt.

Besonders attraktiv sind dabei Open-Source-Frameworks, die nicht nur flexibel und anpassbar, sondern auch entwicklerfreundlich gestaltet sind. Eine intuitive und gut dokumentierte API ermöglicht es Entwicklern, das System schnell zu verstehen und effektiv zu nutzen. Zudem geben sie Entwicklern die Freiheit, das System an ihre spezifischen Anforderungen anzupassen und weiterzuentwickeln.

1.2 Problemstellung

Unity ist eine der weltweit am häufigsten genutzten Spieleentwicklungsplattformen und bietet eine umfassende Umgebung für die Erstellung von 2D- und 3D-Spielen [9]. Aufgrund ihrer Zugänglichkeit und Flexibilität wird Unity von einer breiten Nutzergemeinde genutzt, die von Indie-Entwicklern bis hin zu großen Studios reicht. Allerdings fehlen in Unity standardmäßig integrierte Funktionen zum Speichern und Laden von Spieldaten. Diese Limitierung stellt besonders bei der Entwicklung komplexer Spiele eine zentrale Herausforderung dar. Die Komplexität ergibt sich zum einen durch die hohe Anzahl an Objekten, die gespeichert werden müssen, und zum anderen durch die komplexen Datenstrukturen, die sich durch verschachtelte, zyklische Objekthierarchien auszeichnen, welche zudem zahlreiche Referenzen beinhalten.

Entwickler stehen daher vor der Entscheidung, entweder eigene Systeme zur Datenpersistenz zu entwickeln oder auf externe Lösungen zurückzugreifen. Dabei ergibt sich häufig die Wahl zwischen kostenintensiven, kommerziellen Lösungen, die erweiterte Funktionalitäten bieten, und kostenlosen Alternativen, die jedoch in ihren Möglichkeiten oft eingeschränkt sind. Dies zeigt sich beispielsweise am Fehlen fortschrittlicher Funktionen, wie der Unterstützung komplexer Objektstrukturen. Während kommerzielle Softwarelösungen umfangreiche Funktionen bereitstellen, sind diese meist mit hohen Kosten verbunden und dadurch für kleinere Entwicklerteams schwer zugänglich. Ein weiteres Defizit besteht in der fehlenden Unterstützung für die Versionierung von Spielständen, obwohl diese in der Entwicklung von Spielen eine zentrale Bedeutung einnimmt.

Ein neues Framework könnte genau an dieser Stelle ansetzen und dazu beitragen, die Entwicklungszeiten zu verkürzen sowie den Arbeitsaufwand für Entwickler deutlich zu reduzieren.

1.3 Zielsetzung und Forschungsfrage

Das Ziel dieser Arbeit besteht in der Entwicklung eines entwicklerfreundlichen und flexiblen Speichersystemes für die Unity-Engine. Entwicklerfreundlichkeit bezeichnet dabei die Eigenschaft, dass es Entwicklern eine einfache, effiziente und intuitive Nutzung ermöglicht. Ziel der Entwicklerfreundlichkeit ist es, die Entwicklungszeit zu verkürzen, die Produktivität zu steigern und die Lernkurve möglichst gering zu halten, um es Entwicklern zu erleichtern, das System schnell zu verstehen und zu nutzen. Neben der Entwicklerfreundlichkeit wird dabei ein besonderer Wert auf Erweiterbarkeit sowie die notwendige Flexibilität gelegt, um das System sowohl für einfache als auch für komplexe Spiele nutzbar zu machen. Um den Anforderungen komplexer Spiele gerecht zu werden, ist es unerlässlich, dass sowohl komplexe Objektstrukturen als auch die Versionierung von Spielständen unterstützt werden. Das System wird als Open-Source-Lösung unter der MIT-Lizenz bereitgestellt, um eine breite Akzeptanz und Weiterentwicklung innerhalb der Entwicklergemeinschaft zu fördern.

Die Entscheidung, Unity als Entwicklungsplattform zu wählen, basiert auf den spezifischen Anforderungen dieser Arbeit. Zwar könnte ein C#-basiertes System theoretisch plattformübergreifend, beispielsweise für Engines wie Godot, nutzbar gemacht werden, jedoch unterscheiden sich die Spiele-Engines erheblich in ihrer Architektur und Objektverwaltung. Ein allgemein gehaltenes Speichersystem würde daher nicht die erforderliche Funktionalität und Tiefe bieten, die für die Zielsetzung dieser Arbeit notwendig ist.

Vor diesem Hintergrund ergibt sich die zentrale Forschungsfrage:

Wie muss ein Speichersystem in der Entwicklungsumgebung Unity konzipiert werden, um eine entwicklerfreundliche Erfahrung zu gewährleisten, welche sowohl für einfache als auch komplexe Spiele unterstützt wird?

1.4 Hypothesen

Basierend auf der Zielsetzung und der Forschungsfrage zum Speichern und Laden in Unity werden drei Hypothesen aufgestellt, die unterschiedliche Ansätze untersuchen. Im Rahmen der ersten Hypothese ist definiert, dass die Nutzung von Attributen zur Markierung von Objekten für das Speichern und Laden die Entwicklerfreundlichkeit erhöht. Durch die Einführung von Attributen, die auf Reflection basieren, kann ein solches System stark vereinfacht und das Prototyping beschleunigt werden. Reflection bietet die Möglichkeit, "[...] Informationen zu geladenen Assemblys und den hierin definierten Typen wie Klassen, Schnittstellen und Werttypen [...] abzurufen. Sie können auch mithilfe von Reflektion Typeninstanzen zur Laufzeit erstellen, diese aufrufen und darauf zugreifen."[8]. Diese Methode ist besonders effizient bei einfachen Spielen, jedoch könnte die Performance bei komplexeren Szenarien aufgrund der höheren Rechenzeit von Reflection beeinträchtigt werden. Im Nachfolgenden wird dieses Konzept als *Attribute-Saving* bezeichnet.

H1: Die Nutzung von Attributen zur Markierung von Objekten für das Speichern und Laden in Unity erhöht die Entwicklerfreundlichkeit, ist jedoch in komplexeren Szenarien leistungstechnisch weniger effizient.

Die zweite Hypothese konzentriert sich auf ein komponentenbasiertes System, das auf eine höhere Flexibilität und Modularität ausgelegt ist. Durch die Wiederverwendbarkeit von Code könnten Speicher- und Ladeprozesse verallgemeinert und somit auf verschiedene Objekte in verschiedenen Kontext übertragen werden. Da dieses System ohne Reflection auskommt, wie es beim Attribute-Saving genutzt wird, ist eine bessere Performance zu erwarten. Allerdings wird ein höherer Implementationsbedarf im vergleich zum Attribute-Saving erwartet. Zudem könnte diese Vorgehensweise zusätzliche Anpassungsmöglichkeiten für Entwicklern ermöglichen. Dieses Konzept wird im Nachfolgenden als *Component-Saving* bezeichnet.

H2: Das Component-Saving erhöht die Flexibilität und Performance im Vergleich zum Attribute-Saving, erfordert jedoch mehr Implementationsaufwand und bietet zusätzliche Anpassungsmöglichkeiten für Entwickler.

Die letzte Hypothese fokussiert sich auf die komplexen Objekthierarchien: die automatische Wiederherstellung von Referenzen und die Instanziierung von Objekten. Eine Automatisierung dieses Prozesses könnte Entwicklern Zeit sparen und die Fehleranfälligkeit verringern, da die manuelle Verwaltung von Referenzen oft zeitaufwendig und fehlerbehaftet ist.

H3: Die Automatisierung der Wiederherstellung von Referenzen und der Instanziierung komplexer Objekthierarchien reduziert die Fehleranfälligkeit und spart Entwicklern Zeit im Vergleich zur manuellen Verwaltung.

1.5 Aufbau der Arbeit

Zur Überprüfung der Hypothese und Beantwortung der Forschungsfrage ist der Aufbau dieser Arbeit wie folgt gegliedert: Im theoretischen Teil werden zunächst die Spieleentwicklungsumgebung Unity sowie relevante Technologien und sicherheitsrelevante Aspekte umfassend erläutert. Im Anschluss daran werden die durchgeführten Experteninterviews dargestellt und die daraus abgeleiteten Anforderungen formuliert. Auf Basis dieser Anforderungen erfolgt eine Analyse von Good Practices und Bad Practices bestehender Frameworks. Aufbauend auf diesen Erkenntnissen wird ein Speichersystem implementiert und die entwickelte Lösung präsentiert. Abschließend wird das entstandene Framework gezielt durch Tests evaluiert und anhand der zuvor definierten Anforderungen bewertet.

In dieser Arbeit werden aus Gründen der Lesbarkeit bei Personenbezeichnungen das generische Maskulinum verwendet. Es werden dabei ausdrücklich alle Geschlechter mit inkludiert.

Kapitel 2

Theoretischer Hintergrund

2.1 Entwicklungsumgebung

2.1.1 Unity

Die Unity Engine wurde im Juni 2005 von Unity Technologies veröffentlicht. Eine Game-Engine bietet Entwicklern eine Vielzahl von Werkzeugen und Funktionen, die es ermöglichen, unterschiedliche Aspekte eines Vieodspiels zu erstellen. Zu diesen Funktionen zählen unter anderem das darstellen von Grafik, Physiksimulationen, Audio, Animationen sowie Skriptmechanismen zur Realisierung der Spiellogik. Ursprünglich für Mac OS X entwickelt, unterstützt die Engine mittlerweile eine breite Palette von Plattformen. Dank ihrer freien Zugänglichkeit ist die Engine nicht nur für große Unternehmen, sondern auch für kleine Firmen, Indie-Studios, private Entwickler und Hobbyisten zugänglich. Die stetig wachsende Community sorgt für eine breite Verfügbarkeit von Lösungen für bekannte Herausforderungen. Darüber hinaus wird Unity zunehmend in Bereichen wie Simulationen und Projekten mit Visualisierungsanforderungen eingesetzt.

2.1.2 Editor als Entwicklungsumgebung

Im Unity Editor steht den Entwicklern eine vollständig integrierte Entwicklungsumgebung zur Verfügung, die alle notwendigen Werkzeuge zur Spieleentwicklung bietet. Diese Werkzeuge benötigen jedoch Inhalte, die als sogenannte Assets bezeichnet werden. Gemäß der offiziellen Definition auf der Unity-Website wird ein Asset als ein Objekt, das in einem Spiel oder Projekt verwendet werden kann, beschrieben. Ein Asset kann aus einer Datei stammen, die außerhalb von Unity erstellt wurde, wie beispielsweise ein 3D-Modell, eine Audiodatei, ein Bild oder andere von Unity unterstützte Dateiformate. Zudem können Assets auch direkt innerhalb von Unity erstellt werden, darunter etwa ein Animator-Controller, ein Audio-Mixer oder eine Render-Textur (vgl. [30]). Innerhalb des Asset Stores von Unity können erstellte Assets distributiert werden. Auf diese Assets kann mithilfe der Projekt-View zugegriffen werden.

Zusätzlich stellt Unity eine Szenenansicht bereit, die an etablierte 3D-Softwarelösungen angelehnt ist. Diese Ansicht ermöglicht eine interaktive Visualisierung der von Nutzern erstellten Spielwelt (vgl. [31]). Der dreidimensionale Raum in Unity folgt einem euklidischen Koordinatensystem mit drei Achsen: Während die x- und z-Achse die horizontale Ebene darstellen, gibt die y-Achse die vertikale Position an. Die Spielansicht (Game View) dient dazu, das Spiel oder die Software in ihrer laufenden Form zu testen. Weitere zentrale Fenster des Unity Editors umfassen die Konsole, den

Inspektor und die Hierarchieansicht. Die Konsole zeigt Programmlogs an, der Inspektor ermöglicht die Bearbeitung der Eigenschaften von ausgewählten Objekten, und die Hierarchieansicht stellt die Struktur der *GameObjects* in der Szene dar, was die Navigation und Organisation dieser Elemente erleichtert. Die UI des Editors ist in Abbildung **??** dargestellt.

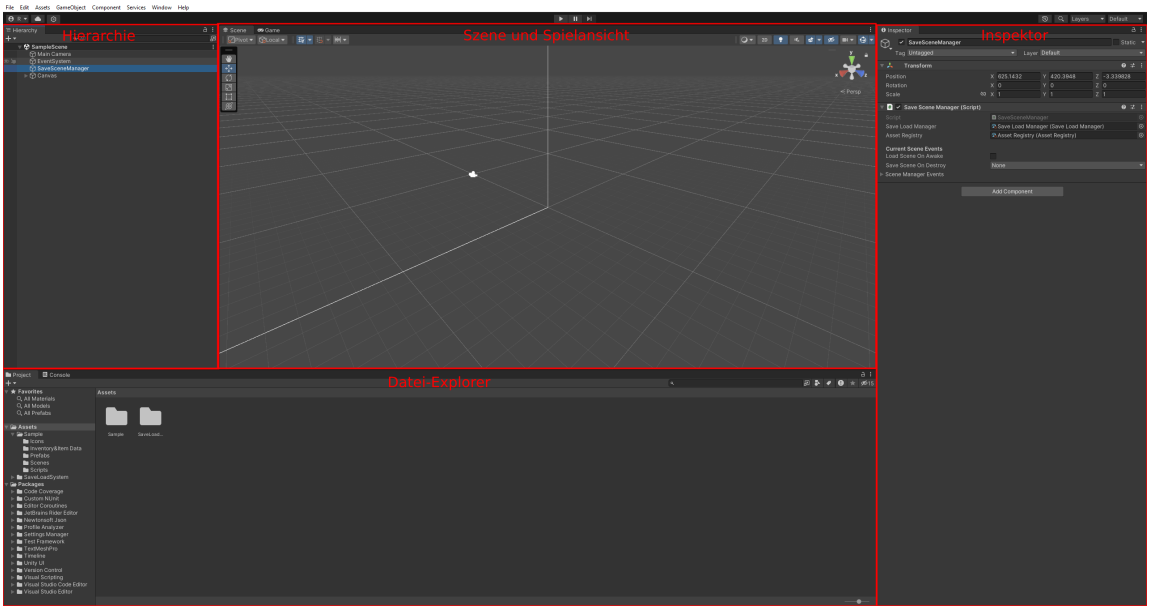


Abbildung 2.1: Unity Editor-UI

2.1.3 Konzepte der Engine

Das grundlegende Prinzip der Unity Engine besteht darin, GameObjects in der Szenenansicht zu platzieren und ihnen über sogenannte *Components* spezifische Funktionen zuzuweisen Robert Nystrom beschreibt in seinem Buch Game Programming Patterns die Rolle von Components folgendermaßen: „Allow a single entity to span multiple domains without coupling the domains to each other.“ [17]. Dies verdeutlicht, dass eine Component eine eigenständige Funktionseinheit ist, die einem GameObject hinzugefügt wird. Typische Components umfassen beispielsweise die Darstellung von Objekten oder das Einbinden von entwicklerdefinierter Skripte. Jedes Objekt in Unity ist ein GameObject, dem Components zugeordnet werden, um ihm bestimmte Funktionen zu verleihen. Ohne diese Components bleibt ein GameObject funktional eingeschränkt und dient in erster Linie als Container für die verschiedenen Funktionen.

Um GameObjects flexibel in verschiedenen Szenen verwenden zu können, bietet Unity das Konzept der *Prefabs*. Ein Prefab ist ein GameObject, das als Asset im Projektordner gespeichert wird. Dadurch kann es als Vorlage beliebig oft in unterschiedlichen Szenen instanziiert werden. Änderungen, die an einem Prefab vorgenommen werden, werden automatisch auf alle Instanzen dieses Prefabs in den entsprechenden Szenen übertragen. Dies ermöglicht eine effiziente Wiederverwendung und Anpassung von GameObjects, ohne dass jedes Objekt einzeln bearbeitet werden muss.

Ein weiteres wichtiges Konzept im Zusammenhang mit GameObjects ist das *Parenting* oder die *Parent-Child-Hierarchie*. Dabei wird ein GameObject als Parent eines anderen GameObjects definiert, wodurch das *Child* GameObject sämtliche Bewegungen, Rotationen und Skalierungen des Parent-Objekts automatisch übernimmt (vgl. [31]). Ein anschauliches Beispiel hierfür ist ein Auto: Wenn eine Person in ein Auto einsteigt und das Auto fährt, bewegt sich die Person mit dem Auto. In diesem Fall wird die Person zum Child des Autos und übernimmt dessen Bewegungen und Transformationen entsprechend.

2.1.4 Scripting

Unity-Anwendungen benötigen *Skripte*, um auf Spielereingaben zu reagieren und den Ablauf von Ereignissen im Spiel zur richtigen Zeit zu steuern. Die Programmiersprache C# ist der Standard für das Skripting in Unity. Skripte werden in das interne Unity-System eingebunden, indem sie von der Klasse *MonoBehaviour* abgeleitet werden (vgl. [31]). Diese von MonoBehaviour abgeleiteten Skripte dienen als Grundlage für die Ausführung spezifischer Aufgaben. Da sie in den Lebenszyklus von Unity integriert sind, beinhalten sie Methoden, die zu bestimmten Zeitpunkten aufgerufen werden, wie beispielsweise 'Awake()', 'Start()' und 'Update()'. Darüber hinaus können Skripte auf Assets zugreifen, die im Unity-Inspektor zugewiesen werden können. Zusätzlich lassen sich Parameter wie Vektoren, Floats oder Booleans definieren, die im Inspektor verändert werden können.

Neben MonoBehaviour gibt es in Unity eine weitere wichtige Klasse namens *ScriptableObject*. Laut der Unity-Dokumentation handelt es sich bei einem ScriptableObject um einen Datencontainer, der große Datenmengen speichern kann (vgl. [31]). Um ein Skript zu einem ScriptableObject zu machen, muss es von der Klasse ScriptableObject erben. Ein ScriptableObject wird wie ein Asset behandelt, was bedeutet, dass es nach der Implementierung als Datei im Projektordner gespeichert wird. Ähnlich wie bei MonoBehaviour können auch hier Referenzen zu Assets und Parameter im Unity Editor festgelegt werden. Einer der Hauptanwendungsfälle für ScriptableObjects ist es, den Speicherbedarf eines Projekts zu reduzieren, indem Kopien von Werten vermieden werden (vgl. [31]). Dies wird erreicht, indem ein ScriptableObject als gemeinsame Referenz für mehrere Skripte verwendet wird, sodass dieselben Daten an verschiedenen Stellen im Projekt genutzt werden können, wie in Darstellung 2.3 dargestellt. Der gespeicherte Datentyp kann dabei auch eine C#-Aktion sein, was die Erstellung eines Event-Systems ermöglicht. Darüber hinaus kann auch Spiel-Logik, wie beispielsweise ein Inventarsystem, innerhalb eines ScriptableObjects implementiert werden.

2.2 Serialisierung

Serialisierung (o. a. Marshalling) ist ein grundlegender Prozess in der Informatik, bei dem komplexe Datenstrukturen oder Objekte in ein Format umgewandelt werden, das zur externen Speicherung oder Übertragung geeignet ist. Die Serialisierung ermöglicht es, Daten in einer Form zu speichern oder zu übertragen, die ohne großen Aufwand wieder in den ursprünglichen State zurückversetzt werden kann, was als Deserialisierung (o. a. Unmarshalling) bezeichnet wird. Dabei wird „das rekonstruierte Objekt zu einem semantisch identischen Klon des originalen Objektes“ (vgl. [10]). Dieser Prozess ist in Abbildung 2.4) dargestellt.

So „lassen sich Serialisierungsformate in zwei Hauptkategorien unterteilen: textbasiert und binär“ [10]. Binärformate, wie zum Beispiel ProtoBuf von Google, wandeln die Daten in konkrete Bytes

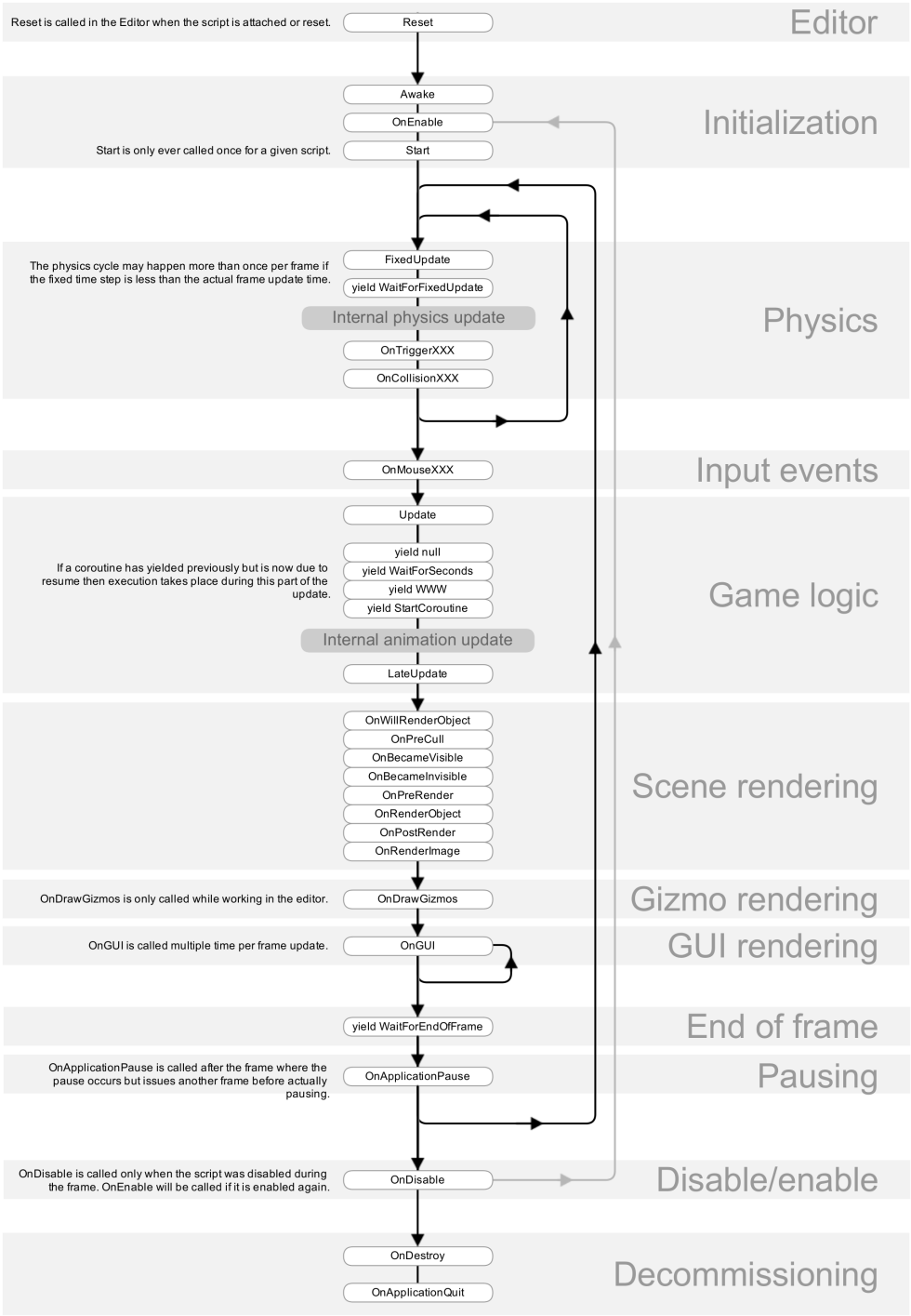


Abbildung 2.2: Lebenszyklus eines Scriptes; Bildquelle [31]

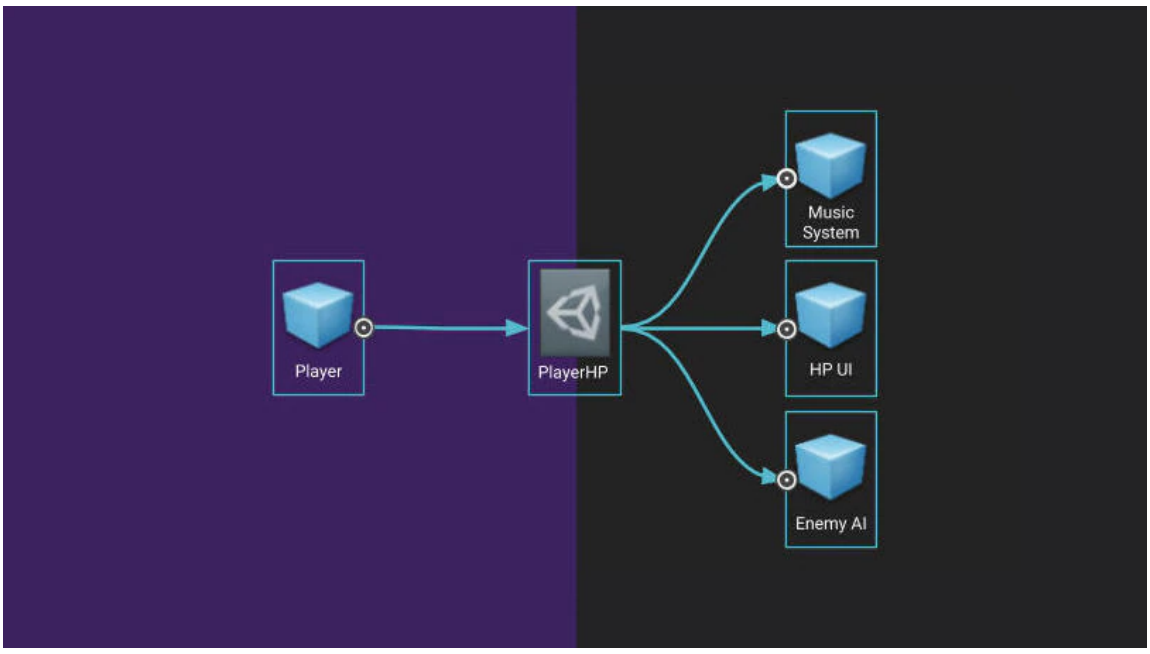


Abbildung 2.3: Anwendungsfall ScriptableObject; Bildquelle [32]

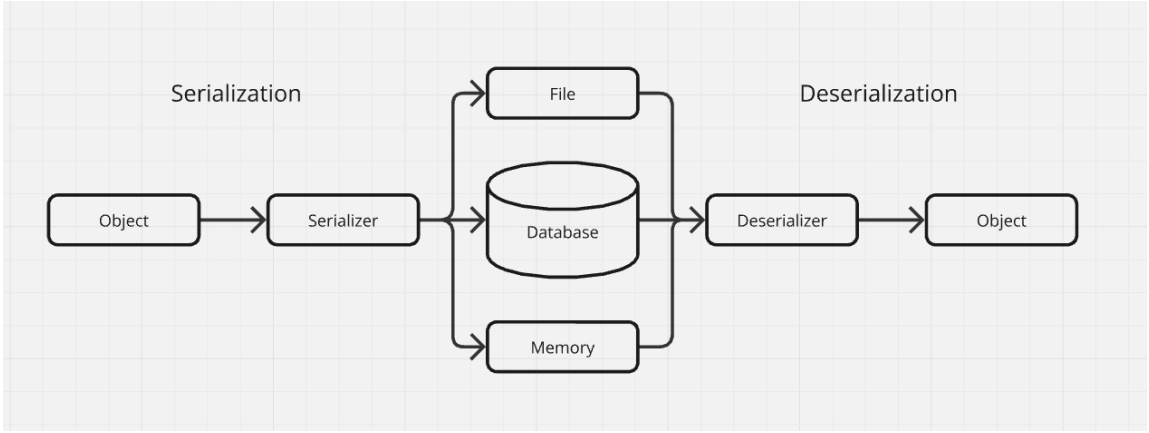


Abbildung 2.4: Serialisierung; Bildquelle [13]

um, was sie besonders effizient und kompakt macht. Im Gegensatz dazu sind textbasierte Formate wie JSON (JavaScript Object Notation) und XML (eXtensible Markup Language) für Menschen lesbar, weisen jedoch aufgrund dieser Lesbarkeit den Nachteil eines erhöhten Daten-Overheads auf.

Die Serialisierung lässt sich in zwei primäre Anwendungsbereiche unterteilen:

- **Datenübertragung zwischen Systemen:** Die Serialisierung spielt eine zentrale Rolle bei der Kommunikation zwischen unterschiedlichen Systemen, insbesondere bei der Übertragung von Daten über Netzwerke. Sie ermöglicht es, Daten in einem standardisierten Format zu übermitteln, das vom Empfängersystem korrekt interpretiert werden kann.

- **Speicherung von Daten:** Serialisierung ist essenziell für die langfristige Speicherung von Daten in verschiedenen Speichermedien wie Datenbanken, Dateien oder anderen Speicherlösungen. Sie erlaubt es, komplexe Datenstrukturen in einem formatgerechten und speicheroptimierten Format abzulegen. Dies ist von fundamentaler Bedeutung für diese Arbeit, da die Entwicklung eines entwicklerfreundlichen Speichersystems im Fokus steht.

Der Anwendungsfall, bei dem „das rekonstruierte Objekt zu einem semantisch identischen Klon des originalen Objektes“ (vgl. [10]) wird, eignet sich zudem, um Deep Copies von Objekten zu erstellen. Dies wird erreicht, indem die Daten zunächst serialisiert und anschließend deserialisiert werden, wodurch eine exakte Replik des ursprünglichen Objekts entsteht, die unabhängig von diesem agiert.

2.2.1 Vergleich von Serialisierungsformaten

In dem Paper von *Sumaray und Makki (2012)*[28] wurden verschiedene Serialisierungsformate hinsichtlich ihrer Leistung und Dateigröße verglichen, wobei der Fokus auf der Geschwindigkeit auf Android-Geräten lag. Untersucht wurden dabei die Formate XML, JSON, ProtoBuf und Thrift. Für die Tests haben die Autoren eine Android-App entwickelt, die das Serialisieren und Deserialisieren von Objekten, darunter einem Buch und einem Video, simulierten.

Die Ergebnisse der Studie zeigen, dass XML die schlechteste Leistung erbracht hat: „[...] XML is the slowest of the formats for serializing, and takes more than five times the amount of time to serialize than it takes the other formats.“[28, S. 5]. Zudem erzeugt XML die größten Dateien: „[...] XML is the largest, followed by JSON, then Thrift, with ProtoBuf being the smallest.“ [28, S. 4].

JSON vermeidet die Nachteile von XML, behält jedoch dessen Vorteile wie breite Nutzerbasis, menschliche Lesbarkeit und weit verbreitete Anwendung bei. Die binären Formate ProtoBuf und Thrift erweisen sich als schneller und erzeugen kleinere Dateien, sind jedoch weniger flexibel, da die im Binärformat gesendeten Daten nur geparst werden können, wenn der Empfänger über die *.proto* - oder *.thrift* Dateien verfügt. (vgl. [28, S. 6])

Zusammenfassend lassen sich die wichtigsten Ergebnisse wie folgt darstellen: „XML should be avoided unless necessary, as JSON is a superior alternative; [...] When serializing data for storage purposes, or when designing a new web service from the ground up, it is advantageous to use one of the binary formats due to their superior speed and size“ [28, S. 6].

2.2.2 Herausforderungen der Serialisierung

In ihrer Studie diskutieren Grochowski, Breiter und Nowak (2019) die technischen Herausforderungen der Serialisierung. Dies steht vor dem Hintergrund einer neuen C++-Bibliothek, welche vorgestellt wird. Zunächst thematisieren die Autoren die grundsätzlichen Schwierigkeiten der Serialisierung: „The portable serialization is not a straightforward process because different processor architectures (big-endian, little-endian) lead to a different binary representation of numbers and other objects, which potentially hinders portability. Objects accessed by reference should be properly restored in terms of inheritance and multi-base inheritance. [...] Complex structure, where single object could be referenced multiple times by various pointers and references, needs to be restored properly. Various object collections should be supported, including lists and dictionaries“ (vgl. [10]). Dies ist besonders relevant, da sie die technischen Herausforderungen und Komplexitäten zur Analyse und Entwicklung von Frameworks aufzeigt.

Darüber hinaus diskutieren die Autoren ein grundsätzliches Problem in der Serialisierung: „The perfect solution, meeting all requirements, does not exist, because the requirements are contradictory: The fastest serialization/deserialization process is achieved for binary format, but it is not portable between platforms. The most compact is also binary format [...]. The text formats, like JSON or XML, are portable and self-descriptive, but serialization/deserialization needs additional data processing, and archive takes significantly more space than the binary one and in result might be slower to transmit if needed“ [10] (vgl. [10]). Dies unterstreicht die Notwendigkeit der Abwägung des Datenformats, welches im Rahmen dieser Arbeit verwendet wird.

2.3 Manipulationssicherheit

Die Sicherheit spielt bei der Entwicklung von Software, wie beispielsweise bei der Serialisierung und Deserialisierung von Daten, eine zentrale Rolle. Da in dieser Arbeit Serialisierungsstrategien eine grundlegende Bedeutung haben, wird in diesem Kapitel der theoretischen Hintergrund von relevanten Sicherheitsaspekten für die Serialisierung thematisiert, damit diese bei der Deserialisierung nicht Manipuliert werden kann.

2.3.1 Deserialisierungsschwachstellen

In dem Paper *Security Vulnerabilities in Some Popular Object-Oriented Programming Languages* (2023)[19] werden die Sicherheitslücken, die in mehreren beliebten objektorientierten Programmiersprachen (OOPLs) bestehen, untersucht. In diesem Paper werden Deserialisierungsschwachstelle behandelt: „Insecure deserialization is a type of vulnerability that arises when an attacker can manipulate the serialized object and cause unintended consequences in the program's flow.“ [19, S. 34]. Dies kann zu einer Reihe von Sicherheitsrisiken führen, einschließlich Remote Code Execution (RCE), bei der ein Angreifer in der Lage ist, beliebigen Code auf dem Zielsystem auszuführen. Das Paper stellt außerdem einen Ansatz zur Prävention dieser Sicherheitslücke vor. Demnach sollten Entwickler vermeiden, komplexe Objekte zu serialisieren. Stattdessen empfiehlt es sich, einfache Datentypen wie Strings und Arrays zu verwenden, um das Risiko unsicherer Deserialisierungen zu minimieren. [19, S. 34]

Neben dieser Empfehlung gibt es weitere Richtlinien von der OWASP, einer weltweit anerkannten gemeinnützigen Organisation, die sich der Verbesserung der Sicherheit von Software widmet. Laut OWASP kann das Risiko erheblich reduziert werden, indem man native Serialisierungsformate

vermeidet. Durch den Wechsel zu einem reinen Datenformat wie JSON oder XML verringert sich die Wahrscheinlichkeit, dass benutzerdefinierte Deserialisierungslogik zu bösartigen Zwecken umfunktioniert wird (vgl. [6]). Darüber hinaus empfiehlt OWASP, dass Anwendungen, die vor der Deserialisierung wissen, welche Nachrichten verarbeitet werden müssen, diese als Teil des Serialisierungsprozesses signieren sollten. Auf diese Weise könnte die Anwendung entscheiden, keine Nachrichten zu deserialisieren, die keine authentifizierte Signatur aufweisen (vgl. [6]).

Die OWASP empfiehlt noch ein weiteres Vorgehen: Um Sicherheitsrisiken bei der Deserialisierung von Datenströmen zu minimieren, sollte der Datenstrom nicht zulassen, dass er den Typ des Objekts bestimmt, in den er deserialisiert wird. In Fällen, in denen es dennoch erforderlich ist, Datenströme zu deserialisieren, die ihren eigenen Typ definieren, sollten die zulässigen Typen streng eingeschränkt werden. Dies kann erreicht werden, indem der Deserializer ausschließlich Typen instanziiert, die ihm im Voraus bekannt sind. Es ist jedoch zu beachten, dass diese Praxis weiterhin Risiken birgt, da viele native .NET-Typen potenziell gefährlich sein können. Ein Beispiel hierfür ist der Typ System.IO.FileInfo. Wenn FileInfo-Objekte deserialisiert werden und auf Dateien auf dem Server verweisen, können sie deren Eigenschaften ändern, beispielsweise auf *schreibgeschützt* setzen, was ein potenzielles Denial-of-Service-Risiko darstellt [6].

Auch im Kontext eines Speichersystems in Spielen sind Sicherheitsrisiken bei der Deserialisierung relevant. Eine gängige Funktion von Spielen ist das Speichern von Spieldateien auf der eigenen Festplatte, um den Fortschritt des Spielers festzuhalten. Da diese Dateien lokal gespeichert werden, können Spieler auf sie zugreifen, sie manipulieren und möglicherweise mit anderen Spielern teilen. Wird diese manipulierte Datei dann von einem anderen Spieler geladen, könnte sie ungewollten Code auf dessen System ausführen und Schaden verursachen. Dieses Szenario verdeutlicht die Bedeutung sicherer Deserialisierungspraktiken, um zu verhindern, dass manipulierte Spieldateien zu einem Sicherheitsrisiko werden.

2.3.2 BinaryFormatter

Mit dem Release von .NET 9 wird der BinaryFormatter in C# aufgrund von Deserialisierungsschwachstellen nicht mehr unterstützt: „Der Typ BinaryFormatter ist riskant und wird für die Datenverarbeitung nicht empfohlen. Anwendungen sollten so bald wie möglich aufhören, BinaryFormatter zu verwenden, auch wenn Sie der Auffassung sind, dass die verarbeiteten Daten vertrauenswürdig sind. BinaryFormatter ist unsicher und kann nicht sicher gemacht werden.” [5]. Ein Angreifer könnte dadurch erfolgreich eine Denial-of-Service-Attacke (DoS) durchführen, Informationen preisgeben oder beliebigen Code ausführen. Microsoft beschreibt das Ausführen von *BinaryFormatter.Deserialize()* in einer einfachen Analogie „als eigenständige ausführbare Datei und deren Starten entspricht.”[5].

Aus diesem Grund wird im Rahmen dieser Arbeit auf die Verwendung des BinaryFormatters vollständig verzichtet. Stattdessen werden alternative, sicherere Methoden zur Serialisierung und Deserialisierung von Daten eingesetzt, die den aktuellen Sicherheitsstandards entsprechen. Empfehlungen von Microsoft umfassen dabei die Libraries *System.Text.Json* oder *XmlSerializer*.

2.3.3 Verschlüsselung von Daten

Um Deserialiserungsschwachstellen zu vermeiden, ist die Verschlüsselung der serialisierten Daten relevant. Verschlüsselung ist eine grundlegende Technik zur Sicherung von Daten, indem sie

in eine unlesbare Form umgewandelt werden, die nur mit einem speziellen Schlüssel wieder entschlüsselt werden kann. Ziel bei der Verschlüsselung ist es, „die Vertraulichkeit und Integrität von Daten zu gewährleisten. Es geht darum, geheime Informationen geheim zu halten, sie zu schützen und unbefugte Änderungen zu verhindern (vgl. [7, S. 344]).

