HOCHSCHULE
MITTWEIDA
University of Applied Sciences

Fakultät Angewandte Computer- und Biowissenschaften

Professur Medieninformatik

Bachelorarbeit

Effizientere Lösung repetitiver Aufgaben durch Editor Tools in der Unity Engine

Niklas Räder

Mittweida, den 6. Oktober 2024

Erstprüfer: Prof. Dr.-Ing. Christian Roschke
Zweitprüfer: Manuel Heinzig, M. Sc.

Räder, Niklas
Effizientere Lösung repetitiver Aufgaben durch Editor Tools in der Unity Engine
Bachelorarbeit, Fakultät Angewandte Computer- und Biowissenschaften
Hochschule Mittweida — University of Applied Sciences, Oktober 2024

Referat

Diese Arbeit befasst sich mit der Entwicklung von Editor Tools in der Unity Engine zur Lösung repetitiver Aufgaben. Hierfür wurden für vier ausgewählte repetitive Aufgaben der Spieleentwicklung mit Unity, Editor Tools entwickelt. Um die Distribution der Editor Tools per UPM Packages zu vereinfachen, wurde ein Konzept für eine Kollektion aus zusammenarbeitenden Packages entwickelt. Die Auswirkungen der erarbeiteten Tools auf die Effizienz des Nutzers, wurden durch eine Evaluation mit festen Rahmenbedingungen ermittelt. Zusätzlich wurden die Testpersonen über einen längeren Zeitraum auf Verhaltensänderungen im Umgang mit den entwickelten Tools geprüft. Die in der Arbeit angeführten Ergebnisse belegen, dass die zugrundeliegende These, repetitive Aufgaben durch Editor Tools effizienter lösen zu können, zutrifft. Die Arbeit bietet eine gute Grundlage zur Entwicklung weiterer Editor Tools und weist wichtige Möglichkeiten zur Lösung zukünftig anfallender repetitiver Aufgaben auf.

Name: Räder, Niklas
Studiengang: Medieninformatik für Interaktives Entertainment
Seminargruppe: MI20w2-B
English Title: More efficient solution of repetitive tasks through editor tools in the Unity Engine

Inhaltsverzeichnis

1. Einführung und Motivation

Videospiele sind als modernes Unterhaltungsmedium aus der Freizeit vieler Menschen nicht mehr wegzudenken. Im Vergleich zu anderen Unterhaltungsmedien, wie Rundfunk und Fernsehen, entstanden die ersten Videospiele Ende der 50er. Sie dienten den Technikenthusiasten als Experiment im Umgang mit dem neu entwickelten Medium Computer. Mit den Entwicklungen der folgenden Jahre wurden Mitte der 1970er Jahre Spielhallen-Klassiker wie Pong und Space Invaders erstmals als Unterhaltungsmedium zur Verfügung gestellt. Um die Videospiele in die Wohnungen der Spieler zu bringen, entstanden in den 1980er Jahren die ersten Heimcomputer und Spielekonsolen. Seit diesem goldenen Zeitalter hat sich die Welt um die Videospiele stetig weiterentwickelt. [ddGBe]

Eine dieser Entwicklungen war die für das 1993 veröffentlichte Spiel Doom entwickelte Doom-Engine. Sie gilt als die erste Game-Engine (engl. für ein Programm zur Entwicklung von Spielen). Die Doom-Engine konnte keine 3D-Assets (engl. für dreidimensionale Objekte), sondern lediglich 2D-Assets (engl. für zweidimensionale Grafiken) darstellen, dies jedoch mit erstaunlicher Geschwindigkeit. Aufgrund dieser Geschwindigkeit und einer von den Entwicklern erstellten Illusion wurde mit Doom eines der ersten, für den Spieler dreidimensionalen Spielen veröffentlicht. Aufgrund der Vorzüge einer eigenen Engine folgten der Doom-Engine schnell weitere Game-Engines, wie beispielsweise die Ultima Underworld Engine oder die Voxel Engine. Jedes dieser Programme ermöglichte die Entwicklung von Spielen durch die Bereitstellung technischer Grundfunktionen. Zu diesen Grundfunktionen lassen sich das Rendern von Objekten, Animationen, Physikberechnungen, Nutzereingaben sowie auch die Kollisionsberechnung zwischen Objekten zählen. In der modernen Spieleentwicklung ist die Entwicklung einer eigenen Engine nur notwendig, wenn zur Umsetzung der Spielidee technische Funktionen benötigt werden, die von keiner bestehenden Engine abgedeckt werden. Vergleichend ist festzustellen, dass sich das Kernziel der Game-Engines von der Doom-Engine bis zu den modernen Engines wie Unreal Engine, Unity oder Godot nicht verändert hat. Game-Engines stellen technische Grundfunktionen sowie Werkzeuge zur Erstellung von digitalen Medien zur Verfügung, um den Entwicklern größtmöglichen Fokus auf die kreative Umsetzung zu ermöglichen. [PGB12]

Durch die modernen Game-Engines ist das Entwickeln von Videospielen Vielen zugänglich geworden. Es braucht kein Hintergrundwissen im Bereich des Rendering oder der Informatik, um ein Spiel zu erstellen. Durch beispielsweise die Bereitstellung des Blueprint Systems von Unreal Engine 5, muss der Entwickler kein Wissen im Bereich C++ mitbringen, um Spielmechaniken zu implementieren. Aufgrund des einfachen Zugangs zu diesen Programmen, entsteht jährlich eine umfangreiche Anzahl an Videospielen. Dies spiegelt sich in den steigenden Zahlen der neu erscheinenden Indie-Spiele (Begriff für Spiele, welche unabhängig von der Finanzierung großer Publisher entwickelt werden) wieder [Ste]. Indie-Spiele werden immer beliebter, da sie regelmäßig mit Innovation und einzigartigen stilistischen Mitteln überraschen. Dies ist notwendig, um aus der

Masse der veröffentlichten Spiele hervorzustechen. Zu den wohl bekanntesten Indie-Titeln gehören “Stardew Valley“ oder “Among Us“. Beide Titel wurden von je einer Person beziehungsweise einem kleinen Team aus Entwicklern erschaffen. Durch die geringe Anzahl der Entwickler können die Entwicklungskosten gering gehalten werden, wodurch Indie-Spiele zu einem günstigeren Preis angeboten werden können, als ein Vollpreis-Titel eines bekannten Studios. [ddGBe21]

Auch renommierte Studios, wie beispielsweise “Epic Games“, “CD Project Red“ oder “miHoYo“ nutzen bestehende Engines zur Entwicklung neuer Spiele. Durch die Verwendung von Engines in großen Entwicklerteams, entstanden neue Arbeitsbereiche. Technical-Artist ist einer der entstandener Beruf. Er beschäftigt sich unter anderem mit der Entwicklung von Materialien, Animationen, visuellen Effekten und der Leistungsoptimierung des Spiels. Diese Tätigkeiten werden in den meisten Game-Engines nativ zur Verfügung gestellt, wodurch die Arbeit der Technical-Artists eng mit der verwendeten Engine verbunden ist. Auch die Verwaltung sowie Qualitätsprüfung der importierten Assets muss in der Engine gewährleistet werden. Der notwendige Aufwand steigt mit der Anzahl der Assets an und beschränkt sich meist auf repetitive Aufgaben.

Jene repetitiven Aufgaben effektiver zu lösen, ist eine weitere Aufgabe eines Technical-Artists. Hierzu werden Hilfsprogramme (im weiteren Tools genannt) erstellt, welche repetitive Abläufe einkürzen und Prozesse automatisieren. Funktionen, die nicht nativ von der Game-Engine bereitgestellt werden, können mittels Tools nachträglich hinzugefügt werden. Die Nutzung der Engine für das bestehende Projekt maßzuschneidern, um die Arbeit effizient und leicht zu gestalten, ist der Kernschwerpunkt der Tool-Entwicklung.

1.1. Aufgabenbeschreibung

Diese Arbeit beschäftigt sich mit der Entwicklung und Implementation von Tools für den Editor der Unity Engine (fortan Editor-Tools genannt). Die Steigerung der Effizienz durch die entstandenen Tools ist anhand von Probandentests zu evaluieren. Hierbei liegt der Schwerpunkt auf der zeitlichen Effizienzsteigerung sowie der Verringerung des benötigten Arbeitsaufwandes.

Schwerpunkte der Arbeit sind neben der Erstellung von vier Editor-Tools und der Auswertung dieser durch Probandentests, die Entwicklung einer vereinfachten Distribution der Tools sowie die Konzipierung der Schnittstellen mit dem Nutzer. Die Bereitstellung der Tools wird für den Nutzer vereinfacht, indem ein zentrales Unity Package als Schnittstelle zu den verbleibenden Funktionen dient.

1.2. Zielstellung und Aufbau der Arbeit

Hauptziel dieser Arbeit ist die Erstellung möglicher Lösungen für ausgewählte repetitive Aufgaben innerhalb der Unity Engine durch die Entwicklung von Editor-Tools. Aufbauend auf den Resultaten soll zudem aufgezeigt werden, wie weitere repetitive Aufgaben in Zukunft einfacher verbessert werden können.

Die Arbeit setzt sich aus insgesamt sechs Kapiteln zusammen. Im ersten Kapitel wird zunächst kurz auf die Geschichte der Videospiele sowie die Entstehung der ersten Game Engines eingegangen. Nachdem die Auswirkungen der Game Engines sowie die Kernaufgaben der Tool-Entwicklung erläutert wurden, wird eine Grundlage für die kommenden Kapitel geschaffen. Es wird erneut detailliert erklärt, welche Vorteile eine Game Engine bietet und was eine solche definiert. Im besonderen wird hierbei auf die Unity Engine eingegangen. Anschließend wird die Bedeutung der Tool-Entwicklung im generellen Kontext sowie bei der Software-Entwicklung und innerhalb anderer Engines dargestellt.

Kapitel drei beschäftigt sich zunächst mit dem Konzept einer Kollektion individueller Unity Packages (Begriff für importierbare Inhalte) zur vereinfachten Nutzung für den Entwickler. Anschließend werden mögliche Lösungen für die ausgewählten repetitiven Aufgaben konzipiert. Um die Effizienz und Nutzbarkeit zu untersuchen, wird eine Evaluation geplant.

Kapitel vier befasst sich mit der erfolgten technische Umsetzung. Hierbei wird zunächst auf den Aufbau der Unity Packages, die Kernfunktionen der Kollektion, sowie die Umsetzung des User Interface eingegangen. Anschließend werden in Kapitel fünf die Ergebnisse der Evaluation hinsichtlich der Effizienzsteigerung untersucht. Um die Arbeit abzuschließen, werden die Ergebnisse anschließend zusammengefasst und Möglichkeiten zur vereinfachten Lösung zukünftig anfallender repetitiver Aufgaben aufgezeigt.

2. Grundlagen

2.1. Game Engines

Eine Game Engine beschreibt eine Software zur Entwicklung von Videospielen. Die Hauptaufgabe einer Game Engine ist die Bereitstellung wiederverwendbarer Komponenten zur Entwicklung verschiedener Spiele. Hierzu gehören nicht nur Funktionen, welche den Entwicklungsprozess beschleunigen sondern auch Systeme, welche in nahezu jedem Spiel benötigt werden und somit den Grundstein eines jeden Videospiels bilden. Zu diesen gehören das Rendering (Grafikausgabe), Physikberechnungen, Audio, Animationen, die Benutzeroberfläche, die Verarbeitung von Nutzereingaben sowie die Verarbeitung und Speicherung von zur Entwicklung genutzter Assets. Basierend auf den Spezialisierungen der Engine werden weitreichendere Grundfunktionen zur Verfügung gestellt.

Der Unterschied zwischen Game Engine und Spiel ist in der Architektur zu erkennen. Während ein Spiel, spezialisierte, fest niedergeschriebene Logik besitzt, welche nur für dieses Spiel entwickelt wurde, besitzt eine Game Engine flexible Logik. Systeme und Funktionen können durch reine Datenanpassungen an die gewünschten Anforderungen angepasst werden, ohne den dahinter liegenden Code zu verändern. Durch diese datengetriebene Architektur können vordefinierte Systeme je nach Spiel für verschiedene Zwecke verwendet werden. *"Arguably a data-driven architecture is what differentiates a game engine from a piece of software that is a game but not an engine."* [Gre18, Kapitel 1.3 *"What is a Game Engine?"*] Diese Definition einer Game Engine hebt erneut hervor, dass die Anpassbarkeit und die Wiederverwendbarkeit die prägnanten Merkmale einer Game Engine sind.

Game Engines können speziell für ein Genre oder eine Plattform entwickelt werden. In einem solchen Fall eignen sich jene Engines für die Entwicklung eines Spiels des Genres besonders. Die meisten Game Engines jedoch streben eine Plattformunabhängigkeit sowie die Unterstützung aller Genres an. Dies führt meist jedoch zu Einbußen in der Optimierung der Spiele, wie es Jason Gregory in seinem Buch mit folgendem Zitat beschreibt. *"It's safe to say that the more generalpurpose a game engine or middleware component is, the less optimal it is for running a particular game on a particular platform."* [Gre18, Kapitel 1.3 *"What is a Game Engine?"*]

2.2. Unity Engine

Die Unity Game Engine oder kurz Unity ist gemeinsam mit der Unreal Engine und Godot eine der populärsten Game Engines der modernen Spieleentwicklung. Seit der Version Unity 1.0 im Jahr 2005 entwickelte sich die Engine stetig weiter. Unity spezialisiert sich hierbei nicht auf einen

bestimmten Aspekt der Spieleentwicklung oder eine spezielle Plattform, sondern ermöglicht die Entwicklung einer großen Vielfalt von Spielen. Durch die Game Objects und das Komponenten-System auf welches nachfolgend eingegangen wird, wird die Entwicklung jeglicher Genres ermöglicht. Das Ziel der Unity Engine ist es, die Spieleentwicklung zu vereinfachen und eine breite Basis an Funktionen bereitzustellen [Sch24, S.41].

Durch die einfache Bedienung und die Zugänglichkeit der Engine für die Betriebssysteme Windows, Mac und Linux ist jeder in der Lage, die Unity Engine zu verwenden. Ebenfalls ist die Nutzung der Engine zunächst kostenfrei. Nach Überschreitung eines festgelegten Schwellwerts durch die Einnahmen des veröffentlichten Spiels sowie finanzieller Förderungen, muss eine kostenpflichtige Lizenz erworben werden. Aufgrund dieser Vorteile verwenden viele Hobbyisten sowie auch professionelle Entwickler und Studios die Unity Engine, um ihre Spiele zu entwickeln. Bekannte Beispiele hierfür sind *miHoYo* mit *Genshin Impact*[1], *Blizzard Entertainment* mit *Hearthstone*[2] oder das Spiel *Ori and the Blind Forest*[3] von *Moon Studios*.

Da Unity bereits seit 2005 entwickelt wird, sind die bestehenden offiziellen Versionen der Engine sehr stabil. Ebenfalls durch die lange Lebenszeit der Engine sowie der großen Anzahl an Entwicklern, die die Engine nutzen, ist eine reiche Menge an Informationen erhältlich. Auftretende Probleme sind oft schnell durch Forum-Einträge und die offizielle Dokumentation lösbar. [Sch24, S.41f]

Nachfolgend werden einige, in der Arbeit verwendete, wichtige Begrifflichkeiten erläutert. Die folgenden Begrifflichkeiten sind im Kontext der Unity Engine zu verstehen.

Game Object und Komponenten
Mit Unity erstellte Spiele bestehen aus Szenen. Eine solche Szene kann die gesamte Spielwelt, Teile dessen oder einzelne Level beinhalten. Als Game Object werden im Allgemeinen alle Objekte innerhalb einer solchen Szene bezeichnet. Das Game Object selbst ist das Grundelement eines jeden Objektes im Spiel und wird anschließend durch die genutzten Komponenten, hinsichtlich seiner Optik sowie Logik definiert. [Sch24, S.74f]

Während ein Game Object als leere Hülle [Sch24, S.77] bezeichnet werden kann, definieren die Komponenten (in der Engine Components genannt) die Eigenschaften und das Verhalten dieser leeren Hülle. Hierbei ist jede Komponente möglichst in sich abgeschlossen und beschränkt sich auf die von ihr ausgeführten Tätigkeit. So beispielsweise gibt es die Komponente des “MeshFilters“, welche ein “Mesh“ (dreidimensionales optisches Objekt) speichert. Das Hinzufügen der MeshFilter Komponente verzeichnet keine Änderung am Game Object. Erst mit dem Hinzufügen einer “MeshRenderer“ Komponente erhält das Game Object ein neues Aussehen. Die MeshRenderer Komponente ist lediglich zuständig ein gegebenes Mesh zu rendern. Nur durch die Bereitstellung des Meshs durch die MeshFilter Komponente kann das gewünschte Aussehen erzielt werden. Das Zusammenspiel zwischen den in sich abgeschlossenen Komponenten ist daher von integraler Bedeutung.

[1] Genshin Impact, miHoYo, besucht am 03.10.2024
[2] Hearthstone, Blizzard Entertainment, besucht am 03.10.2024
[3] Ori and the Blind Forest, Moon Studios, besucht am 03.10.2024

Basierend auf den Grundprinzipien einer datengetriebenen Architektur besitzt jede Komponente individuelle Eigenschaften, welche die Funktionsweise der Komponente genauer definieren ohne den zugrundeliegenden Code zu bearbeiten. [Sch24, S.76]

Prefab
Prefabs sind eine von Unity bereitgestellte Art von Objekten, welche die Arbeit mit sich wiederholenden Objekten stark vereinfacht. Beim Duplizieren eines Game Objects, welches mehrfach in einer Szene vorkommen soll, wird eine identische Kopie des ursprünglichen Objekts erstellt. Diese behält alle Komponenten sowie deren Einstellungen bei. Sollte das Objekt nun verändert werden müssen, so müssen die einzelnen Anpassungen der Werte auf jedem der duplizierten Objekte sowie dem ursprünglichen Objekt vorgenommen werden. Um diese Arbeit zu minimieren besteht die Möglichkeit, ein Game Object in ein Prefab umzuwandeln. Hierbei wird eine exakte Kopie des Objekts in den Projektdateien als Prefab abgespeichert. Dieses Prefab kann wie ein normales Game Object bearbeitet werden. Prefabs können nach ihrer Erstellung in Szenen eingefügt werden. Eingefügte Prefabs werden zu Prefab Instanzen. Jede Prefab Instanz übernimmt selbstständig alle Änderungen, die an der Prefab Datei vorgenommen werden, was den Arbeitsaufwand deutlich senkt. Ebenfalls können Instanzen lokale Überschreibungen besitzen, bei welchen sie die Werte der Prefab Datei durch ihre lokalen Werte überschreiben. [Sch24, S.395-407]

MonoBehaviour
Die Unity Engine stellt eine Vielzahl an Komponenten zur Verfügung. Allerdings ist bei der Entwicklung der allermeisten Spiele die Erstellung eigener Komponenten unabdingbar. Diese individuellen Komponenten beinhalten die für das Spiel spezifische Logik. Casey Hardman beschreibt MonoBehaviours in seinem Buch wie folgt. *"[...]just know that the " : MonoBehaviour" part is what makes the class a script that can be added as a component, not just a normal class."* [Har23, Kapitel 10 *"Working with Scripts"*] MonoBehaviours sind eine von Unity vordefinierte Klasse, von welcher geerbt werden muss, um ein Skript als Komponente zu definieren. Ohne die geerbten Funktionen des MonoBehaviours ist es nicht möglich das Skript einem Game Object anzufügen.

Playmode und Editmode
Die Erstellung von Anwendungen findet im Unity Editor statt. Standardmäßig befindet sich der Editor im Editmode. Dies bedeutet, dass die Szenen des Spiels bearbeitet werden können und das Programm sich allgemein im Bearbeitungsmodus befindet. Um nicht vor jedem Testlauf das Spiel builden (in eine ausführbare Datei wandeln) zu müssen, besteht die Möglichkeit, das Spiel im Unity Editor zu simulieren. Während der Editor das Spiel simuliert werden alle vorgenommenen Änderungen an den Szenen nicht gespeichert. Dieser Zustand ist als Playmode bekannt, da der Entwickler in dieser Zeit das simulierte Spiel spielt.

Inspector
Der Inspector ist ein Editor-Fenster des Unity Editors. Er ist ein integraler Bestandteil, um die Objekte des Spiels zu bearbeiten. Durch die Auswahl eines Game Objects wird dieses im Inspector angezeigt. Dem Entwickler steht es nun frei die Informationen aller auf dem Objekt liegenden Informationen auszulesen und zu bearbeiten. [Har23, Kapitel 2 *"Unity Basics"*]

Scriptable Object
Scriptable Objects sind eine weitere von Unity vordefinierte Klasse, von welcher Skripts erben können. Während MonoBehaviours der Erstellung neuer Komponenten dienen, so werden Scriptable Objects genutzt, um Daten abzulegen. Das Skript, welches von der Klasse der Scriptable Objects erbt, legt die Struktur sowie die enthaltenen Daten des Objekts fest. Anschließend können im Projekt Instanzen des erstellten Skripts in Form von Scriptable Objects erstellt werden. Die Eigenschaften dieser Objekten können über den Inspector bearbeitet werden. Anschließend können die enthaltenen Daten beispielsweise in einem MonoBehaviour verwendet werden. [Sch24, S.383]

2.3. Tool-Entwicklung

Jeder ausgeübte Beruf besitzt seine eigenen Tools [RF12, S.1]. Das Wort Tools ist in dieser Aussage breit definierbar. Es bezeichnet Werkzeuge, welche aus der Verbindung von theoretischem Wissen und der praktischen Anwendung entstehen. Diese Werkzeuge können traditionell in physischer Form zur Anwendung kommen wie beispielsweise der Reflexhammer, welcher über einen Zeitraum von mehreren hundert Jahren entwickelt wurde [IVG24]. Im Kontext dieser Arbeit jedoch bezeichnen Tools Software Produkte und in der Software Entwicklung genutzte Werkzeuge. Tools entwickeln sich ähnlich des Prinzips der natürlichen Selektion. Durch Veränderungen der Anforderungen, Nachfrage nach einer bestimmten Funktion oder der Verbesserung konkurrierender Produkte, befinden sich Tools in einem ständigen Wandel. Ein Tool kann hierbei verworfen, an die gegebenen Voraussetzungen angepasst oder mit innovativen Funktionen erweitert werden, um die Entwicklung voranzutreiben. Das Ziel dieses stetigen Wandels ist es, die bestmögliche Optimierung zur Verrichtung der Arbeit zu erreichen. [RF12, S.1f]

Wissenschaftliche Arbeiten belegen, dass die Entwicklung von Tools sowie auch der zuvor genannte, stetige Prozess zur Verbesserung dieser, einen zusätzlichen Arbeitsaufwand erfordern. Dieser entsteht meist durch die Integration neuer Versionen sowie der Bewältigung technischer Herausforderungen [AT24, S.274f]. Neben der Entwicklung der Tools braucht es anfänglich auch zusätzlichen Aufwand für die Schulung der Nutzer, damit diese den richtigen Umgang mit den Tools erlernen. Hierbei ist es möglich, dass die größte Schwierigkeit nicht im Verständnis des Tools, sondern der Akzeptanz gegenüber eines neuen Werkzeuges liegt [Ver18, S.57ff]. Trotz dieses Mehraufwandes bestätigen Studien, dass die Entwicklung von Tools [GH89, S.15f] sowie die Einarbeitung von Nutzern in neue Systeme [Pra22, S.4722] langfristig zu Zeiteinsparungen und Kostensenkungen führen kann, wodurch die anfänglich anfallenden Zusatzkosten sowie der Arbeitsaufwand kompensiert werden.