In einer Studie von 2021 wurden verschiedene gängige symmetrische verschlüsselungsalgorithemn auf ihre Performance verglichen: AES, 3DES, Blowfish und Twofish. Symmetrische Verschlüsselungsalgorithmen basieren darauf, dass „derselbe Schlüssel für die Ver- und Entschlüsselung verwendet wird“ (vgl. [7, S. 344]). Im Gegensatz dazu verwenden asymmetrische Algorithmen (Public-Key) „verschiedene Schlüssel – einen für die Verschlüsselung und einen anderen für die Entschlüsselung“ (vgl. [7, S. 344]). Dabei wurden die folgenden zentralen Erkenntnisse gewonnen: „The result shows that AES is the most efficient in encryption and decryption from execution time point of view. When the files size is less than 10MB, 3DES comes in the second level and Blowfish in the third, but when the file size exceeds 10MB Blowfish gives a much better result than 3DES in both of the encryption and deception. Twofish gained the worst results at all. Regarding the encryption memory utilization, we noticed that AES and 3DES consumed less memory and relatively they utilized same amount of memory. While Blowfish and Twofish take more memory, and they have the biggest ciphertext size.“ [7, S. 348].

2.3.4 Schwachstellen bei Verschlüsselung

Trotz der Stärke moderner Verschlüsselungsalgorithmen gibt es jedoch Schwachstellen, die ausgenutzt werden können. In einem Paper wird dies als „[...] probably one of the most severe vulnerabilities that can happen in an application“ [19, S. 35] definiert. In dem Artikel *A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions* (2023) wird darauf hingewiesen, dass Kryptographische Angriffe auftreten, wenn ein Angreifer versucht, ein Verschlüsseltes System zu kompromittieren, indem er Schwächen im Code, Chiffre, kryptografischen Protokoll oder im Schlüsselverwaltungsschema identifiziert (vgl. [18]). Dabei ist die korrekte und genaue Implementierung von Kryptographie äußerst entscheidend für ihre Wirksamkeit. Nach Angaben der OWASP führt ein kleiner Fehler in der Konfiguration oder im Code dazu, dass der Großteil des Schutzes aufgehoben wird und die Kryptografie-Implementierung nutzlos wird (vgl. [20]). Die OWASP bietet eine Reihe von Empfehlungen, die in diesem Abschnitt vorgestellt werden. Zu den wichtigsten Aspekten zählen die Schlüsselgenerierung, die Lebensdauer und Rotation, die sichere Speicherung, die Trennung von Schlüsseln und Daten sowie die Verschlüsselung gespeicherter Schlüssel. Darüber hinaus wird der Umgang mit Secrets thematisiert.

- **Schlüsselgenerierung**: Verschlüsselungsschlüssel sollten mithilfe einer kryptografisch sicheren Methode zufällig generiert werden. Es wird empfohlen, keine leicht erratbaren Wörter oder zufällige Zeichenfolgen, die durch unstrukturierte Eingaben erzeugt wurden, zu verwenden (vgl. [20]).

- **Lebensdauer und Rotation von Schlüsseln**: Die Lebensdauer und Rotation von Schlüsseln ist ein wichtiger Aspekt der Sicherheit. Verschlüsselungsschlüssel sollten regelmäßig gewechselt (rotiert) werden. Gründe für den Wechsel können sein, wenn der Schlüssel kompromittiert wurde oder dies vermutet wird, wenn die festgelegte Nutzungsdauer (auch *Kryptoperiode* genannt) abgelaufen ist, wenn der Schlüssel eine bestimmte Menge an Daten verschlüsselt hat oder wenn sich die Sicherheitslage ändert, beispielsweise durch neu entdeckte Schwachstellen im verwendeten Algorithmus (vgl. [20]).

- **Speicherung von Schlüsseln**: Die sichere Speicherung von Schlüsseln stellt eine der größten Herausforderungen dar. Einige grundlegende Richtlinien sollten dabei beachtet werden: Schlüssel dürfen nicht im Quellcode der Anwendung hartkodiert werden, sie sollten nicht in Versionskontrollsysteme eingecheckt werden, und Konfigurationsdateien, die Schlüssel enthalten, sollten mit restriktiven Berechtigungen geschützt werden. Darüber hinaus sollte das Speichern von Schlüsseln in Umgebungsvariablen vermieden werden, da diese unbeabsichtigt offengelegt werden könnten (vgl. [20]).
- **Trennung von Schlüsseln und Daten**: Verschlüsselungsschlüssel sollten, wenn möglich, getrennt von den verschlüsselten Daten aufbewahrt werden. Dies bedeutet beispielsweise, dass die Daten in einer Datenbank und die Schlüssel im Dateisystem gespeichert werden. So kann ein Angreifer, der nur Zugang zu einem der beiden Systeme hat, nicht sowohl auf die Schlüssel als auch auf die Daten zugreifen (vgl. [20]).
- **Verschlüsselung gespeicherter Schlüssel**: Es wird empfohlen, die Schlüssel selbst in verschlüsselter Form zu speichern. Dies erfordert den Einsatz von zwei separaten Schlüsseln: einem Schlüssel zur Verschlüsselung der eigentlichen Daten (Data Encryption Key, kurz *DEK*) und einem weiteren Schlüssel zur Verschlüsselung des DEK (Key Encryption Key, kurz *KEK*). Der KEK sollte dabei separat vom DEK gespeichert werden, um sicherzustellen, dass ein Angreifer nicht beide Schlüssel leicht zusammenführen kann (vgl. [20]).

In der Softwareentwicklung bezieht sich der Begriff *Secrets* auf vertrauliche oder sensible Informationen, die für den Betrieb von Anwendungen und Diensten notwendig sind, aber nicht öffentlich zugänglich sein dürfen. Im Kontext der Verschlüsselung zählen insbesondere Schlüssel zu den sogenannten Secrets. Darüber hinaus empfiehlt OWASP spezielle Vorgehensweisen für den sicheren Umgang mit Secrets, um deren Schutz und Integrität zu gewährleisten, welche im nachfolgenden Präsentiert werden:

- **Zugriffskontrolle**: Nicht alle Benutzer sollten Zugang zu allen Secrets haben. Es sollte das Prinzip der geringsten Privilegien angewendet werden, sodass nur autorisierte Personen Zugriff erhalten (vgl. [24]).
- **Automatisierung**: Der manuelle Umgang mit Secrets erhöht das Risiko von Fehlern und Lecks. Es ist ratsam, den menschlichen Eingriff in die Verwaltung von Secrets zu minimieren oder vollständig zu automatisieren (vgl. [24]).
- **Speicherhandhabung**: Geheimnisse sollten so kurz wie möglich im Speicher gehalten werden, und der Zugriff sollte auf ihren Speicherbereich eingeschränkt sein (vgl. [24]).
- **TLS Everywhere**: Secrets sollten niemals im Klartext übertragen werden. Die Verwendung von TLS (Transport Layer Security) für alle Übertragungen ist heutzutage ein Muss (vgl. [24]).

2.4 Speichern und Laden in Games

Das Speichern in Spielen stellt eine zentrale Mechanik dar. In dem Buch *Fundamentals of Game Design* (2010) wird der Begriff folgendermaßen definiert: „Saving a game takes a snapshot of a game world and all its particulars at a given instant and stores them away so that the player can later load the same data, return to that instant, and play the game from that point." [1, S. 279]. Das Speichern von Spielständen „würde es den Spielern erlauben oder sie sogar dazu verpflichten,

Spiele zu spielen, die viele hundert Mal länger dauern als die Viertelminuten in den Spielhallen" (vgl. [14, S. 488]). Aus einer designtechnischen Perspektive bietet das Speichern jedoch weitere Vorteile: Einerseits, dass der Spieler sich von katastrophalen Fehlern erholen kann und andereseits, um den Spieler das Erkunden von alternativen Stretegien zu ermöglichen (vgl. [1, S. 279–280]). Dabei gibt es verschiedene Arten, wie ein Speicherstand erstellt wird: „They may be triggered when certain things are done, reached, or crossed, or they may be selected by players. This save triggering can become an opportunity for particularly tense game play as areas or times between save points are navigated carefully." [14, S. 488–489].

2.4.1 State Based Saving

Aus technischer Perspektive umfasst der Speichervorgang das Aufzeichnen und Serialisieren von Daten, die den State des Spiels zum Zeitpunkt der Speicherung abbilden. Hierzu zählen in der Regel Variablen wie Position der Spielfigur, Inventarobjekte und verschiedene Statusparameter. Die Herausforderung liegt dabei in der effizienten und verlustfreien Speicherung dieser Daten, insbesondere in komplexen Open-World-Spielen oder solchen, die große Mengen an dynamischen Objekten und Interaktionen verwalten müssen.

Für die Erstellung von Snapshots und dem anschließenden Speichern von Daten eignet sich ein spezifisches Design Pattern: das Memento Pattern. Die *Gang of Four* definiert dieses Pattern als: „Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later" [22, S. 303]. Das Memento Pattern besitzt drei Hauptbeteiligte: Originator, Memento und Caretaker. Der *Originator* ist das Objekt, dessen State gespeichert und wiederhergestellt werden soll. Es erstellt ein Memento, das den aktuellen State sichert. Später kann der State mithilfe des Mementos wiederhergestellt werden. Das *Memento* ist ein Datenobjekt, das den gespeicherten State des Originators enthält. Es gibt keine Methoden, die von außen zugänglich sind, um den State direkt zu ändern oder einzusehen, sodass die Kapselung gewährleistet ist. Der *Caretaker* ist ein Objekt, welches Mementos verwaltet und dafür sorgt, dass sie bei Bedarf wieder an den Originator übergeben werden, um dessen State wiederherzustellen. „When a caretaker wants to record the state of the originator, [...] it first asks the originator for a memento object" [22, S. 304]. Der Caretaker kennt jedoch nicht den Inhalt des Mementos. Eine konkrete Visualisierung ist in 2.5 zu sehen. Durch dieses Entwurfsmuster wird einerseits das Prinzip der Kapselung gewahrt, während gleichzeitig eine einfache Wiederherstellung des States ermöglicht wird. Allerdings bringt das Memento-Muster einen spezifischen Nachteil mit sich: „Eine hohe Anzahl von Mementos erfordert mehr Speicherplatz und belastet gleichzeitig den Betreuer zusätzlich" (vgl. [22, S. 318]).

2.4.2 Command Based Saving

Eine weniger verbreitete Alternative besteht darin, nicht den konkreten State des Spieles zu speichern, sondern stattdessen die Eingaben des Spielers. Dieser Ansatz reduziert den Speicheraufwand erheblich, da anstelle der Speicherung vieler Variablen lediglich eine chronologische Liste der Benutzeraktionen (Inputs) oder Befehle gespeichert wird. Die Wiederherstellung des Systems kann durch das erneute Ausführen dieser gespeicherten Eingaben in chronologischer Reihenfolge erfolgen, um den ursprünglichen State zu rekonstruieren. Dies ist besonders dann effizient, wenn sich der State des Systems hauptsächlich durch Benutzerinteraktionen verändert und sich der Anfangszustand des Systems leicht reproduzieren lässt.

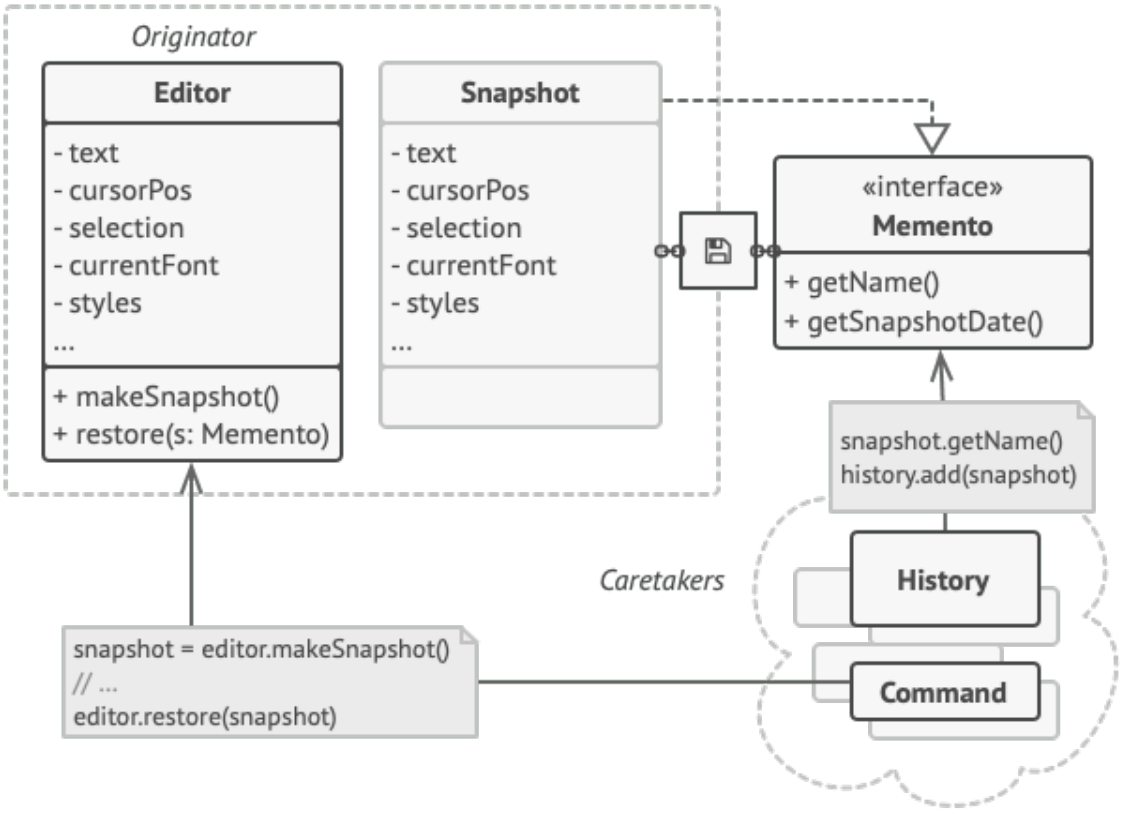


Abbildung 2.5: Memento Pattern; Bildquelle [21]

Das zuvor vorgestellte Memento Design Pattern kann auch als Grundlage für das Command Based Saving genutzt werden. Dabei lässt sich das Konzept des Memento Patterns leicht abwandeln: Anstatt den kompletten States des Systems zu speichern, können die Eingaben oder Aktionen des Benutzers erfasst werden. Diese Aktionen werden dann genutzt, um den State bei Bedarf zu rekonstruieren. Dadurch können beispielsweise Replays leicht implementiert werden. Auch für diesen Ansatz gibt es ein passendes Design Pattern: das Command Pattern. Die *Gang of Four* definiert dieses Pattern mit folgenden Worten: „Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queues, or log requests, and supports undoable operations.“ [22, S. 263]. Dieses Konzept wird in Abbildung 2.6 veranschaulicht.

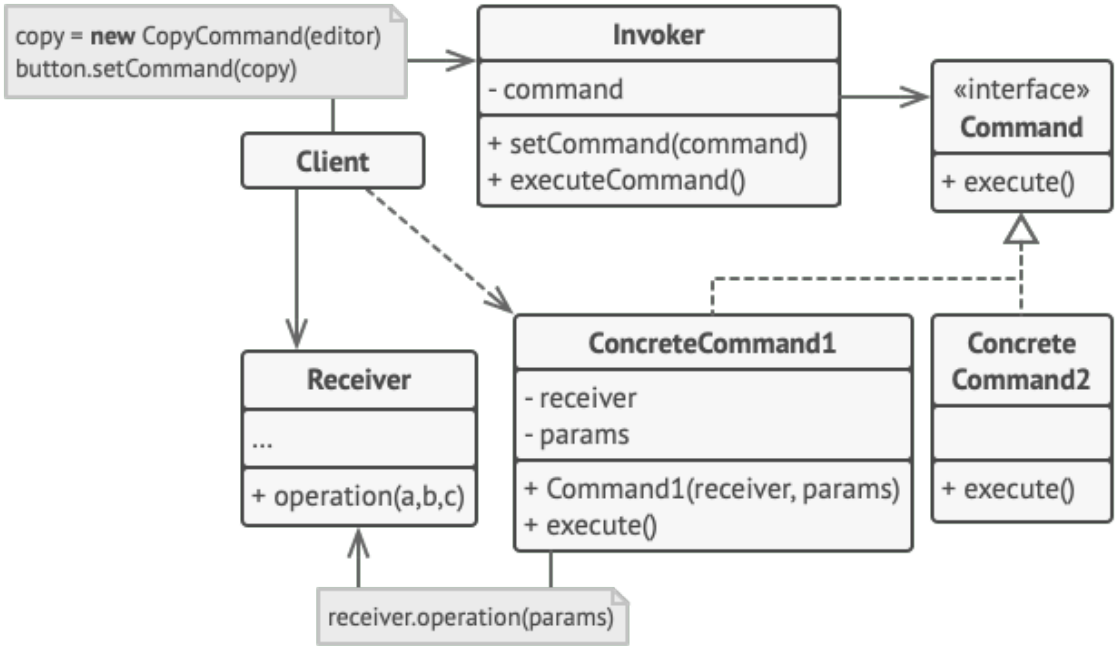


Abbildung 2.6: Command Pattern; Bildquelle [21]

2.4.3 Implementierungen in Game Engines

Dieser Abschnitt beschäftigt sich damit, wie verschiedene Spiele-Engines das Speichern und Laden von Spielständen umsetzen. Die Ansätze der Engines unterscheiden sich teils erheblich: Während einige integrierte Lösungen bereitstellen, erfordern andere eine eigene Implementierung. Im Folgenden werden die Speichermechanismen der weit verbreiteten Engines Unreal, Godot und Unity genauer betrachtet.

Unreal Engine

Die Unreal Engine ist eine leistungsstarke und weit verbreitete Spiel- und Entwicklungsplattform, die von Epic Games entwickelt wird. Ihre Ursprünge reichen bis ins Jahr 1998 zurück, jedoch wurde die Engine erst im Jahr 2014 mit der Veröffentlichung von Unreal Engine 4 für den breiten Markt zugänglich. Seither hat sie sich zu einer der vielseitigsten Echtzeit-Engines für die Erstellung interaktiver 3D-Umgebungen entwickelt. Dank ihrer Flexibilität und Leistung wird sie in einer

Vielzahl von Branchen eingesetzt, darunter in der Videospielindustrie, Architekturvisualisierung, Filmproduktion und Forschung.

Speichern und Laden in der Unreal Engine

Die Unreal Engine verwendet zur Implementierung von Features das sogenannte *Blueprint Visual Scripting*. In der Dokumentation beschreibt Epic Games dieses System folgendermaßen: „The Blueprint Visual Scripting system in Unreal Engine is a complete gameplay scripting system based on the concept of using a node-based interface to create gameplay elements from within Unreal Editor." [33]. Demnach ist ein wesentlicher Vorteil der Blueprint Visual Scripting-Methode, dass Entwickler auch ohne tiefere Programmierkenntnisse in der Lage sind, komplexe Speicherlogiken zu erstellen. Durch den Einsatz von *Nodes* können gewünschte Funktionalitäten aus einer Menge an Nodes gewählt werden. Eine dieser unterstützten Funktionalitäten ist ein System für das Speichern von Spielständen: die sogenannte SaveGame-Klasse, die als Blueprint oder im Code implementiert werden kann. Diese Variabilität der Unreal Engine ermöglicht einerseits eine einfache Implementierung des Speicherns und Ladens von Spielständen und andererseits eine skalierbare Lösung für komplexere Anforderungen. Die richtige Herangehensweise hängt hierbei immer von dem Umfang des Projektes und den Programmierkenntnissen der Entwickler ab.

Godot

Godot ist eine Open-Source und plattformübergreifende Game Engine, die seit ihrer Veröffentlichung im Jahr 2014 eine zunehmende Verbreitung in der Entwicklung von interaktiven Anwendungen und Videospielen findet. Sie bietet eine umfassende und integrierte Entwicklungsumgebung, die die Erstellung von 2D- und 3D-Spielen ermöglicht. Durch die Unterstützung verschiedener Programmiersprachen, insbesondere ihrer eigenen Sprache GDScript, erleichtert Godot die flexible und modulare Entwicklung. Ein wesentlicher Vorteil der Engine ist ihre kostenlose Verfügbarkeit unter der MIT-Lizenz, wodurch sie für kommerzielle und nicht-kommerzielle Projekte gleichermaßen attraktiv ist.

Speichern und Laden in Unity und Godot

Sowohl Unity als auch Godot bieten im Gegensatz zur Unreal Engine kein vorgefertigtes, umfassendes System für das Speichern und Laden von Spielständen an. Dies bedeutet, dass Entwickler in beiden Engines entweder eine eigene Lösung zur Verwaltung von Speichermechanismen implementieren oder auf bereits existierende, von der Community erstellte Tools und Bibliotheken zurückgreifen müssen. Der Entwicklungsprozess für das Speichern von Spielständen erfordert daher ein tieferes Verständnis der Engine-Architektur sowie der Funktionsweise von Dateioperationen und Datenpersistenz.

Für einfache Anforderungen, wie das Speichern von Spieler-Konfigurationen, bieten beide Engines jedoch native Möglichkeiten. In Unity kann hierfür das *PlayerPrefs* System verwendet werden. PlayerPrefs ermöglichen das Speichern in Form von Kay-Value-Paaren. Dieses System ist für einfachere Szenarien gut geeignet, allerdings ohne Verschlüsselung, weswegen Unity empfielt PlayerPrefs nicht für sensible Daten zu verwenden [31].

Godot bietet für ähnliche Zwecke das *ConfigFile* System an. ConfigFile ermöglicht das Lesen und Schreiben von Konfigurationsdaten in einer strukturierten Textdatei, was sich insbesondere für die Verwaltung von Spielereinstellungen eignet. Es bietet Entwicklern eine höhere Flexibilität bei der

Organisation und Strukturierung der gespeicherten Daten, ist jedoch ebenfalls primär für einfache Konfigurationen konzipiert und nicht für das Speichern komplexer Spielstände gedacht.

Kapitel 3

Methodik

In der vorliegenden wissenschaftlichen Arbeit wird eine qualitativ-iteratives Vorgehensweise angewendet, um ein fundiertes und praxisnahes System zu entwickeln, das auf den spezifischen Anforderungen von Entwicklern in Unity basiert. Die gewählte Methodik vereint qualitative und analytische Ansätze, um eine umfassende und praxisorientierte Lösung zu erarbeiten.

Zunächst wird ein Experteninterview durchgeführt, um fundierte Einblicke in die Anforderungen der zuvor definierten Zielgruppe zu gewinnen. Durch die systematische Erhebung und Auswertung der Expertenmeinungen sowie die Heranziehung von Hypothesen können Anforderungen identifiziert werden, die als Grundlage für die anschließende Analyse bestehender Systeme dienen. Diese Anforderungsanalyse stellt sicher, dass die Untersuchung nicht nur theoretisch fundiert, sondern auch praxisnah gestaltet ist.

Auf Basis der gewonnenen Erkenntnisse wird ein eigenes System entwickelt, welches im Anschluss anhand der zuvor erhobenen Anforderungen evaluiert wird. Die Durchführung von User-Tests stellt dabei sicher, dass das entwickelte System nicht nur theoretisch den Anforderungen entspricht, sondern auch in der Praxis gut nutzbar ist. Dies wird durch den Einsatz eines System Usability Scale (SUS) Fragebogens überprüft. In Kombination mit dem theoretischen Hintergrund gewährleistet diese methodische Vorgehensweise eine fundierte, praxisorientierte Entwicklung, die sowohl auf wissenschaftlicher Genauigkeit als auch auf praktischer Relevanz basiert. Dies ist besonders in Hinblick auf die geringe Anzahl an Primärquellen relevant.

In den folgenden Kapiteln wird detailliert auf die einzelnen Schritte dieser Methodik eingegangen, beginnend mit der Planung und Durchführung der Experteninterviews, gefolgt von der systematischen Anforderungsanalyse und der anschließenden Bewertung der existierenden Systeme.

3.1 Experteninterview

Das anonymisierte, semi-strukturierte Experteninterview wurde als methodisches Werkzeug ausgewählt, um spezifisches Fachwissen und praktische Erfahrungen von Experten zu erfassen, die für die Entwicklung einer fundierten Anforderungsanalyse von entscheidender Bedeutung sind. Im Anhang .2 dieser Arbeit sind die Ergebnisse dokumentiert. Ein zentrales Ziel dieses Ansatzes bestand darin, sicherzustellen, dass das entwickelte System genau auf die Bedürfnisse

und Erwartungen der Entwicklergemeinde zugeschnitten ist. Durch die gezielte Einbindung von Experten konnte das Interview dazu beitragen, potenzielle Hürden und Schwierigkeiten bereits im Vorfeld zu identifizieren und geeignete Lösungsansätze zu entwickeln. Dies ist von entscheidender Bedeutung für die Ermittlung von Anforderungen, die sich an realen Herausforderungen orientieren und die durch die vorhandene Literatur oder theoretische Rahmenwerke allein nicht vollständig erfasst werden können.

3.1.1 Expertenselektion

Für das Projekt wurden insgesamt sechs Experten ausgewählt. Die Rekrutierung der Experten erfolgte ausschließlich über Kontakte im akademischen Umfeld, welche durch gemeinsame Forschungsprojekte sowie durch die Zusammenarbeit mit Dozenten an Hochschulen entstanden sind.

Die Auswahl der Expertengruppe erfolgte basierend auf spezifischen, vorab definierten Kriterien, die sich aus der Zielgruppe des Projekts ableiten. Ein zentrales Auswahlkriterium war die langjährige Erfahrung der Experten im Umgang mit der Entwicklungsplattform Unity sowie der Programmiersprache C#. Es wurde zudem Wert darauf gelegt, dass die Experten über Kenntnisse in der Serialisierung und im Speichern von Datenstrukturen verfügen, da diese Aspekte für das Projekt von besonderer Relevanz sind.

Zur Sicherstellung der Diversität wurde auf eine ausgewogene Verteilung zwischen Vertretern der Industrie und der Wissenschaft geachtet. Die Hälfte der ausgewählten Experten ist aktuell in der Industrie tätig und bringt praxisnahe Erfahrungen ein, während die andere Hälfte im wissenschaftlichen Bereich arbeitet.

3.1.2 Format des Interviews

Vor den Interviews wurde ein Leitfaden mit offenen Fragen erstellt, der als Grundlage für die Gespräche diente. Um die Klarheit und Verständlichkeit der Fragen sowie deren logische Abfolge zu gewährleisten, wurde eine Pilotphase durchgeführt. Diese Pilotphase ermöglichte es, eventuelle Unklarheiten oder Missverständnisse in den Fragen zu identifizieren und zu korrigieren.

Die Interviews deckten mehrere zentrale Themenbereiche ab. Zunächst wurden grundlegende demographische Daten der Teilnehmer erhoben, um die Diversität und Repräsentativität der Stichprobe sicherzustellen. Anschließend richteten sich die Fragen auf aktuelle und existierende Ansätze, wobei die Teilnehmer nach ihrer Kenntnis und Erfahrung mit bestehenden Systemen und Praktiken in ihrem Fachgebiet befragt wurden. Ein wesentlicher Schwerpunkt der Interviews lag zudem darauf, die Herausforderungen und Probleme zu erörtern, auf die die Experten in der Vergangenheit gestoßen waren. Zudem lag ein weiterer Schwerpunkt der Befragung auf den Anforderungen und Erwartungen der Experten an ein potenzielles neues System. Wenn sich während des Interviews herausstellte, dass zu bestimmten Themen zusätzliche oder detaillierte Informationen erforderlich waren, wurden spontan neue Fragen formuliert und gestellt.

Während des gesamten Interviewprozesses wurden die Inhalte aller Interviews zeitgleich schriftlich protokolliert, um eine präzise und vollständige Dokumentation der Antworten sicherzustellen.

3.1.3 Analyse der Daten

Die Analyse der Interviewdaten erfolgte anonymisiert mittels einer Analyse, die sich an den Prinzipien der *Qualitativen Inhaltsanalyse nach Mayring* orientierte [15]. Die protokollierten Inhalte des Interviews wurden gesammelt und die gleichen Aussagen gekürzt. Die Aussagen wurden zunächst auf Karten sortiert und anschließend in thematische Cluster eingeteilt. Dabei entstanden folgende Bereiche: Demographischer Hintergrund, Erfahrungen und Fähigkeiten, Verwendete Tools, Challenges, Ideen, Verwendete Serialisierungsformate, Anforderungen und Zusätzliche Überlegungen. Im weiteren Verlauf wurden übergeordnete Cluster formuliert, die sich an den Anforderungen, Ideen und Problemen orientieren. Ein besonderer Fokus lag dabei darauf, die Oberbegriffe so zu definieren, dass sie als Anforderungen für die Anforderungserhebung dienen können und gleichzeitig die gesetzten Hypothesen beinhalten. Auf diese Weise konnten zentrale Themen und Muster für die späteren Schritte definiert oder mittels dem theoretischen Hintergrund gestützt werden. Dabei wurde Wert darauf gelegt, dass alle Karten in entsprechende Kategorien eingeordnet wurden, auch wenn es vorkam, dass einzelne Punkte in mehrere Kategorien eingeordnet werden konnten.

3.1.4 Ergebnisse

Zunächst werden die definierten Kategorien behandelt, die direkt für die Anforderungsanalyse relevant sind. Dazu gehören die folgenden Kategorien: Speicheroptimierung, Performance, Versionierung, plattformübergreifende Kompatibilität, Sicherheit, Komplexität der Datenstrukturen, Fehlerbehebung und Datenkonsistenz. Darüber hinaus wurden Aspekte identifiziert, die nicht unmittelbar als Kategorien für Anforderungen eingeordnet werden können. Dazu gehören Networking und Datenbanken, Architektur sowie seltene Serialisierungsstrategien.

Speicheroptimierung

Eine der zentralen Anforderungen, die von den Experten mehrfach erwähnt und als wesentlich erachtet wurde, ist die Minimierung der Speicherplatznutzung. Diese Optimierung hat nicht nur direkte Auswirkungen auf die Speicheranforderungen, sondern beeinflusst auch die Ladezeiten. Besonders in Umgebungen mit begrenztem Speicherplatz, wie bei mobilen Geräten, ist Speicheroptimierung von größter Bedeutung.

Die Wahl des Dateiformats spielt dabei eine entscheidende Rolle: Für die Speicheroptimierung werden binäre Formate bevorzugt, da sie einen geringeren Speicherbedarf aufweisen und näher an der Maschinenebene arbeiten, indem sie konkrete Bytes speichern. Ein weiteres Dateiformat, das allerdings von den Experten der Industrie als Industriestandard angesehen wird, ist JSON. Die menschliche Lesbarkeit wird hierbei dem erhöhten Speicherbedarf bevorzugt. Um die Stärken von sowohl binären Formaten und JSON zu kombinieren, hat ein Experte eine innovative Idee genannt: Die mögliche Umwandlung zwischen binären Formaten und JSON. Weitere erwähnte Serialisierungsansätze, welche zudem als Kategorie definiert wurden, umfassen XML, das mittlerweile hauptsächlich für Visualisierungszwecke verwendet wird, sowie CSV und spezifische Formate wie Chronos GLTF für programmierbare Daten. Zudem wurden selbst entwickelte Serialisierer in speziellen Anwendungsfällen erwähnt.

Ein weiterer wesentlicher Faktor ist die Datenkompression, die zur zusätzlichen Verringerung des Speicherplatzbedarfs beiträgt. Auch architektonische Überlegungen spielen eine Rolle, insbesondere wie Daten strukturiert werden, um den verfügbaren Speicher effizient zu nutzen. Ein

konkreter Ansatz, den ein Experte aus dem Model-View-Controller (MVC) Konzept abgeleitet hat, besteht darin, nur das Datenmodell und bestimmte Zustände von Klassen zu speichern. Das bedeutet, dass große Assets nicht gespeichert werden. Stattdessen werden lediglich Referenzen zu diesen Ressourcen gespeichert, wobei die Klassen wissen, wo die benötigten Assets zu finden sind. Durch diese Vorgehensweise wird der Speicherbedarf erheblich reduziert und die Verwaltung der Daten vereinfacht. Der Autor des Buches „Java Desing Patterns" (2016) definiert das MVC-Muster folgendermaßen: „you separate the user interface logic from the business logic and decouple the major components in such a way that they can be reused efficiently." [22, S. 437].

Komplexität der Datenstrukturen

Als Negativbeispiel für die Komplexität der Datenstrukturen innerhalb der Entwicklungsumgebung Unity wurden von den Experten die PlayerPrefs genannt. Dies liegt daran, dass die gespeicherten Datentypen schwer nachvollziehbar seien, was es schwierig macht, den Überblick zu behalten. Zudem sind PlayerPrefs anfällig für Manipulationen, weshalb sie sich in der Regel nur für das Speichern von Benutzereinstellungen eignen.

Das MVC-Konzept aus der Speicheroptimierung kann auch hier angewendet werden: Anstatt vollständige Objekte zu speichern, sollten nur Referenzen zu den entsprechenden Assets gespeichert werden. Ein weiterer Experte nannte eine benutzerdefinierte Lösung als Beispiel, bei der einige Teile lokal gespeichert werden, während der Großteil durch Referenzen verwiesen wird. Dabei wurden IDs als möglicher Ansatz zur Implementierung dieser Referenzen erwähnt.

Dieses Konzept kann darauf ausgerichtet werden, dass sämtliche Objektinstanzen als Referenz markiert werden können. Nach Angabe eines Experten seien ein zentrales Problem bei der Verwaltung komplexer Datenstrukturen zyklische und verschachtelte Objekte. Diese können bei der Serialisierung und Deserialisierung zu Endlosschleifen führen. Ein genannter Lösungsansatz des Experten besteht darin, nach einer bestimmten Anzahl von Abhängigkeiten die Serialisierung zu stoppen oder jedes zyklische Objekt nur einmal zu berücksichtigen.

Darüber hinaus wurde die Unterstützung von Vererbung beim Speichern von Experten mehrfach als wichtig hervorgehoben. Dies impliziert, dass Klassen, die von Basisklassen abgeleitet sind, korrekt einbezogen werden müssen. Wenn ein Basistyp in einer Klasse definiert ist, jedoch ein polymorphisches Objekt verwendet wird, muss die korrekte Instanz des Objekts gespeichert werden.

Performance

Eine der zentralen Anforderungen, die im Rahmen der Experteninterviews deutlich hervorgehoben wurde, ist die Notwendigkeit einer hohen Performance des Speichersystems. Das System sollte ressourcenschonend bleiben und gute Ladezeiten bieten. Wie bereits zuvor erwähnt, wird dies durch eine effizientere Speicherbelegung unterstützt. Darüber hinaus betonte ein Experte die Bedeutung thread-sicherer asynchroner Prozesse. Diese Prozesse sind unerlässlich, um sicherzustellen, dass ressourcenintensive Aufgaben im Hintergrund ausgeführt werden können, ohne dabei den Haupt-Thread zu blockieren. Da Unity Single-Threaded agiert, ist das Blockieren des Haupt-Threads in Unity besonders problematisch. Dadurch werden alle zentralen Aufgaben, wie das Rendering der Grafik, die Verarbeitung der Benutzerinteraktionen und die Spielphysik, im Haupt-Thread abgewickelt. Wenn dieser Thread durch zeitintensive Dateioperationen blockiert

wird, kann dies zu Rucklern, Einfrieren der Benutzeroberfläche oder einer stark beeinträchtigten Bildrate (Framerate) führen.

Versioning

Das Thema der Versionierung wurde sowohl in den Herausforderungen als auch in den Anforderungen mehrfach genannt. Insbesondere im Kontext der langfristigen Wartung und Weiterentwicklung von Spielen ist eine sorgfältige Versionierung von entscheidender Bedeutung. Der Fortschritt der Spieler muss nach neuen Updates erhalten bleiben, da der Verlust von Spielfortschritten erhebliche negative Auswirkungen auf die Spielererfahrung haben kann. Aufgrund dieser kritischen Bedeutung wurde das Thema Versionierung von den Experten wiederholt betont und als wesentliches Element für die Systemanforderungen hervorgehoben.

Plattformübergreifende Unterstützung

Die Kompatibilität zwischen verschiedenen Systemarchitekturen wurde von Experten als eine weitere Herausforderung dargestellt, insbesondere wenn unterschiedliche Plattformen und Technologien eingesetzt werden. Diese Inkompatibilität stammt unter anderem durch verschiedene Byte-Reihenfolgen (Endianess) und verschiedene Kulturen.

Fehlerbehandlung und Datenkonsistenz

Ein weiteres Problem, das von einem Experten im Zusammenhang mit dem Speichern und Laden von Daten definiert wurde, ist die Differenzierung der Fehlerquellen. Es ist essentiell, präzise zu unterscheiden, ob Fehler durch Ladeprozesse oder durch normales Gameplay bzw. Benutzereingaben verursacht wurden. Diese Unterscheidung gestaltet sich oft als schwierig, ist jedoch unerlässlich, um eine effektive Fehlerbehebung zu gewährleisten.

Ebenso sei es von Bedeutung, dass Serialisierungs- und Deserialisierungsprozesse einheitlich und konsistent ablaufen. Eine Experte hob hierbei die Rolle von kontextbezogener Speicherung hervor: Hierbei werden Daten innerhalb eines spezifischen Kontextes gespeichert, der beim Laden der Daten erneut verwendet wird. Diese Methode gewährleistet, dass die Daten in ihrem ursprünglichen Zusammenhang interpretiert und verarbeitet werden können, was die Integrität und Konsistenz der gespeicherten Informationen zusätzlich stärkt.

Ferner wurde betont, dass Backups von Speicherdaten von großem Vorteil sind, um die Integrität zu gewährleisten. Diese Backups bieten eine zusätzliche Sicherheitsebene und ermöglichen es, im Falle eines Fehlers oder einer Korruption der Daten schnell auf eine frühere, funktionsfähige Version zurückzugreifen. Dies sei gerade für mobile Plattformen interessant.

Networking und Datenbanken

Ein auffälliger Trend, der sich aus den Interviews abzeichnete, war die Präferenz für selbst entwickelte Lösungen. Insbesondere bei Projekten, die eine enge Integration mit Netzwerken und Datenbanken erfordern, wurde deutlich, dass individuelle Lösungen als unverzichtbar angesehen werden. Dabei ergeben sich zwei Optionen: Entweder war der Experte direkt an der Entwicklung dieser Lösungen beteiligt, oder es wurden bereits bestehende, aber maßgeschneiderte Systeme eingesetzt. Diese Eigenentwicklungen boten den Vorteil, dass sie exakt auf die spezifischen Anforderungen des jeweiligen Projekts zugeschnitten werden konnten. Gleichzeitig wurde aus den

Interviews deutlich, dass solche Implementierungen für das Speichern und Laden eine größere Herausforderung darstellen.