2.3.1. Tool-Entwicklung in der Software-Entwicklung

Software-Tools können, basierend auf ihrer Funktionsweise, grob in die drei Kategorien, kognitive *(cognitive)*, ergänzende *(augmentive)* und aufzeichnende *(notational)* Tools eingeteilt werden. Zu den Vorteilen moderner Software-Tools gehören die verbesserte Qualität des Endproduktes und die erhöhte Effizienz des Nutzers. Ebenfalls legen moderne Tools Wert auf die Verbesserung der

Nutzerzufriedenheit bei Ausführung ihrer Arbeit. Entgegen dieser Vorteile stehen, wie bereits genannt, die anfallenden Kosten zur Entwicklung und Instandhaltung der Tools sowie der Schulung der Nutzer. [GH89, S.1-3]

Kognitive Tools beschreiben weniger eine entstandene Software als Prinzipien und Techniken, welche zur Entwicklung von Software verwendet werden. Sie verbessern die Problemlösefähigkeit des Nutzers oder ermöglichen es ihm, zuvor unlösbare Probleme durch neue theoretische Ansätze zu lösen. [GH89, S.1]

Ergänzende Software-Tools helfen dem Nutzer wie ein Hammer oder ein Schraubendreher einem Handwerker helfen [GH89, S.1]. Die Tools sind meist selbst vollwertige Programme, welche dem Nutzer die Verwendung bestehender Funktionen vereinfachen oder gänzlich neue Funktionalitäten bereitstellen. Durch die effizientere Nutzung der gegebenen Funktionen oder Bereitstellung neuer Funktionalitäten, kann die Produktivität des Nutzers gesteigert werden. Aufzeichnende Tools hingegen dienen als Medium, Problemlösungen sowie Konzepte zu kommunizieren. Sie können hierbei in Form einer entwickelten Software oder Methodik, wie beispielsweise die verwendete Sprache oder bestimmte Notationen vorkommen. Durch die Verwendung effizienter Kommunikationsmedien kann der Nutzer seine Vorstellungen verständlicher ausdrücken. [GH89, S.1f]

Ein ideales Beispiel für die Fusion aus ergänzenden und aufzeichnenden Tools sind CAD-Programme, welche große Anwendung in der Architektur finden. Diese CAD-Programme ermöglichen es, detaillierte Architekturzeichnungen anzufertigen. Diese Zeichnungen können anschließend für den Bau des Gebäudes weiterverwendet werden. Durch die Weiterentwicklung der CAD-Programme jedoch, können diese ebenfalls als Medium genutzt werden, um Entwürfe mit wissenschaftlichen Disziplinen wie der Informatik, Geometrie, Materialkunde und Kybernetik zu verbinden. Durch diese Verbindung ist es den Architekten nicht nur möglich, ihre Arbeit effizienter zu verrichten, sondern sie können ihre Vorstellungen verständlicher und präziser kommunizieren. [SB11]

Ein weiteres Beispiel für Software-Tools sind Game Engines. Da in den vorangegangenen Kapiteln bereits erarbeitet wurde, dass eine Engine dem Entwickler Funktionalitäten bereitstellt, welche dieser verwenden kann um seiner Arbeit effizienter nachzugehen, können Game Engines hinlänglich als ergänzendes Tool angesehen werden. Game Engines befinden sich wie andere Software-Tools in einem stetigen Prozess des Wandels, in welchem sie sich an die Bedürfnisse der Entwickler und die aktuellen Technologien anpassen. Ausschweifend können Game Engines ebenfalls als aufzeichnendes Tool bezeichnet werden, da sie ähnlich wie die zuvor genannten CAD-Programme zur Visualisierung und Übermittlung der Idee des Entwicklers verwendet werden können.

2.3.2. Tool-Entwicklung in der Unity Engine

Auf Grundlage der Entwicklungen der vergangenen Jahre ist die Unity Engine nicht mehr nur Game Engine, sondern expandiert ebenfalls in andere Industrien, welche von der Nutzung der Engine profitieren. Zu diesen Industrien gehören beispielsweise die Filmindustrie, Animationen sowie Anwendungen im VR/AR Bereich. Dieser Trend ist nicht nur bei der Unity Engine sondern auch anderen Game Engines wie der Unreal Engine 5 auffindbar. Der Vorteil der Unity Engine

gegenüber seiner Konkurrenz ist die überaus ausgeprägte Modularität. [Kok21, Kapitel 1 *"Getting Started"*]

Unter dieser Modularität versteht man die Auftrennung der einzelnen Funktionen in individuelle Pakete. Diese Pakete nennt man auch Packages. Unity unterstützt zwei Arten von Packages. "Asset Packages" sind eine Ansammlung von Dateien, welche aus einem bestehenden Projekt exportiert und in ein anderes importiert werden können. Sie sind am ehesten vergleichbar mit *.zip* Dateien. Die zweite Form von Packages bilden die "UPM Packages". UPM steht hierbei für Unity Package Manager, da diese Packages, direkt über Unitys Package Manager, aus dem Unity internen Register oder per URL importiert werden können. Sie enthalten ebenfalls eine Menge von Dateien. Diese können jedoch einfacher aktualisiert und von dem Entwickler des Packages instand gehalten werden.

Neue von Unity entwickelte Systeme werden zumeist nicht direkt in die Versionen der Engine integriert, sondern sind als UPM Package erhältlich. Dies bringt den Vorteil, dass nicht kompatible, voneinander unabhängige Systeme für die gleiche Unity Version existieren können und nicht notwendigerweise aufeinander abgestimmt werden müssen, da der Entwickler die Kontrolle darüber behält, welche Funktionen in sein Projekt integriert werden sollen. Desweiteren können für neue Versionen entwickelte Funktionen auf diesem Weg teilweise, unter zusätzlichem Aufwand, für ältere Versionen der Engine nachjustiert werden. Die Entwicklung solcher modularen Tools kann man beispielsweise an der Entwicklung des "ShaderGraphs" erkennen. Dies ist ein Tool für die Unity Engine entwickelt von Unity, welches eine grafische Oberfläche zur Erstellung von Shadern bereitstellt. Der Entwickler muss hierbei nicht länger eigenen Shadercode schreiben. Die Nutzung des ShaderGraphs ist jedoch nur möglich, wenn das Projekt die von Unity entwickelten neueren Render Pipelines wie die URP (Universal Render Pipeline) verwendet. Die URP ist ebenfalls ein eigenständiges System, welches als UPM Package verfügbar ist und eine weitreichende Bandbreite an neuen Tools bereitstellt. [Kok21, Kapitel 1 *"Getting Started"*]

Nicht nur den Entwicklern von Unity ist dieses modulare System zugänglich. Die verwendeten Schnittstellen (APIs) und Systeme, wie beispielsweise das UI Toolkit, stehen allen Nutzern der Engine zur Verfügung. Hierdurch wird die Entwicklung neuer Funktionen in Form von Tools stark gefördert. Die Entwicklung brillanter Tools kann ebenfalls dazu führen, dass Unity die Funktionen des Tools in die Engine nativ integriert. So beispielsweise wurde 2017 das "TextMesh Pro"[4] Tool von Stephan Bouchard in Unity aufgenommen und gilt seither als Standard für Schrift in der Benutzeroberfläche.

Die Flexibilität der Engine ermöglicht eine einfache Entwicklung von Tools. Gleichzeitig schafft sie allerdings auch die Notwendigkeit Tools zu erstellen. Da jedes Projekt andere Systeme und Funktionen von Unity, als UPM Packages importiert hat und externe Tools importiert haben kann, müssen viele Tools projektspezifisch angepasst werden. Um Kompatibilitätsprobleme sowie die individuellen Bedürfnisse des Unity Projektes zu erfüllen, werden oft interne Tools erstellt. Tools, welche veröffentlicht und für den generalisierten Gebrauch entwickelt wurden, müssen dafür Sorge tragen, keine Kompatibilitätsprobleme zu verursachen und alle benötigten Grundsysteme bereitzustellen.

[4] Integration von TextMesh Pro in Unity, besucht am 05.10.2024

Um diese Schwierigkeiten zu überwinden, entwickelte Unity die Funktion Abhängigkeiten (Dependencies) anzugeben. Dies sind Packages, also Systeme und Funktionen, die von Unity oder externen Quellen entwickelt wurden. Während des Imports des Packages in welchem das Tool enthalten ist, werden dann die benötigten Dependencies ebenfalls importiert. Somit wird vermieden, essentielle und häufig verwendete Systeme und Funktionen in die Packages zu implementieren, da diese über eine angegebene Dependency importiert werden können.

2.3.3. Tool-Entwicklung in anderen Engines

Die Entwicklung von Tools zur Erweiterung einer Game Engine, um neue Funktionen zu erzeugen ist kein auf die Unity Engine lokalisiertes Phänomen. Auch in anderen Engines unterstützen sich Entwickler gegenseitig durch die Entwicklung neuer Tools. Als Beispiele solcher Engines sollen in diesem Fall die Godot Engine und Unreal Engine aufzeigen, wie die Tool-Entwicklung in anderen Engines stattfindet.

Godot Engine
Im Gegensatz zu Unity und Unreal ist die Godot Engine ein Open-Source Projekt. Dies bedeutet, dass die Engine von keiner kommerziellen Firma entwickelt wird. Der Quellcode der Engine ist öffentlich und frei zugänglich. Jeder Entwickler kann Änderungen an den Funktionen der Engine vornehmen. Diese Änderungen jedoch müssen zunächst von einer Vielzahl anderer Entwickler abgesegnet werden, bevor diese in die veröffentlichte Version übernommen werden. Aus diesem Grund sind viele individuelle Entwickler für die Kontribution neuen Quellcodes, dem Überprüfen gestellter Änderungsanfragen sowie der generelle Beteiligung an der Entwicklung der Engine nötig. [Sal22, S.15]

Ähnlich wie bei Unity ist eine Kernausrichtung der Godot Engine, häufig verwendete Funktionen kompakt in das Hauptprogramm zu integrieren, während weniger häufig genutzte Funktionen in Packages ausgelagert werden [Sal22, S.15]. Die Packages oder in Godot Plugins genannt, werden meist zunächst von individuellen Entwicklern erstellt. Da Godot jedoch ein Open-Source Projekt ist, sind ebenfalls die Plugins eigenständige Open-Source Projekte.

Die Distribution von Tools in Godot unterscheidet sich stark von der Distribution innerhalb Unitys. Während Unity die zwei Optionen des Veröffentlichens über den eigenen Asset Store (Website) sowie die Distribution über GitHub Adressen anbietet, bedient sich Godot eines manuelleren Ansatzes. Die offizielle Dokumentation der Godot Engine[5] empfiehlt die Veröffentlichung von Tools in der dafür erstellten Asset Libary (Website). Diese Website umfasst eine Vielzahl hilfreicher Tools und bietet Möglichkeiten diese herunter zu laden. Die herunter geladenen Plugin Dateien müssen anschließend in das Addons Verzeichnis des Projektes verschoben und in den Einstellungen aktiviert werden. Um die Verbindung mit dem Projekt herzustellen und um als Plugin erkannt zu werden, benötigt dieses speziell dafür erstellte Dateien, ähnlich der package.json Datei welche bei der Entwicklung eines UPM-Packages in der Unity Engine benötigt wird. Nicht jedes Plugin muss öffentlich zugänglich sein. Durch die Erstellung eines lokalen Plugins im Addons Verzeichnis des Projektes können projektspezifische Plugins entworfen werden.

[5] Godot Dokumentation zur Installation eines Plugins, besucht am 05.10.2024

Da die Godot Engine erst 2014 veröffentlicht wurde [Sal22, S.14], bestehen bisher im Vergleich zu der umfassenden Lektüre und Diskussionsvielfalt der Unity oder Unreal Engine wenige Hilfestellungen für die Erstellung von Plugins. Durch die Gemeinschaft der mitwirkenden Entwickler jedoch werden die Schnittstellen zur Entwicklung von Plugins stetig verbessert. Ebenfalls nehmen die Veröffentlichungen wissenschaftlicher Arbeiten, welche die Godot Engine als Tool nutzen zu. So beispielsweise zeigt eine wissenschaftliche Arbeit im Bereich des "Deep Reinforcement Learning", dass die Godot Engine genutzt werden kann, um eine realistische und sichere Trainingsumgebung bereitzustellen [RM24]. Durch die Entwicklung lokaler Plugins konnte eine Kommunikation zwischen der Engine (virtuelle Welt) und den Ergebnissen und Simulationen der realen Welt hergestellt werden.

Unreal Engine 5

Die Unreal Engine von Epic Games ist eine der wohl bekanntesten Game Engines. Ähnlich wie in der Entwicklung der Unity Engine kann man erkennen, dass die Unreal Engine, allem voran die neueste Version, die Unreal Engine 5, vermehrt in anderen Industrien wie der Filmindustrie Verwendung findet. [BR23]

Um sich der erweiterten Zielgruppe anzupassen, verbessert Epic Games ihre Engine weitgehend auch hinsichtlich der Bedürfnisse der Filmproduktion. Da der primäre Aspekt der Engine jedoch der einer Game Engine ist, besteht nicht nur für die Entwickler von Spielen, sondern auch die Entwickler digitaler Filmstudios die Notwendigkeit spezialisierte Tools zu erstellen. Aus diesem Grund werden die Schnittstellen und Funktionen zur Entwicklung von Tools stetig verbessert. Anfänglich wurden Tools in der Unreal Engine in C++ geschrieben. Unter Verwendung von C++ konnten die bereitgestellten Funktionen der Engine angesprochen werden. Der Vorteil von C++ ist die Schnelligkeit und Präzision mit welcher die Tools agieren können. Jedoch ist dafür zunächst ein intensives Wissen über die Programmiersprache C++ sowie den Aufbau und die bestehenden Funktionen der Engine notwendig. Um die Entwicklung von Tools zugänglicher zu machen, wurden später Blueprint Lösungen bereitgestellt. Das Blueprints System ist eine visuelle Skripting Umgebung, für dessen Benutzung kein Vorwissen im Bereich einer Programmiersprache notwendig ist. Lediglich ein Verständnis über die genutzte Logik wird vorausgesetzt. Durch die Anbindung des Blueprint Systems an die internen Funktionen des Editors wird die Entwicklung von Tools stark zugänglicher.

Ähnlich wie in Unity ist es mittels geschriebener Tools möglich, eigene vollends neue Fenster innerhalb des Editors zu erschaffen. Diese Fenster stellen die Anbindung zu den entstandenen Funktionen dar und agieren als Schnittstelle zu dem Nutzer des Tools. Den Funktionen, welche durch das Blueprint System geschaffen werden können, sind keine Grenzen gesetzt. Beispielsweise ist die Automatisierung der Assetverwaltung eine oft umgesetzte und enorm wichtige Funktion. Dies zeigen etablierte Tools aus dem Unreal Engine Marketplace (Website) wie der One Click Asset Manager[6].

Ein weiteres Beispiel für die Umsetzung eines Tools in der Unreal Engine 5 zeigt eine wissenschaftliche Arbeit von 2022. Durch die Verwendung der von Unreal bereitgestellten Systeme wurden Tools

[6] One Click Asset Manager, besucht am 05.10.2024

für Reinforcement Learning entwickelt. Die Arbeit betont erneut die einfache Bedienung des Blueprint Systems. Jedoch zeigt es auch, dass eine Balance zwischen der Nutzung des Blueprint Systems und der Entwicklung von Schnittstellen mittels C++, integral für die Entwicklung umfangreicher Systeme ist. [SR22]

Eine weitere Möglichkeit der Tool-Entwicklung innerhalb der Unreal Engine 5 ist die Verwendung von Python. Mittels der Programmiersprache Python ist es möglich, Programme zu schreiben, welche außerhalb der Game Engine laufen. Hierbei können diese auf alle Schnittstellen zugreifen, welche auch über das Blueprint System erreichbar sind. Der Vorteil einer Python Anwendung außerhalb des Editors ist die Möglichkeit weitere Programme einzubinden. So ist es beispielsweise möglich, in einem Vorgang einer Build Pipeline Unreal interne Funktionen aufzurufen oder Änderungen an Assets vorzunehmen. Anschließend an die Tätigkeiten innerhalb der Engine kann der Prozess der Pipeline außerhalb der Engine fortgesetzt werden.

2.4. Objektorientierte Programmierung

Objektorientierte Programmierung, kurz OOP, ist ein Paradigma, nach welchem Programmierkonzepte auf Basis von Objekten erstellt werden. Ein Objekt beschreibt hierbei die Instanz einer Klasse, welche die eigenen Daten sowie das eigene Verhalten definiert. Mit der Entwicklung von OOP durch die Abstraktion realer Entitäten, wurde die Welt der Softwareentwicklung prägend verändert. Das Ziel objektorientierter Programmierung ist die Entwicklung eines flexiblen und wiederverwendbaren Softwaredesigns durch seine Hauptprinzipien der Kapselung, Vererbung, Abstraktion und des Polymorphismus. [AC98, S.7-10]

Kapselung
Unter dem Begriff Kapselung versteht man, den Zugriff auf Daten einer Klasse möglichst stark zu regulieren. Im Idealfall ist der Zugriff auf die Daten der Klasse nur durch Methoden der Klasse möglich, in welchen diese die eingegebenen Parameter auf Kompatibilität und Sicherheitsverstöße überprüfen. Ebenfalls der Zugriff auf die Methoden sollte reguliert werden. Die allermeisten Funktionen sowie Daten einer Klasse sollten nur der privaten Klasse oder den erbenden Kinder-Klassen zugänglich sein. [SS14, S.276f]

Vererbung
Die Vererbung ist eines der zentralen Konzepte der OOP. Hierbei leitet eine Subklasse die Eigenschaften und Methoden einer Superklasse ab. Die Superklasse oder auch Basisklasse genannt definiert Eigenschaften und Funktionen, welche in allen abgeleiteten Klassen benötigt werden, spezifiziert diese jedoch nicht. Hierdurch kann der in der Basisklasse niedergeschriebene Code in allen Subklassen verwendet werden ohne die betroffenen Methoden neu zu definieren. Ebenfalls können die abgeleiteten Methoden und Eigenschaften durch weitere ergänzt oder überschrieben werden. Durch Vererbung lässt sich Redundanz im geschriebenen Code vermindern und hierarchische Beziehungen zwischen den Klassen aufbauen. Dies wiederum fördert die Wiederverwendbarkeit und Erweiterbarkeit der Klassen selbst. [SS14, S.267-275]

Polymorphismus
Polymorphismus beschreibt das Konzept, bei welchem eine Methode oder Objekt in verschiedenen Kontexten unterschiedliche Funktionsweisen annehmen kann. Durch das Überladen (engl. Overload) einer Methode oder eines Objektes mit demselben Namen jedoch unterschiedlichen Parametern können beispielsweise mehrere Instanzen einer Methode erstellt werden. Diese werden anschließend, basierend auf den variablen Parametern, aufgerufen. Neben dem Überladen, können Methoden oder Objekte ebenfalls überschrieben (engl. Overwrite) werden. Hierbei wird beispielsweise eine bestehende Methode einer abgeleiteten Klasse mit einer gleichnamigen Methode überschrieben. Polymorphismus ermöglicht es mehrere Objektarten durch dieselbe Schnittstelle zu manipulieren. [SS14, S.278-283]

Abstraktion
Als Abstraktion bezeichnet man den Prozess der Verallgemeinerung. Hierbei werden Objekte und Prozesse auf die wichtigsten Merkmale reduziert. Abstrakte Klassen und Interfaces bilden die Abstraktionsebene von Klassen. Sie definieren grundlegende Methoden. Jede hiervon erbende Klasse implementiert eine eigene Funktionsweise für die vorgegebenen festgelegten Methoden. Durch die Abstraktion wird die Programmstruktur modularer aufgebaut und kann somit leichter gewartet werden. [SS14, S.277f]

OOP findet sich in Unity in fast allen Skripts wieder. Sofern ein Skript spiel-relevante Logik enthält, muss dieses oder ein Skript, welches die Funktionen des Skriptes aufruft, zwangsweise auf einem Game Object liegen. Um als Komponente auf ein Game Object angewandt zu werden, muss ein Skript von der Klasse MonoBehaviour erben. Ab diesem Zeitpunkt fällt das Skript unter die objektorientierte Programmierung und es ist angebracht, die Prinzipien der OOP anzuwenden. Dies sorgt für eine effizientere und leichtere Wartung, weniger Komplikationen in Verbindung mit anderen Systemen und einem leichteren Verständnis der Codebase. [Sch24, S.435-438]

2.5. Bisherige Kenntnisse

Viele Tool-Entwickler beobachten das Nutzerverhalten und die Beschwerden der Entwickler. Aus diesen Erkenntnissen und meist eigenen Erfahrungen entstehen anschließend neue Editor-Tools. Beispielsweise kann der Standard Inspector nicht alle Datentypen einer Klasse serialisieren. Komplexere Datentypen wie Interfaces oder Dictionaries sind für den Entwickler im Unity Editor nicht einsichtig. Um dieses Problem zu lösen entwarfen Tool-Entwickler den Odin Inspector[7]. Dies ist ein Inspector, welcher ähnlich seinem Vorbild agiert, jedoch einige Verbesserungen aufweist, welche von Unity nicht implementiert werden. Die Verbesserungen beruhen hierbei auf den Rückmeldungen der Entwickler und haben zum Ziel, den Arbeitsfluss jener Entwickler langfristig zu vereinfachen. Durch diese Vereinfachung kann dauerhaft effizienter gearbeitet werden.

Ein Beispiel für eine klassische repetitive Aufgabe eines Entwicklers ist die Umbenennung von Assets in der Engine. Für die Gewährleistung der Qualität der Assets sind in vielen Projekten feste Regeln für die Benennung von Assets einzuhalten. Bei der Einarbeitung neuer Assets in das

[7] Odin Inspector and Serializer, besucht am 05.10.2024

Projekt kann dieser Prozess entweder durch Tools automatisiert oder händisch abgearbeitet werden. Aus diesem Grund wurden Tools wie beispielsweise das “Multi-Rename Tool“ von DevLocker[8] entwickelt. Es ermöglicht mehrere Assets unter der Angabe spezifischer Parameter automatisiert umzubenennen. Die Verwendung eines solchen Tools kann die Effizienz eines Entwicklers deutlich steigern.

Durch die Beobachtung von Entwicklern und durch eigene Erfahrungen ist einigen Entwicklern ebenfalls aufgefallen, dass viel Entwicklungszeit für das Wechseln zwischen verschiedener Szenen aufgewandt wird. Ein Entwickler muss bei jedem Szenenwechsel zunächst die gewünschte Szene im Projektverzeichnis suchen und anschließend laden. Bei ausreichend großen Projekten kann diese Tätigkeit, trotz Verwendung der integrierten Suchfunktion, eine große Menge an Entwicklungszeit beanspruchen. Das “Scenes In Project“ Tool von DevLocker[9] erstellt ein Editor-Fenster, von welchem aus alle Szenen des Projektes einfach erreichbar sind und automatisch gestartet werden können.