Security

Im Rahmen dieser Arbeit wurde im theoretischen Hintergrund die Deserialisierungs-Vulnerability im Detail analysiert und mögliche Angriffsszenarien sowie Schutzmaßnahmen untersucht. Es wurde aufgezeigt, wie die Deserialisierung in Anwendungen genutzt werden kann, um unberechtigten Zugriff auf Daten oder Systeme zu erlangen. Dieses Thema wurde auch bei den Experteninterviews angeschnitten, wobei verschiedene Lösungsansätze diskutiert wurden. Hierbei ist ein Lösungsansatz, dass der Deserialisierer die Daten überprüft, wobei beispielsweise die Datentypen der Daten geprüft werden. Weichen diese Datentypen von den erwarteten Datentypen ab, kann dies als Indiz für eine mögliche Manipulation gewertet werden.

Besonders im Kontext von kompetitiven Multiplayer-Spielen wurde von den Experten die Sicherheit der Daten als ein zentrales Anliegen hervorgehoben. Im Kontext mit einem Speichersystem ist dies relevant, um den Schutz vor unbefugter Manipulation der Daten zu gewährleisten. Die Implementierung von Verschlüsselung stellt hierbei eine wirksame Maßnahme dar, um die Integrität der gespeicherten Daten zu sichern. Es wurde jedoch darauf hingewiesen, dass die Anwendung von Verschlüsselung die Möglichkeit des Moddings einschränkt. Dies ist potenziell nicht für alle Spiele gewünscht, weswegen es optional ist.

Ein Vorschlag, der im Rahmen der Experteninterviews diskutiert wurde, ist die Definition spezifischer Objekte, die verschlüsselt werden sollen. Diese selektive Verschlüsselung könnte es ermöglichen, sicherheitsrelevante Daten gezielt zu schützen, ohne die Flexibilität des Systems unnötig einzuschränken. Ein weiterer Ansatz betrifft die Vergabe von IDs für das Verschlüsseln von spezifischen Objekten, wie z.B. erhaltbare gegenstände. Dieses Vorgehen ähnelt dem Konzept der Speicherplatzreduktion durch Referenzen im MVC-Ansatz.

In Bezug auf die Priorisierung wurde von den Experten betont, dass zunächst die Funktionalität des Systems über die Sicherheit gestellt werden sollte. Dies impliziert, dass die grundlegende Funktionsweise und Entwicklerfreundlichkeit des Systems sichergestellt sein muss, bevor Sicherheitsmechanismen in vollem Umfang implementiert werden.

Architektur

Im Bereich der Systemarchitektur wurden von den Experten vielfältige Anforderungen, Herausforderungen und Innovationen thematisiert, um eine möglichst skalierbare und anpassungsfähige Lösung zu entwickeln. Als Beispiel wurde die Unreal Engine von einem Experten hervorgehoben. Wie bereits im theoretischen Hintergrund thematisiert, ermöglicht die Engine durch ihre Abstraktion des Codings mittels Visual Scripting eine entwicklerfreundliche Implementierung.

Bezüglich der Speicherung von Daten äußerte ein Experte Bedenken gegenüber einem reinen Command Based Speichersystem. Es wurde jedoch die Möglichkeit einer hybriden Lösung in Betracht gezogen, die die Vorteile eines state-basierten und eines command-basierten Modells kombiniert, um eine effektivere Speicherung zu gewährleisten.

Neben dem hybriden Ansatz wurden weitere Konzepte diskutiert, darunter der Einsatz von Künstlicher Intelligenz, die eigenständig relevante Änderungen im System erkennen und speichern

kann. Ein anderer Ansatz konzentrierte sich auf den Editor, sodass Werte für den Inspektor gespeichert und geladen werden können. Dieser Ansatz ermöglicht es, während der Laufzeit den aktuellen Stand des Systems zu speichern und später wiederherzustellen. Allerdings wurde dieses Konzept bereits in kostenpflichtigen Implementierungen umgesetzt [27]. Eine weiterer Möglichkeit ist die Verwendung vieler kleine JSON-Dateien zur getrennten Speicherung spezifischer Datensätze, was die Modularität des Systems fördert.

3.2 Anforderungserhebung

Die Anforderungsanalyse basiert auf den kategorisierten Erkenntnissen des Experteninterviews und dem theoretischen Hintergrund. Dabei haben die aufgestellten Hypothesen H1, H2 und H3 einen maßgeblichen Einfluss auf die Erhebung der funktionalen und nicht-funktionalen Anforderungen an das Speichersystem. Diese bildet eine essentielle Grundlage für die Auswahl und Bewertung sowohl bestehender Systeme als auch hinsichtlich des in dieser Arbeit entwickelten Systems. Dadurch wird sichergestellt, dass die Anforderungen nicht nur auf theoretischen Grundlagen und Hypothesen basieren, sondern auch eine hohe Praxisrelevanz besitzen.

Die systematische Kategorisierung und Priorisierung der Anforderungen erfolgt in zwei Hauptkategorien: funktionale Anforderungen und nicht-funktionale Anforderungen. Funktionale Anforderungen beziehen sich auf die konkreten Aufgaben und Funktionen, die das System erfüllen muss, während nicht-funktionale Anforderungen Aspekte wie Sicherheit, Leistung, Entwicklerfreundlichkeit und Wartbarkeit umfassen.

Im Nachfolgenden werden die einzelnen Kategorien des Experteninterviews untersucht und ihre Relevanz als Anforderung sowie deren Definition evaluiert.

3.2.1 Speicheroptimierung

Im Rahmen der durchgeführten Experteninterviews hat sich die Speicheroptimierung als ein zentraler und mehrfach diskutierter Aspekt herausgestellt, der für die Entwicklung des Systems von hoher Relevanz ist. Dabei bestätigen die Experteninterviews die Erkenntnisse einer Studie, in der Performance-Tests von verschiedenen Serialisierungsformaten durchgeführt wurden [28].

- **Anforderung**: Das System muss den Speicherbedarf durch den Einsatz effizienter Datenstrukturen, optimierter Algorithmen und geeigneter Kompressionsstrategien minimieren.
- **Testfall**: Um die Einhaltung dieser Anforderung zu überprüfen, wird der Speicherverbrauch anhand eines konkreten Anwendungsfalls eines Inventarsystems gemessen.
- **Validierung**: Die Validierung erfolgt durch die Messung des Speicherbedarfs vor und nach der Anwendung von Kompressionsmethoden sowie durch Belastungstests.

3.2.2 Komplexität der Datenstrukturen

Aufgrund der Hypothese H3 wurde eine Anforderung definiert, die sich auf die Automatisierung der Wiederherstellung von Referenzen fokussiert. Zudem wurde die Komplexität von Datenstrukturen und Datentypen im Rahmen der Experteninterviews mehrfach als wichtig erachtet. Insbesondere wurde die Fähigkeit des Systems diskutiert, ein Objekt entgegenzunehmen und die

zugehörigen Datentypen autonom zu behandeln. Dabei wurde auch das Konzept der speicherbaren Referenzen sowie die damit verbundenen Herausforderungen thematisiert, was darauf hinweist, dass das Speichern von Referenzen signifikante Vorteile bietet. Besonders gestützt wird dies durch die Arbeit von Grochowski, Breiter und Nowak (2019): „A proper complete serialization should follow all references used in the object. Otherwise deserialized object would not be a semantic copy of its source." [10]. Demnach kann eine vollständige Deserialisierung nur unterstützt werden, wenn Referenzen korrekt berücksichtigt werden, was Objekte wie Prefabs und Components von Unity einschließt.

- **Anforderung**: Das System muss in der Lage sein, komplexe und verschachtelte Datenstrukturen eigenständig zu speichern und zu laden. Dies umfasst die Unterstützung sowohl von Reference-Type- und Value-Type Objekten als auch von Asset-Referenzen.
- **Testfall**: Speichern und Laden komplexer Datenstrukturen, einschließlich verschachtelter Objekte und zirkulärer Referenzen.
- **Validierung**: Durchführung einer Integritätsprüfung des Game-States nach dem Laden, um die korrekte und vollständige Wiederherstellung der Daten sicherzustellen.

3.2.3 Performance

Die Performance ist ein wesentlicher Bestandteil des Systems. Es muss mit ausreichender Geschwindigkeit erfolgen, um die Benutzererfahrung nicht zu beeinträchtigen. Dies ist gegeben, wenn keine merklichen Verzögerungen oder Unterbrechungen im Spielfluss auftreten. Um dies zu gewährleisten, muss das Schreiben und Lesen von Dateien auf der Festplatte asynchron erfolgen, wie bereits im Experteninterview diskutiert wurde. Darüber hinaus darf beim Erstellen eines Snapshots während des Speichervorgangs in keinem Frame mehr als 16 ms Rechenzeit beansprucht werden, um eine konstante Bildrate von 60 FPS sicherzustellen. Da die Experten jedoch priorisiert haben, dass der Fokus zunächst auf die Sicherstellung der Grundfunktionalität gelegt wird, ist das Zuweisen der geladenen Daten von geringerer Relevanz, da in diesem Fall ein Ladebildschirm eingesetzt werden kann.

- **Anforderung**: Das Schreiben und Lesen von Dateien auf der Festplatte muss vollständig asynchron ablaufen, und die Erstellung von Snapshots darf pro Frame nicht mehr als 16 ms Rechenzeit in Anspruch nehmen.
- **Testfall**: Zur detaillierten Analyse der Performance wird die Zeit gemessen, die für das Speichern und Laden unterschiedlicher Spielstände benötigt wird. Diese Messungen werden in einer finalen, mit Unity erstellten Anwendung durchgeführt, indem die Rechenzeit des Speichersystems erfasst wird.
- **Validierung**: Zum einen müssen die gemessenen Zeiten mit den in den Anforderungen festgelegten Zeitrahmen übereinstimmen. Zum anderen ist sicherzustellen, dass asynchrone Prozesse eingesetzt werden, um eine optimale Performance zu gewährleisten.

3.2.4 Versioning

Das Thema Versioning wurde sowohl als bedeutende Herausforderung als auch als ein wichtiges Anliegen identifiziert. In Bezug auf die Priorisierung wurde jedoch deutlich, dass die Grundfunktionalität des Systems zunächst im Vordergrund steht.

- **Anforderung**: Das System muss abwärtskompatibel sein, sodass Spielstände, die in früheren Versionen des Spiels erstellt wurden, in neueren Versionen korrekt geladen werden können.

- **Testfall**: Das System wird auf die Fähigkeit überprüft, Spielstände zu laden, die in früheren Versionen gespeichert wurden.

- **Validierung**: Durchführung einer Integritätsprüfung für das Importieren von Spielständen mit älteren Versionen in eine neuere Version.

3.2.5 Plattformübergreifende Unterstützung

Obwohl die plattformübergreifende Unterstützung aufgrund des starken Platform-Supports von Unity als relevant erachtet wird, wurde dieses Thema in den Interviews nur von zwei Experten angesprochen. Aus dem theoretischen Hintergrund geht hervor, dass JSON durch seine plattformunabhängige, textbasierte Struktur auf verschiedenen Systemen funktioniert, während binäre Formate zwar effizienter, aber aufgrund der unterschiedlichen Endianness und plattformspezifischen Anpassungen komplexer zu implementieren sind.

- **Anforderung**: Das System muss plattformübergreifend funktionieren und in der Lage sein, Spielstände auf verschiedenen Plattformen wie Windows, Mac, Linux, iOS und Android zu speichern und zu laden.

- **Testfall**: Der Testfall beinhaltet das Speichern eines Spielstands auf einer Plattform und das anschließende Laden dieses Spielstands auf einer anderen Plattform (Windows, Mac, Linux, iOS, Android).

- **Validierung**: Die Validierung erfolgt durch einen erfolgreichen Integritätstest, bei dem derselbe Game-State auf verschiedenen Plattformen gespeichert wird.

3.2.6 Fehlerbehandlung und Datenkonsistenz

Die Themen Fehlerbehandlung und Datenkonsistenz lassen sich in zwei separate Anforderungen unterteilen. Die Berücksichtigung dieser Anforderungen ist entscheidend für die Stabilität und Zuverlässigkeit des Speichersystemes.

Fehlerbehandlung

- **Anforderung**: Das System muss in der Lage sein, den Entwickler auf auftretende Fehler aufmerksam zu machen und gleichzeitig robust gegenüber Fehlern und Abstürzen zu reagieren.

- **Testfall**: Überprüfung des Systemverhaltens bei Fehlern während des Speicherns und Ladens, wie beispielsweise bei plötzlichem Abbruch oder Datenkorruption.

- **Validierung**: Durch gezieltes Einführen von Fehlern werden die Fehlermeldungen sowie die Sicherheitsmaßnahmen untersucht.

Datenkonsistenz

- **Anforderung**: Das System muss sicherstellen, dass die gespeicherten Daten konsistent und vollständig sind.
- **Testfall**: Speichern und Laden eines einfachen Spielstands mit unterschiedlichen Datensätzen.
- **Validierung**: Durchführung einer Integritätsprüfung der geladenen Daten.

3.2.7 Networking und Datenbanken

In den Experteninterviews wurden Networking und Datenbanken wiederholt als zentrale Themen angesprochen. Trotz dieser häufigen Nennung wird dies jedoch nicht als Anforderung definiert. Der Hauptgrund hierfür liegt in der Tatsache, dass die damit verbundenen Herausforderungen eine hohe Spezifität und Komplexität aufweisen, die über den Rahmen dieser Arbeit hinausläuft. Stattdessen fokussiert sich die Analyse auf übergeordnete Themenbereiche, um eine breitere Anwendbarkeit und Relevanz zu gewährleisten. Dies verhindert, dass die Betrachtung in zu spezialisierte technische Details abgleitet, die möglicherweise nicht für alle Anwendungsfälle gleichermaßen von Bedeutung sind.

3.2.8 Security

Trotz der Entscheidung, Networking- und Datenbankanbindungen nicht als Teil der Anforderungen zu definieren, bleibt die Sicherheit des Systems von hoher Bedeutung, insbesondere aufgrund potenzieller Deserialisierungs-Schwachstellen. Aus dem theoretischen Hintergrund geht hervor, dass unsichere Deserialisierung zu Sicherheitsrisiken wie Remote Code Execution (RCE) führen kann. Dies ist besondes in Kombination mit der Anforderung *Komplexität der Datenstrukturen* Problematisch, wenn Informationen über Objekttypen gespeichert werden. Die OWASP empfielt hierbei den Einsatz sicherer Praktiken, wie die Validierung und Signierung von Daten während des Serialisierungsprozesses sowie die Verwendung bewährter Formate wie JSON. Dennoch wurde von den Experten die allgemeine Funktionalität und Entwicklerfreundlichkeit zunächst von höherer Priorität eingestuft.

- **Anforderung**: Das System muss sicherstellen, dass gespeicherte Daten vor Manipulation geschützt sind.
- **Testfall**: Überprüfung der Sicherheitsmechanismen, die gespeicherte Daten vor unbefugter Manipulation schützen.
- **Validierung**: Durchführung einer manuellen Modifikation der gespeicherten Daten und anschließender Versuch, diese Daten zu laden, um zu prüfen, ob die Manipulation erkannt und wirksam verhindert wird.

3.2.9 Architektur

Aus den Experteninterviews geht hervor, dass das System skalierbar sein muss. In Verbindung mit dem Ziel dieser Arbeit, ein entwicklerfreundliches System zu entwickeln, ergeben sich daraus spezifische Anforderungen, die die Systemarchitektur betreffen.

Entwicklerfreundliche API

Auf Grundlage der Forschungsfrage wurden Hypothesen zur entwicklerfreundlichen Implementierung von Speicher- und Ladefunktionalitäten aufgestellt. Zur Überprüfung dieser Hypothesen werden im Rahmen dieser Anforderung die Hypothesen H1 und H2 adressiert.

- **Anforderung**: Das Speichersystem soll eine intuitive API bereitstellen, die es Entwicklern ermöglicht, Daten mit minimalem Aufwand zu speichern und zu laden. Dabei sollen die Konzepte aus den Hypothesen H1 und H2 implementiert werden.
- **Validierung**: Die Entwicklerfreundlichkeit der API wird durch Nutzer-Tests überprüft, bei denen die Handhabung der API evaluiert und durch Reviews der Testpersonen bewertet werden.

Dokumentation

- **Anforderung**: Das System muss umfassend dokumentiert sein, um die Wartung und Weiterentwicklung zu erleichtern.
- **Validierung**: Die Qualität der Dokumentation wird durch Nutzer-Tests überprüft, bei denen die Entwicklerfreundlichkeit und Vollständigkeit der Dokumentation bewertet werden.

3.2.10 Anforderungstabelle

Im nachfolgenden sind alle Anforderungen in einer Tabelle für funktionale und nicht funktionale Anforderungen gelistet.

Funktionale Anforderungen

Funktionale Anforderungen beschreiben, was ein System tun muss, d.h., sie spezifizieren die Hauptfunktionen und Aufgaben, die das System erfüllen soll. Sie sind meist benutzerzentriert und definieren spezifische Verhaltensweisen oder Leistungen, die vom System erbracht werden sollen.

Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen beschreiben wie gut das System diese Aufgaben erfüllt. Diese Anforderungen sind nicht auf die Hauptfunktionen des Systems fokussiert, sondern eher auf seine Qualität und Effizienz.

3.3 Analyse existierender Frameworks

Ausgehend von den im Rahmen der Anforderungsanalyse ermittelten Kriterien zielt diese Methode darauf ab, bestehende Systeme systematisch zu evaluieren. Der Vergleich dieser Systeme mit den festgelegten Anforderungen liefert wichtige Erkenntnisse für die Weiterentwicklung des neuen Frameworks. Dabei dienen die in diesem Kapitel analysierten Good Practices sowie aufgedeckten Lücken als Grundlage. Diese Methode leistet somit einen wesentlichen Beitrag zur Entwicklung eines Frameworks, das die definierten Anforderungen erfüllt.

Funktionale Anforderungen	Beschreibung
Entwicklerfreundliche API	Das Speichersystem soll eine intuitive API bereitstellen, die es Entwicklern ermöglicht, Daten mit minimalem Aufwand zu speichern und zu laden. Dabei sollen die Konzepte aus den Hypothesen H1 und H2 implementiert werden.
Datenkonsistenz	Das System muss sicherstellen, dass die gespeicherten Daten konsistent und vollständig sind.
Komplexität der Datenstrukturen	Das System muss in der Lage sein, komplexe und verschachtelte Datenstrukturen eigenständig zu speichern und zu laden. Dies umfasst die Unterstützung sowohl von Reference-Type- und Value-Type Objekten als auch von Asset-Referenzen.
Versioning	Das System muss abwärtskompatibel sein, sodass Spielstände, die in früheren Versionen des Spiels erstellt wurden, in neueren Versionen korrekt geladen werden können.
Plattformübergreifende Unterstützung	Das System muss plattformübergreifend funktionieren und in der Lage sein, Spielstände auf verschiedenen Plattformen wie Windows, Mac, Linux, iOS und Android zu speichern und zu laden.
Fehlerbehandlung	Das System muss in der Lage sein, den Entwickler auf auftretende Fehler aufmerksam zu machen und gleichzeitig robust gegenüber Fehlern und Abstürzen zu reagieren.

Tabelle 3.1: Funktionale Anforderungen

Nicht-funktionale Anforderungen	Beschreibung
Dokumentation	Das System muss umfassend dokumentiert sein, um die Wartung und Weiterentwicklung zu erleichtern.
Performance	Das Schreiben und Lesen von Dateien auf der Festplatte muss vollständig asynchron ablaufen, und die Erstellung von Snapshots darf pro Frame nicht mehr als 16 ms Rechenzeit in Anspruch nehmen.
Speicheroptimierung	Das System muss den Speicherbedarf durch den Einsatz effizienter Datenstrukturen, optimierter Algorithmen und geeigneter Kompressionsstrategien minimieren.
Security	Das System muss sicherstellen, dass die gespeicherten Daten vor Manipulation geschützt sind.

Tabelle 3.2: Nicht-funktionale Anforderungen

3.3.1 Auswahl der Frameworks

Bei der Auswahl zur Analyse der Speichersysteme wurde eine repräsentative Auswahl getroffen, um verschiedene Ansätze und Technologien für Unity abzudecken. Es wurden sowohl fünf kostenlose als auch zwei kostenpflichtige Frameworks gewählt, wobei bei letzterem als Auswahlkriterium eine umfangreiche Dokumentation vorhanden sein muss. Die Frameworks wurden entweder aus dem Unity Asset Store oder von öffentlichen GitHub Projekten gewählt. Projekte, die veraltet sind oder auf überholten Technologien wie dem BinaryFormatter basieren, wurden ausgeschlossen. Ein Projekt gilt in diesem Kontext als veraltet, wenn es seit mindestens vier Jahren nicht mehr gewartet wurde. Die kostenlosen Frameworks wurden durch jakobdufault [12], DerKekser [4], coryleach [3], Clayton Industries [11] und AlexMeesters [2] entwickelt. Die kostenpflichtigen Frameworks durch SteveSmith.Software [26] und [16].

3.3.2 Identifikation von Good Practices

Die Analyse bestehender Speichersysteme hat eine Reihe von Good Practices hervorgebracht, die in unterschiedlichen Bereichen der Softwarearchitektur und -sicherheit relevant sind.

Datenkompression und Verschlüsselung

Eine häufig verwendete Good Practice ist die Anwendung von Gzip-Kompression zur effizienten Speicherung von Daten. Darüber hinaus setzen viele Frameworks auf AES-Verschlüsselung, um die Sicherheit gespeicherter Daten zu gewährleisten. Die Schlüssel für die Verschlüsselung werden in der Regel über die API bei den Methoden zum Speichern und Laden übergeben. Einige Tools bieten die Möglichkeit, diese Schlüssel im Code oder in einem ScriptableObject zu speichern, wobei ein Framework empfiehlt, die Schlüssel zur Verbesserung der Sicherheit von einem Server abzurufen [16].

Plattformunterstützung

Alle analysierten Systeme unterstützen die Plattformen, welche Unity bedienen möchte. Eine Ausnahme stellt ein Framework dar, das zusätzlich die Integration mit Datenbanken ermöglicht, wodurch jedoch die Plattformunterstützung eingeschränkt ist. Diese Integration wird allerdings nur in der kostenpflichtigen Variante des Frameworks angeboten [2].

Unterstützung von Datentypen

Die Frameworks zeigen eine ausgewogene Unterstützung von Datentypen. Hinsichtlich der aufgestellten Hypothese H3 versuchen einige Frameworks, möglichst viele Typen zu unterstützen, was insbesondere durch die Verwendung von Referenzen und Reflection gut gelingt. Hierbei werden meist auch Assets als Referenzen behandelt. Andere Frameworks beschränken sich auf eine spezifische Auswahl unterstützter Typen. Ein auffälliger Trend ist, dass die kostenpflichtigen Frameworks auch die Speicherung von Referenzen unterstützen. Hierzu wurde in der Doku eines Frameworks angegeben, dass im Hintergrund *Global Unique identificators* (GUID) vergeben werden, um Referenzen zu verwalten [16].

Sicherstellung der Datenkonsistenz

Viele der analysierten Frameworks beinhalten JUnit-Tests, um die Konsistenz der gespeicherten Daten sicherzustellen. Darüber hinaus unterstützen vier Tools das Speichern und Laden von

aktiven und instanziierten Prefab-Instanzen. Dazu gehören die beiden kostenpflichtigen Assets [16] [26], während die zwei kostenfreien Assets das Feature ohne die Möglichkeit des Speicherns von Referenzen bieten [4] [2].

Architekturansätze

Die Architektur der Speichersysteme variiert, wobei die meisten Tools eine statische Klasse als Kommunikationsmittel verwenden, die für das Speichern und Laden von Daten in einem bestimmten Ordner zuständig ist. Ein weiteres Framework geht über diesen grundlegenden Ansatz hinaus und bietet erweiterte Funktionen wie die Durchführung von Speicher- und Ladeoperationen bei bestimmten Ereignissen, die Verwendung von *Save Slots* oder die Möglichkeit des Design Patterns Observer [2], bei dem Events zu einer Klasse hinzugefügt werden können, welche zu einen definierbaren Zeitpunkt von dieser Klasse ausgelößt wird. Diese erweiterte Funktionalität reduziert den Implementierungsaufwand für Entwickler, erfordert jedoch ein tieferes Verständnis des Tools.

Im Rahmen der Analyse der Frameworks wurde deutlich, dass die definierten Hypothesen bereits als Konzepte verwendet werden, was ihre Relevanz unterstreicht. Zwei der Frameworks nutzen das in der Hypothese H2 genannte Konzept des Component-Savings, bei dem Entwickler ein Save-Component schreiben, welcher konkret für das Speichern für definierbare Daten zuständig ist. Damit dieses Savable-Component registriert wird, muss dasselbe GameObject einen Registrator-Component enthalten, welches die Savable-Components auf demselben GameObject registriert. Ein Framework verwendet für die Erstellung von Savable-Components Interfaces [2] und das andere benutzt Attribute [4]. Mit den Attributen können somit nicht nur Methoden für das Speichern markiert werden, sondern auch einzelne Fields und Properties, welches Hypothese H1 stützt. Dieser modulare Aufbau fördert die Kapselung von Speicher- und Ladeprozessen und macht das System zudem erweiterbar. Es lässt sich zudem leicht in bestehende Unity-Projekte integrieren, ohne dass umfangreiche Anpassungen erforderlich sind. Zudem können Unity-Components nicht direkt mit zusätzlichem Code erweitert werden, was durch den Einsatz von Component-Saving umgangen wird.

Eine besondere Ausnahme bietet ein kostenpflichtiges Framework, welches direkt im Unity Inspektor Fields und Properties für Klassen anzeigt, welche speicherbar sein sollen. Dadurch können gezielt Daten für das Speichern ohne Programmieraufwand integriert werden. Dies ist besonders für Designer der Anwendung relevant, die keine oder wenig Programmiererfahrung haben [16].

3.3.3 Lücken und Verbesserungspotenzial

Im Rahmen der Analyse wurden mehrere Lücken und Verbesserungspotenziale identifiziert, die für die spätere Entwicklung eines Speichersystems relevant sind. Hierbei muss stark zwischen kostenpflichtigen und kostenlosen Frameworks unterschieden werden, damit die spezifischen Stärken und Schwächen beider Kategorien klar hervortreten und in die Entwicklung eines neuen Frameworks einfließen kann.

Versionierung

Ein zentrales Defizit aller untersuchten Frameworks ist das fehlende Versioning. Keines der Tools unterstützt die Möglichkeit, ältere Spielstände in neuen Versionen des Spiels zu laden, was eine

wesentliche Anforderung für langfristige Projekte darstellt.

Lücken bei den Kostenpflichtige Frameworks

Die beiden analysierten kostenpflichtigen Frameworks weisen zwar einige fortschrittliche Funktionen auf, zeigen jedoch auch Schwächen in spezifischen Bereichen. Beide Frameworks verzichten auf ein komponentenbasiertes System und bieten stattdessen eine grundlegende API, die eine direkte Interaktion mit Speicher- und Ladefunktionen ermöglicht. Dieser Ansatz erleichtert zwar die Implementierung und bietet eine hohe Generalisierbarkeit, beeinträchtigt jedoch die Modularität und Flexibilität, die ein komponentenbasiertes System bieten könnte.

Ein weiteres Problem zeigt sich bei der Verwaltung von Referenzen in einem Framework. Easy Save [16] organisiert Referenzen durch einen Referenzmanager. Allerdings kann es in Szenarien mit sehr komplexen Abhängigkeiten oder in Fällen, in denen die Reihenfolge des Ladens entscheidend ist, zu Schwierigkeiten kommen. In manchen Fällen erfordert das Auflösen fehlender Referenzen mehrere Lade-Durchgänge, was den Prozess verkompliziert [16].

Das andere Framework *Save for Unity Complete* [26] verwendet hingegen ein proprietäres binäres Speicherformat, das speziell für dieses Framework entwickelt wurde. Änderungen nach der Erstellung der Speicherdatei invalidieren die Integrität der Speicherdatei, was die Fehlersuche und die Erweiterung des Systems erschwert. Gleichzeitig bietet dieses System keine integrierte Verschlüsselung, sodass diese von den Entwicklern manuell implementiert werden muss. Für diese Implementierung wurden jedoch bereits Vorbereitungen getroffen, die den Prozess erleichtern sollen.

Lücken bei den Kostenfreie Frameworks

Die kostenfreien Systeme priorisieren Einfachheit, was ihre Fähigkeit einschränkt, komplexe Szenarien, insbesondere solche im Zusammenhang mit Referenzen, effektiv zu bewältigen. Darüber hinaus verfügen einige dieser Systeme über eine unzureichende Dokumentation, was für Entwickler, die auf fortgeschrittene Funktionen angewiesen sind oder Unterstützung benötigen, eine erhebliche Herausforderung darstellen kann. Dies kann durch eine Erweiterung der unterstützten Datentypen mit guter Dokumentation verbessert werden.

Ein architektonisches Problem zeigt sich auch bei dem Framework, welches Component-Saving mit Attributen verwendet [4]. Während ein Interface eine Methode für das Speichern und eine für das Laden definieren kann, erfordert der Einsatz von Attributen, dass zwei separate Methoden entsprechend markiert werden. Fehlt ein Attribut für eine der Methoden, kann dies zu Problemen bei der Konsistenz der Serialisierung und Deserialisierung führen. Diese Inkonsistenz stellt eine potenzielle Fehlerquelle dar, die die Integrität der Daten gefährdet und die Zuverlässigkeit des Systems beeinträchtigen kann. Darüber hinaus wird durch diese Herangehensweise erwartet, dass der Vorteil der höheren Performance durch den Verzicht auf Reflection verloren geht.

Kapitel 4

Design und Implementierung eines neuen Speichersystems

Die Entwicklung des eigenen Speichersystems basiert auf den Erkenntnissen aus dem vorangegangenen theoretischen Hintergrund, der Anforderungsanalyse und der abschließenden Systemanalyse. Ziel dieser Herangehensweise war es, die in der Analyse identifizierten Lücken und bewährten Methoden zu identifizieren. Diese Erkenntnisse wurden direkt in die Entwicklung eines neuen Frameworks einbezogen, um praxisnah den spezifischen Anforderungen gerecht zu werden. Dabei wurde die Architektur des entwickelten Speichersystems stark von den aufgestellten Hypothesen beeinfluss.

Im Rahmen dieser Arbeit wurde das Speichersystem mittels Unity und C# implementiert. Der erstellte Code ist in einem öffentlichen Repository zugänglich, das im Anhang .1 verlinkt ist. Neben dem Speichersystem wurde als Beispielanwendungsfall ein Inventarsystem entwickelt, welches zur Veranschaulichung dient und gleichzeitig Aufschluss über die Performance des Systems bietet. Dies ermöglicht es, die Funktionsfähigkeit und Effizienz des Speichersystems unter realen Bedingungen zu bewerten und zu demonstrieren.

Die Struktur dieses Kapitels orientiert sich an den definierten Anforderungen. Zunächst wird auf die Umsetzung der entwicklerfreundlichen API eingegangen, um ein umfassendes Verständnis für den Aufbau und die Funktionsweise des Systems zu ermöglichen. Da die Dokumentation erst im Rahmen der User-Tests relevant ist, wird diese Anforderung erst in der Auswertung behandelt. Im Fokus stehen dabei einerseits die wichtigsten Bestandteile des Systems und andererseits die Hypothesen H1 und H2. In späteren Kapiteln wird die Implementierung der übrigen Anforderungen detailliert ausgeführt.

4.1 Grundkomponenten

Zunächst werden in diesem Kapitel die wichtigsten verschiedenen Bestandteile präsentiert. Diese sind einerseits Components auf GameObjects, welche sich auf eine gesammte Szene oder ein einziges Objekt beziehen und andererseits ScriptableObjects, die als Manager fungieren.

4.1.1 SaveLoadManager

Der SaveLoadManager fungiert als Kern des Speichersystems und ist als ScriptableObject implementiert. Seine Hauptaufgabe besteht darin, globale Einstellungen für das Speichern und Laden von Daten zu verwalten und den Speicherprozess für alle aktiven Szenen oder eine Auswahl von Szenen zu steuern. Er wurde als ScriptableObject implementiert, damit Entwickler verschiedene Einstellungen für das Speichern und Laden erstellen können. Zu den globalen Einstellungen zählen die Versionsnummer, die Konfiguration des Standardnamens und Dateityps, die Art des Integritätscheck, Datenkompression und Datenverschlüsselung.

Der SaveLoadManager führt Speicheraktionen auf der Grundlage einer aktuellen Speicherdatei aus, in die aktiv geschrieben wird. Diese Speicherdatei kann explizit über Methoden des Systems festgelegt werden oder automatisch verwaltet sein, wenn Speicher- und Ladefunktionen genutzt werden. Wird kein spezifischer Dateiname angegeben, verwendet das System einen vordefinierten Standarddateinamen. Das Speichersystem erstellt für jeden Speichervorgang zwei Dateien: eine Meta-Datei und die eigentliche Save-Datei mit den gespeicherten Daten. Die Meta-Datei enthält Metadaten wie das Erstellungsdatum des Speicherstands, die Version der Speicherdatei und einer checksum, um die Integrität der gespeicherten Daten zu überprüfen. Darüber hinaus können Entwickler benutzerdefinierte Daten zur Meta-Datei hinzufügen, was eine detailliertere und spezifischere Verfolgung von Speicherständen ermöglicht.

Neben dem standardmäßigen Speichern und Laden bietet das System erweiterte Funktionen wie das Snapshotting von Szenen. Dabei wird der aktuelle State einer Szene im Cache zwischengespeichert, ohne dass die Daten vollständig auf die Festplatte geschrieben werden. In Kombination mit diesem Cache können weitere Funktionen genutzt werden, wie das Laden des Cache-Inhalts in eine Szene oder das Löschen von Daten einer Szene im Cache. Um diese Caches auf der Festplatte zu speichern und wieder zu laden, stehen entsprechende Methoden zur Verfügung. Für komplexere Szenarien gibt es eine Methode, mit der eine Szene neu geladen und anschließend die Speicherdatei geladen wird. Dies stellt sicher, dass die Szene auf ihren ursprünglichen State zurückgesetzt wird, bevor die gespeicherten Daten angewendet werden. Diese Funktion ist besonders nützlich, wenn das ursprüngliche Setup der Szene wiederhergestellt werden muss, bevor spezifische Änderungen aus einem Speicherstand geladen werden.

Darüber hinaus bietet das System verschiedene Events, in die Entwickler ihre Methoden einbinden können. Diese Events ermöglichen es, benutzerdefinierte Logik zu bestimmten Zeitpunkten des Speicher- und Ladeprozesses auszuführen, beispielsweise vor oder nach dem Speichern und Laden einer Szene. Dies ist besonders hilfreich, um nach dem Laden einer Szene die Benutzeroberfläche zu aktualisieren.

4.1.2 SaveSceneManager Component

Der *SaveSceneManager* ist eine Component, die speziell für die Verwaltung von Speicherprozessen innerhalb einzelner Szenen verantwortlich ist. Jede Szene sollte genau einen *SaveSceneManager* enthalten, um ihren State unabhängig zu verwalten und sicherzustellen, dass szenenspezifische Daten korrekt gespeichert und wiederhergestellt werden.

4.1.3 Asset- und Prefab-Registry

Die Asset- und Prefab-Registry ist ein zentraler Bestandteil des Speichersystems, der alle zu persistierenden Assets mithilfe der UnityEngine Serialisierung speichert, sodass sie später über eine GUID abgefragt werden können. Dies ist erforderlich, um sicherzustellen, dass diese Assets serialisiert werden können.

Die Prefab-Registry ist ein ScriptableObject, das automatisch alle Prefabs erfasst, die mit dem Savable-Component markiert sind. Falls diese Registry noch nicht existiert, wird sie automatisch erstellt. Sie wird für das *Dynamic Prefab Loading* verwendet, indem das System basierend auf dem gespeicherten State notwendige Prefabs instanziert und nicht benötigte Prefabs entfernt. Das Dynamic Prefab Loading wird in einem späteren Abschnitt ausführlich behandelt.

Die Asset-Registry hingegen verwaltet und verfolgt alle zu persistierenden Assets des Systems. Sie speichert Referenzen zu Ressourcen wie Texturen, Materialien und anderen nicht-Prefab Assets, um sicherzustellen, dass diese beim Wiederherstellen eines Speicherstands korrekt geladen werden. Durch die zentrale Verwaltung dieser Assets kann das System Abhängigkeiten effizient organisieren und das Risiko von fehlenden Ressourcen beim Laden minimieren.

Sowohl die Asset-Registry als auch die Prefab-Registry identifizieren Objekte anhand ihres Typs. Entwickler müssen sicherstellen, dass die Typen der als speicherbar markierten Objekte mit dem Typ des jeweiligen Assets in der entsprechenden Registry übereinstimmen. Beispielsweise muss ein Field, das ein Prefab speichert, vom Typ Savable sein, da die Prefab-Registry nur Prefabs identifizieren kann, die diesem Typ entsprechen.

4.1.4 Savable-Component

Neben den serialisierbaren Assets durch die Registrys müssen auch Objekte innerhalb der Szene serialisierbar sein. Hierfür ist das Savable Component zuständig, da sie eindeutige IDs den GameObjects und Transforms innerhalb einer Szene zuweist. Zudem erkennt dieses Component automatisch Skripte, welche das Attribute-Saving oder das Component-Saving implementieren und weist auch diesen eine GUID zu. Um zu überprüfen, welche Objekte eines GameObject für das Speichern und Laden verfolgt werden, können Entwickler die aktuelle *Savable List* auf der Savable Component einsehen.
Ferner bietet das Savable-Component ein Field namens *Save References*, das es Entwicklern ermöglicht, beliebigen Objekten GUIDs zuzuweisen. Diese Funktion ist in erster Linie für Components auf dem GameObject vorgesehen, das die Savable Component enthält, kann jedoch für sämtliche Unity-Objects verwendet werden.

4.2 H1: Attribute-Saving

Um die Hypothese H1 zu testen, wurde das Konzept des Attribute-Saving entworfen, das durch die Nutzung von Attributen zur Markierung von Objekten die Entwicklerfreundlichkeit erhöhen soll. Diese Hypothese gewinnt zusätzlich an Bedeutung durch das bereits implementierte Speichersystem von DerKekser, das ebenfalls das Konzept des Attribute-Saving verwendet [4]. Die Markierung von Objekten durch Attribute stellt einen effizienten, flexiblen und klar strukturierten Ansatz zur Realisierung von Speichern und Laden dar. Es gibt zwei Hauptattribute, die Entwickler verwenden können, um Daten für das Speichern und Laden zu markieren.