Ein weiterer Aspekt der Spieleentwicklung mit Unity, in welchem bestehende Tools mit effizienteren Alternativen zu den nativen Funktionen zu finden sind, ist die Gestaltung der Audio. Eine hinlänglich bekannte Sound-Engine für Unity ist FMOD[10]. Diese Engine ermöglicht es Entwicklern und Designern wesentlich einfacher, komplexe Sound Events einzubauen, wozu zuvor ein Programmierer notwendig war. Dadurch ist es den Entwicklern möglich, sich stärker auf andere Aspekte der Spieleentwicklung zu konzentrieren, während die Designer die Einarbeitung der Audio bearbeiten. [Abb18, S.48]

Die genannten Beispiele belegen, dass bereits eine große Menge Expertise und Erfahrungen im Bereich der Tool-Entwicklung in Verbindung mit der Unity Engine besteht. Entwickler nutzen ihre eigenen Erfahrungen sowie die Rückmeldungen andere Entwickler, um neue Tools zu entwickeln und erhoffen sich eine Steigerung der Effizienz sowie eine Erleichterung für den Nutzer. Die Auswirkungen von Software-Tools wurde bereits in wissenschaftlichen Arbeiten untersucht. Messungen der Effizienzsteigerung im Vorher-Nachher-Vergleich geschieht im Bereich der Unity Engine jedoch selten.

[8] Multi-Rename Tool von DevLocker, besucht am 05.10.2024
[9] Scenes In Project von DevLocker, besucht am 05.10.2024
[10] FMOD Website, besucht am 05.10.2024

3. Konzeption

Aufbauend auf dem, Grundwissen aus Kapitel 1 und 2, werden die folgenden Kapitel die Konzepte der praktischen Arbeit erläutern. Zunächst wird aufgezeigt, wie die Editor-Tools dem Nutzer durch ein zentrales Unity Package bereitgestellt werden. Anschließend beschäftigt sich die Arbeit mit dem erstellten User Interface und weist auf merkliche Unterschiede zwischen dem UI Toolkit und dem IMGUI System von Unity hin. Nach Erläuterung der Grundkonzepte wird näher auf die ausgewählten repetitiven Aufgaben und deren Problemstellung sowie mögliche Lösungen eingegangen. Abschließend geht es um den Aufbau und die Konzeption der Evaluation und dem dafür notwendigen Unity Package.

3.1. Tools als Unity Package Collection

Es gibt zwei Arten von Packages, welche von Unitys Package Manager unterstützt werden. “Asset Packages“ sind Kollektionen von Assets, welche aus einem Unity Projekt exportiert wurden. Die exportierten Elemente behalten hierbei ihre ursprüngliche Ordnerstruktur, sowie ihre Metadaten bei. Alternativ besteht die Möglichkeit zur Erstellung von UPM Packages, welches das native Format für Unitys Package Manager ist.

Zunächst soll sich mit der Erstellung von Asset Packages befasst werden. Um ein solches zu erstellen ist zunächst die Entwicklung der gewünschten Funktionen, der Assets und aller notwendigen Dateien innerhalb der Unity Engine von Nöten. Nachdem alle benötigten Dateien in einem Unity Projekt erstellt wurden, werden diese exportiert. Über die Menüschaltfläche “Assets/Exportieren“ öffnet sich das Exportfenster. In diesem sind alle zu exportierenden Assets auszuwählen. Anschließend, sorgt die Einstellung “Include dependencies“ dafür, dass beim Betätigen der Export Schaltfläche alle referenzierten Assets ebenfalls exportiert werden. [Tec24]

Aufgrund der geringen Voraussetzungen ist die Erstellung eines Asset Packages verglichen zur Erstellung eines UPM Packages leichter. Bei dieser Methode der Distribution ist jedoch zu beachten, dass bei Veränderung eines einzelnen Objektes alle Dateien erneut exportiert und auf entsprechendem Wege an den Nutzer weitergeleitet werden müssen.

Die Erstellung eines UPM Packages erfordert anfänglich einen größeren Arbeitsaufwand, bietet jedoch im weiteren Verlauf der Entwicklung die Möglichkeit zur Automatisierung und Integration einer Software zur Versionskontrolle. Im nachfolgenden Abschnitt wird der Prozess zur Erstellung eines UPM Packages erläutert, um den benötigten Aufwand im Vergleich zur Distribution per Asset Package zu evaluieren.

Der initiale Schritt zur Erstellung eines UPM Packages ist das Erzeugen eines lokalen, in das Projekt eingebetteten Package. Dies kann durch die Nutzung des von Unity bereitgestellten "Package Development Packages" vereinfacht werden. Das genannte Package wird mittels des Package Managers unter Verwendung der URL "com.unity.upm.develop" importiert. Anschließend besitzt der Package Manager die Funktion zur Erstellung eines neuen UPM Packages basierend auf dem dafür entwickelten Unity Template. Die Funktion ist unter dem gleichen Menüreiter zu finden, welcher zum Importieren des Packages dient. Nach Erstellung des lokalen Packages kann diesem nun eine Versionskontrolle hinzugefügt werden. Aufgrund des Bekanntheitsgrades und der einfachen Bedienung ist GitHub für diesen Schritt zu empfehlen. Die Versionskontrolle ist im gleichen Verzeichnis wie der package.json Datei anzubringen. Das Verwenden der Versionskontrolle ist essentiell für die Distribution an den Nutzer, da dieser nach erfolgreicher Verknüpfung mit GitHub, alle Daten von dort herunterladen kann. Um die Erstellung des lokalen Packages abzuschließen, muss die package.json Datei mit entsprechenden Daten befüllt werden. Zu diesen Daten gehören unter anderem der Name des Packages, der verschönerte Name zur Anzeige für den Nutzer, die minimal notwendige Unity Version, die Version des Packages sowie Informationen über den Versionsverlauf und die Dokumentation des Packages. Desweiteren kann auf weitere Packages verwiesen werden, welche beim Import des Packages ebenfalls importiert werden sollen. [Kok21, S. 225-253]

Der Aufwand zur Erstellung eines UPM Packages ist deutlich höher als bei der Erstellung eines Asset Packages. Jedoch birgt ein UPM Package den Vorteil der Integration einer Versionskontrolle, sowie eine effizientere Lösung zur Wartung des Packages. Die Veränderung eines einzelnen Assets kann mittels Versionskontrolle auf das, auf GitHub bestehende, Repository angewandt werden. Ein Nutzer kann anschließend die Update Funktion des Package Managers nutzen, um die Änderungen auf sein Projekt anzuwenden.

Es lässt sich ableiten, dass die Erstellung von Asset Packages bei kleinen Packages, welche nicht häufig überarbeitet werden, effizient und günstig ist. Bei umfangreich großen Packages, mit konstanter Überarbeitung hingegen, wird diese Methode der Distribution zunehmend ineffizient. Hier ist die Distribution mittels UPM Packages empfehlenswert. Diese Erkenntnis ist von integraler Bedeutung für das zu erarbeitende Konzept.

Um die Distribution zu verbessern sowie eine höhere Reichweite zu erhalten, können Packages im Unity Asset Store publiziert werden. Dies ist eine Website, auf welcher Assets sowie Tools käuflich oder kostenlos erworben werden können. Der Nutzer kann diese in seinen Unity Projekten ohne die *.unitypackage* Datei oder Git URL importieren. Von einer solchen Publikation der entstehenden Tools wird jedoch im Rahmen dieser Arbeit abgesehen, da dies die Entwicklung von Marketing-Material beinhalten würde.

Auf Basis der bisherigen Erkenntnisse ist die Distribution mittels UPM Packages vorteilhafter, da die entstehenden Packages über längere Dauer gewartet werden sollen und teilweise einen großen Umfang haben werden. Als Versionskontrolle soll Git mit der Plattform GitHub verwendet werden. Auf diesem Weg wird eine Mehrzahl von UPM Packages entstehen. Um diese in das Projekt zu installieren, benötigt der Nutzer die zugehörigen GitHub URLs. Da diese dem Nutzer ohne weitere Recherche nicht bekannt sind, muss eine Lösung gefunden werden, dem Nutzer das Importieren der Packages ohne URLs zu ermöglichen. Die simpelste Möglichkeit wäre das Erstellen einer Website,

welche als Verzeichnis der URLs zu den jeweiligen Packages agiert. Dies jedoch setzt erneut das Vorwissen über die Existenz der Website und weitere Recherchen auf Seiten des Nutzers voraus. Eine weitere Möglichkeit wäre, alle entstehenden Tools in ein einziges UPM Package zu implementieren. Jedoch widerspräche diese Option der modularen Entwicklung und würde die Beanspruchung von Speicherplatz durch ungenutzte Tools unterstützen. Alle entstehenden Tools sollen in separaten Packages implementiert werden, um ungewollte Speichernutzung zu minimieren. Die Installation der Packages soll ohne weitere Vorkenntnisse und nur innerhalb der Unity Engine stattfinden.

Um diese Ziele zu erfüllen, werden die zu konzipierenden Packages als Package Collection entwickelt. Der Kern dieser Kollektion ist ein Haupt-Package, welches Kernfunktionen zur Bedienung der folgenden Packages bereitstellt und deren Installation ermöglicht. Die Installation soll über eine eigene Implementation eines Package Managers ermöglicht werden. Neben dem internen Package Manager soll das Haupt-Package über eine Benutzeroberfläche verfügen, welche Informationen, Hilfestellungen sowie Einstellungen der einzelnen Packages beinhaltet. Jene Oberfläche wird dynamisch mit den Informationen der importierten Packages gefüllt und dient ausschließlich als Grundgerüst. Die Logik der Funktionen sowie die benötigten Dateien zur Anzeige in der Benutzeroberfläche, werden dynamisch aus den Packages ausgelesen. Zu den bereitgestellten Grundfunktionen sollen das Speichern von Einstellungen sowie das Anzeigen von Fenstern und erstellten Benutzeroberflächen mittels USS (Unity Style Sheet) Dateien gehören.

3.2. User Interface

Der Schwerpunkt bei der Konzipierung der Benutzeroberfläche soll darauf liegen, dem Nutzer die Bedienung der Funktionen möglichst einfach, intuitiv und effizient zu ermöglichen. Die Komplexität der Oberflächen soll möglichst gering sein, während alle Funktionen eindeutig erklärt werden. Ebenfalls soll eine barrierefreie Nutzung durch die Verwendung von hellen und dunklen Designs geschaffen werden.

Nach der erstmaligen Veröffentlichung des UI Toolkits in Unity 2019.1 muss bei der Erstellung von Tools zunächst erwägt werden, welches System für die Erstellung des User Interfaces verwendet werden soll. Neben dem UI Toolkit, welches das neuere Systeme bietet, steht das ältere IMGUI System. Im nachfolgenden Kapitel sollen die Unterschiede der Systeme aufgezeigt werden. Anschließend wird evaluiert, welches der Systeme zur Entwicklung der zu konzipierenden Packages besser geeignet ist.

3.2.1. Vergleich IMGUI System und UI Toolkit

IMGUI steht für “Immediate Mode Graphical User Interface“. Dies beschreibt ein Programmierparadigma, welches zur Erstellung von grafischen Benutzeroberflächen verwendet wird. Der Kernaspekt des Systems besteht darin, dass die Zustandsinformationen der Oberfläche nicht über dessen Lebensdauer gespeichert werden. Hierfür legt der Entwickler das Aussehen sowie die Funktionsweisen der Benutzeroberfläche mittels Code fest. Ungeachtet dessen, ob Veränderungen durch den Endnutzer an der Oberfläche getätigt wurden, wird die Benutzeroberfläche jeden Frame neu

gezeichnet und verliert hierbei jegliche Informationen des vorangehenden Frames. Aus diesem Grund spricht man von einem sofortigen Stil, da der Entwickler das Aussehen direkt für jeden Frame festlegt.

Durch diesen deklarativen Stil ist der Code einer solchen Benutzeroberfläche meist kompakt, prägnant und leicht zu verstehen. Oberflächen, welche mittels Unitys IMGUI System erstellt wurden, werden in allen Elementen des Editors unterstützt, da die Oberfläche des Editor selbst das IMGUI System verwendet. In kommenden Versionen des Unity Editors hingegen sollen die noch bestehenden IMGUI Systeme durch das UI System des UI Toolkits ersetzt werden. Dieses wird der neue Standard für Benutzeroberflächen im Editor und Laufzeit-Anwendungen. Dennoch werden IMGUI Oberflächen in kommenden Versionen weiterhin funktionieren, da das UI Elements System eine native Unterstützung für IMGUI implementiert.

Die Implementation einer Benutzeroberfläche mittels IMGUI erfolgt mittels C# unter der Verwendung mehrerer von Unity bereitgestellter APIs. Diese APIs enthalten vorgegebene Methoden zur Erstellung einfacher, visueller Elemente wie beispielsweise einem Knopf. Alle Parameter, wie das Aussehen der Elemente, werden mit dieser Methode übermittelt. Die Verwendung der bereitgestellten Methoden ist für simple Anwendungsfälle einfach und effizient. Bei komplexen Oberflächen mit individuellen Elementen verliert das System jedoch an Effizienz, da diese unter größerem Aufwand neu implementiert werden müssen.

Entgegen dem IMGUI steht das UI System des UI Toolkit. Hierbei handelt es sich um ein RMGUI System, welches für "Retained Mode Graphical User Interface" steht. Entgegen den Grundsätzen des IMGUI basiert dieses auf den Prinzipien der OOP. Hierbei wird die Benutzeroberfläche einmalig definiert. Die erstellten Objekte werden über die Lebenszeit der Oberfläche beibehalten und verlieren ihre Zustände nicht. Die erschaffenen Elemente werden lediglich neu gezeichnet, wenn dies, beispielsweise durch das Betätigen einer Schaltfläche, erforderlich ist. Jegliche Zustandsänderungen werden durch Callback Methoden weitergegeben. Dadurch ist die Codebase einer solchen Oberfläche komplex und schwerer zu verstehen.

Durch Unitys UI Toolkit ist neben der Implementation unter Verwendung von C# auch das Erstellen einer Oberfläche über den UI Builder möglich. Der UI Builder ist eine grafische Schnittstelle, in welcher Benutzeroberflächen aus bestehenden visuellen Elementen, zusammengestellt werden können. Diese visuellen Elemente sind modulare Objekte, welche mittels Code oder als UXML Datei in andere visuelle Elemente eingefügt werden können. Unity stellt grundlegende Elemente wie Eingabefelder, Knöpfe oder Listen bereit. Durch die Möglichkeit, bestehende visuelle Elemente miteinander zu verbinden, können komplexere Oberflächen mit wenig Aufwand realisiert werden. Neben den Vorteilen eines RMGUI bietet das UI Elements System etablierte Praktiken der Webprogrammierung. Maßgebend hierfür ist die Verwendung von USS (Unity Style Sheet) Dateien. Dieses Format ähnelt dem CSS (Cascading Style Sheets) Format und ermöglicht die Definition von Klassen, welche auf Objekte übertragen werden können. Diese Klassen enthalten die stilistischen Parameter der Objekte. Durch das Verwenden von Pseudo States können zudem stilistische Änderungen an interaktiven Objekten vorgenommen werden.

[Kok21, S. 29-69, S. 71-110] [SF21, Kapitel 3 *"Inventory and Advanced UIs"*]

3.2.2. Auswahl des UI Toolkit für die UI-Entwicklung

Auf Grundlage der genannten Informationen eignet sich die Verwendung des IMGUI Systems durch die einfache und effiziente Implementation vorrangig für simple Editor Tools. Das UI Toolkit hingegen eignet sich für komplexere Editor Tools sowie für Laufzeit Oberflächen besser. Der Schwerpunkt der zu konzipierenden Benutzeroberflächen liegt auf der einfachen, intuitiven und effizienten Bedienung durch den Nutzer. Dies ist mit beiden Systemen unter ähnlichem Aufwand möglich.

Jedoch ist die Implementation eines eigenen Farbschemas sowie eines einheitlichen Regelsatzes für die UI, unter Verwendung des IMGUI Systems mit Mehraufwand verbunden. Hierfür müssten GUIStyles implementiert werden, welche bei Erstellung der einzelnen Elemente angewandt werden. Zudem wird die Benutzeroberfläche viele interaktive und komplexe Elemente enthalten. Aufgrund dieser Einschätzung ist das UI Toolkit besser geeignet die Benutzeroberflächen zu erstellen. Zudem lassen sich die Farbschemata sowie Regelsätze der visuellen Elemente unter Verwendung der USS Dateien einfacher realisieren.

Fürsprechend für das UI Toolkit ist zudem, dass einige der zu konzipierenden Oberflächen Elemente enthalten, welche dauerhauft auf dem Bildschirm des Nutzers angezeigt werden. Unter Verwendung des IMGUI Systems werden diese Elemente häufiger aktualisiert, welches Leistung beansprucht. Um die benötigte Leistung gering zu halten, sollte die Benutzeroberfläche nur aktualisiert werden, wenn dies notwendig ist. Gegen die Verwendung des UI Toolkits spricht, dass die zu Grunde liegende Editor Oberfläche das IMGUI System verwendet und daher, für wenige Elemente, IMGUI in den zu konzipierenden Oberflächen eingebaut werden muss. Jedoch soll das UI Elements System der neue Standard der Engine werden. Aus diesem Grund beschränkt sich diese Limitierung lediglich auf die für die zur Implementation genutzte Version. Neuere Versionen werden weitreichendere Unterstützung in den noch bestehenden IMGUI Bereichen des Editors besitzen.

Nachteilig muss angebracht werden, dass eine Implementation, unter Verwendung des UI Toolkits, nicht mit früheren Unity Versionen kompatibel ist. Trotz dieser negativen Aspekte wird das UI Elements System des UI Toolkits zur Erstellung der Benutzeroberflächen verwendet.

3.2.3. Umsetzung der definierten Konzeptgrundlagen

Um die zu erstellenden Oberflächen intuitiv nutzen zu können, muss der Nutzer wiederkehrende Elemente wie Eingabefelder, Knöpfe oder Hilfestellungen schnell erkennen können. Hierfür ist ein einheitliches Aussehen der Elemente zwingend notwendig. Die Vorteile eines solchen einheitlichen Aussehens sollen kurz anhand eines Beispiels belegt werden. Das Beispiel beschreibt das Aussehen eines Textelements. Ein solches Textelement besitzt eine von zwei Farben. Die erste Farbmöglichkeit ist die des normalen Textes, ein fast volles Weiß. Da diese Farbe prägnant ist und sich von dem festgelegten Hintergrund stark abhebt, wird diese für Textstellen verwendet, welche dem Nutzer notwendige Informationen vermitteln. Im Gegensatz dazu, kann ein Textelement ein schwaches Weiß besitzen. Dieser Farbton hebt sich weniger vom festgelegten Hintergrund ab und zieht wenig Aufmerksamkeit auf sich. Diese Farbe kann verwendet werden, um dem Nutzer zusätzliche

Informationen zu bieten, welche jedoch nicht zwingend notwendig sind. Neben der Grundfarbe des Textelements sind alle interaktiven Textstellen in einer noch prägnanteren Hauptfarbe gestaltet. Diese wiederkehrenden Regeln ermöglichen es dem Nutzer in kürzester Zeit, die interaktiven Textstellen zu identifizieren und wichtige von optionalen Informationen zu trennen.

Dieses Konzept soll auf alle visuellen Elemente angewandt werden. Zu dessen Realisierung, werden die festgelegten Regeln in USS Klassen festgehalten und auf die jeweiligen Elemente angewandt.

Um die Komplexität der zu erstellenden Oberflächen zu minimieren, sollen nur notwendige Informationen angezeigt werden. Oberflächen, welche mehrere Funktionen bereitstellen, sollen lediglich die Schaltflächen, welche für die ausgewählte Funktion benötigt werden, anzeigen. Am Beispiel des zu entwickelnden internen Package Managers, soll im folgenden Abschnitt ein solches Vorgehen demonstriert werden.

Die Benutzeroberfläche des internen Package Managers soll Schaltflächen zum Importieren, Aktualisieren sowie Entfernen des ausgewählten Packages bereitstellen. Die Import-Schaltfläche wird nur angezeigt, wenn das gewählte Package noch nicht importiert ist. Gleiches gilt auch für die Entfernen-Schaltfläche, welche nur angezeigt wird, sollte das Package bereits im Projekt vorhanden sein. Die Schaltfläche für die Aktualisierung hingegen wird nur gezeigt, wenn das Package importiert ist und ein Update zur Verfügung steht. Es ist ebenfalls notwendig, dem Nutzer Elemente zu zeigen, obwohl er nicht mit diesen interagieren kann, so beispielsweise die Import- und Entfernen-Schaltfläche. Nach dem Betätigen dieser werden sie nicht direkt ausgeblendet, da der Nutzer in diesem Fall davon ausgehen wird, dass der Vorgang abgeschlossen ist. Die Schaltfläche wird lediglich ausgegraut. Auch bei der Wahl eines anderen Packages bleibt die jeweilige Schaltfläche ausgegraut, bis der bestehende Vorgang beendet ist. Hierdurch wird dem Nutzer vermittelt, dass ein Vorgang läuft und kein neuer gestartet werden kann.

Um die Nutzung der Oberflächen einfach und intuitiv zu gestalten, soll allen interaktiven Elementen eine klare und einfache Beschreibung gegeben werden. Dies sorgt für ein schnelles Verständnis beim Nutzer. Elemente, deren Funktion ausführlicher beschrieben werden muss, werden mit Tooltips versehen. Diese Tooltips erklären kurz und prägnant die genaue Funktion des jeweiligen Elements.

Die Nutzung von Tool-Funktionen über Benutzeroberflächen bietet die Möglichkeit, dem Nutzer weitreichende Informationen über die Funktion zu vermitteln und Einstellungen tätigen zu lassen. Jedoch ist dies für Nutzer, welche bereits ausführlich mit der Funktionalität des Tools vertraut sind, ineffizient. Bei Funktionen, welche keiner Einstellungen bedürfen ist es wichtig, Diese ohne das vorherige Öffnen der Oberfläche nutzen zu können. Dies ist in Unity durch Menu Items (Schaltflächen im Programmkopf) möglich. Der große Vorteil der Menu Items ist, dass ihnen Tastenkombinationen zur Ausführung zugewiesen werden können. Aus diesem Grund sollen alle Hauptfunktionen, bei welchen keine Einstellungen von Nöten sind, über eine Benutzeroberfläche und über Menu Items verwendbar sein.

Der einheitliche Stil der Benutzeroberflächen soll sich von der Farbpalette des Unity Editors abheben. Die Abweichung soll für den Nutzer deutlich erkennbar sein, jedoch nicht das Allgemeinbild des Editors stören. Aus diesem Grund ist die Farbpalette an die des Editors angelehnt, die Hauptfarbe (Magenta) hebt sich jedoch stark von der Hauptfarbe des Editors (Blau) ab. Die Verwendung einer

abweichenden Farbpalette soll dem Nutzer ermöglichen leichter aufzuzeigen, welche Funktionen durch die Installation der Packages hinzugefügt wurden.