- **[Savable]**: Das *Savable* Attribut kann auf einzelne *Fields* oder *Properties* angewendet werden, um sie für die Speicherung zu kennzeichnen. Dadurch können Entwickler gezielt festlegen, welche Daten gespeichert werden sollen, ohne die gesamte Klasse für das Speichern zu markieren.
- **[SavableObject]**: Wenn das *SavableObject* Attribut auf eine Klasse angewendet wird, werden automatisch alle Fields und Properties innerhalb dieser Klasse für die Speicherung markiert. Dieses Attribut ist besonders für Data Transfer Objects (DTOs) geeignet, die dazu dienen, Daten gekapselt zwischen verschiedenen Schichten einer Anwendung zu transportieren, ohne *Business Logic* zu enthalten. Entwickler können hierbei den Parameter *declaredOnly* verwenden, um festzulegen, ob nur die Fields und Properties der deklarierten Klasse berücksichtigt werden sollen. Dies ist insbesondere für MonoBehaviour-Klassen nützlich, da diese nicht direkt serialisiert werden können.

Die Attribute [Savable] und [SavableObject] gelten für alle Zugriffsmodifikatoren, einschließlich *public-*, *private-* und *static* Fields und Properties. Dies bietet eine hohe Flexibilität bei der Festlegung, welche Daten gespeichert werden sollen. Ein Beispiel der Nutzung ist in Abbildung 4.1 dargestellt.

Zusätzlich können diese Attribute zusammen mit dem Component-Saving verwendet werden. Wenn sowohl Attribute-Saving als auch das Component-Saving in einer Klasse verwendet werden, ist es allerdings möglich, denselben Wert zweimal zu speichern und zu laden.

4.3 H2: Component-Saving

Um einen modulareren und flexibleren Ansatz für das Speichern zu ermöglichen, wurde zudem das in Hypothese H2 genannte Component-Saving implementiert. Diese Flexibilität führt zu weiteren Vorteilen: Unity-Components können nicht direkt mit zusätzlichem Code erweitert werden, wodurch Attribute-Saving nicht verwendbar ist. Um diese Einschränkung zu umgehen, bietet das Component-Saving eine robuste Lösung, um diese Components speicherbar zu machen. Zudem ermöglicht das Component-Saving eine erweiterte Kontrolle im Vergleich zur den Attributes. Dies ist besonders dann nützlich, wenn spezielle Logik oder zusätzliche Schritte erforderlich sind, um den State eines Objekts korrekt zu speichern oder wiederherzustellen. Das Component-Saving dieses Frameworks orientiert sich an der architektonischen Herangehensweise des Frameworks von Alexmeesters [2]. Demnach wird ein Interface eingesetzt, welches für alle Klassen verwendet werden kann.

Um ein Unity-Component speicherbar zu machen, muss das ISavable-Interface in einer Klasse definiert und die Methoden 'OnSave()' und 'OnLoad()' Implementiert werden. In diesen Methoden kann die gewünschte Speicher- und Ladefunktionalität festgelegt werden. Dies erfolgt unter Verwendung des bereitgestellten SaveDataHandler und LoadDataHandler. Für das Speichern und Laden gibt es zwei verschiedene Herangehensweisen: *Value-Saving* und *Reference-Saving*. Beim Value-Saving wird versucht, die Daten direkt in ein serialisierbares Format zu konvertieren, während beim Reference-Saving die Objekte durch eine GUID serialisiert werden. Diese beiden Konzepte werden in einem späteren Kapitel näher definiert.

Das Value-Saving wird mit den Methoden 'SaveAsValue()' und 'LoadValue<T>()' durchgeführt. Das Reference-Saving wird durch die Methoden 'TrySaveAsReferencable()' und 'TryLoadReferencable()' durchgeführt. Allerdings erfordert das Reference-Saving im Vergleich zum Value-Saving

```
// This class demonstrates the use of Savable attributes within a MonoBehaviour.
No asset usages
public class SaveAttributeExample : MonoBehaviour
{
    // The 'position' field is marked as [SerializeField] to be editable in the Unity Inspector
    // and [Savable] to be included in the save process.
    [SerializeField, Savable]
    private Vector3 position;  Serializable

    // The 'ExampleObject' property is marked with [Savable] to ensure it is included with save and loading.
    [Savable]
    private ExampleDataTransferObject ExampleObject { get; set; }
}

// This class is marked as a savable object using the [SavableObject] attribute.
// All fields and properties within this class will be automatically marked for saving.
[Serializable, SavableObject]
1 usage
public class ExampleDataTransferObject
{
    // These public fields will be saved and loaded automatically due to the SavableObject attribute.
    public string Name;  Serializable
    public int Health;  Serializable

    // Since ExampleObject is a custom class, it needs to be instantiated during loading.
    // This is done using Activator.CreateInstance(), which requires a parameterless constructor.
    // The parameterless constructor allows the object to be instantiated without providing arguments.
    public ExampleDataTransferObject() {}

    public ExampleDataTransferObject(string name, int health)
    {
        Name = name;
        Health = health;
    }
}
```

Abbildung 4.1: Attribute-Saving Example

einen zusätzlichen Schritt: Intern wird zunächst sichergestellt, dass alle Objekte existieren. Erst im Anschluss werden die Referenzen zugewiesen. Das bedeutet, dass das Laden von Referenzen zunächst nur eine GUID zurückgibt. Um diese Pfade in das gewünschte Objekt umzuwandeln sobald alle Objekte erstellt sind, bietet der LoadDataHandler die Methode 'EnqueueReferenceBuilding()' an. Diese Methode ermöglicht es, im Rahmen einer Queue die Umwandlung einer oder mehrerer GUIDs in das tatsächliche Objekt vorzunehmen. Der erste Parameter der 'EnqueueReferenceBuilding()' Methode enthält die GUIDs, während der zweite Parameter eine *Action* definiert, die aufgerufen wird, sobald ein passendes Objekt zu einer GUID gefunden wird. Dieses Objekt wird als Parameter an die Action übergeben und dient der eigentlichen Zuweisung zum gewünschten Objekt beim Loading. Dies wird am Beispiel der SaveParent-Component in Abbildung 4.2 veranschaulicht, welches sicher stellt, dass Parent-Child-Hierarchien gespeichert und geladen werden können.

```
2 asset usages  Robin Jaspers *
public class SaveParent : MonoBehaviour, ISavable
{
    // Method called during the save process to store relevant data.
    0+1 usages  Robin Jaspers *
    public void OnSave(SaveDataHandler saveDataHandler)
    {
        // Check if the current object has a parent and if the parent was successfully added to the saveDataHandler as a referencable object.
        if (transform.parent == null || !saveDataHandler.TrySaveAsReferencable(uniqueIdentifier:"parent", transform.parent))
        {
            Debug.LogWarning($"The {nameof(Savable)} object {name} needs a parent with a {typeof(Savable)} component to support Save Parenting!");
            return;
        }

        // Save the sibling index of the current transform. This helps maintain the order of the object in the hierarchy.
        saveDataHandler.SaveAsValue(uniqueIdentifier:"siblingIndex", transform.GetSiblingIndex());
    }

    // Method called during the load process to restore data and references.
    0+1 usages  Robin Jaspers *
    public void OnLoad(LoadDataHandler loadDataHandler)
    {
        // Attempt to retrieve the parent reference from the loadDataHandler.
        if (!loadDataHandler.TryLoadReferencable(identifier:"parent", out GuidPath parent)) return;

        // Retrieve the sibling index stored during the save process.
        var siblingIndex = loadDataHandler.LoadValue<int>(identifier:"siblingIndex");

        // Enqueue a reference-building action to set the parent and sibling index once the parent reference is resolved.
        loadDataHandler.EnqueueReferenceBuilding(parent, onReferenceFound:foundObject =>
        {
            // Cast the found object to Transform and set it as the parent of the current object.
            transform.parent = (Transform)foundObject;

            // Set the sibling index to maintain the order in the hierarchy.
            transform.SetSiblingIndex(siblingIndex);
        });
    }
}
```

Abbildung 4.2: Beispiel des Component-Savings

Existierende Implementierungen

Angenommen, das Child eines GameObject verfügt über ein Savable-Component mit einem SaveParent-Component, die es ermöglicht, den Parents zu speichern und nach dem Laden wiederherzustellen. Um sicherzustellen, dass das Child den ursprünglichen parent korrekt identifizieren und die Verbindung zu diesem wiederherstellen kann, muss der Transform des Parents zu der *Save Reference* Liste des parents hinzugefügt werden. Dieser Schritt gewährleistet, dass das Speichersystem den Transform des Parents erkennt und das Child beim erneuten Laden des Spiels

seine Beziehung zum richtigen Parent präzise wiederherstellen kann.

Zu den weiteren implementierten Components zählen das SavePosition-Component, mit dem die Position eines Objekts gespeichert werden kann, sowie das SaveVisibility-Component, das den Status speichert, ob ein Objekt aktiv ist.

4.4 Type-Converter

Das *Type-Converter* System ähnelt dem Component-Saving. Der Unterschied besteht darin, dass beim Type-Converter konkrete Typen in ein serialisierbares Format umgewandelt werden. Diese Typen müssen beim Laden zudem innerhalb des Type-Converters initialisiert werden. Dadurch wird es möglich, für spezifische Typen, die von den Entwicklern nicht direkt geändert werden können, sowohl das Referenzieren von normalen Objekten als auch von Unity-Objects zu unterstützen.

Sowohl der Type-Converter als auch das Component-Saving implementieren die Methoden 'OnSave()' und 'OnLoad()' und stellen jeweils einen SaveDataHandler und LoadDataHandler zur Verwaltung des Speichern und Ladens bereit. Der entscheidende Unterschied liegt in der Art und Weise, wie die Datenumwandlung gehandhabt wird: In der 'OnSave()' Methode eines Type-Converters wird der zu konvertierende Typ in ein speicherbares Format überführt. In der 'OnLoad()' Methode wird das Objekt aus den gespeicherten Daten rekonstruiert und zurückzugeben.

Der Type-Converter ist bereits für mehrere Unity-Typen implementiert, darunter Color32, Color, Vector2, Vector3, Vector4 und Quaternions. Neben der Behandlung von Unity-Typen wird der Type-Converter auch für die Serialisierung von Referenzen innerhalb von Collections wie Array, List, Dictionary, Stack und Queue verwendet. Diese Fähigkeit stellt sicher, dass komplexe Datenstrukturen, einschließlich solcher mit verschiedenen Referenzen, genau serialisiert und deserialisiert werden können, wodurch die Datenintegrität gewahrt bleibt.

Verwendung

Um einen einen neuen Type-Converter zu erstellen, können Entwickler entweder von der Klasse BaseConverter<T> erben oder direkt das IConvertible-Interface implementieren und anschließend ihre 'OnSave()' und 'OnLoad()' Methoden für die Serialisierung und Deserialisierung implementieren:

- **Von BaseConverter erben**: Für die meisten Anwendungsfälle reicht es aus, von der Klasse BaseConverter<T> zu erben. Das generische T steht für den Typ, für den die konvertierung durchgeführt werden soll. Diese Klasse bietet grundlegenden Funktionen für das konvertieren von Objekten.

- **IConvertible implementieren**: In bestimmten Spezialfällen kann es vorkommen, dass der BaseConverter<T> fehlschlägt. In solchen Fällen müssen Entwickler das IConvertible-Interface direkt implementieren. Dabei muss auch die Methode 'CanConvert()' definiert werden. Diese Methode bestimmt, unter welchen vorraussetzungen der Typkonverter verwendet werden soll. Wenn 'CanConvert()' true zurückgibt, wird dieser Type-Converter für die Umwandlung genutzt.

Nach der Erstellung des Type-Converters muss dieser im Speichersystem registriert werden. Dies geschieht, indem der erstellte Converter dem statischen Konstruktor der Klasse *TypeConverterRegistry* hinzugefügt wird.

4.5 H3: Technische Umsetzung komplexer Datenstrukturen

Eine zentrale Herausforderung bei der Implementierung komplexer Speicherlösungen besteht in der Wiederherstellung verschachtelter Objekthierarchien. Hypothese H3 nimmt an, dass bei der Deserialisierung die korrekte Rekonstruktion aller Referenzen gewährleistet sein muss, um den State des Spiels exakt wiederherzustellen, wodurch die Fehleranfälligkeit reduziert wird und Entwicklern Zeit spart. Um diese Hypothese zu testen, wurde im System ein Mechanismus implementiert, der beim Speichern von Objekten automatisch ihre Referenzen verfolgt und beim Laden korrekt wiederherstellt.

Um Referenzen zu unterstützen, müssen alle Objekte mittels einer eindeutige GUID identifizierbar sein. Diese GUID darf sich nicht ändern, da die Objekte sonst nicht mehr gefunden werden können. Für die Vergabe von GUIDs bei Assets sind die Asset-Registry und die Prefab-Registry verantwortlich. Innerhalb einer Szene können den einzelnen GameObjects und Transforms über eine Savable Component GUIDs zugewiesen werden. Um zusätzliche Components referenzieren zu können, bietet das Savable-Component eine Registry für Components (im nachfolgenden Savable-Component Registry genannt) an, in die auch andere Unity-Objects eingebunden werden können.

Die Struktur der finalen, serialisierten JSON folgt einem spezifischen Schema: In erster Instanz existiert eine Liste aller relevanten Szenen. Ein Datencontainer einer Szene enthalten zwei zentrale Elemente: Zum einen die platzierten Prefabs und zum anderen ein Datenobjekt, das in Form eines Dictionaries vorliegt. Dieses Dictionary verwendet als Key einem sogenanntem *GuidPath*, welche als struct implementiert ist und eine GUID repräsentiert. Der Value des Dictionaries ist ein *SaveDataBuffer*, der als Container für ein Referenzobjekt dient. Diese Architektur ist bewusst horizontal gestaltet, was bedeutet, dass ein SaveDataBuffer keine rekursive Struktur aufweist, in der ein weiterer SaveDataBuffer verschachtelt werden kann. Dies bedeutet im Umkehrschluss, dass kein Referenzobjekt direkt ein anderes Referenzobjekt enthält. Beim Serialisieren werden alle als speicherrelevant markierten Objekte iteriert, und für jedes dieser Objekte wird ein neuer SaveDataBuffer erstellt. Diese Herangehensweise stellt sicher, dass auch verschachtelte oder zyklische Objekte serialisierbar sind. Allerdings ist auf diese Weise auch Polymorphismus unterstützt. Wenn ein Objekt eine Referenz auf ein anderes Objekt enthält, wird diese Referenz während der Serialisierung durch eine GUID ersetzt. Bei der Deserialisierung wird zunächst überprüft, ob alle benötigten Objekte existieren. Anschließend werden alle Referenzen auf Grundlage der GUIDs des SaveDataBuffers zwischeneinander aufgebaut.

Eine Instanz der Klasse SaveDataBuffer stellt eine serialisierte Referenz eines Objektes dar. Sie enthält zwei wesentliche Fields: die Save-Strategy, die das zugrunde liegende Speicherkonzept angibt, und SavableType, welches den Typ des speicherbaren Objekts repräsentiert. Im Rahmen der Save-Strategie wird differenziert, ob die Speicherung für Typen von UnityEngine.Object, durch das Attribute-Saving, das Component-Saving, den Type-Converter oder die Konvertierung des Objekts durch Newtonsoft JSON erfolgen soll. Die Klasse SaveDataBuffer verwendet zwei Haupt-Datencontainer: Einerseits das Field *GuidPathSaveData*, welches für das Speichern von Referenzen

(Reference-Saving) zuständig ist, und zum anderen das Field *SerializableSaveData*, welches das Value-Saving unterstützt. Diese Fields werden nur initialisiert, wenn sie benötigt werden, um den Speicher effizient zu nutzen.

4.5.1 Dynamic Prefab Loading

Ein zentrales Merkmal des Speichersystems ist die Fähigkeit, Referenzen zu speichern und diese beim Laden wiederherzustellen. In einer Unity-Anwendung können Prefabs zur Laufzeit je nach Bedarf instanziiert und zerstört werden. Wenn jedoch ein Prefab beim Laden fehlt, könnten die zugehörigen Referenzen nicht korrekt wiederhergestellt werden. Um dies zu verhindern, speichert das System aktive Prefabs, sodass dieselben Prefabs beim nächsten Ladevorgang vorhanden sind und die Referenzen intakt bleiben. Dies stellt sicher, dass eine Szene nahtlos und ohne Datenverlust rekonstruiert werden kann. Dies wird im Rahmen des Frameworks als Dynamic Prefab Loading bezeichnet.

In diesem Algorithmus werden die aktuellen Prefabs mit den benötigten Prefabs verglichen. Es werden erforderliche Prefabs instanziiert und überflüssige zerstört, wodurch unterschiedliche Speicherstände geladen werden können, ohne dass die Szene neu geladen werden muss.

Um sicherzustellen, dass das Speichersystem speicherbare Prefabs identifizieren kann, benötigen die gewünschten Prefabs das Savable-Component. Alle speicherbaren Prefabs werden automatisch der Prefab-Registry hinzugefügt. Falls ein Prefab mit der Savable-Component kein Dynamic Prefab Loading unterstützen soll, kann dies über den Inspektor mit dem Field *Dynamic Prefab Spawning Disabled* deaktiviert werden.

Prefabs mit Overrides

Wenn möglich, wird empfohlen, die Zerstörung von Prefabs mit Overrides zu vermeiden. Entwickler können Prefabs mit Overrides identifizieren, indem sie nach Modifikationen in der Prefab-Instanz im Vergleich zu ihrer ursprünglichen Definition suchen. In Unity sind diese Überschreibungen typischerweise durch eine blaue Linie im Inspektor markiert, die darauf hinweist, dass bestimmte Eigenschaften vom Anfangszustand des Prefabs abweichen.
Prefabs mit Overrides erfordern eine besondere Behandlung, da das Dynamic Prefab Loading keine Overrides speichern kann. Wenn ein Prefab mit Overrides zerstört wird, geht es beim laden verloren, da das ursprüngliche Prefab im Anfangszustand neu instanziiert wird. Sollte die Zerstörung von Prefabs mit Overrides unvermeidbar sein, werden die folgenden Lösungsansätze empfohlen:

- **Prefabs inaktiv setzen**: Anstatt die Prefabs zu zerstören, sollten sie mittels 'GameObject.SetActive()' auf inaktiv gesetzt werden. Entwickler können das SaveVisibility-Component verwenden, um den aktiven/inaktiven State der Prefabs zu speichern und zu laden. Dies stellt sicher, dass die Prefabs, einschließlich ihrer Überschreibungen, intakt bleiben und nicht dauerhaft verloren gehen.

- **Szene neu laden**: Vor dem Laden eines gespeicherten States sollte die gesamte Szene neu geladen werden, um sie auf ihren ursprünglichen State zurückzusetzen. Dieser Prozess stellt sicher, dass alle Prefabs in ihrer ursprünglichen Form instanziiert werden.

4.5.2 Save-Strategy

Die Save-Strategie, welche beispielsweise in einem SaveDataBuffer serialisiert wird, gibt dem System den Kontext mit welcher Herangehensweise ein Objekt Serialisiert und Deserialisiert werden soll. Diese Unterscheidung ist notwendig, da Objekte, die von der Klasse UnityEngine.Object erben, nicht ohne Weiteres serialisiert werden können. Beispielsweise ist es nicht möglich, GameObjects mit dem standardmäßigen C#-Konstruktor zu instanziieren, da hierfür eine spezifische Methode names *Instantiate* benötigt wird.

Zudem wird die Save-Strategy im Rahmen des SaveDataBuffers auf der Festplatte gespeichert, um festzulegen, wie ein Objekt später deserialisiert werden soll. Dies ermöglicht es dem System, die Art der Deserialisierung zu identifizieren. Insgesamt stehen fünf verschiedene Save-Strategys zur Verfügung:

- **[UnityObject]**: Diese Strategie wird verwendet, wenn ein Objekt von UnityEngine.Object erbt. Sie ist besonders für Objekte konzipiert, die von MonoBehaviour oder ScriptableObject erben. Sie umfasst sowohl das Attribut-Saving als auch das Component-Saving. Beim Deserialisieren wird automatisch die korrekte Instanz gefunden, sofern diese bereits existiert.
- **[AttributeSaving]**: Diese Strategie wird für das Attribute-Saving und das Component-Saving verwendet. Sie ist für Klassen ausgelegt, die nicht von Unity-Typen erben, und kann auch für Structs genutzt werden. Bei der Deserialisierung wird eine neue Instanz von dem benötigten Objekt erstellt und mit den gespeicherten Daten initialisiert, was eine nahtlose Wiederherstellung des States ermöglicht. Intern wird dies durch Reflection mittels der Methode 'Activator.CreateInstance()' erreicht, die jedoch einen parameterlosen Konstruktor für Klassen erfordert. Für Structs hingegen ist kein parameterloser Konstruktor notwendig, da sie automatisch mit ihren Standardwerten instanziiert werden.
- **[Component-Saving]**: Diese Strategie wird ausschließlich für das Component-Saving verwendet, da das Attribut-Saving bereits durch die vorherige Strategie abgedeckt ist. Abgesehen davon funktioniert sie auf die gleiche Weise wie die vorherige Strategie: Sie nutzt ebenfalls Reflection mittels 'Activator.CreateInstance()' zur Instanziierung und Initialisierung von Objekten.
- **[Type-Converter]**: Diese Strategie wird verwendet, wenn ein Typ durch einen *I* serialisierbar gemacht werden soll. Sie kann für beliebige Typen verwendet werden, erfordert jedoch, dass die Instanz eines Objekts manuell innerhalb des TypeConverter's erstellt wird.
- **[JsonSerialize]**: Diese Strategie nutzt direkt die *Newtonsoft.Json*-Serialisierung und -Deserialisierung für Klassen. Das hat den Vorteil, dass das System jederzeit über eine Herangehensweise zur Serialisierung und Deserialisierung von Objekten verfügt.

4.5.3 Zusammensetzung der GUID

Das Savable-Component ist verantwortlich für die Identifizierung aller Components eines GameObjects, die entweder das Attribute-Saving oder das Component-Saving implementieren und sich auf demselben GameObject befinden, dem das Savable-Component hinzugefügt wurde. Die GUID dieser identifizierten Components setzt sich aus zwei Teilen zusammen: Zum einen aus einer eindeutigen GUID der Savable-Component innerhalb einer Szene und zum anderen aus einer separaten GUID, die jeweils für die identifizierten Components generiert wird. Für jede dieser

identifizierten Components wird ein neuer SaveDataBuffer erstellt, welcher in die richtige Szene des zu serialisierenden DTOs hinzugefügt wird.

Durch diesen Ansatz wird sichergestellt, dass die Serialisierung von Objekten in einer Szene immer auf Components basiert, die von einer Savable-Component erkannt wurden. Jede GUID eines Szenenobjekts beginnt daher mit dem Schema: [SavableSceneID]/[ComponentID]. Da nur in den erkannten Components Objekte zum Speichern und Laden markiert werden können, ist gewährleistet, dass auch für verschachtelte Objekte innerhalb dieser identifizierten Objekte eine GUID generiert wird.

Sobald ein Objekt als Referenz mithilfe des Attribute-Savings oder Component-Savings markiert und gespeichert wird, handelt es sich um verschachtelte Referenzen. Ein neuer SaveDataBuffer wird im DTO angelegt, falls diese Referenz zuvor nicht vorhanden war. Um dieser Referenz eine neue GUID zuzuweisen, wird folgendermaßen vorgegangen: Zunächst wird die SaveDataBuffers Repräsentation des Objekts gesucht, in dem die Referenz markiert wurde. Anschließend wird die GUID dieses SaveDataBuffers verwendet. Die gefundene GUID wird dann mit einer weiteren GUID kombiniert: Beim Attribute-Saving wird der Name des Fields oder der Property als Identifikator verwendet, während beim Component-Saving der Identifikationsname der jeweiligen Methode genutzt wird. Dadurch entsteht der folgende Aufbau der GUID:

[SavableSceneGUID]/[ComponentGUID]/[unique Identifier]/[unique Identifier]/[unique Identifier]...

Nicht für alle referenzierbaren Objekte wird ein eigener SaveDataBuffer erstellt. Für einige Objekte wird lediglich die GUID im aktuellen SaveDataBuffer gespeichert. Dies betrifft Referenzen auf Objekte, die nicht szenenbasiert sind: Objekte in der Asset-Registry, Prefab-Registry und der Savable-Component Registry. Dieser Ansatz stellt sicher, dass nur relevante Daten auf der Festplatte gespeichert werden. Gleichzeitig wird der Speicherverbrauch optimiert, indem Referenzierungsmöglichkeiten effizient genutzt werden. Dieses Konzept hat den Ursprung aus den Experteninterviews.

4.5.4 Value-Saving und Reference-Saving

In diesem Kapitel wird die Funktionsweise zu Value-Saving und Reference-Saving näher erläutert.

Reference-Saving: Wenn das Speichersystem ein Objekt als Referenz speichert, wird eine neue Instanz eines SaveDataBuffers erstellt, welche in die korrekte Szene des zu serialisierenden DTO's hinzugefügt wird. Obwohl dieser Ansatz Flexibilität bietet, ist er im Allgemeinen langsamer und verbraucht mehr Speicher, da der Verwaltungsaufwand für Referenzen höher ist. Daher wird empfohlen, referenzbasiertes Speichern sparsam einzusetzen und nur dann zu verwenden, wenn es für komplexe Objekte oder Beziehungen notwendig ist, die nicht als einfache Werte dargestellt werden können.

Value-Saving: Im Gegensatz dazu vereinfacht Value-Saving diesen Prozess. Anstatt einen neuen, separaten SaveDataBuffers für jede Referenz zu erstellen, wird das Objekt direkt in einem SaveDataBuffers ohne Referenz serialisiert. Dies ist möglich, da sichergestellt ist, dass immer ein aktueller SaveDataBuffers existiert. Jedes Objekt ist in erster Instanz an einem MonoBehaviour gebunden. Dieser Ansatz macht den Speicherprozess effizienter, indem nicht nach dem Identifikator des zu

referenzierenden Objektes beim Laden gesucht werden muss. Allerdings erfordert Value-Saving, dass der richtige Typ direkt angewendet wird, da es im Gegensatz zum referenzbasierten Speichern die dynamische Natur polymorpher Typen nicht unterstützt.

Beim Einsatz des Attribute-Savings werden Objekte mittels Reference-Saving gespeichert. Dies trifft nicht zu, wenn sie bestimmte Kriterien erfüllen: Sie dürfen keine MonoBehaviour- oder ScriptableObject-Instanzen sein, nicht das Component-Saving implementieren und nicht Teil des Type-Converter Systems sein. In diesem Fall werden sie mittels Value-Saving gespeichert.

4.5.5 Single-Scene and Multi-Scene Setup

Das Speichersystem ist darauf ausgelegt, sowohl Single-Scene als auch Multi-Scene Setups effizient zu verwalten und bietet somit Flexibilität für unterschiedliche Spielarchitekturen. Es gibt jedoch eine Einschränkung bei Multi-Scene Setups: Referenzen können beim Speichern und Laden nur innerhalb derselben Szene aufgebaut werden. Wenn ein Objekt in Szene A gespeichert wird, wird es in Szene B als eigenständige Instanz behandelt, selbst wenn die Objekte dieselbe Referenz besitzen. Die Entscheidung, Spieldaten szenenunabhängig zu speichern, beruht auf der Tatsache, dass nicht garantiert werden kann, dass zu jeder Zeit dieselben Szenen gleichzeitig aktiv sind. Diese Herangehensweise gewährleistet, dass jede Szene unabhängig von anderen verwaltet werden kann, was die Konsistenz und Zuverlässigkeit des Speichersystems in dynamischen Spielszenarien erhöht.

Es besteht jedoch die Möglichkeit, szenenübergreifende Referenzen durch den Einsatz von ScriptableObjects zu verwalten. In diesem Kontext agiert das ScriptableObject als DTO. Um diesen Ansatz umzusetzen, sind folgende Schritte erforderlich:

- **Festlegung einer Szene als Ursprung**: Es sollte eine einzelne Szene festgelegt werden, die als Ausgangspunkt für das Speichern und Laden der relevanten Daten dient.
- **Markierung der Objekte für das Speichern**: Zusätzlich ist es erforderlich das ScriptableObjects in der Asset-Registry aufzunehmen.
- **Registrierung in Registry**: Wenn Entwickler das Speichern von Daten auf ScriptableObjects unterstützen möchten, müssen diese der Asset-Registry hinzugefügt werden. Dies liegt daran, dass das ScriptableObject sowohl Asset als auch Script ist.
- **Nutzung des ScriptableObject als Brücke**: Nach dem Speichern und Laden können die im ScriptableObject gespeicherten Daten in anderen Szenen verwendet werden.

Wenn ein ScriptableObject in zwei Szenen gespeichert wird, wird es zweimal geladen. Dies könnte nützlich sein, wenn die Szenen gemeinsam geladen werden und verschiedene Objekte enthalten, die auf das ScriptableObject angewendet werden müssen. Generell wird jedoch empfohlen, diese Vorgehensweise zu vermeiden und stattdessen das ScriptableObject in zwei separate Objekte aufzuteilen, um Konflikte zu verhindern und eine sauberere Datenstruktur zu gewährleisten.

4.6 Versioning

Die Implementierung des Versionierungssystems umfasst derzeit lediglich eine Versionsnummer im SaveLoadManager. Bei der Deserialisierung wird diese Versionsnummer mit der lokalen Version

verglichen und nur akzeptiert, wenn eine Übereinstimmung vorliegt. Diese Nummer basiert auf dem Konzept des Semantic Versioning, bei dem die Versionsnummer einem spezifischen Format folgt: MAJOR.MINOR.PATCH. Im Rahmen der Definition des Semantic Versioning werden die Bedeutungen der einzelnen Werte wie folgt definiert:

- MAJOR wird erhöht, wenn API-inkompatible Änderungen veröffentlicht werden,
- MINOR wird erhöht, wenn neue Funktionalitäten, die kompatibel zur bisherigen API sind, veröffentlicht werden, und
- PATCH wird erhöht, wenn die Änderungen ausschließlich API-kompatible Bugfixes umfassen [25].

Zusätzlich wurde bei der Implementierung des Speichersystems auf einen weiteren Aspekt besonders geachtet: Die Deserialisierung basiert auf den in der geladenen Datei enthaltenen Daten und nicht auf die Datenstruktur der Anwendung. Beim Laden werden die deserialisierten Daten in die Datenstruktur der Anwendung überführt. Dies bietet den Vorteil, dass der Standartwert innerhalb der Anwendung bei neuen und unbekannten Daten verwendet wird.

Es existiert für das Speichersystem derzeit keine Implementierung zur Migration von einer alten Version auf eine neue Version. Eine solche Migration ist bislang lediglich konzeptionell vorgesehen, wie im Folgenden erläutert wird.

4.6.1 Konzept

Im Zentrum des Konzepts für das Versionierungssystems steht ein Konverter, der die Umwandlung von einer definierten Version in eine andere ermöglicht. Dieser Konverter führt die Versionierung direkt auf die geladene JSON-Datei durch. Dies erfolgt bevor die JSON in das DTO Objekt deserialisiert wird. Das ist erforderlich, da nicht garantiert werden kann, dass die Daten der JSON ohne vorherige Anpassungen erfolgreich in Instanzen der aktuellen Version konvertiert werden können. Sollten zwischen der veralteten JSON-Version und der aktuellen Systemversion mehrere Updates liegen, werden die Konverter in der entsprechenden Reihenfolge auf die JSON-Daten angewendet.

Jeder Konverter definiert explizit, von welcher Version in welche konvertiert wird. Die Konvertierung erfolgt dabei typbasiert, d.h., Entwickler spezifizieren für einen bestimmten Typ, welche Änderungen vorgenommen werden müssen. Dies ist möglich, da in jedem SaveDataBuffer der Typ des Objekts gespeichert ist. Für die Umwandlung der einzelnen Typen stehen verschiedene Operationen zur Verfügung:

- **Änderung der GUID von Membern**: Der Begriff *Member* bezieht sich dabei auf die Fields und Properties beim Attribute-Saving. Beim Component-Saving sind damit jene Member gemeint, die durch Methoden serialisierbar gemacht wurden. Im Falle des Attribute-Savings entspricht die GUID den Namen der Fields und Properties, während sie beim Component-Saving die in den Methoden definierten Identifikatoren repräsentiert.
- **Änderung des Namens des Klassentyps**: Die Umbenennung eines Typs soll direkt im Konverter vorgenommen werden.
- **Änderung des Typs der Member**: Dies geschieht beim Reference-Saving indirekt durch das ändern des Namens des Klassentyps. Bei dem Value-Saving muss die Konvertierung direkt erfolgen, indem das gewünschte Objekt der JSON in ein JObject umgewandelt wird, um dann die entsprechende Änderung vorzunehmen.

- **Hinzufügen und Entfernen von gespeicherten Membern**: Dabei unterstützt das Entfernen von Membern die Initialisierung von Standartwerten der Anwendung. Das liegt daran, dass durch das Entfernen von Membern die in der Anwendung definierten Standardwerte verwendet werden können.
- **Änderung des gespeicherten Werts**: Anpassungen am gespeicherten Wert müssen ebenfalls durch den Konverter durchgeführt werden.
- **Änderung der GUIDs**: Darüber hinaus muss gewährleistet sein, dass sämtliche GUIDs, die im Inspector gesetzt werden können, ausgetauscht werden. Dies betrifft die GUIDs der Registrys sowie der Savable-Components.

Durch diesen Konvertierungsmechanismus wird sichergestellt, dass die Daten über verschiedene Versionen hinweg konsistent bleiben und das System in der Lage ist, Updates und Veränderungen flexibel zu integrieren. Neben der Aufwärtskompatibilität ist durch diese Herangehensweise auch die Abwärtskompatibilität denkbar.

4.7 Implementierung weiterer Anforderungen

Im Vergleich zu den vorherigen Systemen nehmen diese Aspekte eine untergeordnete Rolle ein oder lassen sich leicht umsetzen.

4.7.1 Datenkonsistenz und Fehlerbehandlung

Um die Integrität der Save-Datei zu gewährleisten, verwendet das System einen Integrity-Check. Dabei wird ein Integrity-Key in einer Meta-Datei abgelegt, der mit der entsprechenden Save-Datei abgeglichen wird. Dieser Abgleich gewährleistet, dass die gespeicherten Daten unverändert und konsistent sind. Zur Generierung des Integrity Keys wurden drei verschiedene Algorithmen verwendet: SHA-256 Hashing, CRC-32 und Adler-32. Diese Algorithmen unterscheiden sich hinsichtlich ihrer Kollisionsresistenz und des benötigten Rechenaufwands, wobei SHA-256 eine höhere Kollisionsresistenz aufweist, jedoch auch rechenintensiver ist, während CRC-32 und Adler-32 eine schnellere Berechnung ermöglichen, aber anfälliger für Kollisionen sind.

Zusätzlich wurde eine weitere Sicherheitsmaßnahme implementiert, die sicherstellt, dass der Schreibprozess auf die Festplatte nur nach erfolgreicher Serialisierung der Daten erfolgt. Das bedeutet, dass die Daten zunächst gesammelt, verschlüsselt, auf Integrität überprüft und komprimiert werden, bevor sie endgültig auf die Festplatte geschrieben werden. Diese Methode minimiert das Risiko von Datenkorruption während des Speicherprozesses.

Darüber hinaus benachrichtigt das System den Entwickler durch zahlreiche Fehlermeldungen über potenzielle Probleme. Zum Beispiel wird eine Fehlermeldung ausgegeben, wenn eine Referenz zu einem benötigten Asset nicht gefunden werden kann.

4.7.2 Speicheroptimierung

Zur Reduzierung der Dateigrößen verwendet das System die Gzip-Datenkompression. Diese effiziente Methode der verlustfreien Kompression stellt sicher, dass große Datenmengen ohne Beeinträchtigung der Datenintegrität deutlich verkleinert werden können, was besonders bei begrenztem Speicherplatz von Vorteil ist. Die Wichtigkeit dieses Aspekts wurde auch in den

Experteninterviews hervorgehoben, da die Speicherressourcen auf mobilen Geräten oft limitiert sind und effiziente Speicherlösungen daher einen entscheidenden Vorteil darstellen.

4.7.3 Performance

Die Performance des Systems wird durch eine klare Trennung zwischen der Erstellung eines Snapshots der Szene und dem eigentlichen Schreiben und Lesen der Daten auf die Festplatte optimiert. Dabei wird jede dieser Aktionen in einer Queue registriert, um sicherzustellen, dass sie in der korrekten Reihenfolge ausgeführt wird. Das Erstellen und Anwenden der Snapshots auf die Szene erfolgt innerhalb eines Frames. Sämtliche Dateioperationen, die das Schreiben und Lesen von der Festplatte betreffen, werden vollständig asynchron ausgeführt, um sicherzustellen, dass diese Vorgänge den Haupt-Thread nicht blockieren. Dieser asynchrone Ansatz trägt wesentlich zur Aufrechterhaltung einer flüssigen Spielperformance bei, da zeitintensive Dateioperationen im Hintergrund abgewickelt werden und somit das System flüssig läuft.

4.7.4 Plattformübergreifende Unterstützung

Wie bereits in den Anforderungen definiert, ist die plattformübergreifende Kompatibilität ein zentraler Aspekt. Da Unity bereits weitreichend verschiedenen plattformen unterstützt, muss ein Speichersystem diesem Anspruch gerecht werden. JSON als universelles Format stellt sicher, dass die serialisierten Daten in unterschiedlichen Umgebungen korrekt verarbeitet werden können.

4.7.5 Security

Zur Gewährleistung der Datensicherheit wird im System die AES-Verschlüsselung eingesetzt, um die gespeicherten Daten vor unbefugtem Zugriff zu schützen. Entwickler haben die Möglichkeit, den Schlüssel und das Salt entweder direkt über den Inspector des SaveLoadManagers zu konfigurieren oder eine Methode zu verwenden, um diese Werte zu cachen. Der Cache-Mechanismus erlaubt es, bei besonders sicherheitskritischen Anwendungen den Schlüssel und das Salt beispielsweise von einem externen Server abzurufen. Diese Herangehensweise bietet sowohl Flexibilität als auch eine erhöhte Sicherheit bei der Datenverwaltung, da die sensiblen Schlüssel nicht lokal gespeichert werden müssen und somit die Gefahr eines unbefugten Zugriffs minimiert wird.

Zu Beginn der Implementierung wurde der BinaryFormatter für die Serialisierung eingesetzt, jedoch stellte sich heraus, dass dieses Verfahren das System erheblich für Sicherheitsrisiken anfällig macht. Um diese Schwachstellen zu vermeiden, wurde auf das JSON-Format umgestellt. Diese Entscheidung basiert nicht nur auf der höheren Sicherheit bei der Deserialisierung, sondern auch auf der breiten plattformübergreifende Kompatibilität von JSON, die als zentrale Anforderung definiert wurde.