Um die Nutzerfreundlichkeit und Barrierefreiheit der Oberflächen zu verbessern, soll das Aussehen jener durch den Nutzer einstellbar sein. Hierzu werden alle Benutzeroberflächen in jeweils einem dunklen und einem hellen Design zur Verfügung gestellt. Zunächst passen sich die Oberflächen an die Einstellungen des Editors an. Allerdings soll die Möglichkeit bestehen, dass der Nutzer das Design individuell einstellt. Einige der Oberflächen fügen, für eine effizientere Bedienung, der Toolbar des Unity Editors Schaltflächen hinzu. Da diese Elemente dauerhaft sichtbar sind, soll es dem Nutzer möglich sein, diese zu deaktivieren. Zudem soll er entscheiden können, ob jene Schaltflächen mit einem Icon oder Titel beschriftet sind. Diese Konzepte sollen dem Nutzer die größtmögliche Individualisierung der Oberflächen ermöglichen.

3.2.4. Verwendung webbasierter Programmierprinzipien

Wie bereits benannt, ermöglicht das UI Elements System die Verwendung webbasierter Programmierprinzipien. Ermöglicht wird dies durch die Nutzung von “Unity Extensible Markup Language“, kurz UXML. Dies ist ein Format für Textdateien, welches die Struktur sowie den Stil einer Benutzeroberfläche enthält. UXML soll technisch weniger versierten Entwicklern die Erstellung von Oberflächen erleichtern. Während sich UXML am ehesten mit HTML vergleichen lässt, bietet das UI Toolkit ebenfalls ein Äquivalent zu CSS. Dies ist das “Unity Style Sheet“ Format, kurz USS. Die Syntax beider Formate CSS und USS ist identisch, während USS zusätzliche Funktionen und Individualisierungen enthält, um sich besser in Unity zu integrieren. Genutzt werden USS Dateien, um, wie bei CSS, Klassen festzulegen, welche anschließend auf Elemente der Oberfläche angewandt werden können. Jene Klassen definieren die stilistischen Attribute der Elemente.

USS Klassen können auf jedes erstellte visuelle Element einer UXML Datei angewandt werden. Hierfür muss lediglich die USS Datei in der UXML Datei referenziert sein. Jede UXML Datei kann beliebig viele USS Dateien referenzieren. Ebenfalls können visuelle Elemente beliebig viele USS Klassen besitzen. Die Anwendung jener Klassen erfolgt, wie bei CSS, hierarchisch. Das bedeutet, dass beispielsweise durch das Festlegen eines Font Attributes in einem visuellen Element ebenfalls alle Kinder-Objekte dieses Elementes beeinflusst werden. Hat ein Kind-Objekt jedoch eine lokale Definition für jenes Attribut, hat diese eine höhere Priorität. Ist ein visuelles Element im Besitz mehrerer Klassen, so ist die Priorität dieser abhängig von ihrer Reihenfolge in den jeweiligen USS Dateien sowie der Reihenfolge, in welcher die USS Dateien innerhalb der UXML Datei referenziert werden.

Zunächst soll sich mit der Vereinfachung einer einheitlichen Gestaltung auseinandergesetzt werden. Hierzu ist zu erwähnen, dass Unitys UI Toolkit in dem bereitgestellten UI Builder bereits einige visuelle Elemente vordefiniert. Hierzu gehören beispielsweise Knöpfe, Eingabefelder oder Listen. Jene Elemente wiederum bestehen aus weiteren Grundelementen. Diese jedoch sind im Besitz bestimmter USS Klassen, welche bei Nutzung das Aussehen der Elemente durch Pseudo States manipulieren können. So ist es beispielsweise möglich, Knöpfe farbig zu umranden, wenn der Nutzer den Hover State auslöst. Durch das Erstellen einer eigenen USS Datei, können diese

Klassen überschrieben werden. Durch das Überschreiben der bereits existierenden Klassen ist es ohne weiteren Aufwand möglich, die von Unity vordefinierten Objekte zur Erstellung der eigenen Benutzeroberflächen zu verwenden.

Das Prinzip eines Rootelements bezeichnet allgemein das hierarchisch erste Element. Jenem Element unterliegen alle anderen Elemente. Die Definition von stilistischen Attributen auf dem Rootelement hat zur Folge, dass diese ebenfalls an die Kinder-Objekte übertragen werden. So definiert ein im Rootelement festgelegter Font dieses Attribut für alle Elemente der Oberfläche, welche keine expliziten lokalen Definitionen besitzen. Zusätzlich zu dieser Eigenschaft ist es Ziel, dass das Rootelement die Werte für alle festgelegten Attribute, wie beispielsweise Farbwerte, besitzt. Hierzu wird jenes mit einer Klasse versehen, in welcher alle Farben der Oberfläche als Variable definiert werden. In allen weiteren Klassen, welche auf Kinder-Objekten des Rootelements angewandt werden, werden nun keine konkreten Farbwerte definiert, sondern auf die im Rootelement festgelegten Variablen verwiesen [Gra18, Kapitel 1 *"Cascade, specificity, and inheritance"*, Kapitel 2 *"Working with relative units"*].

Die Nutzung eines Rootelements bietet bereits bei der Erstellung der Oberflächen folgende Vorteile. Zunächst spart das einmalige Definieren der beispielsweise Farbwerte Aufwand und Zeit, da diese nicht in jeder erstellten Klasse neu definiert, sondern lediglich referenziert werden. Zudem ermöglicht dies die Änderung von Werten mittels eines "single point of entry". Beispielsweise muss bei Änderung der Textfarbe einmalig der Farbwert in der Klasse des Rootelements abgeändert werden. Diese Änderung überträgt sich anschließend auf alle Klassen, welche die Variable verwenden.

Ein Wechsel zwischen dunkel und hellem Design einer Oberfläche, wird durch das beschriebene Verhalten des Rootelements und dessen Klasse ermöglicht. Es soll eine dunkle sowie eine helle Rootelement-Klasse erstellt werden. Die darin definierten Variablen sollen identisch benannt sein, jedoch an das jeweilige Design angepasste Werte besitzen. Alle weiteren USS Klassen lesen ihre Werte aus den Variablen der Rootelement-Klassen aus. Die gleiche Benennung der Variablen ermöglicht das Austauschen der Klasse des Rootelements zur Laufzeit, ohne dass die restlichen Elemente die Referenzen zu diesen verlieren. Durch den Austausch der beispielsweise dunklen mit der hellen Klasse, ändern alle betroffene Elemente ihre Farbe gemäß der neuen Rootelement-Klasse.

3.3. Repetitive Aufgaben in Unity

3.3.1. Speichern einer Szene

Die Erstellung der Laufzeit-Umgebung durch beispielsweise die Entwicklung verschiedener Level, erfolgt innerhalb der Unity Engine in Szenen Objekten. Dies sind Dateien, welche alle Informationen über die in ihnen befindlichen Game Objects enthalten. Eine große Rolle bei jener Erstellung der Umgebung spielt das Level- oder Environment-Design. Ein Entwickler muss hierbei häufig Szenen mit einer Großzahl an Assets füllen, um die Spielwelt lebhaft wirken zu lassen. Dieser Prozess ist sehr aufwändig und zeitintensiv. Bei solch langen Tätigkeiten kommt es häufig vor, dass aufgrund technischer Probleme oder kleinen Fehlern, Fortschritt verloren geht. In dem konkreten Fall des

Leveldesigns, geht dabei der Fortschritt an der Szene verloren. Um solchen Verlusten entgegenzuwirken, ist ein regelmäßiges Speichern der Szenen Datei notwendig. Unity bietet, gegensätzlich zu anderen Programmen, keine Funktion zur automatischen Sicherung der Szenen. Aus diesem Grund ist der Entwickler dazu angehalten, regelmäßig manuell zu speichern. Erfahrene Unity Entwickler mögen dies aufgrund der langanhaltenden Notwendigkeit gewohnt sein. Jedoch sorgt diese fehlende Funktion meist bei unerfahrenen Entwicklern zu häufigem Verlust ihrer Daten.

Da das Speichern der bearbeiteten Szene eine Tätigkeit ist, welcher jeder Entwickler innerhalb der Unity Engine nachkommen muss, soll für diese repetitive Aufgabe eine automatisierte Funktion entwickelt werden. Diese soll sich an bestehenden Funktionen anderer Programme orientieren.

Das automatische Speichern der Szene soll zeitlich abhängig sein. Jenes Zeitintervall soll für den Nutzer einstellbar sein. Zudem soll der Nutzer wählen können, ob beim Speichern der Szene die aktuelle Szene überschrieben oder eine Kopie in einem festgelegten Verzeichnis erstellt werden soll. Dies bietet dem Nutzer zusätzlich die Möglichkeit, das Tool als vereinfachte Versionskontrolle für Szenen zu verwenden, indem er regelmäßige Zwischenstände erstellt. Grafisch soll die Funktion über zwei Wege verwendbar sein. Zunächst soll eine neue Schaltfläche in der Toolbar der Unity Engine eingebettet werden, mit welcher die Funktion gestartet und pausiert werden kann. Der Tooltip dieser Schaltfläche soll die benötigte Zeit bis zur nächsten Sicherung anzeigen. Desweiteren soll eine detailliertere Übersicht in Form eines Editor-Fensters erstellt werden. Diese zeigt, neben den Schaltflächen zum Starten und Pausieren der Funktion, die verbleibende Zeit des aktuellen Intervalls an. Hierdurch wird dem Nutzer ermöglicht, selbst über die ihm zur Verfügung stehenden Informationen zu entscheiden. In jedem Fall soll beim Pausieren der Funktion eine Warnung in Form eines auditiven Signals sowie einer Konsolen-Nachricht erscheinen. Dies soll den Nutzer davor schützen, die Funktion ungewollt zu deaktivieren.

3.3.2. Automatisches Starten der Hauptmenü Szene

Eine weitere Tätigkeit, deren Anwendung sich in den Bereichen des Level- und Environment-Designs häuft, ist das Wechseln der Szenen innerhalb des Unity Editors, um die Anwendung für Testzwecke zu starten. Dies beschränkt sich nicht nur auf die genannten Bereiche. Ebenfalls davon betroffen sind alle Entwickler, welche eine bestimmte Szene bearbeitet haben und diese anschließend zum Testen über das Hauptmenü starten müssen. Hierbei muss der Entwickler zunächst die Änderungen in der bearbeiteten Szene speichern und die Hauptmenü Szene über das Projekt-Verzeichnis öffnen. Nach dem Starten der Anwendung und Testen der betroffenen Szene, muss der Entwickler die zu bearbeitende Szene erneut über das Verzeichnis finden und öffnen. Bei jedem Öffnen einer Szene ist eine kurze Ladezeit zu berücksichtigen, welche von dem Computer des Entwicklers abhängig ist. Das eben beschriebene Szenario tritt nicht in allen Unity Projekten auf, da dies von dem Aufbau und der entstandenen Codebase abhängig ist. In vielen Projekten hingegen ist ein solches Verhalten ab einem gewissen Zeitpunkt unvermeidbar, da das Starten über das Hauptmenü oft wichtige, zum Start der Anwendung notwendige, Objekte mittels eines Ladebildschirms initialisiert.

Trotz dessen, dass nicht alle Entwickler von jener Problematik betroffen sind, soll ein Tool als effizientere Alternative zum häufigen Szenenwechsel entwickelt werden, da dies in Projekten, in welchen es der Fall ist, viel Entwicklungszeit kostet. Fürsprechend ist zudem, dass die Unity Engine bereits eine Funktion für dieses Verhalten bereitstellt. Jedoch ist die Funktion nicht intuitiv und lediglich über selbst geschriebene Skripts erreichbar, wodurch sie nur von wenigen Entwicklern genutzt wird.