Kapitel 5

Auswertung des Speichersystems

Im Rahmen dieser Arbeit wurde eine umfassende Evaluation des entwickelten Systems durchgeführt. Die Evaluationsmethodik bestand aus zwei zentralen Komponenten: einer Anforderungsanalyse, die als Referenzrahmen diente, sowie einem User-Test, der die praktische Bewertung des Systems ermöglichte. Ziel dieser Untersuchung war es, zu überprüfen, inwieweit die im Vorfeld definierten Anforderungen erfüllt wurden und wie entwicklerfreundlich das System in der Praxis ist. Durch die Kombination beider Ansätze konnte eine detaillierte Bewertung der Stärken und Schwächen des Systems vorgenommen werden, um einen potenziellen Optimierungsbedarf zu identifizieren und zukünftige Entwicklungsschritte abzuleiten.

5.1 User-Tests

Die User-Tests dienten dazu, die Entwicklerfreundlichkeit und die intuitive Bedienbarkeit des entwickelten Speichersystems in einer praxisnahen Umgebung zu evaluieren. Dabei lag der Fokus auf der Erfüllung der definierten Anforderungen: eine Entwicklerfreundliche API und gute Dokumentation. Hierbei wurde nicht nur die technische Funktionalität bewertet, sondern auch die Usability im Hinblick auf die Interaktion der Benutzer mit dem System in verschiedenen Anwendungsszenarien. Alle im Rahmen der Benutzer-Tests erhobenen Daten sind dem Anhang .3 beigefügt.

5.1.1 Selektion der Tester

Es wurden Fünf User-Tests durchgeführt. Die Rekrutierung der Teilnehmenden erfolgte auf der selben Weise wie bei den Experteninterviews: ausschließlich über Kontakte im akademischen Umfeld. Die Kriterien wurden hier im Gegensatz zu den Experteninterviews nur auf umfangreiche Kentnissen in der Entwicklung von objektorientierten Programmiersprachen angesetzt.

Die Teilnehmenden des User-Tests wiesen eine breite Diversität hinsichtlich ihres beruflichen Hintergrunds und ihrer Erfahrung auf. Ihr Alter lag zwischen 21 und 50 Jahren, wobei die Mehrheit in die Kategorie 21–35 Jahre fiel. Beruflich waren die Teilnehmenden in unterschiedlichen Positionen tätig, von Studierenden und wissenschaftlichen Mitarbeitenden bis hin zu Softwareentwicklern und UI Team Leads, was ein breites Spektrum an Expertise abdeckt – von Berufseinsteigern bis hin zu Führungskräften.

Die Erfahrung im Bereich objektorientierter Programmierung variierte stark: Einige Teilnehmende hatten erst zwei Jahre Erfahrung, während andere bis zu 16 Jahre in Sprachen wie C# oder Java arbeiteten. Ähnlich vielfältig war die Erfahrung mit der Unity-Entwicklungsplattform, die von keiner bis zu sechs Jahren reichte. Diese Bandbreite an beruflicher Expertise und Fachwissen gewährleistet unterschiedliche Perspektiven und trägt zur Validität der Untersuchungsergebnisse bei.

5.1.2 Format des User-Tests

Zu Beginn des Tests wurde eine demografische Umfrage durchgeführt, um grundlegende Informationen über die Teilnehmenden zu erfassen. Die Erhebung der persönlichen Daten diente dem Zweck, die Testergebnisse im Kontext der Nutzererfahrung besser einordnen zu können. Alle Daten der User-Tests wurden dabei anonym erhoben und nur zur statistischen Auswertung genutzt. Wie bei den Experteninterviews wurde zunächst eine Pilotrunde durchgeführt, um mögliche Unklarheiten zu identifizieren und sicherzustellen, dass die Aufgabenstellungen verständlich und durchführbar sind. Auf Basis der Ergebnisse der Pilotrunde wurden kleinere Anpassungen vorgenommen, bevor die endgültigen User-Tests mit den Teilnehmern durchgeführt wurden.

Im Fokus der User-Tests stand die Implementierung von Speichern und Laden anhand des Unity Templates *2D Platformer Microgame* sowie eines *Inventarsystems*. Beide Aufgaben ergänzten sich, indem sie unterschiedliche Aspekte des Systems beleuchteten und eine umfassende Evaluation der Entwicklerfreundlichkeit und Flexibilität des Speichersystems sowie der begleitenden Dokumentation ermöglichten. Die Teilnehmer sollten während des Tests ihre Gedankengänge verbal äußern (Methode des lauten Denkens), welche zeitgleich Protokolliert wurden. Zur Unterstützung wurden den Teilnehmern Fragen zum System beantwortet und Hilfestellungen in Form von Tipps bereitgestellt. Nach Abschluss der jeweiligen Implementierungen erhielten die Teilnehmer eine kurze Reflexionsphase, in der sie ihre Vorgehensweise und die durchgeführten Tätigkeiten rekapitulieren konnten. Den Teilnehmern stand hierfür die erstellte Dokumentation zur Verfügung. Anschließend wurde eine Umfrage zur Bewertung ihrer Erfahrung durchgeführt. Um die Usability des Systems systematisch und vergleichbar zu erfassen, wurde hierfür der standardisierte System Usability Scale (SUS)-Fragebogen verwendet [29]. Der SUS-Fragebogen ermöglicht es, eine quantitative Bewertung der Entwicklerfreundlichkeit des Systems vorzunehmen und gewährleistet durch seine weit verbreitete Anwendung eine Vergleichbarkeit der Ergebnisse mit anderen Studien und Systemen.

2D Platformer Microgame Implementierung

Die erste Aufgabe des User-Tests wurde im Rahmen des von Unity angebotenen 2D Platformer Microgame Projekts durchgeführt. Der Test konzentrierte sich hauptsächlich auf die Konfiguration des Systems und die Hypothese H1, die die Implementierung des Component-Savings beinhaltet. Diese Aufgabe erforderte keinen Programmieraufwand. Für die Implementierung wurde den Teilnehmern eine konkrete Schritt-für-Schritt-Anleitung bereitgestellt, die detailliert erklärte, wie das System umzusetzen ist. Dabei mussten die Teilnehmer das Speichersystem in das Projekt integrieren, was die Einrichtung des SaveLoadManagers, der Asset-Registry und der Prefab-Registry umfasste. Ein weiterer Schwerpunkt lag auf der Rolle des SaveSceneManager-Components und des Savable-Components, die für das Speichern und Laden von Spielobjekten unerlässlich sind.

Das Ziel bestand darin, mithilfe des Component-Saving die folgenden Bereiche für das Speichern

und Laden zu markieren: die Position des Spielercharakters und der Gegner sowie den Status der sammelbaren Tokens, ob diese bereits vom Spieler aufgenommen wurden. Abschließend sollten die Teilnehmer die implementierten Speicher- und Ladefunktionen testen, um deren Funktionsfähigkeit sicherzustellen.

Inventar Implementierung

Für die zweite Aufgabe des User-Tests wurde ein spezifischer Use-Case entwickelt, der die Stärken und Schwächen des Speichersystems aufzeigen sollte. Dieser Use-Case konzentrierte sich auf die Implementierung eines Inventarsystems. Im Gegensatz zum ersten Test liegt der Schwerpunkt des zweiten User-Tests auf der Programmierung und tieferen Integration des Speichersystems. Das Ziel bestand darin, dass die Teilnehmer mithilfe des Attribute-Saving die Speicher- und Ladefunktionen für komplexere Objekte implementieren. Diese Aufgabe diente insbesondere der Analyse der Hypothesen H2 und H3. Dieser Test erforderte deutlich mehr Programmieraufgaben und verlangte von den Teilnehmern, den Code anzupassen und zu erweitern. Die bereitgestellten Schritt-für-Schritt-Anweisungen waren absichtlich weniger detailliert, um die Teilnehmer zu ermutigen, eigenständig zu agieren und ein tieferes Verständnis für die Funktionsweise des Systems zu entwickeln.

Das Inventar verfügt einerseits über eine UI, in der die einzelnen Items dargestellt werden, und andererseits über zwei Buttons: Mit einem Button kann ein zufälliges Item hinzugefügt werden, während der zweite Button dazu dient, ein zufälliges Item aus dem Inventar zu entfernen. Ziel dieser Aufgabe ist es, sicherzustellen, dass nach dem Laden einerseits alle Objekte intern korrekt wiederhergestellt werden und andererseits die UI auf den aktuellen Stand aktualisiert wird. Zusätzlich muss es nach dem Laden möglich sein, alle existierenden Items im Inventar mithilfe des Buttons zu löschen.

5.1.3 Analyse der Daten

Der Artikel von Jeff Sauro bietet eine fundierte Grundlage zur Bewertung von SUS-Fragebögen [23]. In diesem Artikel wird der Wert von 68 als durchschnittlich eingestuft. Werte über 68 deuten auf überdurchschnittliche Usability hin, während niedrigere Werte Verbesserungspotenzial aufzeigen. Werte ab 85 werden als „exzellent“ eingestuft, während 50 bis 70 „marginal akzeptabel“ ist. Werte unter 50 weisen auf erhebliche Usability-Probleme hin, die einer Überarbeitung bedürfen.

Analyse der SUS-Daten

Die Durchschnittswerte der SUS-Scores zeigen deutliche Unterschiede in der wahrgenommenen Entwicklerfreundlichkeit zwischen den beiden Aufgaben. Aufgabe 1 erzielte einen durchschnittlichen SUS-Wert von 78,5, was auf eine überdurchschnittlich hohe Entwicklerfreundlichkeit hindeutet. Die Gesammtbewerungen pro Teilnehmer sind in Abbildung 5.1 dargestellt. Im Gegensatz dazu ist in Abbildung 5.2 dargestellt, dass der durchschnittliche SUS-Wert für Aufgabe 2 bei 63,0 liegt. Dies ist unter dem als akzeptabel geltenden Schwellenwert von 68 und weißt dadurch auf potenzielle Usability-Probleme hin.

Die Verteilung der SUS-Scores verdeutlicht diese Diskrepanz weiter. Während die Bewertungen für Aufgabe 1 mit einer maximalen Variation von 10 sich ähneln, zeigt Aufgabe 2 eine breitere Streuung mit mehreren Teilnehmern, die das System unter 70 bewerteten. Dies deutet auf eine

größere Variation in den Benutzererfahrungen hin, die vermutlich durch unterschiedliche Vorkenntnisse und Schwierigkeiten bei der Implementierung des Inventarsystems verursacht wurde.

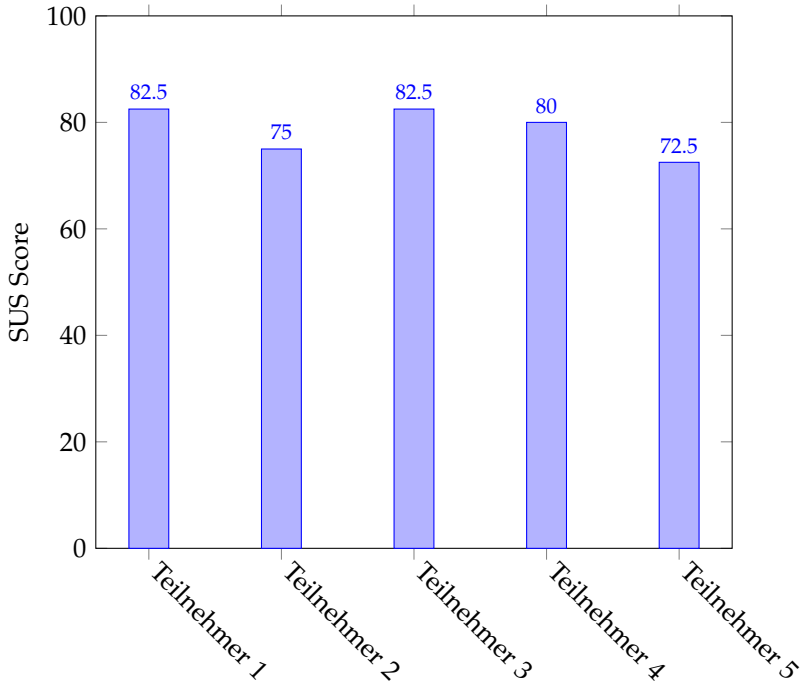


Abbildung 5.1: SUS Scores für Aufgabe 1

Die Analyse der einzelnen SUS-Fragen bietet weitere Einblicke in die spezifischen Herausforderungen, die in den beiden Aufgaben auftraten. In Aufgabe 1 gaben die Teilnehmer an, das System als leicht zu bedienen wahrzunehmen (Durchschnittswert 4,4), während in Aufgabe 2 dieser Wert signifikant niedriger ausfiel (3,6). Ein weiteres signifikantes Ergebnis betrifft die Frage, ob die Teilnehmer das Gefühl hatten, ohne Unterstützung eines technischen Experten auszukommen. In Aufgabe 2 war der Bedarf an technischer Unterstützung (Durchschnittswert 2,8) deutlich höher als in Aufgabe 1 (1,8). Auch das Selbstvertrauen der Nutzer in Bezug auf die Nutzung des Systems war in Aufgabe 2 geringer (2,6 gegenüber 3,4 in Aufgabe 1). Die Teilnehmer gaben außerdem an, dass sie in Aufgabe 2 mehr lernen mussten, bevor sie das System effektiv nutzen konnten (3,0 im Vergleich zu 2,4 in Aufgabe 1). Diese Ergebnisse lassen sich auf die höhere Komplexität und die geringere Unterstützung durch die Dokumentation der zweiten Aufgabe zurückführen. Darüber hinaus deuten sie auf eine steilere Lernkurve und möglicherweise unzureichende Dokumentation oder Benutzeranleitung für fortgeschrittene Implementierungen hin. Die konkreten Werte für alle Fragen des SUS-Fragebogens sind in Tabelle 5.1 abgebildet.

Analyse der Bearbeitungszeiten

Die Bearbeitungszeiten der beiden Aufgaben zeigt signifikante Unterschiede in der benötigten Zeit. Diese Unterschiede lassen sich auf mehrere Faktoren zurückführen, darunter die Komplexität der Aufgaben, die Herangehensweise der Teilnehmenden (einige nutzten die Aufgaben, um sich intensiver mit dem System vertraut zu machen) sowie ihre Vorkenntnisse und Erfahrungen.

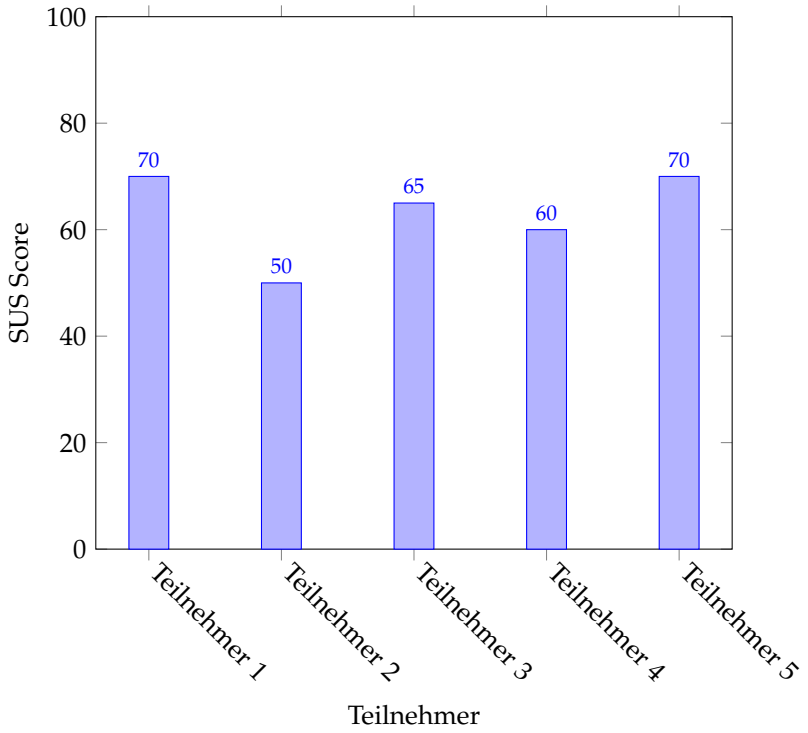


Abbildung 5.2: SUS Scores für Aufgabe 2

SUS Frage	**Aufgabe 1 Durchschnitt**	**Aufgabe 2 Durchschnitt**
I think that I would like to use this system frequently.	4,0	3,8
I found the system unnecessarily complex.	2,0	2,6
I thought the system was easy to use.	4,4	3,6
I think that I would need the support of a technical person to use this system.	1,8	2,8
I found the various functions in this system were well integrated.	4,2	4,0
I thought there was too much inconsistency in this system.	1,4	1,6
I would imagine that most people would learn to use this system very quickly.	4,6	3,6
I found the system very cumbersome to use.	1,6	2,4
I felt very confident using the system.	3,4	2,6
I needed to learn a lot of things before I could get going with this system.	2,4	3,0

Tabelle 5.1: SUS Fragen mit Durchschnittswerten für Aufgabe 1 und Aufgabe 2

Die Bearbeitungszeit für Aufgabe 1 (2D Platformer Microgame) variierte zwischen 22 und 45 Minuten. Im Vergleich dazu benötigten die Teilnehmer für Aufgabe 2 (Inventar Implementierung) mehr Zeit. Auch dies lässt sich auf die höhere Komplexität und die geringere Detaillierung der bereitgestellten Anweisungen zurückzuführen, was eine intensivere Auseinandersetzung mit dem System und eine tiefere Programmierarbeit erforderte. Diese Ergebnisse unterstreichen den Bedarf nach einer klareren Dokumentation und einer stärkeren Unterstützung bei komplexeren Implementierungsaufgaben, um das Erlernen des Systems für Entwickler zu erleichtern.

Qualitative Rückmeldungen

Die qualitativen Rückmeldungen der Teilnehmer bestätigen die Ergebnisse der Umfragen. Während die meisten Rückmeldungen zu Aufgabe 1 überwiegend positiv waren, gab es bei Aufgabe 2 mehrere Hinweise auf Verwirrung und Bedarf an zusätzlicher Anleitung. Die Teilnehmer wiesen darauf hin, dass die Verwendung der Attribute, das Referenzieren von Assets mittels der Asset-Registry und die Event-Callbacks nicht immer klar war. Die Rückmeldungen lassen darauf schließen, dass sowohl die Dokumentation verbessert werden muss, mehr Anwendungsbeispiele notwendig sind und die generelle Usability des Frameworks optimiert werden sollte.

5.2 Bewertung der Funktionalen Anforderungen

5.2.1 Entwicklerfreundliche API

Die Anforderung einer Entwicklerfreundlichen API wurde maßgeblich durch die Hypothesen H1 und H2 beeinflusst. Das im Rahmen der Hypothese H2 untersuchte Component-Saving wurde von den Teilnehmern als entwicklerfreundlich und flexibel bewertet, was durch einen hohen SUS-Wert von 78,5 bestätigt wurde. Im Gegensatz dazu zeigte das im Rahmen der Hypothese H1 untersuchte Attribute-Saving Schwächen in der Usability und erzielte einen vergleichsweise niedrigeren SUS-Wert von 63,0. Es wurde als weniger intuitiv empfunden und erfordert Optimierungen, um den Implementierungsaufwand zu reduzieren und eine bessere Nutzererfahrung zu bieten. Deswegen wurde die Anforderung, eine intuitive API für das Speichersystem bereitzustellen, teilweise erfolgreich erfüllt.

Optionen für Verbesserungen

Um die Entwicklerfreundlichkeit des Attribute-Saving zu erhöhen, sollte ein alternativer Ansatz verfolgt werden, der stärker an bekannte Unity-Konzepte wie *SerializeField* und *Serializable* angelehnt ist. Dadurch könnten Entwickler auf vertraute Mechanismen zurückgreifen, was die Lernkurve senkt und die Implementierung intuitiver macht. Für Fields könnte das Attribut [SaveField] und für Properties [SaveProperty] eingeführt werden. Für Klassen sollte ein [Savable]-Attribut genutzt werden, um explizit festzulegen, welche Daten gespeichert werden sollen. Im Gegensatz zum aktuellen markieren von Klassen sollten dabei nur öffentliche Fields und Properties gespeichert werden, es sei denn, sie sind durch das Attribut [SaveIgnore] ausgeschlossen. Private Fields und Properties können explizit mit [SaveField] oder [SaveProperty] markiert werden. Um die Bedienung weiter zu erleichtern, sollten diese Attribute in der integrierten Entwicklungsumgebung (IDE) ausgegraut sein, wenn die zugrundeliegende Klasse nicht speicherbar oder serialisierbar ist. Dies würde die Funktionsweise des Unity Serialization-Systems nachahmen. Ein wichtiger Unterschied wäre jedoch, dass auch MonoBehaviours und ScriptableObjects explizit mit dem [Savable]-Attribut markiert werden müssten. Dies ist mittels des Unity Serialization-Systems

nicht erforderlich, da alle Components standartmäßig erkannt und für den Inspector serialisiert werden.

Beim Component-Saving wurde die Entwicklerfreundlichkeit von den Teilnehmern bereits als hoch eingestuft. Daher soll die API um eine größere Auswahl an vorgefertigten Component-Saving-Skripten erweitert werden.

Das Callback-System hingegen wurde von den Teilnehmenden als ineffizient und komplex beschrieben. Der UI-Team-Lead schlug vor, die Implementierung zu verbessern, indem das Savable-Component automatisch relevante Interfaces auf demselben GameObject erkennt und aufruft. Dadurch könnte der Entwicklungsaufwand reduziert und die Implementierung von Events effizienter gestaltet werden, was letztlich die Usability des Systems erhöht.

Auch das manuelle Hinzufügen von Assets zur Asset-Registry wurde von den Teilnehmenden als umständlich und unübersichtlich wahrgenommen. Sie wünschten sich eine Automatisierung, ähnlich wie beim Dynamic-Prefab-Loading in Unity, um den Prozess zu vereinfachen und den Aufwand für die Entwickler zu reduzieren.

Zusätzlich wurde empfohlen, das Setup des Systems weiter zu automatisieren, um den Implementierungsprozess zu vereinfachen. Zudem wurde angemerkt, dass der Aufbau der generierten JSON-Dateien optimiert werden sollte, um die Lesbarkeit und Struktur zu verbessern. Ein weiterer Vorschlag war, die Möglichkeit zu schaffen, direkt im Unity-Inspector einzelne Fields und Properties als speicherbar zu markieren. Dies würde den Vorteil bieten, dass Entwickler diese Konfiguration visuell vornehmen könnten, ohne zusätzlichen Code schreiben zu müssen, was den Workflow deutlich vereinfachen und beschleunigen würde.

5.2.2 Datenkonsistenz

Die Überprüfung der erfolgreichen Umsetzung der Anforderung zur Datenkonsistenz erfolgt durch eine Integritätsprüfung, wobei die Methodik dieser Prüfung auch für andere Anforderungen der Anforderungsanalyse verwendet wird. Diese basiert auf der Generierung eines Integritätskeys sowohl beim Speichern als auch beim Laden der Daten. Durch den Vergleich dieser beiden Keys wird die Konsistenz der Daten validiert. Stimmen die Keys überein, gilt die Integritätsprüfung als erfolgreich und die Konsistenz der gespeicherten Daten ist gewährleistet. Zur Berechnung der Integritätskeys wurden drei Algorithmen implementiert: SHA256, CRC32 und Adler32. In den durchgeführten Tests haben alle drei Algorithmen die Datenkonsistenz erfolgreich sichergestellt, wodurch die Anforderung der Datenkonsistenz vollständig erfüllt wurde.

5.2.3 Komplexität der Datenstrukturen

Zur Validierung dieser Anforderung wurden Integritätsprüfungen anhand der Musterlösungen der beiden Aufgaben aus den User-Tests durchgeführt. Die Testergebnisse bestätigen, dass das System die Anforderung zur Speicherung und Wiederherstellung komplexer Datenstrukturen vollständig erfüllt. Dabei wurden sowohl Reference-Type und Value-Type Objekte als auch die Referenzen von Unity Objekte korrekt verarbeitet.

Diese erfolgreichen Integritätstests zeigen zudem für Hypothese H3, dass durch die Automatisierung der Wiederherstellung potenzielle Fehlerquellen vermieden werden können. Allerdings

ist dies nur teilweise der Fall, da es für Entwickler nicht immer klar ist, welche Assets zur Asset-Registry hinzugefügt werden müssen. Der erhöhte Zeitaufwand entsteht lediglich beim Markieren von Referenzen beim Hinzufügen von Assets zur Asset-Registry, sodass der zusätzliche Aufwand für die Verwaltung von Referenzen minimal bleibt und insgesamt eine Zeitersparnis für Entwickler darstellt. Durch die Automatisierung des Hinzufügens von Assets zur Asset-Registry könnte der Prozess weiter optimiert werden, um sowohl die Fehleranfälligkeit weiter zu reduzieren als auch den Zeitaufwand für Entwickler zu minimieren, was die Hypothese H3 zusätzlich stützen würde.

5.2.4 Versioning

Die Anforderung, dass das System abwärtskompatibel sein muss, sodass Spielstände aus früheren Versionen in neueren Versionen korrekt geladen werden können, wurde bisher nur konzeptionell umgesetzt. Aktuell verfügt das System über eine Grundfunktionalität, die eine Error Warning ausgibt, wenn eine Spielstandsversion nicht mit der aktuellen Version des Systems kompatibel ist. In einem solchen Fall blockiert das System das Laden der Save-Datei.

Zur Validierung des finalen Versionierungssystems ist vorgesehen, eine Integritätsprüfung durchzuführen, bei der eine Spielstand-Datei aus einer älteren Version in eine neuere Version importiert wird. Diese Prüfung wird es ermöglichen, festzustellen, ob die gespeicherten Daten vollständig und korrekt geladen werden können, ohne dass Datenverlust oder Inkonsistenzen auftreten.

Somit ist die Anforderung derzeit nur teilweise umgesetzt, und es sind weitere Schritte notwendig, um eine vollständige Abwärtskompatibilität zu gewährleisten.

5.2.5 Plattformübergreifende Unterstützung

Die Validierung der plattformübergreifenden Unterstützung wurde mittels Integritätsprüfungen auf verschiedenen Plattformen durchgeführt. Die Tests beschränkten sich auf die Betriebssysteme Windows und Android, da aufgrund von Ressourcenmangel keine weiteren Betriebssysteme, wie macOS, iOS oder Linux, zur Verfügung standen. Diese Einschränkung führt dazu, dass keine Aussagen über die Kompatibilität des Systems mit diesen nicht getesteten Betriebssystemen getroffen werden können. Um die plattformübergreifende Unterstützung umfassend zu bestätigen, wären zusätzliche Tests auf den nicht geprüften Plattformen erforderlich. Die durchgeführten Integritätsprüfungen zwischen den beiden Unity-Backends Mono und IL2CPP sowie zwischen Windows und Android mit Mono verliefen jedoch erfolgreich, was bestätigt, dass die Anforderung zur plattformübergreifenden Unterstützung innerhalb der getesteten Umgebungen erfüllt wurde.

5.2.6 Fehlerbehandlung

Die Anforderung, dass das System den Entwickler auf auftretende Fehler aufmerksam macht und gleichzeitig robust gegenüber Fehlern und Abstürzen reagiert, wurde nur teilweise erfüllt. Zwar verfügt das System über Mechanismen zur Erkennung und Benachrichtigung von Fehlern, jedoch existieren in bestimmten Bereichen noch Schwächen, die im Folgenden näher erläutert werden.

Eine zentrale Funktion ist, dass der Schreibprozess auf die Festplatte erst nach erfolgreicher Serialisierung, Verschlüsselung, Integritätsimplementation und Kompression der Daten erfolgt. Dies verringert das Risiko von Datenkorruption während des Speichervorgangs. Zudem werden Entwickler benachrichtigt, wenn beispielsweise eine Referenz zu einem benötigten Asset nicht

gefunden werden kann. Allerdings fehlt eine API, die Entwicklern ermöglicht, auf Fehler zu reagieren. Eine solche API könnte beispielsweise über Callback-Funktionen Entwicklern ermöglichen, bei Fehlern spezifische Schritte gezielt zu implementieren.

Zudem wurden verschiedene Testfälle durchgeführt, um das Verhalten des Systems bei Fehlern während des Speichern und Laden zu überprüfen. Zu diesen Fehlern gehören plötzliche Abbrüche oder Datenkorruption. Um Abbrüche zu simulieren, wurde die Funktion 'Application.Quit()' an verschiedenen Stellen des Speicherprozesses verwendet. Wurde der Abbruch direkt nach dem asynchronen Schreiben der Daten auf die Festplatte ausgelöst, konnte der Schreibvorgang ohne Datenkorruption abgeschlossen werden. Erfolgt der Abbruch jedoch unmittelbar vor dem eigentlichen Schreiben auf die Festplatte, traten in einigen Fällen korrupte Daten auf.

Beschädigte Save-Dateien wurden durch die implementierte Integritätsprüfung erfolgreich erkannt, und der Ladeprozess wurde in diesen Fällen abgebrochen. Dies ist allerdings nur der Fall, wenn der Integritätstest verwendet wird und der Integritätskey selbst nicht beschädigt ist.

Obwohl das System in vielen Fällen Fehlermeldungen korrekt ausgibt, erkennt es nicht immer zuverlässig fehlerhafte Referenzen zu Assets. In einigen Fällen, in denen Referenzen nicht korrekt gesetzt waren, wurden keine Fehlermeldungen ausgegeben, was zu einer unvollständigen Fehlererkennung führte.

5.3 Bewertung der nicht-funktionalen Anforderungen

5.3.1 Dokumentation

Die Anforderung, eine umfassende Dokumentation für das Speichersystem bereitzustellen, wurde teilweise erfolgreich erfüllt. Die vorhandene Dokumentation deckt grundlegende Aspekte des Systems ab und bietet Entwicklern eine solide Grundlage für die Implementierung. Allerdings haben die Ergebnisse der User-Tests gezeigt, dass in bestimmten Bereichen Verbesserungsbedarf besteht. In der zweiten Aufgabe, die sich auf das komplexere Attribute-Saving konzentrierte, wurde deutlich, dass detailliertere Anweisungen benötigt werden, um Entwicklern eine klarere Anleitung zu bieten.

Optionen für Verbesserung

Die Teilnehmer des User-Tests betonten besonders, dass für die erhöhte Komplexität des zweiten User-Tests die Dokumentation klarer und verständlicher aufbereitet werden muss, um Missverständnisse zu vermeiden und den Einstieg für Entwickler zu vereinfachen.

Für das neue Konzept des Attribute-Savings, der nun stärker an Unitys Serializable-System angelehnt ist, muss zudem die Dokumentation grundlegend überarbeitet werden. Ein Teilnehmer der User-Tests wies auch ohne dieses neue Konzept darauf hin, dass hervorgehoben werden muss, dass das Speichersystem unabhängig vom Serializable-System funktioniert. Diese Unterscheidung ist mit dem neuen Konzept besonders wichtig, da beide Systeme strukturelle Ähnlichkeiten aufweisen, jedoch unterschiedliche Anwendungsbereiche und Einschränkungen haben.

Zusätzlich sollte die Dokumentation durch mehrere Beispielprojekte ergänzt werden. Diese Praxisbeispiele könnten Entwicklern helfen, das System in verschiedenen Anwendungsszenarien

zu verstehen und schneller in die Arbeitsweise des Speichersystems einzutauchen. Dies würde nicht nur die Effizienz der Implementierung erhöhen, sondern auch die Lernkurve für neue Nutzer deutlich abflachen.

5.3.2 Performace

Zur Bewertung der Performance wurden sowohl die Geschwindigkeit des Erstellens von Snapshots als auch das Laden dieser Snapshots in die Szene gemessen. Hierfür wurde das Inventarsystem mit dem IL2CPP-Backend in eine ausführbare Anwendung kompiliert. IL2CPP (Intermediate Language to C++) ist ein von Unity entwickeltes Backend, das C#-Code zunächst in eine Zwischensprache (Intermediate Language, IL) und anschließend in C++ übersetzt. Durch die Kombination von C++ mit Ahead-of-Time (AOT) Compiling kann eine gesteigerte Performance erzielt werden und die Anwendung lässt sich präzise messen. Eine alternative Option wäre die Verwendung des Mono-Backends gewesen. Dieses basiert jedoch auf Just-in-Time (JIT) Compiling, wodurch während des ersten Durchlaufs zusätzliche Rechenzeit für die JIT-Übersetzung erforderlich wäre, was die Messung der Ausführungszeiten erschwert. Aus diesem Grund wurde das IL2CPP Backend verwendet.

Es wurde bewusst entschieden, das Schreiben und Lesen der Daten auf der Festplatte nicht in die Performance-Tests einzubeziehen, da diese Operationen nicht innerhalb eines einzelnen Frames ausgeführt werden und somit den Haupt-Thread nicht blockieren. Daher wären sie für die Bewertung der Performance im Rahmen der definierten Anforderung nicht aussagekräftig.

Die Performance-Tests wurden auf einem System mit den folgenden Spezifikationen durchgeführt:

- CPU: Intel i5-13600KF 3.50 GHz
- RAM: 32,0 GB DDR4
- GPU: MSI GeForce RTX 4070

Ergebnisse

Zunächst ist die Komplexität der zweiten Aufgabe zu analysieren und zu definieren. Diese Aufgabe verwendet hauptsächlich das Attribute-Saving. Für das Speichern von Parents der UI-Items wird hingegen das Component-Saving verwendet. Dies ist notwendig, um sicherzustellen, dass die instanziierten Prefabs als Child des Parents korrekt dargestellt werden. Neben den Parents wird das Dynamic-Prefab-Saving für die UI-Item Prefabs angewendet. Zusätzlich werden alle Items gespeichert. Die Referenz zu diesen Items wird sowohl in einem Inventar ScriptableObject als auch auf den instanziierten Prefabs selbst abgelegt.

In dieser Auswertung wurden die Speicher- und Ladezeiten sowie die Garbage Collection Allokationen (GC-Alloc) für das Speichern und Laden von 10, 100 und 1000 Objekten gemessen. Ziel der Analyse ist es, das Skalierungsverhalten dieser Operationen in Bezug auf Zeit und GC-Alloc zu analysieren.

Wie in Tabelle 5.3 dargestellt, steigt die Speicherdauer mit der Anzahl der Objekte. Von 10 auf 100 Objekte beträgt die Zunahme der Speicherdauer das 3,8-fache, während von 100 auf 1000 Objekte die Zunahme das 7,5-fache beträgt. Während das Speichern in den unteren Bereichen (10 bis 100 Objekte) effizienter skaliert, nähert es sich bei größeren Datenmengen (1000 Objekte) einem nahezu linearen Wachstum an. Die GC-Allocation zeigt eine deutliche Zunahme, wobei der

Anzahl der Objekte	GC Alloc Speichern	GC Alloc Laden
10 Objekte	354 KB	445 KB
100 Objekte	2,7 MB	3,9 MB
1000 Objekte	26,1 MB	41,5 MB

Tabelle 5.2: GC-Allokation für Speichern und Laden

Anzahl der Objekte	Test 1	Test 2	Test 3	Test 4	Test 5	Durchschnitt
10 Objekte	3 ms	2 ms	2 ms	2 ms	2 ms	2,2 ms
100 Objekte	9 ms	8 ms	9 ms	8 ms	8 ms	8,4 ms
1000 Objekte	62 ms	63 ms	63 ms	63 ms	65 ms	63,2 ms

Tabelle 5.3: Speichergeschwindigkeit eines Snapshots

Umfang von 354 KB bei 10 Objekten auf 26,1 MB bei 1000 Objekten ansteigt. Die GC-Allokation erhöht sich dabei mit einem Faktor von etwa 7,6 zwischen 10 und 100 Objekten und 9,7 zwischen 100 und 1000 Objekten, was fast linear ist.

Anzahl der Objekte	Test 1	Test 2	Test 3	Test 4	Test 5	Durchschnitt
10 Objekte	6 ms	6 ms	6 ms	7 ms	6 ms	6,2 ms
100 Objekte	46 ms	46 ms	50 ms	45 ms	48 ms	47,0 ms
1000 Objekte	427 ms	426 ms	419 ms	433 ms	428 ms	426,6 ms

Tabelle 5.4: Ladegeschwindigkeit eines Snapshots

Die Ladezeiten steigen ebenfalls signifikant mit der Anzahl der Objekte. Von 10 auf 100 Objekte nimmt die Ladezeit um das 7,6-fache zu, und von 100 auf 1000 Objekte um das 9,1-fache. Die GC-Alloc beim Laden zeigt eine noch größere Zunahme: Sie steigt von 445 KB bei 10 Objekten auf 41,5 MB bei 1000 Objekten. Dabei vergrößert sich die GC-Alloc zwischen 10 und 100 Objekten um den Faktor 8,8 und zwischen 100 und 1000 Objekten um den Faktor 10,6.

Zusätzlicher Optimierungstest

In der ursprünglichen Testumgebung wurde überwiegend das Attribute-Saving verwendet, das stark auf Reflection basiert. Reflection bringt jedoch eine schlechte Performance mit sich. Sowohl im Attribute-Saving als auch im Component-Saving wurde Reflection eingesetzt, obwohl es für das Component-Saving nicht erforderlich war. Im Rahmen der Optimierungen wurde das Attribute-Saving vollständig durch das Component-Saving ersetzt, und der Algorithmus wurde so angepasst, dass beim Laden keine Reflection mehr erforderlich ist. Im bisherigen Algorithmus wurde ein Lookup-Table rekursiv aufgebaut, der alle relevanten Objekte umfassen sollte. Durch die Modifikation des ausschließlichen Component-Savings konnte dieser Prozess der Lookup-Erstellung deutlich vereinfacht werden. Diese Maßnahme war allerdings provisorisch und diente primär dazu, die Auswirkungen von Reflection auf die Speicher- und Ladeprozesse zu testen.

Bei 10 Objekten sank die Speicherdauer von 2,2 ms auf 2,0 ms, was einer Verbesserung von

Anzahl der Objekte	Test 1	Test 2	Test 3	Test 4	Test 5	Durchschnitt
10 Objekte	2 ms	2 ms	2 ms	2 ms	2 ms	2,0 ms
100 Objekte	5 ms	5 ms	5 ms	4 ms	4 ms	4,6 ms
1000 Objekte	25 ms	23 ms	23 ms	23 ms	23 ms	23,4 ms

Tabelle 5.5: Optimierte Speichergeschwindigkeit eines Snapshots

9,1% entspricht, da die ursprüngliche Methode bereits effizient war. Bei 100 Objekten wurde die Speicherdauer um 45,2% auf 4,6 ms reduziert, was zeigt, dass die Optimierung hier deutliche Auswirkungen hatte. Die größte Verbesserung trat bei 1000 Objekten auf, wo die Speicherdauer um 63,0% auf 23,4 ms sank, was den vollen Nutzen der Umstellung auf Component-Saving und die Entfernung von Reflection verdeutlicht. Dies ist in Tabelle 5.5 dargestellt.

Anzahl der Objekte	GC Alloc Speichern
10 Objekte	89 KB
100 Objekte	0,6 MB
1000 Objekte	6 MB

Tabelle 5.6: GC-Allokation für optimiertes Speichern

Die GC-Allokation verringerte sich bei 10 Objekten um 74,9% auf 89 KB, was zeigt, dass die Optimierung auch bei kleinen Datenmengen den Speicherverbrauch deutlich senkt. Bei 100 Objekten reduzierte sich die Allokation um 77,8% auf 0,6 MB, was auf die Eliminierung von Reflection zurückzuführen ist. Wie in Tabelle 5.6 dargestellt, sank bei 1000 Objekten die GC-Allokation um 77,0% auf 6,0 MB, was die Wirksamkeit der Optimierung in speicherintensiven Szenarien unterstreicht.

Schlussfolgerung

Die durchgeführten Optimierungen, insbesondere der Wechsel von Reflection zu Component-Saving, führten zu signifikanten Verbesserungen in der Performance des Systems. Der Verzicht auf Reflection reduzierte nicht nur die Ausführungszeiten, sondern auch die Menge an GC-Allokationen, was vor allem durch den Wegfall des rekursiven Lookup-Tables erreicht wurde. Dies zeigt, dass bereits durch provisorische Anpassungen erhebliche Leistungsgewinne erzielt werden konnten, und deutet darauf hin, dass weitere Optimierungen in diesem Bereich möglich sind. Gleichzeitig ist das Laden von Objekten deutlich zeitintensiver und benötigt eine höhere GC-Alloc als das Speichern. Dies liegt an der Wiederherstellung von Referenzen und dem Dynamic-Prefab-Loading. Gerade in Anbetracht der möglichen Optimierung des Speicherns lässt den Schluss zu, dass auch im Ladevorgang erhebliche Optimierungspotenziale bestehen, die durch eine effizientere Handhabung der Wiederherstellung von Objekten realisiert werden kann.

Im Kontext der Auswertung der Anforderungen der Performance wurden asynchrone Prozesse für das Schreiben und Laden auf der Festplatte implementiert, die durch eine Queue mit dem Snapshot-System ergänzt werden. Bei der unoptimierten Speicherversion konnten im Inventarsystem 100 Items mit einer durchschnittlichen Rechenzeit von 8,4 ms gespeichert werden. Obwohl dieser Wert unter der kritischen Schwelle von 16 ms liegt, die für eine flüssige Bildwiederholrate

von 60 FPS erforderlich ist, muss berücksichtigt werden, dass zusätzlich weitere rechenintensive Aufgaben wie das Rendern der Grafik und die Gameplay-Berechnungen durchgeführt werden müssen. Diese Belastungen können die Gesamtperformance erheblich beeinflussen. Daher sind, obwohl die grundlegende Anforderung erfüllt wurde, weitere Optimierungen notwendig, um das System unter hoher Last nachhaltig performant zu halten.

5.3.3 Speicheroptimierung

Das System muss den Speicherbedarf durch den Einsatz effizienter Datenstrukturen, optimierter Algorithmen und geeigneter Kompressionsstrategien minimieren. Zur Validierung dieser Anforderung wurde der Speicherverbrauch anhand von Aufgabe 2 der User-Tests sowohl mit als auch ohne Kompression untersucht. Die Ergebnisse der Messung des Speicherverbrauchs, wie in Tabelle 5.7 dargestellt, beziehen sich ausschließlich auf die Save-Datei und berücksichtigen nicht die Meta-Datei des Speichersystems. Die Messungen zeigen eine deutliche Reduzierung des Speicherbedarfs durch die Anwendung der Gzip-Kompression. Bei 10 Objekten konnte der Speicherbedarf von 19 KB auf 2 KB reduziert werden, bei 100 Objekten von 169 KB auf 9 KB und bei 1000 Objekten von 1.667 KB auf 123 KB. Diese Reduzierung des Speicherverbrauchs um bis zu 93% bestätigt die Effektivität der implementierten Kompressionsstrategien. Damit wurde die Anforderung der Speicheroptimierung durch die erfolgreiche Anwendung der Gzip-Kompression vollständig erfüllt.

Objekte	One Kompression	Mit Kompression
10 Objekte	19kb	2kb
100 Objekte	169kb	9kb
1000 Objekte	1.667kb	123kb

Tabelle 5.7: Speicherbedarf für Objekte ohne und mit Gzip-Kompression

5.3.4 Security

Angesichts der potenziellen Risiken, die mit der Deserialisierung komplexer Datenstrukturen einhergehen, wie etwa die Gefahr von RCE bei Deserialisierungen, wurde besonderer Wert auf die Implementierung robuster Sicherheitsmechanismen gelegt.

Um zu verhindern, dass gespeicherte Daten manipuliert werden können, nutzt das System optional AES-Verschlüsselung. Entwickler können den Verschlüsselungsschlüssel und das Salt über den Inspector des SaveLoadManagers konfigurieren oder alternativ einen Cache verwenden, der eine anpassbare Übergabe dieser Werte ermöglicht. Ergänzend wurden Integritätsprüfungen implementiert, um manipulierte Daten zu erkennen und deren Verarbeitung zu verhindern.

Darüber hinaus wurde das JSON-Format aufgrund seiner höheren Sicherheit und seiner plattformübergreifenden Kompatibilität eingeführt. Zusätzlich wird durch die Verwendung von IL2CPP-Unterstützung eine besonders schwierige Dekompilation der Skripte erreicht. Dies erhöht die Sicherheit allerdings nur, wenn der Schlüssel und das Salt mindestens über eigenen Code injiziert werden.

Eine Einschränkung besteht in der Notwendigkeit, sicherzustellen, dass nur festgelegte Datentypen innerhalb der gespeicherten Dateien unterstützt werden. Aktuell wird der QualifiedAssemblyName in der Save-Datei abgespeichert. Um die Risiken durch die Manipulation von dem QualifiedAssemblyName zu minimieren, dürfen potenziell gefährliche Objekte nicht deserialisiert werden. Derzeit fehlt jedoch eine solche Funktion im System, der ausschließlich die für das Speichern markierte Datentypen akzeptiert und somit die Deserialisierung von unautorisierten Typen verhindert. Da dieser wesentliche Sicherheitsmechanismus noch nicht implementiert wurde, ist die Anforderung zur Sicherstellung der Datensicherheit nur teilweise erfüllt.

Kapitel 6

Fazit

Im Rahmen dieser Arbeit wurde ein Speichersystem für die Unity-Engine entworfen und entwickelt, wobei der Schwerpunkt auf den Aspekten der Entwicklerfreundlichkeit, Erweiterbarkeit und Flexibilität lag. Ziel der Entwicklung war es, ein System zu konzipieren, das sowohl für einfache als auch für komplexe Spiele effizient eingesetzt werden kann. Das System sollte als Open-Source-Projekt veröffentlicht werden, um es Entwicklern zu ermöglichen, sich mit dessen Funktionsweise vertraut zu machen, es anzupassen und für eigene Projekte weiterzuentwickeln. Zur Erfüllung dieser Anforderungen wurde folgende zentrale Forschungsfrage formuliert:

Wie muss ein Speichersystem in der Entwicklungsumgebung Unity konzipiert werden, um eine entwicklerfreundliche Erfahrung zu gewährleisten, welche sowohl für einfache als auch komplexe Spiele unterstützt wird?

Zur Beantwortung dieser Frage wurde im theoretischen Teil dieser Arbeit zunächst der theoretische Hintergrund umfassend dargestellt. Darauf aufbauend wurden Experteninterviews durchgeführt, deren Ergebnisse in konkrete Anforderungen an das Speichersystem überführt wurden. Diese Anforderungen wurden gezielt so formuliert, dass sie die Überprüfung der aufgestellten Hypothesen ermöglichen. Auf Basis dieser Anforderungen erfolgte eine umfassende Analyse von Good Practices und Bad Practices für existierende Frameworks. Anschließend wurde ein Speichersystem implementiert und das Konzept im Rahmen dieser Arbeit umfassend dargestellt. Abschließend wurde das entwickelte Framework durch gezielte Tests evaluiert und im Hinblick auf die zuvor definierten Anforderungen bewertet.

6.1 Zusammenfassung der Ergebnisse

Im Rahmen der Untersuchung zur Entwicklerfreundlichkeit der API wurden die Hypothesen eingehend evaluiert. Die Analyse des Attribute-Savings, welche auf Hypothese H1 basiert, ergab einen SUS-Score von 63, der leicht unter dem durchschnittlichen Wert von 68 liegt. Dies ist teilweise darauf zurückzuführen, dass von den Teilnehmenden der User-Tests neben der Programmierung auch das Referenzieren von Assets und Callbacks als verbesserungswürdig empfunden wurde. Die Nachvollziehbarkeit des Attribute-Savings wurde von den Testern kritisch bewertet, und es wurde empfohlen, auf bewährte Ansätze zurückzugreifen. Trotz dieser Schwächen wurde von den Testern mehrfach bestätigt, dass diese Methode für einfache Spiele eine schnelle und effiziente Lösung darstellen kann.

Im Vergleich dazu erwies sich das Component-Saving, welche auf Hypothese H2 basiert, als deutlich entwicklerfreundlicher, was durch einen SUS-Score von 78,5 in der ersten Aufgabe des User-Tests belegt wird. Da diese Aufgabe jedoch kein Programmieren beinhaltete, kann diese Aussage nur getroffen werden, wenn die Components des Component-Savings bereits vorhanden und gut dokumentiert sind. Besonders die Modularität dieses Ansatzes ermöglicht eine umfangreiche Skalierbarkeit für sowohl einfache als auch komplexe Spiele. Da dieser Ansatz im Vergleich zum Attribute-Saving eine größere Flexibilität in der Implementierung bietet, ist er gleichzeitig mit einem höheren Implementierungsaufwand verbunden. Zudem wurden die Performance-Vorteile bestätigt, die sich aus der Nichtverwendung von Reflection ergeben.

Die Hypothese H3 im Hinblick auf die Anforderung der Komplexität von Datenstrukturen untersucht und zeigte dabei signifikante Vorteile, da durch diese Unterstützung komplexe Objekthierarchien stark vereinfacht werden können. Experteninterviews bestätigten, dass dieser Ansatz die Implementierung erheblich vereinfacht. Durch die Automatisierung des hinzufügens von Assets zur Asset-Registry kann dies weiter optimiert werden.

Neben den Hypothesen haben sich im Rahmen der technischen Erkenntnisse zentrale Konzepte herausgestellt. Eine der wichtigsten war die Differenzierung zwischen Reference-Saving und Value-Saving, die sich als essenziell für die Architektur des Speichersystems erwies. Diese Unterscheidung ermöglicht eine flexible Balance zwischen Datenintegrität und Performance, je nach Anwendungsanforderung. Ebenfalls wurde die Notwendigkeit von GUIDs zur eindeutigen Identifikation von Objekten hervorgehoben, um eine korrekte Zuordnung und Wiederherstellung beim Speichern und Laden zu gewährleisten. Zur effizienten Handhabung komplexer Datenstrukturen hat sich die Strategie bewährt, Objekte vollständig zu instanziieren, bevor ihre Referenzen aufgebaut werden. Dies vereinfachte die Initialisierungsreihenfolge und verhinderte Probleme mit Referenzen auf noch nicht existierenden Objekten. Besonders relevant war dies bei der Instanziierung von Prefabs, die häufig in größeren Mengen und komplexen Hierarchien erzeugt werden. Zuletzt ist der entwickelte Type-Converter zu nennen. Dieser wurde aus den Prinzipien des Component-Savings abgeleitet und bietet eine Lösung für die Speicherung unveränderbarer Klassen.

Im Vergleich zu existierenden Lösungen bieten kostenfreie Anwendungen kaum oder gar keine Unterstützung für Referenzen. Darüber hinaus fehlt selbst bei kostenpflichtigen Tools häufig die Möglichkeit zur Versionierung von Speicherdaten, was einen zentralen Aspekt dieser Arbeit ausmacht. Um diese Lücke zu schließen, wurde im Rahmen dieser Arbeit einerseits ein Konzept zur Versionierung von Speicherdaten entwickelt und andererseits ein Framework entwickelt, welches das Problem der Referenzen löst.

6.2 Limitationen der Arbeit

Eine wesentliche methodische Limitation dieser Arbeit liegt in der geringen Anzahl durchgeführter Experteninterviews und User-Tests. Obwohl die gewonnenen Erkenntnisse wertvolle und praxisnahe Einsichten lieferten, wäre eine größere Stichprobe erforderlich gewesen, um eine breitere und differenziertere Perspektive auf die Anforderungen an ein Speichersystem in der Spieleentwicklung zu erhalten. Besonders ins Gewicht fällt die niedrige Teilnahmequote bei den User-Tests, an denen lediglich 5 von 17 eingeladenen Personen teilnahmen, was die Aussagekraft der Ergebnisse

deutlich einschränkt.

Ein weiterer methodischer Aspekt betrifft die Durchführung der Experteninterviews. Eine Aufnahme und anschließende Transkription der Interviews hätte tiefere Einblicke ermöglicht, konnte jedoch aus zeitlichen und organisatorischen Gründen nicht realisiert werden. Zudem wäre eine detailliertere Erörterung der Architekturen für Attribute-Saving und Component-Saving im Rahmen der Experteninterviews sinnvoll gewesen. Dies hätte gezielte Ergebnisse in Bezug auf die Hypothesen H1 und H2 geliefert.

Ferner ist die Implementierung von Component-Saving in den User-Tests eine weitere Schwachstelle. Es wurden keine praktischen Implementierungen getestet, weshalb nur Aussagen auf Basis vorgefertigter Components getroffen werden konnten. Dies reduziert die Aussagekraft hinsichtlich der praktischen Anwendbarkeit und Effizienz dieser Architektur in realen Szenarien.

Technologische Einschränkungen betreffen die plattformübergreifenden Tests, die ausschließlich auf Windows und Android durchgeführt wurden. Diese Begrenzung mindert die Aussagekraft in Bezug auf die allgemeine Nutzbarkeit und Performance des entwickelten Speichersystems auf anderen Plattformen wie iOS, Konsolen oder Webumgebungen.

Ein weiterer limitierender Faktor war der begrenzte Zeitrahmen, weshalb das Versioning lediglich auf konzeptioneller Ebene behandelt und nicht vollständig implementiert werden konnte.

6.3 Ausblick

Im Rahmen dieser Arbeit wurde bewusst auf die Speicherung von Daten in Verbindung mit Servern verzichtet, da die damit verbundenen Herausforderungen in Bezug auf Netzwerke, Datenbanken und Synchronisation eine hohe Komplexität und Spezifizität aufweisen, die den Rahmen dieser Untersuchung sprengen würden. Die durchgeführten Experteninterviews haben zudem gezeigt, dass Entwickler in diesem Bereich häufig auf maßgeschneiderte, individuelle Lösungen setzen. Dennoch könnte dieses Thema in zukünftigen Forschungsarbeiten thematisiert werden. Dabei wäre vor allem der Aspekt der netzwerkbasierten Speicherung und Synchronisation von Spieldaten von Interesse. Hierbei wäre es von besonderer Bedeutung, Mechanismen zu entwickeln, die eine nahtlose Synchronisation von Spielständen zwischen mehreren Spielern und Geräten gewährleisten.

Um das in dieser Arbeit entwickelte Framework der breiten Entwickler-Community vorzustellen, müssen die in den User-Tests identifizierten Usability-Probleme behoben werden. Eine ausschlaggebende Maßnahme besteht darin, das Attribute-Saving System des Frameworks an das Konzept des Unity-Serialization anzupassen. Dies würde die Lernkurve für Entwickler verkürzen und die Verbreitung des Frameworks innerhalb der Unity-Community fördern, da vertraute Konzepte genutzt werden könnten. Ein weiteres Optimierungsziel ist das automatische Hinzufügen von Assets in die Asset-Registry, um die manuelle Verwaltung der speicherbaren Assets zu erleichtern. Zudem stellt die Implementierung robuster Fehlerbehandlungsmechanismen ein zentrales Thema dar, da erweiterte Mechanismen dazu beitragen könnten, Datenverluste bei Spielabstürzen zu minimieren. Neben der Fehlerbehandlung gehören hierbei auf erweiterte und bessere Fehlermeldungen dazu. Zusätzlich sollte das Konzept zur Versionierung für das Speichersystem vollständig im Framework implementiert werden, um die Konsistenz und Integrität bei Updates eines Spiels

zu gewährleisten. Neben diesen funktionalen Erweiterungen besteht erhebliches Potenzial in der Performance-Optimierung des Frameworks. Dies ist besonders bei ressourcenintensiven Spielen notwendig.

Neben diesen grundlegenden Zielen gibt es weitere Innovationen, durch die das Sperichersystem verbessert werden kann. Zukünftige Arbeiten könnten sich beispielsweise auf die Erweiterung des Frameworks um cloud-basierte Speicherlösungen konzentrieren. Ein weiterer Ansatz zur Verbesserung der Benutzerfreundlichkeit wäre die Implementierung einer grafischen Benutzeroberfläche, die die einfache Markierung von Objekten für das Speichern ermöglicht. Dies würde insbesondere weniger erfahrenen Entwicklern helfen, die Einstiegshürde zu senken und das Framework insgesamt intuitiver zu gestalten.

6.4 Schlusswort

Die vorliegende Arbeit hat gezeigt, dass die Entwicklung eines flexiblen und entwicklerfreundlichen Speichersystems für die Unity-Engine nicht nur eine technische Herausforderung darstellt, sondern auch einen bedeutenden Beitrag zur Verbesserung der Spieleentwicklungsprozesse leisten kann. Die Verbindung von praxisnaher Entwicklung und theoretischer Analyse führte zu einem System, das sowohl die Anforderungen einfacher als auch komplexer Spiele erfüllt. Durch die Kombination von Attribute- und Component-Saving, der Implementierung von Referenzen und der Berücksichtigung von Versionierungsmechanismen bietet das entwickelte Framework eine solide Grundlage, auf der zukünftige Erweiterungen und Optimierungen aufbauen können.

Literatur

[1] Ernest Adams und Andrew Rollings. *Fundamentals of game design*. 2nd ed. Voices that matter. Berkeley, CA: New Riders, 2010. 675 S. ISBN: 978-0-321-64337-7.

[2] AlexMeesters. *GitHub - AlexMeesters/Component-Save-System: Save system that is made to co-exist with the component system of Unity.* URL: https://github.com/AlexMeesters/Component-Save-System (besucht am 05. 09. 2024).

[3] coryleach. *GitHub - coryleach/UnitySaveLoad: Quickly Save/Load data in Binary or Json formats, and Encrypt it in Unity*. URL: https://github.com/coryleach/UnitySaveLoad (besucht am 05. 09. 2024).

[4] DerKekser. *GitHub - DerKekser/unity-save-system: Save System is a simple save system for Unity*. URL: https://github.com/DerKekser/unity-save-system?tab=readme-ov-file (besucht am 05. 09. 2024).

[5] *Deserialisierungsrisiken bei der Verwendung von BinaryFormatter und verwandten Typen - .NET — Microsoft Learn*. URL: https://learn.microsoft.com/de-de/dotnet/standard/serialization/binaryformatter-security-guide (besucht am 03. 09. 2024).

[6] *Deserialization - OWASP Cheat Sheet Series*. URL: https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html (besucht am 03. 09. 2024).

[7] Hasan Dibas und Khair Eddin Sabri. „A comprehensive performance empirical study of the symmetric algorithms:AES, 3DES, Blowfish and Twofish“. In: *2021 International Conference on Information Technology (ICIT)*. 2021 International Conference on Information Technology (ICIT). Amman, Jordan: IEEE, 14. Juli 2021, S. 344–349. ISBN: 978-1-66542-870-5. DOI: 10.1109/ICIT52682.2021.9491644. URL: https://ieeexplore.ieee.org/document/9491644/ (besucht am 04. 09. 2024).

[8] gewarren. *Reflektion in .NET — Microsoft Learn*. 5. Apr. 2024. URL: https://learn.microsoft.com/de-de/dotnet/fundamentals/reflection/reflection (besucht am 25. 09. 2024).

[9] Steve Goldstein. *Game Engine Software Statistics 2024*. 2024. URL: https://llcbuddy.com/data/game-engine-software-statistics/ (besucht am 23. 09. 2024).

[10] Konrad Grochowski, Michał Breiter und Robert Nowak. *Serialization in Object-Oriented Programming Languages — IntechOpen*. 29. Aug. 2019. URL: https://www.intechopen.com/chapters/68840 (besucht am 05. 06. 2024).

[11] Clayton Industries. *Quick Save*. URL: https://assetstore.unity.com/packages/tools/utilities/quick-save-107676#description.

[12] jakobdufault. *GitHub - jacobdufault/fullserializer: A robust JSON serialization framework that just works with support for all major Unity export platforms.* URL: https://github.com/jacobdufault/fullserializer (besucht am 05. 09. 2024).

[13] Robin Jaspers. *SerializationProcess*. Google Drive. Online: https://drive.google.com/file/d/1SeUpWTOf8nenOWUEPIjXWQO2sFTpGkQi/view?usp=sharing. 2024.

[14] Henry E. Lowood, Raiford Guins und A. C. Deger. *Debugging game history: a critical lexicon*. Game histories. Cambridge, Mass: The MIT press, 2016. ISBN: 978-0-262-03419-7.

[15] Philipp Mayring und Thomas Fenzl. „Qualitative Inhaltsanalyse“. In: *Handbuch Methoden der empirischen Sozialforschung*. Hrsg. von Nina Baur und Jörg Blasius. Wiesbaden: Springer Fachmedien Wiesbaden, 2019, S. 633–648. ISBN: 978-3-658-21308-4. DOI: 10.1007/978-3-658-21308-4_42. URL: https://doi.org/10.1007/978-3-658-21308-4_42.

[16] Moodkie. *Easy Save - The Complete Save Data & Serializer System*. URL: https://assetstore.unity.com/packages/tools/utilities/easy-save-the-complete-save-data-serializer-system-768.

[17] Robert Nystrom. *Game Programming Patterns*. 1. Auflage. Online: https://gameprogrammingpatterns.com/. Genever Benning, 2014. ISBN: 978-0990582908.

[18] Ömer. *Electronics — Free Full-Text — A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions*. URL: https://www.mdpi.com/2079-9292/12/6/1333 (besucht am 04. 09. 2024).

[19] Fergus U Onu u. a. „Security Vulnerabilities in Some Popular Object-Oriented Programming Languages – Forecast of the Danger Ahead“. In: 16 (2023).

[20] *OWASP Developer Guide — Principles of Cryptography — OWASP Foundation*. URL: https://owasp.org/www-project-developer-guide/draft/foundations/crypto_principles/ (besucht am 04. 09. 2024).

[21] *Refactoring and Design Patterns*. URL: https://refactoring.guru/ (besucht am 05. 09. 2024).

[22] Vaskaran Sarcar. *Java Design Patterns*. Berkeley, CA: Apress, 2016. ISBN: 978-1-4842-1801-3 978-1-4842-1802-0. DOI: 10.1007/978-1-4842-1802-0. URL: http://link.springer.com/10.1007/978-1-4842-1802-0 (besucht am 04. 09. 2024).

[23] Jeff Sauro. *5 Ways to Interpret a SUS Score – MeasuringU*. URL: https://measuringu.com/interpret-sus-score/ (besucht am 18. 09. 2024).

[24] *Secrets Management - OWASP Cheat Sheet Series*. URL: https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html (besucht am 04. 09. 2024).

[25] *Semantic Versioning 2.0.0 — Semantic Versioning*. URL: https://semver.org/lang/de/ (besucht am 15. 09. 2024).

[26] SteveSmith.Software. *Save for Unity Complete*. URL: https://assetstore.unity.com/packages/tools/utilities/save-for-unity-complete-244507#description.

[27] Gollachut Studios. *Easy Editor Save*. URL: https://assetstore.unity.com/packages/tools/utilities/easy-editor-save-257189.

[28] Audie Sumaray und S. Kami Makki. „A comparison of data serialization formats for optimal efficiency on a mobile platform“. In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. ICUIMC '12: The 6th International Conference on Ubiquitous Information Management and Communication. Kuala Lumpur Malaysia: ACM, 20. Feb. 2012, S. 1–6. ISBN: 978-1-4503-1172-4. DOI: 10.1145/2184751.2184810. URL: https://dl.acm.org/doi/10.1145/2184751.2184810 (besucht am 04. 06. 2024).

[29] *SUS — German UPA*. URL: https://germanupa.de/wissen/fragebogenmatrix/sus (besucht am 16. 09. 2024).

[30] Unity Technologies. *Eine kurze Einführung in den Unity Asset Store*. Website. Online: https://unity3d.com/de/quick-guide-to-unity-asset-store. o.D.

[31] Unity Technologies. *Unity Documentation*. Website. Online: https://docs.unity3d.com/Manual/index.html. 2022.

[32] Unity Technologies. *unity-scriptable-object-players-health-points*. Website. Online: https://unity.com/de/how-to/architect-game-code-scriptable-objects. 2020.

[33] *The most powerful real-time 3D creation tool - Unreal Engine*. URL: https://www.unrealengine.com/en-US (besucht am 05. 09. 2024).

Anhang

.1 GitHub Repository

Das Speichersystem wurde in Unity entwickelt. Neben dem Speichersystem ist als Beispielanwendungsfall ein Inventarsystem vorhanden, welches zudem Aufschluss über die Performance des Systems bietet. Der Code ist unter folgendem GitHub-Repository verfügbar:

`https://github.com/AyuCalices/MasterThesisSaveLoad.git`

Das Repository enthält den vollständigen Quellcode sowie eine detaillierte Dokumentation des Projekts unter der MIT-Lizenz. Es wurde zur Speicherung und Verwaltung von Spielständen in Unity entwickelt und kann als Referenz oder zur Weiterentwicklung genutzt werden.

.2 Experteninterviews

Der Anhang der Experteninterviews umfasst sowohl die PDF-Dokumente der einzelnen Interviews als auch die Kategorisierung der Aussagen unter übergeordneten Themenbereichen, wobei die Kategorisierung von Miro auf ein Textdokument übertragen wurde, um eine bessere Lesbarkeit zu gewährleisten.

Umfrage Erfahrung Game Engines

Vielen Dank, dass Sie an dieser Befragung teilnehmen. Ziel dieser Befragung ist es, wertvolle Einblicke für die Entwicklung eines Save and Loading Systems in Unity3D unter Verwendung von C# zu gewinnen. Ihre Antworten werden vertraulich behandelt und anonymisiert ausgewertet. Die Befragung wird etwa 30 Minuten in Anspruch nehmen.

Demografische Fragen

1. Könnten Sie kurz Ihre Erfahrungen und Ihre Rolle in der Entwicklung mit Game Engines beschreiben?
    a. Alle haben erfahrungen mit Unity und demnach auch C#
    b. Mit der Unreal Engine hat nur eine person erfahrung (Julian)
    c. Alle arbeiten in der XR entwicklung
    d. Sebastian: Dozent, spezialisiert auf Analog Haptics und allgemein auf Game Development.
    e. Julian: Spezialisierung auf Serialisierung und Netzwerke.

Hauptfragen

Fragen zum existierenden Ansätzen

1. Welche aktuellen "Save and Loading"-Lösungen nutzen Sie derzeit in Ihren Projekten?
    a. Tools
        i. Externe Datenbanken
        ii. PlayerPrefs
        iii. Chronos glTF für programmierbare Daten.
    b. Serialisierungscontainer
        i. Serialisieren als CSV
        ii. Unity Json: Daten mit einem einzigen Befehl serialisieren.
        iii. Newtonsoft Json ist nicht komprimiert genug.
        iv. Json hat bei großen Dateigrößen immer noch einen vorhandenen Bezeichner (z.B. Feldname), was nicht nötig ist.
        v. Binär wird bevorzugt.
        vi. XML wird mehr für UI-Visualisierung und Datenpräsentation genutzt, weniger für Serialisierung. Es bietet jedoch breite Kompatibilität (ähnlich wie Json), obwohl es als alt angesehen wird.
        vii. Selbst erstellter Serialisierer
    c. Für Save and Loading wird lieber eine eigene Lösung entwickelt, wie z.B. ein Inventarisierungssystem. Fertige Assets werden nicht verwendet, da man lieber selbst schreibt und weiß, wie es funktioniert.
2. Hatten Sie in der Vergangenheit Schwierigkeiten oder Herausforderungen bei der Implementierung von Save and Load in Ihren Spielen? Wenn ja, welche?
    a. Rekursionstiefe und Cyclic Dependencies für Structures, Klassen mit Unity Json. Lösungsansätze:

        i. Ab einer bestimmten Anzahl an Abhängigkeiten hört man auf.
        ii. bei zyklischen objekten wird jedes objekt nur einmal berücksichtigt
        iii. Julian ignoriert alle zyklischen Abhängigkeiten und verwendet das "Newtonsoft cyclic dependency attribute".
    b. Probleme bei Datenbankverbindungen: HTTP-Request/WebSocket, API an Datenbank/Übertragung -> Webform -> Alles nur in Strings (unbeliebt) -> Datentypkonvertierung muss beachtet werden.
    c. Serialisierung/Deserialisierung ist mühselig -> Einheitliche Konvertierung ist notwendig.
    d. Idee zur Datensicherheit: Einzelne Felder verschlüsseln, die man sich aussuchen kann.
    e. Weitere Idee gegen Cheating: Waffen eine ID geben -> Ändern der Stats der Waffe wird erschwert.
    f. Beispiel aus existierenden Spielen: Clicker-Game-Daten werden in einer Art "PlayerPrefs" gespeichert -> Echtgeld-Währung kann ercheatet werden.
    g. Cheat-Engine: Kaum Erfahrung (Sebastian), nur Tutorials genutzt -> Musste nur funktionieren -> Eindruck von einem besseren Hack-Editor.

Anforderungen und Wünsche an ein neues System

1. Welche Features und Funktionen sind Ihnen bei einem "Save and Loading"-System besonders wichtig?
    a. Speicherplatz darf nicht explodieren, z.B. wie bei Minecraft, wo große Datenmengen entstehen -> Das System sollte schlank bleiben und konsistente/bessere Ladezeiten bieten.
    b. Datenkompression: Delta-Kompression für Formate wie Json.
    c. Einmal lesbar und dann noch binär -> Umwandlung zwischen binär und Json.
    d. Diese Daten sollten mit dem Kompressionslevel gespeichert werden -> Wie das geschieht, übernimmt das Save-and-Loading-System (z.B. Float-Kompression).
    e. Weitere Idee: Serialisierte Daten im Editor nutzen: Datensätze direkt im Inspektor sichtbar -> Felder im Editor sehen/bearbeiten -> im Bestehenden Workflow einbinden.
        i. dieser fall nützlich für editor (editor workflow)
        ii. ohne popup window: wo es gespeichert wird.
        iii. Werte für den Inspektor können gespeichert/geladen werden.
        iv. Während der Laufzeit den Stand speichern, um später wieder ladbar zu sein.
        v. Serialisierung für Import/Export / Neustart des Editors.
            1. serialize referenze?
    f. Versioning: Für die Editor-Zeit nicht notwendig, abhängig von der Anwendung Versionierung.
        i. Möglicherweise zu komplex: Komplette Szenen müssen gespeichert werden.
        ii. Git-Versionierung ist beliebter.
        iii. Bei einem zusätzlichen System -> Workflow-Overload.

iv. Vermutung: Versionierung ist mehr Applikationsdesign, wie ich und Entwickler das verwenden. Möglicherweise ohne Hierarchy-Problem nutzbar

Schlussfragen

1. Gibt es sonstige Aspekte, die Sie in Bezug zu dem Thema ansprechen möchten?
    a. auf einen use case konzentrieren
    b. C# klassen & referenzen von monoBehaviours nicht auf beides konzentrieren
    c. networking systeme machen das -> deterministische Randomness für hierarchy anwenden -> Netcode for Gameobject -> NetworkObject (system wie guid ids vergeben werden)
    d. Non prefab parent von objekt ändert sich bei laufzeit

Vielen Dank für Ihre Teilnahme und Ihre wertvollen Beiträge. Ihre Antworten werden helfen, ein neues und benutzerfreundliches System zu entwickeln.

Umfrage Erfahrung Game Engines

Vielen Dank, dass Sie an dieser Befragung teilnehmen. Ziel dieser Befragung ist es, wertvolle Einblicke für die Entwicklung eines Save and Loading Systems in Unity3D unter Verwendung von C# zu gewinnen. Ihre Antworten werden vertraulich behandelt und anonymisiert ausgewertet. Die Befragung wird etwa 30 Minuten in Anspruch nehmen.

Demografische Fragen

1. Könnten Sie kurz Ihre Erfahrungen und Ihre Rolle in der Entwicklung mit Game Engines beschreiben?
    a. CTO Mixed World
    b. Seit c.a. 15 Jahren arbeit mit game engines
    c. anfänglich Unity, Unreal Engine (damals 3), CryEngine
    d. Erstmal Desktop Orientiert -> später Filmentwicklung (tooling) -> verknüpft maya mit unreal -> eigene Game Engine mit Startup rein C++ -> ab etwa 2014 stärker richtung VR bereich und ab 2016 in mixed reality
    e. immer fokus auf entertainment aber hauptsächlich probleme lösen -> mehr non game themen mit game engine lösen -> zwangsläufig mit Unity weil mit HoloLens support

Hauptfragen

Fragen zum existierenden Ansätzen

- Welche aktuellen "Save and Loading"-Lösungen nutzen Sie derzeit in Ihren Projekten?
    a. Bereits von Beginn an intensiv mit Save and Load beschäftigt.
    b. Erste Erfahrungen mit der Unreal Engine, besonders in einem verteilten System, bei dem ein Großteil der Daten außerhalb der Engine gespeichert wurde -> Speicherung erfolgt über einen zentralen Ort wie eine Datenbank.
        i. Seit 2013 lag der Fokus darauf, dass alles, bis hin zu den tiefsten Ebenen, serialisierbar ist.
    c. Beim Übertragen auf die Hololens ist es besonders wichtig, darauf zu achten, was genau gespeichert wird und wie einfache Datenstrukturen gekapselt werden.
    d. Die Unreal Engine erleichtert Save/Load-Prozesse, da sie von der Gesamtstruktur her bereits gut vorbereitet ist -> Alle Datenstrukturen sind dadurch leicht serialisierbar.
    e. JSON wird in der Regel immer verwendet.
    f. Bei der Küchensoftware "Rooms" wurde der komplette Datenteil externalisiert -> Einige Teile sind lokal, aber der Großteil verweist auf Referenzen, wo z.B. Meshes und Texturen in einer Datenbank gespeichert werden. Der Fokus liegt auf asynchronen Prozessen.
    g. Thread-safe und komplett asynchrones Speichern und Laden.

    h. Fabians volumetrisches Video-Plugin stellt extreme Anforderungen an die Speicher- und Ladelösungen.
- Wie denkst du, können Entwickler Deserialisierungs Savety sicherstellen?
    a. Daten, die auf der Festplatte manipuliert werden, werden vom Deserialisierer überprüft, um sicherzustellen, dass keine ungewollten Änderungen vorgenommen wurden.
    b. Die Datenübertragung zwischen Server und Client ist generell abgesichert, da SSH verwendet wird.
    c. Der Client hat nur Leserechte, was die Sicherheit erhöht.
    d. Auf der Serverseite (C# ASP.NET) werden Datentypen überprüft, und auf der Clientseite stellt der Serialisierer sicher, dass die Daten korrekt getypt sind.
    e. Jeder Datenchunk muss einem bestimmten Datentyp entsprechen, was ebenfalls die Sicherheit und Konsistenz der Daten gewährleistet.
    f. Sobald man in unsichere Bereiche geht, wie z.B. Bereiche, die Pointer benötigen, wird es schwieriger, was jedoch selten vorkommt.
    g. In 90% der Fälle reicht die Verschlüsselung der Werte im Editor aus, und der Schlüssel wird nicht offengelegt.

Schlussfragen

1. Gibt es sonstige Aspekte, die Sie in Bezug zu dem Thema ansprechen möchten?
    a. Zunächst das System stabil machen und dann optimieren: Auch Aspekte wie Sicherheit sollten erst nach der grundlegenden Funktionsfähigkeit optimiert werden.
    b. Bei dynamischen Strukturen: Der Entwickler sollte die Möglichkeit haben, bestimmte Teile des Systems nach Bedarf zu aktualisieren.
    c. Datenstrukturen wie Meshes sollten extern gespeichert und geladen werden -> Asset Bundles werden zur Compilezeit für jede Plattform erstellt.
    d. Einschränkungen durch IL2CPP: Es gehen viele Möglichkeiten verloren -> Kann trotzdem alles zuverlässig gespeichert und geladen werden?

Vielen Dank für Ihre Teilnahme und Ihre wertvollen Beiträge. Ihre Antworten werden helfen, ein neues und benutzerfreundliches System zu entwickeln.

Umfrage Erfahrung Game Engines

Vielen Dank, dass Sie an dieser Befragung teilnehmen. Ziel dieser Befragung ist es, wertvolle Einblicke für die Entwicklung eines Save and Loading Systems in Unity3D unter Verwendung von C# zu gewinnen. Ihre Antworten werden vertraulich behandelt und anonymisiert ausgewertet. Die Befragung wird etwa 30 Minuten in Anspruch nehmen.

Demografische Fragen

1. Könnten Sie kurz Ihre Erfahrungen und Ihre Rolle in der Entwicklung mit Game Engines beschreiben?
    a. serialisierung
    b. networking
    c. mensch-maschinen-interaktion (HMI) -> konzeptionell &
    d. geschäftsführer von mixed world (kleines unternehmen)
    e. seit 2017 teilweise entwicklung an privaten netzwerk tool, mit wunsch dass es irgendwann andere entwickler einsetzen zu können
        i. immer wenn upgrade gebraucht wird wurde es auf bedarf angepasst
2. Welche Entwicklungsumgebung und -tools verwenden Sie hauptsächlich in Ihren Projekten?
    a. lieber eigener stack

Hauptfragen

Fragen zum existierenden Ansätzen

1. Welche Berührungspunkte hattest du zu Save and Loading?
    a. Früher mit Arduino arbeitete man oft über serielle Schnittstellen und Streams, bei denen nur Bytes gesendet werden. In der Programmiersprache C ist es ähnlich. Man arbeitet sehr nah an der Hardware und speichert Daten oft direkt als Bytes auf der Maschine.
    b. Das war vor der Zeit von JSON, einem Format, das heutzutage sehr beliebt ist.
    c. Wenn sich die gespeicherten Daten verändert haben, konnte man sie nicht mehr korrekt laden.
    d. JSON hat sich als Standard durchgesetzt, weil es flexibler und verständlicher ist.
    e. Robert findet PlayerPrefs problematisch, weil die Datentypen schwer nachvollziehbar sind, was es komplex macht, wenn man etwas nachträglich ändern möchte.
        i. lieber ganz konkret sagen: das objekt möchte ich speichern
    f. Eine Versionskontrolle für gespeicherte Daten ist wünschenswert, aber oft schwierig umzusetzen.
2. Hatten Sie in der Vergangenheit Schwierigkeiten oder Herausforderungen bei der Implementierung von Save and Load in Ihren Spielen? Wenn ja, welche?

a. Probleme bei der Kompatibilität zwischen verschiedenen Systemarchitekturen.
b. Genaues Wissen darüber, wohin die Daten gespeichert werden sollen: Zunächst in C programmiert, dann in einer Anwendung oder an einem allgemein zugänglichen Ort abgelegt -> Einfaches manuelles Übertragen der Daten.
c. Kulturelle Unterschiede bei der Verwendung von JSON und Endianess

Anforderungen und Wünsche an ein neues System

1. Welchen Ansatz würden Sie bevorzugen?
    a. Mehrere Ebenen: Sowohl die eigentlichen Daten als auch das Datenhandling sind wichtig.
        i. Query-basiertes Vorgehen: Daten abfragen und speichern.
        ii. Der Prozess: Ein bestimmter Datensatz wird identifiziert und soll abgespeichert werden. Inkludiert, ob die Daten überhaupt vorhanden sind.
        iii. Kontextbezogene Speicherung: Daten werden innerhalb eines bestimmten Kontextes gespeichert, der beim Laden wieder verwendet wird.
        iv. Vererbung (Inheritance) wird erwartet.
        v. Das System verwaltet sich selbst, ähnlich wie eine Baumstruktur. (Datentypen, Strukturen, Klassen, Kontext, Kollektionen)
        vi. Aus der MVC-Logik: Nur das Datenmodell und bestimmte Zustände von Klassen speichern, keine Meshes oder Assets -> Stattdessen nur Referenzen zu den Assets, wobei die Klassen wissen, wo sie die benötigten Ressourcen finden.
        vii. Alles Gespeicherte sollte in einer Datei sein, mit Ausnahme der Assets -> Dies setzt voraus, dass Referenzen zu den Assets vorhanden sind.
        viii. Flexibilität bei der Entscheidung, welche Informationen wo gespeichert werden sollen.
2. Würden Sie ein solches System auf Commands basieren lassen?
    a. Im Kontext von Networking
        i. Problematisch beim Networking: Manchmal verschwindet ein Command.
        ii. Commands können den Server überlasten -> Manche Commands kommen nicht an, werden doppelt gesendet oder einfach ignoriert.
    b. Zu gefährlich, da die Integrität des Systems leidet -> Das System kennt die Commands, aber nicht den aktuellen Zustand, und umgekehrt.
    c. Ursprungsidee ist, dass man sich damit was spart: ist das allerdings wirklich der fall?
    d. Ist es für den User einfach zu handhaben?
    e. Eine hybride Lösung könnte sehr interessant sein.
    f. Merken der letzten states für undo & redo und commands genutzt um states davon abzuleiten

g. Beispiel Starcraft: Ein Command-basiertes System, bei dem man nicht an eine beliebige Stelle springen kann -> Das war benutzerfreundlich, da es keine flexible Navigation ermöglichte.

Schlussfragen

1. Gibt es sonstige Aspekte, die Sie in Bezug zu dem Thema ansprechen möchten?
    a. wegen versioning:
        i. Innerhalb einer Szene sind alle Unity-IDs gültig.
        ii. Es gibt eine Äquivalenzliste: Immer wenn auf eine alte ID (z.B. a1) zugegriffen wird, wird diese durch die neue ID (z.B. a2) ersetzt -> Diese Ersetzung erfolgt erst beim Serialisieren, wo alte IDs durch neue ersetzt werden.
    b. Viele Entwickler haben eine Abneigung gegen Monobehaviours, aber die Serialisierung muss auch für diese Klassen funktionieren.
    c. Diese Abneigung kommt oft von Entwicklern, die ursprünglich nicht mit Unity angefangen haben.
    d. Wichtige Fragen: Wer ist die Zielgruppe?
    e. Was passiert, wenn jemand Java integriert hat?
    f. Idee: Künstliche Intelligenz (AI): Änderungen (Deltas) werden gespeichert, um Unterschiede festzuhalten.
        i. AI findet heraus, welche Deltas relevant sind.

Vielen Dank für Ihre Teilnahme und Ihre wertvollen Beiträge. Ihre Antworten werden helfen, ein neues und benutzerfreundliches System zu entwickeln.

Umfrage Erfahrung Game Engines

Vielen Dank, dass Sie an dieser Befragung teilnehmen. Ziel dieser Befragung ist es, wertvolle Einblicke für die Entwicklung eines Save and Loading Systems in Unity3D unter Verwendung von C# zu gewinnen. Ihre Antworten werden vertraulich behandelt und anonymisiert ausgewertet. Die Befragung wird etwa 30 Minuten in Anspruch nehmen.

Demografische Fragen

1. Könnten Sie kurz Ihre Erfahrungen und Ihre Rolle in der Entwicklung mit Game Engines beschreiben?
    a. Seit 2013 als Unity-Entwickler in verschiedenen Studios in Berlin tätig.
    b. Sehr viel Mobile Entwicklung, weil viele Studios in Berlin
    c. Beteiligt an 5 bis 10 Spielveröffentlichungen.
    d. Seit drei Jahren verstärkt auf die Entwicklung für die PC-Plattform fokussiert.
    e. Seit 2021 als Senior Developer aktiv, seit zweieinhalb Jahren in der Rolle des Lead Developers
    f. Arbeit an Kinderspielen für Mobilgeräte, meist kleinere Projekte mit einer Entwicklungsdauer von etwa sechs Monaten.
        i. Mitarbeit an einem Dinosaurier-Spiel für Playmobil.
        ii. Projekt „Arbo Idle Garden“ wurde für den Deutschen Computerspielpreis nominiert.
    g. Beteiligung an der Entwicklung eines VR-Spiels für Etermax.
    h. Letztes Studio musste aufgrund von Budget Problemen schließen.
    i. Derzeitige Tätigkeit in einem neuen Studio.

Hauptfragen

Fragen zum existierenden Ansätzen

1. Welche aktuellen "Save and Loading"-Lösungen nutzen Sie derzeit in Ihren Projekten?
    a. In Studios hat Michele eher selten das erste Spiel gemacht. Demnach waren meist custom solution vorhanden
    b. Die Save-Systeme die online support hatten waren meist schwerer und sind dadurch eher in erinnerung geblieben
        i. football manager: backend, dass automatisch mit unity interagiert -> javascript/mongoDB -> man kann direkt json speichern
    c. Anderer ansatz: viele kleine jsons
        i. json immer wieder iterieren und auf sachen draufhauen
    d. Amazon Datenbank: S3 Buckets -> file storages -> einfach daten reinwerfen
2. Was gefällt Ihnen an Ihrem aktuellen "Save and Loading"-System?
    a. Backups auf mobilen Geräten zu machen ist besonders geschätzt, dass es geht
    b. Dateigröße muss möglichst klein sein (u.a. für mobile)
    c. Aktuell: alles in einen einzelnen json -> durch inheritance reiht sich alles automatisch ein

        i. wollen ungern viel effort in komplexes save system einsetzen
        ii. factorio ähnlich: das level ist das savegame -> werden jetzt schon einige megabyte groß
3. Hatten Sie in der Vergangenheit Schwierigkeiten oder Herausforderungen bei der Implementierung von Save and Load in Ihren Spielen? Wenn ja, welche?
    a. Das Gefährliche ist beim Unterscheiden, woher Bugs kommen: kommen sie, da geladen wurde oder entstanden sie durch das normale gameplay/user input?
    b. Versioning: man kann es spielern nicht erlauben, spielern progress wegzunehmen -> nested versioning so lange bis version stimmt -> version verändern grunddatei behalten und cache benutzen für migrierung
        i. bei football manager war versioning schwerer
        ii. aktuell: validator: machen die daten untereinander sinn?
    c. Es muss sichergestellt werden, dass loading in richtige reihenfolge passiert
    d. Vergabe von Ids für referenzen -> neue ids für alles

Anforderungen und Wünsche an ein neues System

1. Welche Features und Funktionen sind Ihnen bei einem "Save and Loading"-System besonders wichtig?
    a. versioning
    b. Wie gut skaliert es?
2. Gibt es bestimmte Funktionen oder Integrationen (z.B. Cloud-Speicher, Datenbankanbindung, Verschlüsselung), die Sie sich wünschen?
    a. verschlüsselung
        i. nur bei competitive relevant/multiplayer
        ii. flag: ist das cheated
        iii. bei manchen games einfach nicht notwendig
    b. verschlüsselung erschwert modding

Schlussfragen

1. Gibt es sonstige Aspekte, die Sie in Bezug zu dem Thema ansprechen möchten?
    a. Jsons ist industriestandart
    b. pro gameObject DTO’s ein getter/setter

Vielen Dank für Ihre Teilnahme und Ihre wertvollen Beiträge. Ihre Antworten werden helfen, ein neues und benutzerfreundliches System zu entwickeln.

Gruppierung von Statements der Experten

Demographischer Hintergrund der Teilnehmenden:

- 2x Student
- 2x WiMi (Wissenschaftlicher Mitarbeiter)
- 1x HiWi (Hilfswissenschaftler)
- 3x Dozent
- Entwickler in Game Studios mit mehreren Releases
- Geschäftsführer von Mixed World

Erfahrungen und Fähigkeiten:

- Alle Teilnehmer haben Erfahrung mit Unity
- Manche haben Erfahrung mit der Unreal Engine, aber auch CryEngine vorhanden
- Alle Teilnehmer haben XR-Entwicklung gemacht
- Game-Engine-Entwicklungserfahrung
- Spezialisierung mit Serialisierung und Networking mehrfach vorhanden
- Tooling für Filmentwicklung

Verwendete Tools:

- Datenbanken: Nutzung von externen Datenbanken (z.B. S3-Buckets), mehrfach erwähnt
- PlayerPrefs: Verbreitet, aber problematisch, da Datentypen schwer nachvollziehbar sind
- Eigene Lösungen: Präferenz für selbst entwickelte Lösungen, keine fertigen Assets (mehrfach erwähnt)
- Chronos GLTF: Für programmierbare Daten
- Nahe an der Hardware: Speichert Daten oft direkt als Bytes auf der Maschine (Arduino, C)
- Unreal Engine: Konzeptuell gut vorbereitet, um Save und Loading zu integrieren
- Bereits Custom Lösungen: In Projekten vorhanden, in denen Entwickler dazugestoßen sind
- Beispiel einer eigenen Lösung: Einige Teile lokal, viele allerdings über Referenzverweise

Challenges:

- Datenbankprobleme: Herausforderungen bei Datenbankverbindungen und Datenkonvertierung.
- Einheitliche Serialisierung/Deserialisierung: Notwendigkeit einer einheitlichen Lösung.
- Zyklische Abhängigkeiten und Nested Objekte: Lösungsansätze:
    - Objekte in Abhängigkeiten reduzieren.
    - Nur die Referenzen von Objekten speichern, um zyklische Abhängigkeiten zu vermeiden.

- Save and Loading mit Online Support: Meist schwerer umsetzbar.
- Versioning:
    - Schwierig, Änderungen zu implementieren.
    - Gefahr, dass durch falsche Implementierung Fortschritte der Spieler verloren gehen.
- Kompatibilität: Probleme bei der Kompatibilität zwischen verschiedenen Systemarchitekturen.
- Gefahr bei Bugs: Schwierigkeit, zu bestimmen, ob Bugs durch das Laden oder das normale Gameplay entstehen.
- Referenz-IDs: IDs für Referenzen müssen eindeutig vergeben werden (mehrfach erwähnt).
- Richtige Reihenfolge beim Laden: Sicherstellen, dass das Laden in der richtigen Reihenfolge passiert.

Ideen:

- Einzelne Felder verschlüsseln: Möglichkeit, einzelne Felder zu verschlüsseln, sodass sie bei Bedarf ausgetauscht werden können.
- Editor Support:
    - Werte für den Inspektor können gespeichert und geladen werden.
    - Serialisierte Daten im Editor verwenden; Paketinhalte direkt im Inspektor sichtbar.
- Hybridlösung: Kombination aus State-based und Command-based Saving.
- Umwandlung zwischen binär und JSON: Verschlüsselung und Komprimierung möglich.
- Künstliche Intelligenz für Deltas: Änderungen speichern, um festzustellen, welche Deltas relevant sind.
- Cheating verhindern: IDs für Waffen vergeben, um das Ändern von Waffen-Stats zu erschweren.
- Backups von Save-Files: Besonders wichtig für mobile Anwendungen.
- Deserialisierungs-Vulnerabilitäten: Lösungsansätze, um Manipulationen an Daten zu verhindern
    - Überprüfung der Daten bei Deserialisierung.
    - Sicherstellen der Konsistenz der Datenübertragung (z.B. SSH).
    - Verschlüsselung von Werten im Editor, um den Schlüssel geheim zu halten.

Verwendete Serialisierungsformate:

- CSV
- Binär: Beliebt wegen geringerem Speicherbedarf.
- XML: Mehr für UI-Visualisierung und Datenpräsentation sowie breite Kompatibilität.
- JSON:
    - Menschlich lesbar, aber Overhead.
    - JSON wird besonders von Experten in der Industrie als Industriestandard angesehen (mehrfach erwähnt).
    - Unity JSON wird wegen geringerem Speicherbedarf über Newtonsoft bevorzugt, allerdings hat Newtonsoft mehr Features.
    - Verwendung von vielen kleinen JSONs.

- Selbst erstellter Serialisierer.
- Daten mit einem einzigen Befehl serialisieren.

Anforderungen:

- Effiziente Speicherplatznutzung: System sollte schlank bleiben und gute Ladezeiten bieten (mehrfach erwähnt).
- Datenkompression: Wichtige Funktion zur Reduzierung von Dateigrößen (mehrfach erwähnt).
- Thread-safe asynchrone Prozesse: Notwendig für große Datenmengen.
- Konkret sagen: "Das Objekt will ich speichern."
- State-basiertes Save und Loading: Command-basiert stößt auf Ablehnung.
- Versioning: Wichtig für langfristige Wartung und Weiterentwicklung von Spielen, aber oft schwierig umzusetzen (mehrfach erwähnt).
- Security: Verschlüsselung der Daten, besonders relevant bei Multiplayer-Spielen. Erschwert allerdings Modding.
- Skalierbarkeit: System sollte gut skalieren und sich an verschiedene Anforderungen anpassen lassen.
- Dateigröße: Sollte möglichst klein sein, insbesondere für mobile Anwendungen.
- Support von Vererbung: Erwartet (mehrfach erwähnt).

Zusätzliche Überlegungen:

- Bei dynamischen Strukturen: Der Entwickler sollte die Möglichkeit haben, bestimmte Teile des Systems nach Bedarf zu aktualisieren.
- Mehrfacher Wunsch: Zunächst sollte das System stabil sein, danach sollten Security und Performance angegangen werden.

.3 User-Tests

Der Anhang der User-Tests umfasst ein Übersichtsblatt der durchgeführten Tests, die vollständige Dokumentation der Testabläufe, die Ergebnisse der Google Forms-Umfragen sowie die Mitschriften der User-Tests.

Vielen Dank für Ihre Teilnahme am Unity User Test!

Aufgabe 0

Zunächst fülle bitte eine Umfrage zur Person aus. Bitte beachte, dass die Auswertung aller Fragebögen vollständig anonym erfolgt. Deine persönlichen Daten werden vertraulich behandelt und nicht mit den Auswertungen verknüpft. Die erhobenen Informationen dienen ausschließlich dem Zweck des Usability-Tests für das Save-and-Loading-System und werden unter Berücksichtigung aller geltenden Datenschutzbestimmungen verarbeitet. Durch das Ausfüllen der Umfrage erklärst du dich mit diesen Bedingungen einverstanden.

Bei der Bearbeitung der folgenden Fragen bitte laut denken. :)

Aufgabe 1

In der Doku Save Load System Documentation (Seite 15) stehen die konkreten Schritte, um für das Unity Template “2D Platformer Microgame” Save and Loading zu implementieren. Parallel wird die Zeit gestoppt!

Wenn du fertig bist, nimm dir kurz Zeit, um zu verstehen, was du gerade eigentlich gemacht hast. Anschließend gibt es eine kurze Umfrage.

Aufgabe 2

In der Doku Save Load System Documentation (Seite 19) stehen die konkreten Schritte, um für ein Inventar Save and Loading zu implementieren. Ziel ist es, dass nach dem Laden sowohl das UI als auch die Daten wiederhergestellt werden. Bevor die Zeit gestoppt wird, darfst du dich mit dem Projekt vertraut machen!

Wenn du fertig bist, nimm dir kurz Zeit, um zu verstehen, was du gerade eigentlich gemacht hast. Anschließend gibt es eine kurze Umfrage.

Thank you for participating in the Unity User Test!

Task 0

First, please fill out a personal survey. Please note that the evaluation of all questionnaires will be completely anonymous. Your personal data will be treated confidentially and will not be linked to the evaluation. The collected information is used solely for the purpose of the usability test for the Save and Loading System and will be processed in compliance with all applicable data protection regulations. By filling out the survey, you agree to these terms.

Please think out loud when answering the following questions. :)

Task 1

In the Save Load System Documentation (page 15), you will find the specific steps for implementing Save and Loading for the Unity Template "2D Platformer Microgame". Time will be tracked simultaneously!

When you're done, take a moment to understand what you just did. Afterwards, there will be a short survey.

Task 2

In the Save Load System Documentation (page 19), you will find the specific steps to implement Save and Loading for an inventory. The goal is to ensure that after loading, both the UI and the data are restored. Before the time is tracked, you can familiarize yourself with the project!

When you're done, take a moment to understand what you just did. Afterwards, there will be a short survey.

Save Load System Documentation

The Save System is a comprehensive and versatile tool designed specifically for Unity developers who need a reliable solution for saving game data. At its core, the system uses JSON for data serialization, which makes the saved data inherently platform-agnostic, ensuring that it can be easily transferred and used across various environments without compatibility issues.

One of the standout features of this Save System is its ability to handle complex object references. In many games, objects are interconnected, with dependencies and relationships that need to be preserved when saving and loading data. This system excels in managing these relationships, even when they involve nested or circular references, which are often challenging to handle. By accurately restoring these references during the loading process, the system ensures that the game's state is fully reconstructed, including the dynamic reloading of any prefabs that were part of the saved state.

Additionally, the Save System is designed to support as many data types as possible, making it highly adaptable to different game structures and requirements. This flexibility allows developers to save a wide variety of data, from simple variables to complex object graphs, ensuring that all necessary information is preserved.

Moreover, the Save System offers various approaches to saving data, allowing developers to choose the method that best suits their specific needs and the complexity of their game. Whether it's through declarative Attribute Saving, using the ISavable interface, or employing Type Converters, the system provides flexible options to ensure that data is efficiently and accurately preserved across different game states. This means that while developers can quickly implement basic saving and loading functionalities with minimal effort, they also have the option to customize the process.

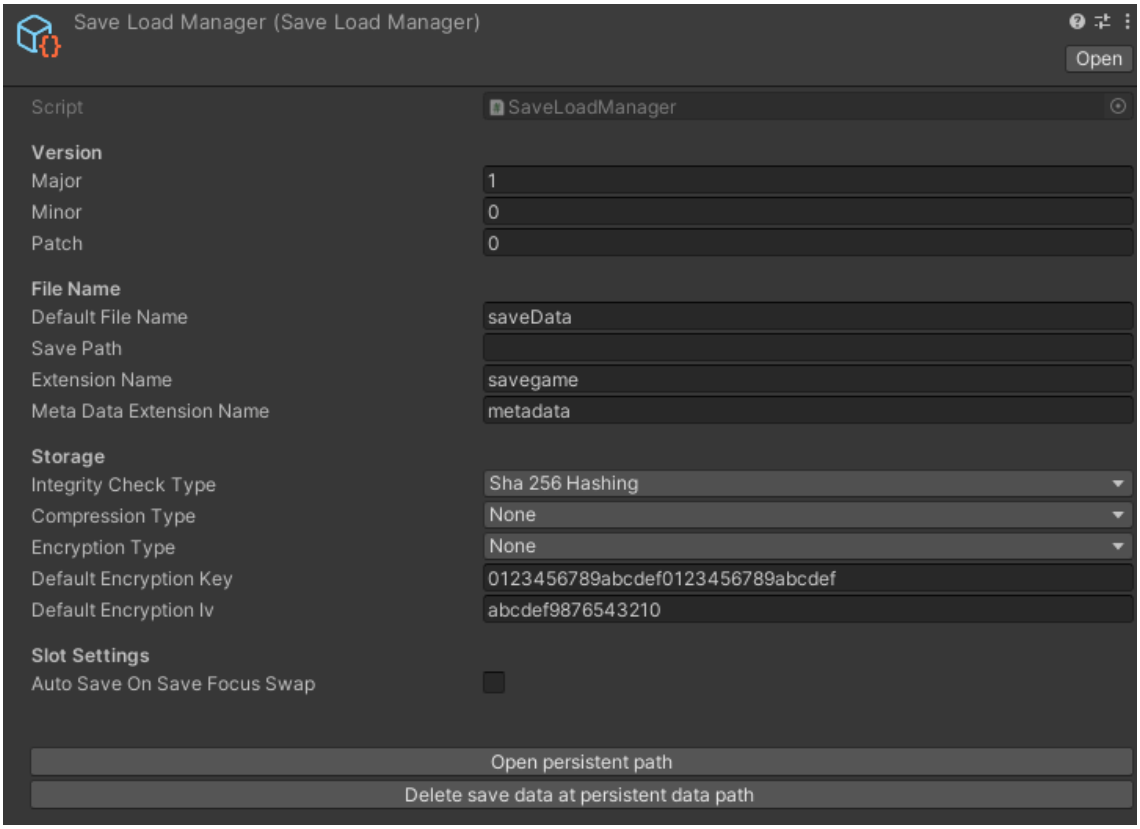
Main Components

Save Load Manager

The **SaveLoadManager** serves as the core of the save system, implemented as a Scriptable Object. Its primary role is to manage and control global settings for saving and loading data and to handle the saving process for all active scenes or a selection of scenes, ensuring data persistence across sessions.

It operates save actions based on a **current save file** that is actively being written to. This save file focus can be set explicitly through methods provided by the system or automatically when using save and load functions. If no specific filename is provided, the system defaults to a predefined save file name.

Beyond standard saving and loading, the system also offers advanced options like **Snapshotting** scenes. This will capture the current state of a scene without fully writing the save to the disk. Additionally, **deleting data** from certain scenes is possible. For more complex scenarios, there is also a method to **reload a scene and then load** the save file, ensuring that the scene is reset to its original state before applying the saved data. This can be particularly useful in cases where the scene's initial setup needs to be restored before loading specific changes from a save.
Additionally, the system provides various events that developers can hook their methods into. These events allow developers to execute custom logic at specific points during the save and load process, such as before or after a scene is saved or loaded, which for example is useful to update UI after loading.



These features provide developers with greater control over how and when game data is managed, allowing for more flexible and robust save management across various gameplay scenarios. All actions related to writing and reading save data to and from disk are fully **asynchronous**, ensuring that these operations do not block the main thread. This async approach helps maintain smooth gameplay performance by offloading potentially time-consuming file operations.

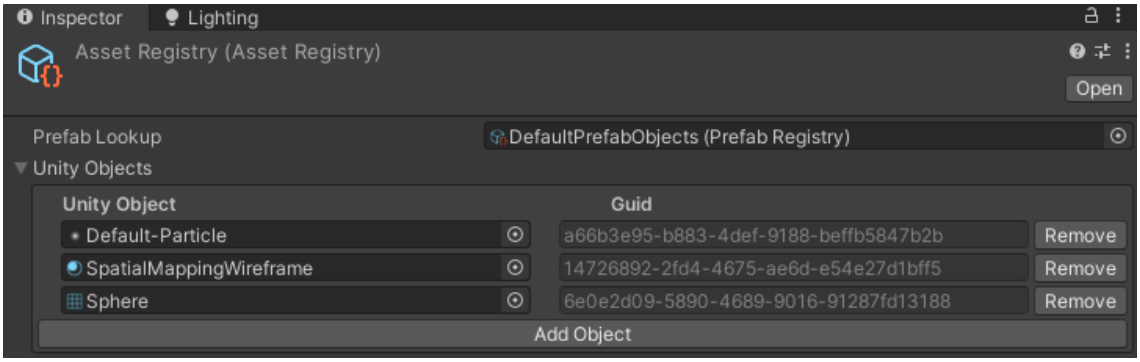
The Save System creates two files for each save operation: a **metadata file** and the actual **save data file**. The metadata file contains essential information such as the creation time of the save, the version of the save file, and a checksum to verify the integrity of the save data.

Additionally, developers can add custom data to the metaData file, allowing for more detailed and specific tracking of save states.

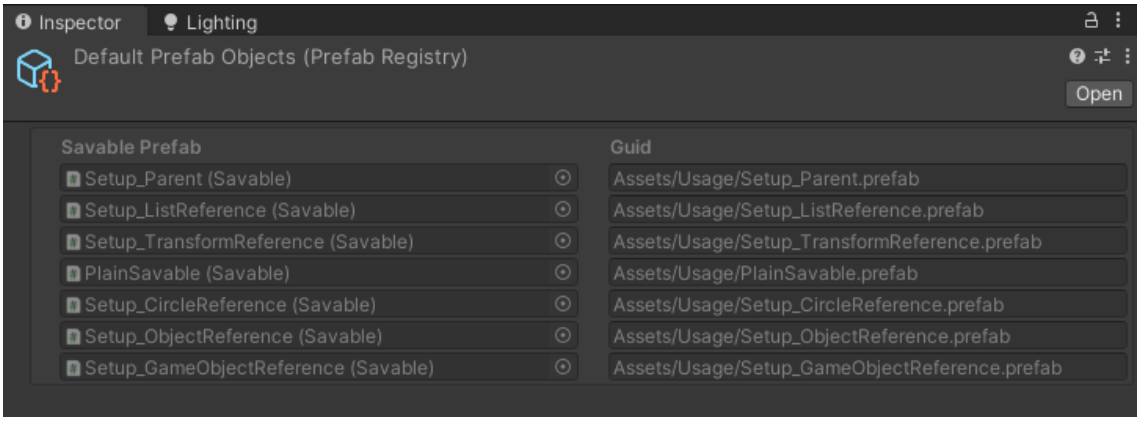
Asset and Prefab Registry

The **Asset Registry** is a crucial part of the save and load system that tracks and manages all assets that need to be persisted. It stores references to assets like textures, materials, and other non-prefab resources, ensuring they are correctly reloaded when a save state is restored. By maintaining a centralized record of these assets, the system can efficiently manage dependencies and reduce the risk of missing resources during loading.

If developers want to support saving data on ScriptableObjects, they must be added to the Asset Registry. This ensures that the ScriptableObjects are properly tracked, and their data is accurately saved and reloaded.



The **Prefab Registry** is a Scriptable Object that automatically keeps track of all prefabs marked with the Savable Component. It is automatically created if it does not already exist. This registry is used for Dynamic Prefab Loading, where the system can instantiate necessary prefabs and remove unnecessary ones based on the saved state.

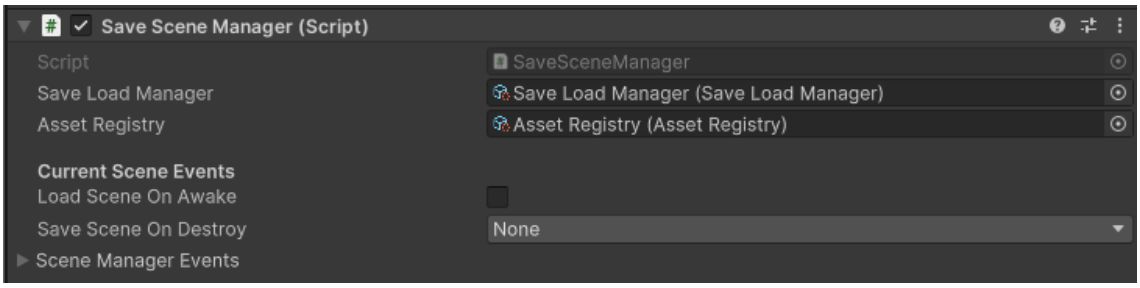


Both the Asset Registry and Prefab Registry identify objects based on their type. This requires developers to ensure that the type of a property or field matches the type of the asset in the corresponding registry. For instance, if a field is intended to store a prefab asset,

it must be of type `Savable`. This is because the Prefab Registry can only identify prefabs that are of the `Savable` type.

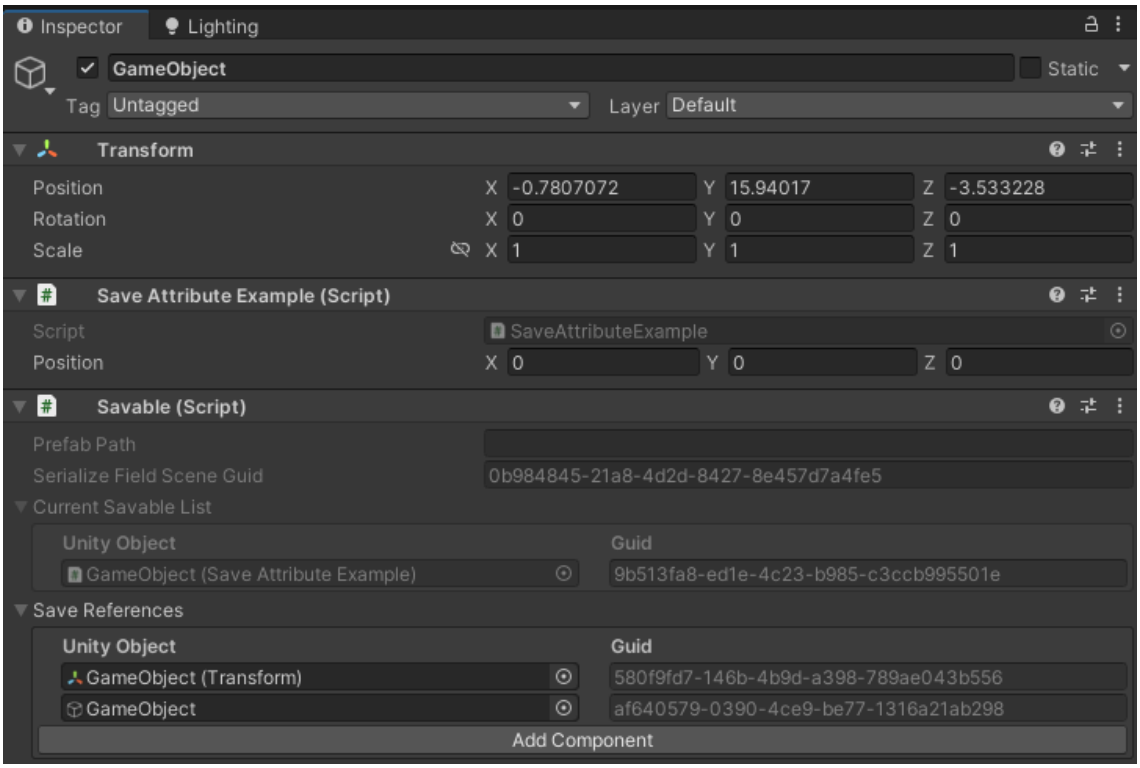
Save Scene Manager Component

The **SaveSceneManager** is a component specifically responsible for managing save processes within individual scenes. Each scene should contain exactly one SaveLoadSceneManager to independently manage its state and ensure that scene-specific data is saved and restored correctly.



Savable Component

The **Savable Component** is crucial for assigning unique IDs to objects within a scene. These IDs are used to identify and restore all references during loading. It automatically detects scripts that implement the Savable attributes or the ISavable interface and assigns them an ID. To check which objects on a GameObject are tracked for save and loading, developers can check the `Current Savable List` on a Savable Component.



Additionally, the Savable Component provides a `Save References` field, allowing developers to assign IDs to arbitrary objects for reference purposes. This feature is particularly intended for components on the GameObject that contains the Savable Component. Let's say the child of a GameObject has a Savable component with a SaveParent component, which allows it to save and restore its parent after loading. To ensure that the child can correctly find and reattach to its old parent, you'll need to add the parent's transform to the parent's `Save Reference` list. This step ensures the Save System recognizes the parent's transform, allowing the child to accurately locate and re-establish its connection to the correct parent when the game is reloaded.

Important Concepts

Single Scene and Multi Scene Setup

The Save System in Unity is designed to efficiently handle both single scene and multi-scene setups, offering flexibility for different game architectures. However, there is one restriction regarding multi-scene setups: references are scene-bound. This means that an object reference saved in one scene cannot be directly linked to an object in another scene. If you save an object in Scene A, it will be treated as a separate instance when loaded in Scene B, even if the objects are identical. This design helps maintain clarity and consistency within the game's data, ensuring that each scene can be managed independently without cross-scene conflicts.

However, there is a way to handle references across scenes by using ScriptableObjects. To do this effectively:

1. **Choose One Scene as the Save and Load Origin:** Decide on a single scene to serve as the save and load origin for the ScriptableObject.
2. **Only Save in the Origin Scene:** Ensure that the ScriptableObject is only saved in this designated scene. This prevents duplication and ensures consistency across your game.
3. **Use the ScriptableObject as a Bridge:** Leverage the ScriptableObject to share data across other scenes, acting as a bridge to maintain consistent information.

Additionally, in order to support saving data on ScriptableObjects, they must be added to the Asset Registry. This ensures that the ScriptableObjects are properly tracked, and their data is accurately saved and reloaded.

The Save System saves the data of a ScriptableObject separately for each scene. If a ScriptableObject is saved in two scenes, it will be loaded twice. This could be useful if the scenes are loaded together and contain different objects that need to be applied to the ScriptableObject. However, this approach is generally not recommended. Instead, consider splitting the ScriptableObject into two separate objects to avoid conflicts and ensure a cleaner data structure.

Value Saving and Reference Saving

The Save System offers two primary methods for saving and loading objects: Value Saving and Reference Saving.

Reference Saving: When the Save System saves an object as a reference, it typically creates a separate data container to store that reference. This approach is particularly useful for handling complex objects and relationships, as it allows the system to maintain object identities and dependencies, such as polymorphic types where different derived classes might be saved and restored. While this approach provides flexibility, it is generally slower and consumes more memory due to the overhead of managing references. Therefore, it is recommended to use referenceable saving sparingly and only when necessary for complex objects or relationships that cannot be represented as simple values.

Value Saving: On the other hand, Value Saving simplifies this process. Instead of creating a new, separate container for each reference, the object is directly converted into JSON format and stored within the existing data structure. This approach makes the saving process more efficient by keeping all related data within a single container, reducing the overhead of managing multiple containers. However, Value Saving requires that the correct type be applied directly, as it doesn't support the dynamic nature of polymorphic types like Reference Saving does.

When using the Attribute Saving feature, objects will be saved using Value Saving if they meet certain criteria: they aren't a MonoBehaviour or ScriptableObject, don't implement the ISavable Interface, and are not part of the Type Converter system.
On the other hand, when using the ISavable interface, developers have the flexibility to choose between Reference Saving and Value Saving. This allows developers to decide the most appropriate saving method for their objects, depending on the specific needs of their

game. For instance, they can opt for Reference Saving to maintain object references and polymorphic types or use Value Saving to directly store object's data within the current reference data container.

How to mark things for saving

Attribute Saving

There are two main Attributes developers can choose from to mark things to be saved.

- `[Savable]`: This attribute can be applied to individual fields or properties to mark them for saving. It allows developers to specify precisely which data should be persisted without saving the entire class.
- `[SavableObject]`: Applying this attribute to a class automatically marks all fields and properties within that class for saving. By doing so, developers can ensure that all relevant data is persisted. This attribute is particularly suited for Data Transfer Objects (DTOs), which are designed to encapsulate and transport data between different layers of an application without any business logic. Additionally, developers need to use the `declaredOnly` parameter to specify if only the fields of the declared class should be considered, excluding fields from base classes. This is especially useful for MonoBehaviour classes, which cannot be serialized directly by Unity. Using `declaredOnly` ensures that the serialization focuses solely on the class's unique fields, avoiding potential issues with inherited fields that are not directly compatible with the save system.

The attributes `[Savable]` and `[SavableObject]` apply to all access modifiers, including public, non-public, static, and instances for fields and properties, offering high flexibility in defining what data to save. For non-MonoBehaviour classes, the system can automatically create new instances of objects and initialize them with saved data, ensuring seamless restoration of the state. Internally it is accomplished using `Activator.CreateInstance`. This requires a parameterless constructor for classes. In contrast, structs do not need a parameterless constructor, as they are automatically instantiated with their default values.

Additionally, these attributes can be used alongside with a ISavable interface. By using both attributes and interface inside a class there is one thing developers must be aware of: it is possible to save and load the same value twice. This should be avoided.

Attribute Save Path

The save system is designed to handle complex data structures by resolving nested objects marked as savable. When nested fields or properties are saved, the system generates a unique identifier (ID) for each reference. Since an object may be used multiple times, the first occurrence path is the origin ID for that object. Every other occurrence will use that path. The ID for nested objects is constructed using the following format:

`[SavableSceneID]/[ComponentID]/[field/property name]/[field/property name]...`

```
// This class demonstrates the use of Savable attributes within a MonoBehaviour.
No asset usages
public class SaveAttributeExample : MonoBehaviour
{
    // The 'position' field is marked as [SerializeField] to be editable in the Unity Inspector
    // and [Savable] to be included in the save process.
    [SerializeField, Savable]
    private Vector3 position;  Serializable

    // The 'ExampleObject' property is marked with [Savable] to ensure it is included with save and loading.
    [Savable]
    private ExampleDataTransferObject ExampleObject { get; set; }
}

// This class is marked as a savable object using the [SavableObject] attribute.
// All fields and properties within this class will be automatically marked for saving.
[Serializable, SavableObject]
1 usage
public class ExampleDataTransferObject
{
    // These public fields will be saved and loaded automatically due to the SavableObject attribute.
    public string Name;  Serializable
    public int Health;  Serializable

    // Since ExampleObject is a custom class, it needs to be instantiated during loading.
    // This is done using Activator.CreateInstance(), which requires a parameterless constructor.
    // The parameterless constructor allows the object to be instantiated without providing arguments.
    public ExampleDataTransferObject() {}

    public ExampleDataTransferObject(string name, int health)
    {
        Name = name;
        Health = health;
    }
}
```

ISavable interface

Unity components can't be directly extended with additional code to support the savable attributes. To address this limitation, the ISavable interface offers a robust solution for making these components savable. Furthermore, the ISavable interface provides the ability to further customize and control the save and load process compared to the Attribute Saving. This is particularly useful when special logic or additional steps are required to correctly save or restore an object's state. You can even add this interface to Non-MonoBehaviour classes, if the instance of the class is marked as a savable.

To make a Unity component savable, implement the ISavable interface in its own MonoBehaviour class and define the `OnSave` and `OnLoad` methods. Inside those methods you can define your custom save and loading for your wanted Unity component. This is done by utilizing the provided `SaveDataHandler` and `LoadDataHandler`. There are two main options for saving and loading data:

To perform Value Saving, use the `SaveAsValue` and `LoadValue<T>` methods. This approach reduces memory usage and improves performance, making it the preferred choice for most scenarios. Reference Saving can be performed by using the `TrySaveAsReferencable` and `TryLoadReferencable` methods. However, Reference Saving requires an additional step compared to Value Saving:

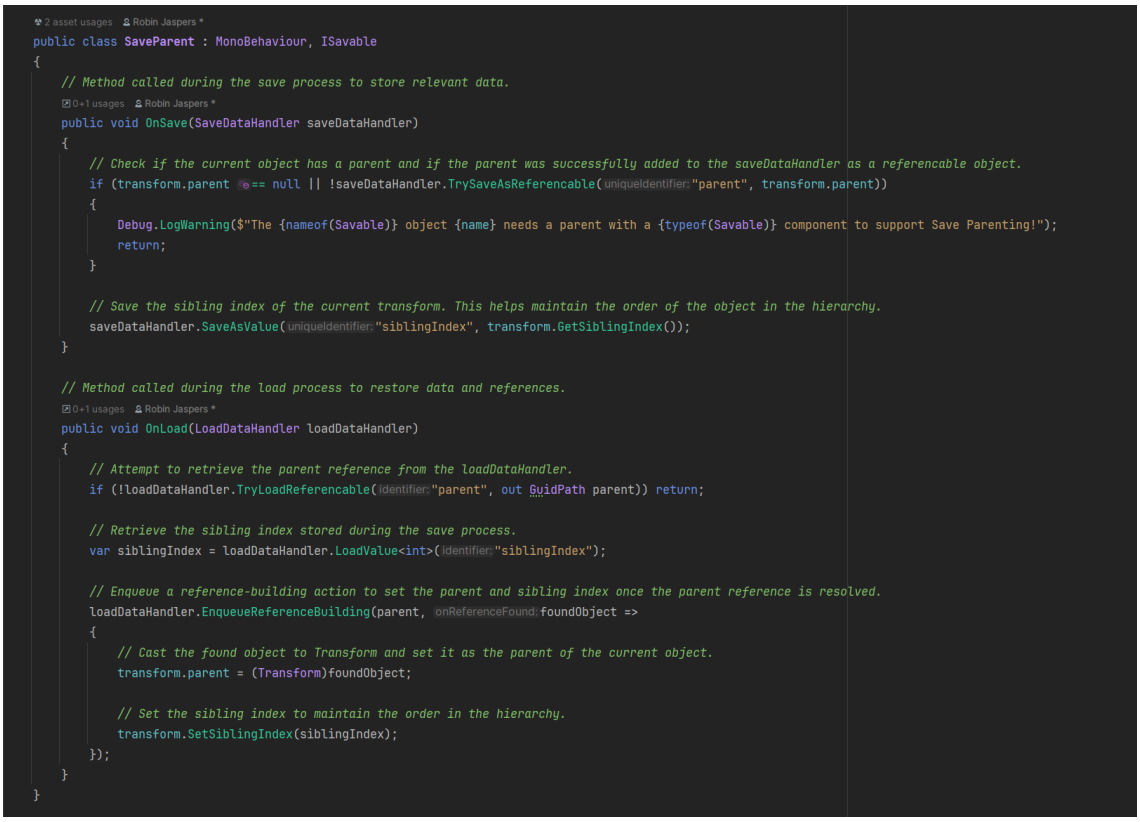
Internally, all objects are created first, and references are resolved afterward. This means that loading references will initially return only the path to the reference. To convert these paths into the desired object once all objects are created, the `LoadDataHandler` includes an additional method called `EnqueueReferenceBuilding`. This method allows you to queue the conversion of one or multiple paths into the actual object. The found object is returned as an action within the `EnqueueReferenceBuilding` method. Inside that action, the developer is responsible for converting the returned object into the required type and resolving the loading logic.

The ISavable interface can be combined with Attribute Saving to mark specific fields or properties as savable. By doing so, it is possible to save and load values twice, which should be avoided.

How to use it

1. **Inherit from ISavable:** Start by inheriting from the ISavable interface in the class you want to make savable. This will allow you to implement the required methods for saving and loading data.
2. **Ensure Proper Tracking:** Make sure that the object implementing ISavable is tracked by a Savable Component. This can be done by adding the ISavable class to the same GameObject that has the Savable Component attached. Alternatively, the object can be identified for saving through another script. For instance, if a field in a MonoBehaviour is marked with the [Savable] attribute, and the object assigned to that field implements ISavable, the Save System will automatically track and save it.

Below is an example demonstrating how the Save and Loading System can save and load the parent of an object, provided that the parent has a Savable Component. This setup ensures that even complex relationships, like parent-child hierarchies, are accurately preserved across save and load operations.



```
2 asset usages  Robin Jaspers *
public class SaveParent : MonoBehaviour, ISavable
{
    // Method called during the save process to store relevant data.
    0+1 usages  Robin Jaspers *
    public void OnSave(SaveDataHandler saveDataHandler)
    {
        // Check if the current object has a parent and if the parent was successfully added to the saveDataHandler as a referencable object.
        if (transform.parent == null || !saveDataHandler.TrySaveAsReferencable(uniqueIdentifier: "parent", transform.parent))
        {
            Debug.LogWarning($"The {nameof(Savable)} object {name} needs a parent with a {typeof(Savable)} component to support Save Parenting!");
            return;
        }

        // Save the sibling index of the current transform. This helps maintain the order of the object in the hierarchy.
        saveDataHandler.SaveAsValue(uniqueIdentifier: "siblingIndex", transform.GetSiblingIndex());
    }

    // Method called during the load process to restore data and references.
    0+1 usages  Robin Jaspers *
    public void OnLoad(LoadDataHandler loadDataHandler)
    {
        // Attempt to retrieve the parent reference from the loadDataHandler.
        if (!loadDataHandler.TryLoadReferencable(identifier: "parent", out GuidPath parent)) return;

        // Retrieve the sibling index stored during the save process.
        var siblingIndex = loadDataHandler.LoadValue<int>(identifier: "siblingIndex");

        // Enqueue a reference-building action to set the parent and sibling index once the parent reference is resolved.
        loadDataHandler.EnqueueReferenceBuilding(parent, onReferenceFound: foundObject =>
        {
            // Cast the found object to Transform and set it as the parent of the current object.
            transform.parent = (Transform)foundObject;

            // Set the sibling index to maintain the order in the hierarchy.
            transform.SetSiblingIndex(siblingIndex);
        });
    }
}
```

Additional ISavable Components

The [ISavable interface] can also be extended with additional components to manage other aspects of a GameObject's state. In addition to the `SaveParent` component, which handles saving and restoring parent-child relationships, there are other pre-built components available, such as `SavePosition` for saving and restoring an object's position, and `SaveVisibility` for managing its active or inactive state. These components provide ready-made solutions for common saving needs, further simplifying the process of making a Unity component savable.

Interface Save Path

By using the Reference Saving, the system generates a unique identifier (ID) for each reference. Since an object may be used multiple times, the first occurrence path is the origin ID for that object. Every other occurrence will use that path. The ID for nested objects is constructed using the following format:

`[SavableSceneID]/[ComponentID]/[unique Identifier]/[unique Identifier]...`

The unique identifier string originates from the parameter defined inside `SaveDataHandler` and `LoadDataHandler` methods.

Advanced Saving

Type Converter

The Type Converter system in the Save System is similar to the ISavable Interface, and it is recommended to familiarize with the ISavable interface system first. The primary purpose of the Type Converter is to convert a specified type directly into a serializable format while maintaining references when loading, making it easier to save and load complex data structures. This system is mainly used for classes that do not provide direct access to change them, such as Unity types.
Both the Type Converter and ISavable interface implement the `OnSave` and `OnLoad` methods, and both provide a `SaveDataHandler` and `LoadDataHandler` for managing the saving and loading processes. The key difference lies in how they handle data conversion: in the `OnSave` method of a Type Converter, you receive the type that needs to be converted into a savable format. In the `OnLoad` method, you are responsible for reconstructing the object from the saved data and returning it.
The Type Converter is already implemented for several Unity types: `Color32`, `Color`, `Vector2`, `Vector3`, `Vector4`, and `Quaternions`. In addition to handling Unity types, the Type Converter is also used for creating references within collections such as `Array`, `List`, `Dictionary`, `Stack`, and `Queue`. This capability ensures that complex data structures, including those that contain multiple references, can be serialized and deserialized accurately, preserving the integrity of the data.

How to Use it

To create a custom Type Converter, developers can either inherit from the `BaseConverter<T>` class or directly implement the `IConvertible` interface, then implement their `OnSave` and `OnLoad` methods for serialization and deserialization:

- **Inheriting from BaseConverter:** For most use cases, inheriting from the `BaseConverter<T>` class is sufficient. The generic T represents the type for which Type Conversion should be enabled. This class provides the foundational functionality needed to create custom type converters, making it easier to handle the conversion process.

- **Implementing IConvertible:** In some specific cases, the `BaseConverter` may not work. In these instances, you will need to implement the `IConvertible` interface directly. When doing so, developers must also define the `CanConvert` method. This method determines whether the Type Converter should handle a given type. If `CanConvert` returns `true`, the Type Converter will use this class for conversion.

```
1 usage  Robin Jaspers *
public class Color32Converter : BaseConverter<Color32>
{
    0+1 usages  Robin Jaspers
    protected override void OnSave(Color32 data, SaveDataHandler saveDataHandler)
    {
        saveDataHandler.SaveAsValue(uniqueIdentifier:"r", data.r);
        saveDataHandler.SaveAsValue(uniqueIdentifier:"g", data.g);
        saveDataHandler.SaveAsValue(uniqueIdentifier:"b", data.b);
        saveDataHandler.SaveAsValue(uniqueIdentifier:"a", data.a);
    }

    0+1 usages  Robin Jaspers *
    public override object OnLoad(LoadDataHandler loadDataHandler)
    {
        var r = loadDataHandler.LoadValue<byte>(identifier:"r");
        var g = loadDataHandler.LoadValue<byte>(identifier:"g");
        var b = loadDataHandler.LoadValue<byte>(identifier:"b");
        var a = loadDataHandler.LoadValue<byte>(identifier:"a");

        return new Color32(r, g, b, a);
    }
}
```

After creating your custom converter, it must be registered with the Save System. This is done by adding the converter type to the static constructor of the `TypeConverterRegistry` class. Be aware that this registration process is expected to change in a future version of the Save System.

```
3 usages  Robin Jaspers *
public static class TypeConverterRegistry
{
    private static readonly List<IConvertable> Factories = new();

    Robin Jaspers *
    static TypeConverterRegistry()
    {
        //collections
        Factories.Add(new ArrayConverter());    //array must be processed before list, due to both inheriting from IList
        Factories.Add(new ListConverter());
        Factories.Add(new DictionaryConverter());
        Factories.Add(new StackConverter());
        Factories.Add(new QueueConverter());

        //unity types
        Factories.Add(new Color32Converter());
        Factories.Add(new ColorConverter());
        Factories.Add(new Vector2Converter());
        Factories.Add(new Vector3Converter());
        Factories.Add(new Vector4Converter());
        Factories.Add(new QuaternionConverter());

        //add your own converter here
    }
```

Dynamic Prefab Loading

A key feature of this Save and Loading System is its ability to save references and restore them during loading. In a Unity application, prefabs can be instantiated and destroyed during runtime according to requirements. However, if a prefab is missing during loading, its associated references may not be restored correctly. To prevent this, the system saves active prefabs so that the same prefabs exist upon the next load, keeping references intact. This ensures that a scene can be reconstructed seamlessly and without data loss.

Dynamic Prefab Loading compares the current prefabs with the required prefabs. It instantiates necessary prefabs and destroys redundant ones, allowing different save states to be loaded without reloading a scene.

To ensure the Save and Loading System can identify savable prefabs, desired prefabs need the Savable Component. All savable prefabs are automatically added to the Prefab Registry. If a prefab with a Savable Component should not support dynamic prefab loading, it can be disabled using the "Dynamic Prefab Spawning Disabled" inspector boolean.

Prefabs with Overrides

When possible, it is recommended to avoid destroying prefabs with overrides. Developers can identify prefabs with overrides by looking for modifications in the prefab instance compared to its original definition. In Unity, these overrides are typically marked with a blue line in the Inspector, indicating that certain properties have been modified from the prefab's default state.
Prefabs with overrides require special handling, as the dynamic prefab loading system cannot save overrides. If a prefab with overrides is missing during loading because it was destroyed during runtime, it will be lost when the original prefab is re-instantiated in its default state. If destroying prefabs with overrides is necessary, the following solutions are recommended:

- **Set Prefabs to Inactive**: Instead of destroying the prefabs, deactivate them by setting them inactive. Developers can use the `SaveVisibility` component to save and load their active/inactive state. This ensures that the prefabs, along with their overrides, remain intact and are not permanently lost.
- **Reload the Scene**: Before loading a saved state, reload the entire scene to reset it to its original condition. This process ensures that all prefabs, including those with overrides, are instantiated in their original form with all initial modifications and configurations intact. This is particularly useful if the scene's state depends heavily on runtime changes that are not captured by the save system.

Future Plans

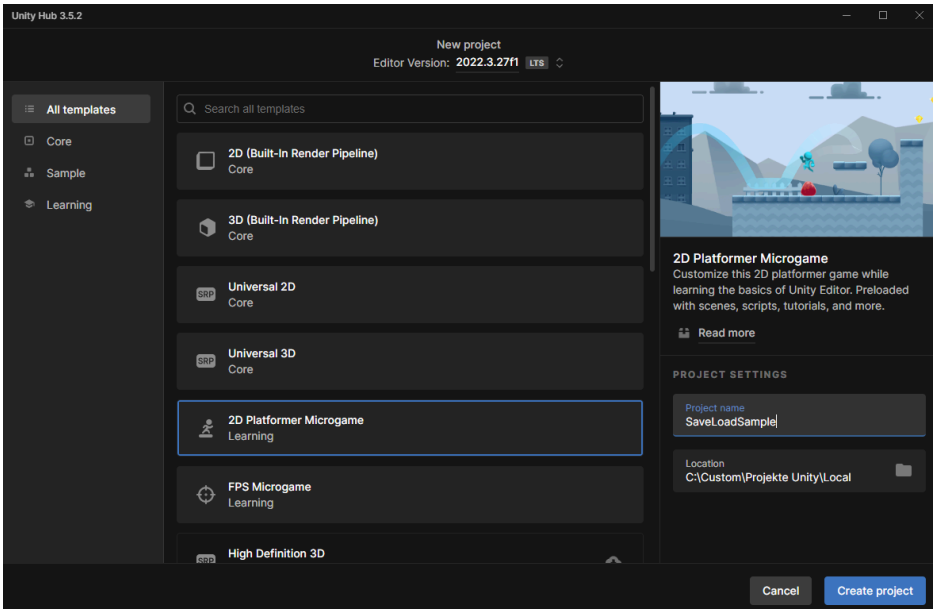
- Save References inspector field on Savable Component: make them resolve itself, so that no user input is needed
- implement a way to easier find save paths inside json and in inspector
- versioning system
- improve security regarding saving of Type.AssemplyQualifiedName (which is currently needed for polymorphic Reference Saving) and make it easy usable for versioning
- improve error handling
- improve adding elements to the Type Converter
- among others

Getting Started with the SaveLoadSystem in Unity

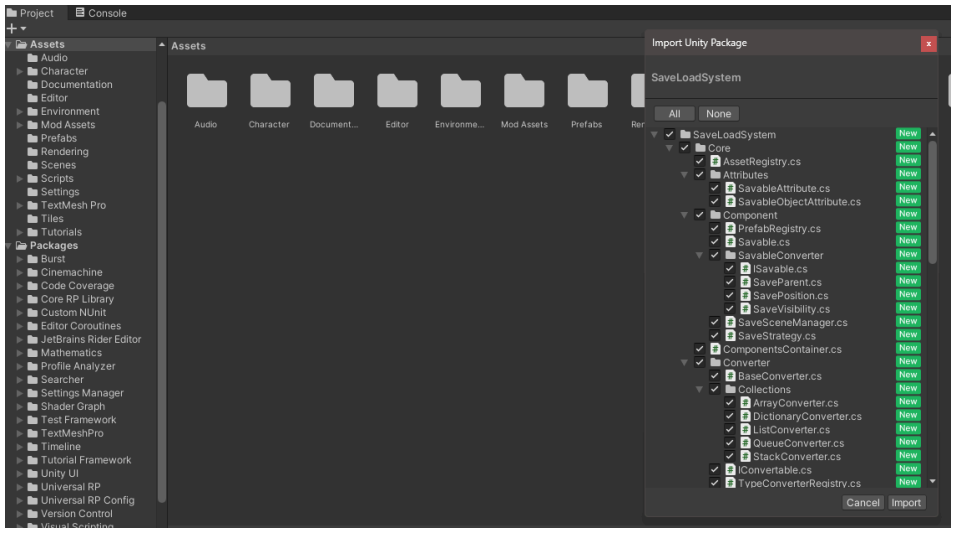
This tutorial will guide you through setting up the SaveLoadSystem in a new Unity project based on the "2D Platformer Microgame" template. Follow these steps to integrate the system and quickly set up saving and loading functionality for key game objects.

Step 1: Download and Import the SaveLoadSystem

1. Download the “SaveLoadSystem.unityPackage” file containing the SaveLoadSystem from here.
2. Open Unity and create a new project using the "2D Platformer Microgame" template.

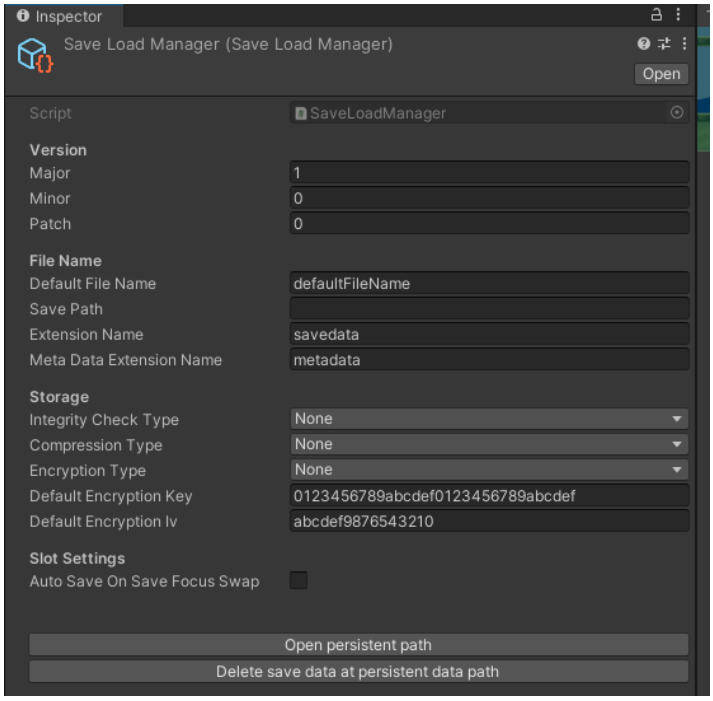


3. Drag the `.unitypackage` file into your Unity project window and import all the contents.



Step 2: Initial Setup

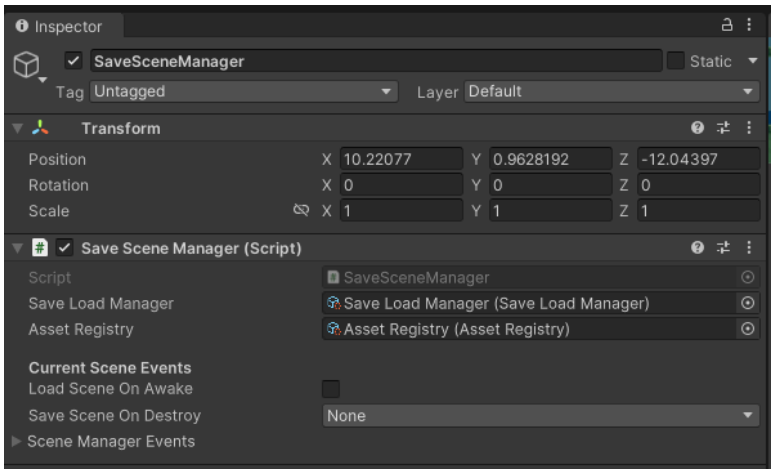
1. **Create Scriptable Objects:**
    - Upon importing the package, a Prefab Registry ScriptableObject will be automatically created
    - Manually create an Asset Registry ScriptableObject.
    - Create a Save Load Manager ScriptableObject.
2. **Configure the Save Load Manager:**
    - Open the Save Load Manager in the Inspector.
    - Set a default file name, extension name, metadata extension name, and version number for the save file.



3. **Link Registries:**
    - In the Asset Registry Inspector, drag the Prefab Registry into the “Prefab Lookup” field.

Step 3: Set Up the Save Scene Manager

1. **Create Save Scene Manager:**
    - In the Sample Scene of the 2D Platformer Microgame, create a new empty GameObject.
    - Name it SaveSceneManager.
    - Add the "Save Scene Manager" component to this GameObject.
    - In the Inspector, link the associated ScriptableObjects (Asset Registry, Save Load Manager) to the relevant fields.



Step 4: Prepare GameObjects for Saving

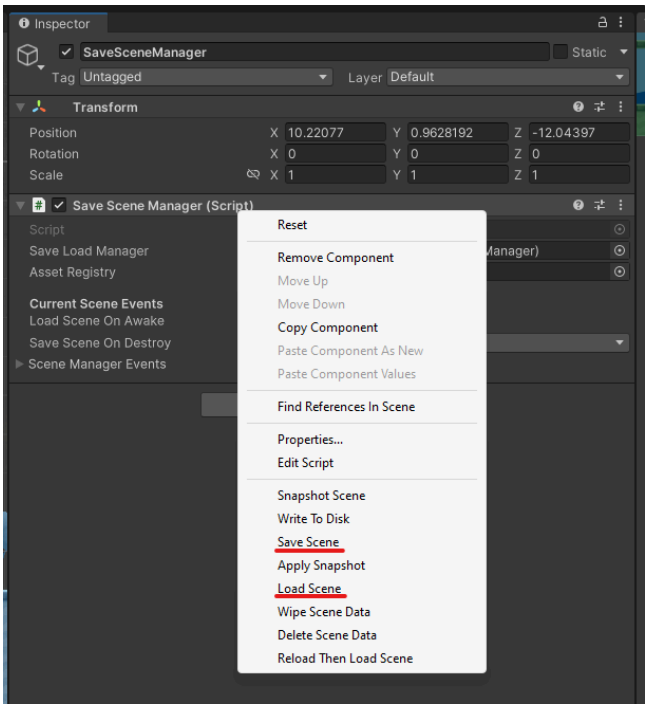
You will need to configure three types of objects for saving: the player prefab, enemy prefab, and all the tokens.

1. **Player Prefab:**
    - Locate the **Player** prefab in the Project window.
    - Open the prefab to make changes directly.
    - Add the Savable component and the **SavePosition** component to the Player prefab.
2. **Enemy Prefab:**
    - Locate the **Enemy** prefab in the Project window.
    - Open the prefab to make changes directly.
    - Add the Savable component and the **SavePosition** component to the Enemy prefab.
3. **Tokens:**
    - In the **Sample Scene** Hierarchy under Tokens, select all token objects (Select the top token, then hold Shift and select the bottom token to highlight all tokens)
    - Add the Savable component and the **SaveVisibility** component to the tokens.

Step 5: Testing the Save and Load Functionality

1. Play and Save the Game:
    - Enter Play mode in Unity and interact with the game to modify the positions and states of the player, enemies, and tokens.
    - Right-click the SaveSceneManager component in the Inspector and select **Save Scene** to save the current state.

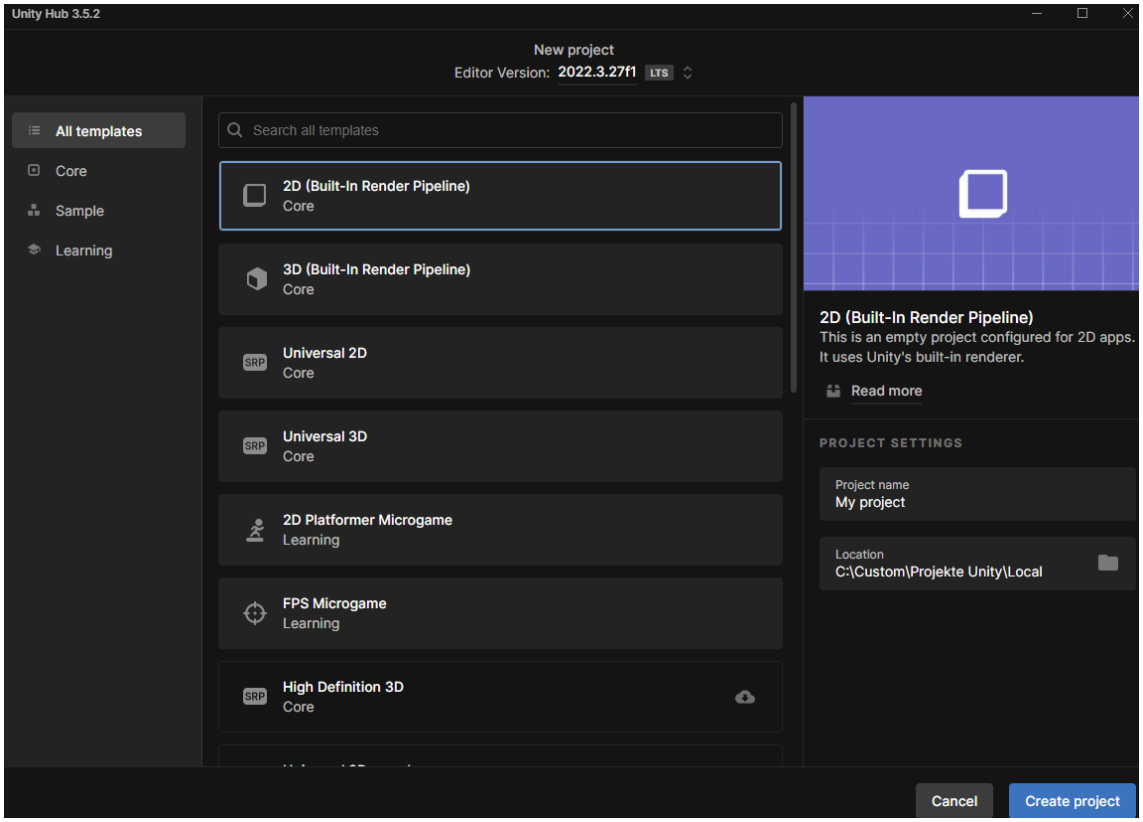
2. **Load the Scene:**
    - Restart Play mode.
    - In the Inspector, click **Load Scene** on the SaveSceneManager component to load the saved state.
    - The player, enemies, and tokens should return to the state they were in when you saved the scene.



Getting Started with Coding

This tutorial focuses on coding with the SaveLoadSystem. You'll learn how to code, configure dynamic prefab loading and manage references effectively. By the end, you'll have a fully functional save and load system for an Inventory. Note that this Inventory is an example and not production code!

1. **Create a New Project:** Start by creating a new project using the "2D Built-In Render Pipeline." Then, import the "Exercise Two" package from the `SaveLoadSystemPackages` directory.



2. **Add "Savable" Component to "InventoryElement" Prefab:** The "InventoryElement" prefab requires a "Savable" component to enable Dynamic Prefab Loading.
3. **Add "SaveParent" to "InventoryElement" Prefab:** Since the "InventoryElement" prefab should maintain the same parent after Dynamic Prefab Loading, you need to add the SaveParent script to it. For the "SaveParent" script to identify its parent reference, the parent must have an ID. Therefore, add a "Savable" component to the parent at "Canvas/Scroll View/Viewport/Content"(this is a hierarchy path) GameObject. After saving and loading, all previously existing InventoryElement prefabs should reappear, though the UI will be empty.
4. **Use the Attribute Saving System:** To save the content of individual class instances, we will use the Attribute Saving System. This system allows fields and properties to be marked so their content, including references, can be saved. Upon loading, all references should be correctly reassigned.

    a. **Apply Attribute to Fields in the “Item” Class:** For the standard C# class “Item”, all fields/properties must be marked as savable. A shortcut for this is the [SavableObject] attribute, which automatically marks all fields and properties. Classes that do not inherit from “MonoBehaviour” or “ScriptableObject”, and do not use Attribute Saving, the ISavable interface, or the Type Converter, will not be saved as references (they will be converted to JSON and directly saved to the current save reference object). In this case, using attributes is necessary to avoid this. Note that even references to collections can be saved. This is internally handled by the Type Converter.
    b. **Apply Attribute [Savable] to Fields/Properties in “Inventory” and “InventoryElement” Scripts:** In the Inventory and InventoryElement scripts, there is a field/property that must be marked with the [Savable] attribute.
    c. **Apply Attribute [Savable] to Fields/Properties in “InventoryView” Script:** The InventoryView script contains elements that must be marked with the [Savable] attribute.
5. **Add “Savable” Component to GameObjects:** To ensure the system recognizes “MonoBehaviour” scripts with attributes, you need to add the “Savable” component to the GameObjects they are attached to. Locate these GameObjects and add the “Savable” component.
6. **Add Necessary Assets for Save and Load System:** To allow assets to be referenced by the save and loading system, you need to add the required assets to the Asset Registry. This includes Sprites as well.
7. **Test Saving and Loading:** When you attempt to save and load now, not only will the prefabs be correctly assigned, but all objects will also be correctly referenced. However, the UI must still be updated to reflect the new values. To do this, find the appropriate event in the “SaveFocus” field of the “SaveLoadManager.” Then, use this event to call the Setup(Item item) method in the “InventoryElement” script, passing the “ContainedItem” property as the parameter.
8. **Done:** Everything should now Save and Load correctly! You should be able to use the button “Delete Random Inventory Item” without issues after loading now! To further check if everything is correctly loaded, you can use the debugging mode inside the inspector.

Fragen zur Person

Wie Alt bist du? *

- ( ) 0-20
- (•) 21-35
- ( ) 36-50
- ( ) 50+

Was ist ihr Beruf *

Software-Entwickler

Wieviel Erfahrung haben Sie mit Unity? (e.g. in Jahren) *

0

Wieviel Erfahrung haben Sie mit objektorientierten Programmiersprachen wie Java, c#...? * (e.g. in Jahren)

2

Fragen zur Person

Wie Alt bist du? *

- ○ 0-20
- ◉ 21-35
- ○ 36-50
- ○ 50+

Was ist ihr Beruf *

Student, Softwareentwickler, IT-Consultant

Wieviel Erfahrung haben Sie mit Unity? (e.g. in Jahren) *

Durch das Masterstudium viele Projekte, ca. 6 Jahre, bisschen im Bachelor, hauptsächlich im Master

Wieviel Erfahrung haben Sie mit objektorientierten Programmiersprachen wie Java, c#...? (e.g. in Jahren) *

7 Jahre, im Bachelor mit C# angefangen, Java erst im Master

Fragen zur Person

Wie Alt bist du? *

- ○ 0-20
- ○ 21-35
- ◉ 36-50
- ○ 50+

Was ist ihr Beruf *

UI (Team) Lead

Wieviel Erfahrung haben Sie mit Unity? (e.g. in Jahren) *

5

Wieviel Erfahrung haben Sie mit objektorientierten Programmiersprachen wie Java, c#...? (e.g. in Jahren) *

16

Fragen zur Person

Wie Alt bist du? *

- ○ 0-20
- ◉ 21-35
- ○ 36-50
- ○ 50+

Was ist ihr Beruf *

Wissenschaftlicher Mitarbeiter Informatik

Wieviel Erfahrung haben Sie mit Unity? (e.g. in Jahren) *

3

Wieviel Erfahrung haben Sie mit objektorientierten Programmiersprachen wie Java, c#...? (e.g. in Jahren) *

10

Fragen zur Person

Wie Alt bist du? *

- ○ 0-20
- ◉ 21-35
- ○ 36-50
- ○ 50+

Was ist ihr Beruf *

Werkstudent

Wieviel Erfahrung haben Sie mit Unity? (e.g. in Jahren) *

2

Wieviel Erfahrung haben Sie mit objektorientierten Programmiersprachen wie Java, c#...? * (e.g. in Jahren)

6

Aufgabe 1 Umfrage

SUS Fragebogenteil

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	◉	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly disagree	○	○	○	○	◉	Strongly agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

Sonstige Fragen

Wie lange war deine Bearbeitungszeit der Aufgabe? *

11 min 19 sek

Gibt es weitere Anmerkungen, welche du in Bezug zu dem Thema ansprechen möchtest?

Richtig geil

Aufgabe 1 Umfrage

SUS Fragebogenteil

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	◉	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	◉	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly disagree	○	○	○	○	◉	Strongly agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

Sonstige Fragen

Wie lange war deine Bearbeitungszeit der Aufgabe? *

9min50

Gibt es weitere Anmerkungen, welche du in Bezug zu dem Thema ansprechen möchtest?

Bug

Aufgabe 1 Umfrage

SUS Fragebogenteil

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	◉	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	◉	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly disagree	○	○	○	○	◉	Strongly agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

Sonstige Fragen

Wie lange war deine Bearbeitungszeit der Aufgabe? *

18min mit Rumprobieren

Gibt es weitere Anmerkungen, welche du in Bezug zu dem Thema ansprechen möchtest?

Crazy idea: Neben dem aktuellen Snapshot-Verfahren wäre ein Stack-Save-Verfahren (Command-Pattern / Replays) ultra.

Aufgabe 1 Umfrage

SUS Fragebogenteil

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

Sonstige Fragen

Wie lange war deine Bearbeitungszeit der Aufgabe? *

4min 50sec

Gibt es weitere Anmerkungen, welche du in Bezug zu dem Thema ansprechen möchtest?

SavableBehaviour als Erweiterung von MonoBehaviour, welches über Attribut markierte fields automatisch saven und loaden kann.

Aufgabe 1 Umfrage

SUS Fragebogenteil

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

Sonstige Fragen

Wie lange war deine Bearbeitungszeit der Aufgabe? *

6min10sec

Gibt es weitere Anmerkungen, welche du in Bezug zu dem Thema ansprechen möchtest?

Aufgabe 2 Umfrage

SUS Fragebogenteil

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	◉	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

Sonstige Fragen

Wie lange war deine Bearbeitungszeit der Aufgabe? *

40 min 20 sek (zu lang weil noob)

Gibt es weitere Anmerkungen, welche du in Bezug zu dem Thema ansprechen möchtest?

Sehr geil

Aufgabe 2 Umfrage

SUS Fragebogenteil

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

Sonstige Fragen

Wie lange war deine Bearbeitungszeit der Aufgabe? *

36

Gibt es weitere Anmerkungen, welche du in Bezug zu dem Thema ansprechen möchtest?

Aufgabe 2 Umfrage

SUS Fragebogenteil

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

Sonstige Fragen

Wie lange war deine Bearbeitungszeit der Aufgabe? *

45min

Gibt es weitere Anmerkungen, welche du in Bezug zu dem Thema ansprechen möchtest?

siehe Notizen. Events verfügbar in der Savable Component und als (Interface) methods für saved Classes wären super handy.

ich denke es wird sehr viel helfen, nochmal genau über die Benennung der Attribute (und evtl anderer Teile des Systems) nachzudenken --> erklärt sich selbst besser.

Aufgabe 2 Umfrage

SUS Fragebogenteil

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

Sonstige Fragen

Wie lange war deine Bearbeitungszeit der Aufgabe? *

22min

Gibt es weitere Anmerkungen, welche du in Bezug zu dem Thema ansprechen möchtest?

Noch etwas verwirrend. Besonders die Zusammenhänge zwischen Savable und Component, und Referenzen untereinander sind am Anfang unübersichtlich. Vielleicht mehr Zeit geben das Projekt vorher kennen zu lernen, oder Skripte zentraler im Projekt platzieren.

Aufgabe 2 Umfrage

SUS Fragebogenteil

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	◉	○	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly.

	1	2	3	4	5	
Strongly disagree	○	○	○	◉	○	Strongly agree

I found the system very cumbersome to use.

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

I felt very confident using the system.

	1	2	3	4	5	
Strongly disagree	○	○	◉	○	○	Strongly agree

I needed to learn a lot of things before I could get going with this system.

	1	2	3	4	5	
Strongly disagree	○	◉	○	○	○	Strongly agree

Sonstige Fragen

Wie lange war deine Bearbeitungszeit der Aufgabe? *

~ 22min

Gibt es weitere Anmerkungen, welche du in Bezug zu dem Thema ansprechen möchtest?

- Add tutorial
- Overall improvement for usability (is already ongoing ig)

Mitschrift User-Test

Anmerkungen

- Kompression und Verschlüsselung kommen gut an.
- Für Replays sind Commands besser geeignet.
- Das JSON ist schwer lesbar.
- Das Component-Saving wird als besser empfunden.
- Schnelle Implementierung möglich, wenn eine gute Dokumentation vorhanden ist oder es erklärt wird.

Anliegen

- Eine textuelle Anzeige im SaveLoadManager, die den SavePath zeigt.
- Mehr Automatisierung beim Aufbau des Setups im Usertest (Aufgabe 1) für das System.
- Automatisches Hinzufügen von Assets zur Asset-Registry.
- Gruppierungen für speicherbare GameObjects mittels Savable-Component.
- Das Formatieren des JSON, damit es direkt lesbar ist.
- Neben JSON auch binäre Formate unterstützen, wie FlatBuffer oder ProtoBuffer.
- Im [SavableObject] den Namen „useInheritance“ statt „declaredOnly“ verwenden.
- Die Benennung von Elementen überarbeiten, insbesondere für das Attribute-Saving.
- In der Dokumentation klar trennen, dass das Unity-Serialisierungssystem nicht mit dem Speichersystem verbunden ist.
- Das Callback-System muss überarbeitet werden, sodass die Savable-Component bestimmte Interfaces auf ihrem GameObject aufrufen kann.

Idee

- Markieren von Objekten zum Speichern mithilfe des Unity-Inspectors.

Bug

- Wenn ein Savable während des Unity-Playmodes zu einem Prefab hinzugefügt wird, wird es nicht korrekt zur Asset-Registry hinzugefügt.

Achtung die Erklärung muss eine Originalunterschrift haben!

Eigenständigkeitserklärung

Ich erkläre hiermit, dass

- Ich die vorliegende wissenschaftliche Arbeit selbständig und ohne unerlaubte Hilfe angefertigt habe,
- ich andere als die angegebenen Quellen und Hilfsmittel nicht benutzt habe,
- ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe,
- die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfbehörde vorgelegen hat.

Berlin, 25.09.2024
Berlin, Datum

R. Jaspers
Unterschrift

Vorname Name