Durch das Nutzen der, von Unity bereitgestellten, Funktion über das zu entwickelnde Tool soll beim Betreten des Playmodes, ungeachtet der aktuell geöffneten Szene, die ausgewählte Szene gestartet werden. Hierdurch muss der Entwickler keinen Szenenwechsel durchführen, um die gerade bearbeitete Szene über die ausgewählte Hauptmenü Szene zu starten. Das Auswählen der zu startenden Szene soll über eine Benutzeroberfläche erfolgen. Die Auswahl soll abgespeichert werden, so dass diese für den nächsten Start der Engine erhalten bleibt. Ähnlich wie bei dem zuvor konzipierten Tool zum automatischen Speichern von Szenen, soll dieses Tool auch über zwei Oberflächen erreichbar sein. Zunächst soll ebenfalls eine Schaltfläche in die Toolbar der Engine eingebettet werden, welche das Starten und Pausieren der Funktion ermöglicht. Nur wenn die Funktion gestartet ist, wird die gewählte Szene gestartet. Andernfalls agiert die Unity Engine wie gewohnt. Die zweite Oberfläche, welche als Editor-Fenster entwickelt wird, soll ebenfalls die Funktionen zum Starten und Pausieren der Funktion bieten. Zusätzlich soll jenes Editor-Fenster den Namen der ausgewählten Szene zeigen, um dem Nutzer auf direktem Wege ergänzende Informationen zu bieten. Diese Informationen sollen außerdem auf indirektem Weg, als Tooltip der Toolbar Schaltfläche, erreichbar sein.

3.3.3. Fehlerkorrektur von Assets

Unity Projekte mit einer großen Anzahl von Assets neigen zunehmend zu höherem Aufwand bezüglich der Instandhaltung jener Assets. In größeren Projekten ist es nicht unüblich, dass häufig Probleme durch Veränderungen an, auf Assets liegender Skripts, entstehen. Durch beispielsweise die Umbenennung einer Variable im Skript, verlieren deren Referenzen auf den Assets ihre Werte. Ebenfalls kommt es vor, dass durch bereits eingetragene Werte neue Logik ungeahnte Fehler verursacht. Solche Probleme führen im schlimmsten Fall zum Absturz beziehungsweise dem Verlust der Spielbarkeit des Spieles. In einem solchen Fall das konkrete Asset ausfindig zu machen und die darauf befindlichen Fehler zu beheben, wird mit wachsender Anzahl der Assets schwieriger. Um dieser Problematik entgegenzuwirken, werden meist Tools entwickelt, welche projekt-spezifische Regeln auf den Assets erzwingen. Da jene Tools zumeist individuell für das eigene Projekt entwickelt werden, fällt die Arbeit der Neuentwicklung in jedem Projekt an.

Um diese Tools zu entwickeln, stellt die Unity Engine bereits einige hilfreiche Methoden und Systeme zur Verfügung. Eines dieser Systeme sind die “AssetImporter“. Das sind Skripts, in welche für bestimmte Arten von Assets Regeln festgelegt werden können. So kann beispielsweise erzwungen werden, dass eine Bilddatei, deren Name mit “[T]“ beginnt, immer ihren alpha Wert als Transparenz benutzt oder ein bestimmtes Farbprofil verwendet. Den Möglichkeiten der AssetImporter sind wenig Grenzen gesetzt, jedoch ist anzumerken, dass die von ihnen erzwungenen Einstellungen automatisch, ohne die Zustimmung des Entwicklers, beim Importieren des Assets geschehen. Daher ist die Einführung von Sonderfällen, in welchen die Regeln außer Kraft gesetzt werden, mit

hohem Aufwand verbunden. Eine weitere Möglichkeit zur Überprüfung der Assets wird durch die "OnValidate" Methode der MonoBehaviour gegeben. Diese Methode wird ausgeführt, wann immer das Objekt validiert wird. Dies ist beispielsweise der Fall, wenn Werte der Objekte verändert wurden, eine Szene mit diesem Objekt geladen oder die Anwendung gestartet wird. Sie ermöglicht dem Entwickler die Überprüfung aller Eigenschaften des Objektes, die Ausgabe der Ergebnisse oder die direkte Fehlerbehebung. Auch diese Methode umfasst großen Aufwand in der Vorbereitung, da für jedes Skript individuelle Regelungen implementiert werden müssen. Zudem ist eine Ausgabe der Fehler als Konsolen Nachricht bei einer Großzahl von Assets weitgehend unübersichtlich.

Um die Instandhaltung großer Assetmengen zu vereinfachen, soll aufbauend auf Unitys OnValidate Methode, ein System entstehen, welches definierte Regeln überprüft und nur bei Zustimmung durch den Entwickler erzwingt. Das System soll auf jedes MonoBehaviour anwendbar sein. Der Nutzer soll den Zustand der Assets grafisch im Inspector erkennen können. Zudem soll er in der Lage sein, alle Assets der aktuellen Szene sowie alle Prefabs des Projektes zusammenfassend zu überprüfen. Der Fokus des Tools soll auf der effizienten Erstellung der Regelsätze sowie intuitiven Lösung der aufgetretenen Probleme liegen.

Die technische Verwendung des zu erstellenden Systems soll über eine Klasse funktionieren, welche von MonoBehaviour erbt. Diese neue Klasse soll in allen zu validierenden Skripten anstelle der MonoBehaviour Klasse verwendet werden. Jene Klasse überschreibt Unitys bestehende OnValidate Methode mit dem neuen Validations-System. Der Nutzer soll in der Lage sein, die Regeln, welche erzwungen werden sollen, durch eine grafische Schnittstelle im Inspector einzustellen. Hierbei stellt das Tool bereits einige vordefinierte Regeln bereit, aus welchen der Entwickler auswählen kann. Zudem soll ihm ermöglicht werden, eigene Regeln zu schreiben, welche anschließend auf gleichem Wege hinzugefügt werden können. Durch das Hinzufügen einer weiteren Komponente, welche den Status eines jeden validierbaren MonoBehaviours des Objektes abliest, lässt sich der Status des Game Objects erkennen. Der Status soll unterscheiden zwischen "OK", wenn keine Probleme gefunden wurden und "Warning", wenn mindestens ein Problem mit der Schwere Warnung gefunden wurde. Ebenfalls soll der Status "Error" angezeigt werden, wenn mindestens ein Problem aufgetreten ist, welches als schwerer Fehler markiert wurde. Neben der Anzeige des generellen Status soll diese Komponente alle gefundenen Fehler anzeigen. Hierbei werden der Name des Fehlers, sowie der Name des fehlerhaften Behaviours sowie die Fehlerbeschreibung angezeigt. Zusätzlich wird zu jedem gefundenen Fehler eine Schaltfläche mit der Aufschrift "Fix" erstellt. Betätigt der Entwickler diese Schaltfläche, so wird die per Code definierte automatische Fehlerbehebung für dieses Problem ausgeführt. Dem Entwickler soll zudem ermöglicht werden, alle aufgetretenen Fehler mit einmal zu beheben. Besitzt ein Fehler keine automatische Fehlerbehebung, so soll dies dem Nutzer durch ein Warnsymbol signalisiert werden und eine manuelle Behebung ist erforderlich.

Neben der zuvor beschriebenen Schnittstelle innerhalb des Inspectors soll ein Editor-Fenster entwickelt werden, welches alle validierbaren Objekte der offenen Szene sowie alle validierbaren Prefabs des Projektes in einem Fenster zusammenfasst. Die gefundenen Objekte werden anhand ihres Status kategorisiert. Der Entwickler soll in der Lage sein, in der Liste der gefundenen Objekte, fehlerhafte Objekte auszuwählen und in einer Übersicht die, im vorgehenden Abschnitt beschriebenen Informationen und Funktionen, zu erhalten. Zudem soll eine Schaltfläche erstellt werden, mit welcher

der Entwickler alle gefundenen Fehler aller fehlerhaften Assets in der Szene sowie dem Projekt durch einmaliges Betätigen, beheben kann.

Die automatische Fehlerbehebung soll durch dieses System um ein Vielfaches vereinfacht werden. Der Hauptteil der Arbeit zur Instandhaltung der Assets liegt unter Verwendung dieses Systems, in der Erstellung neuer Regeln sowie deren Anwendung auf die Assets. Um die Anwendung zu vereinfachen, sollen Scriptable Objects erstellt werden, welche die Informationen über die Regelsätze beinhalten. Diese können aus bestehenden validierbaren Objekten exportiert werden. Ebenfalls kann ein erstellter Regelsatz in ein Behaviour importiert werden. So ist es beispielsweise möglich, einen Regelsatz für Gegner zu erstellen, welcher dann auf alle Gegnertypen angewandt wird. Dieser erzwingt etwa das Vorhandensein einer Rigidbody Komponente sowie die Definition der Lebenspunkte des Gegners in einem bestimmten Intervall. Jeder Gegnertyp kann über diese grundlegenden Regeln weitere lokale Regeln erstellen oder weitere Regelsätze importieren. Mit dieser Erweiterung des Konzepts soll die Haupttätigkeit von nun an auf der Erstellung sinnvoller Regelsätze sowie deren Anwendung auf die Assets liegen. Die Überprüfung der Assets sowie die Behebung gefundener Fehler verläuft anschließend weitestgehend automatisiert.

3.3.4. Vereinfachte Bildschirmaufnahmen

Einer der wichtigsten Grundsteine für das Marketing eines Spiels oder einer Anwendung sind bildliche Aufnahmen des Programms. Diese Bildschirmaufnahmen werden fortan Screenshots genannt. Egal ob für eine private Publikation oder die Veröffentlichung auf bekannten Plattformen wie Steam oder Itch.io, Screenshots sind essentiell, um dem Käufer einen Eindruck des Produkts zu vermitteln. Die Richtlinien der Spieleplattform Steam sind hierbei zudem sehr strikt. Lediglich im Spiel entstandene Screenshots, welche ein ehrliches Abbild des Spielgeschehens, bieten sind erlaubt. Unity bietet keine native Funktion, Aufnahmen des Spielgeschehens anzufertigen. Stattdessen bleibt den Entwicklern übrig, dass Spiel mittels des dafür vorgesehenen Knopfes zu pausieren und unter Verwendung anderer Software jene Aufnahmen zu tätigen. Hierbei ist es schwer, den genauen Zeitpunkt zu pausieren, welchen man dem Käufer präsentieren möchte. Um dies zu umgehen, ist die meist verwendete Methode das Erstellen eines Skripts, welches das Spielgeschehen ohne das Betätigen der Pause Schaltfläche, zum richtigen Zeitpunkt pausiert und dem Entwickler die Chance gewährt, den Screenshot zu schießen. Aufgrund der hohen Nachfrage dieser Funktion gibt es bereits zahlreiche Tools, welche das Aufnehmen von Screenshots innerhalb der Unity Engine erlauben. Viele nutzen Menu Items, welche mit Tastenbelegungen erreichbar sind, um eine Aufnahme des gesamten Editors zu tätigen. Andere wiederum fokussieren sich auf das Game-View Fenster, welches das Spielgeschehen aus Perspektive der Hauptkamera wiedergibt.

Die bereits bestehenden Tools sind funktionell ausreichend und erprobt, jedoch fehlen den meisten Funktionen, welche zur Erstellung von Screenshots für das Marketing, hilfreich sind. So soll es beispielsweise möglich sein, mehrere Kameras zu erstellen, welche den Spielercharakter aus verschiedenen Perspektiven betrachten. Alle Kameras sollen zur gleichen Zeit, durch das Betätigen einer Tastenkombination, Screenshots aufnehmen. Dies ermöglicht dem Entwickler, das Spiel zu spielen und zu geeigneten Momenten eine Aufnahme aus verschiedenen Perspektiven zu tätigen. Zudem sollen die Aufnahmen die exakten Bilddaten der Kameras mit gegebenen Postprocessing

darstellen können. Um die Nachbearbeitung der Screenshots zu vereinfachen, soll es zusätzlich möglich sein, aus verschiedenen Typen des Hintergrunds auszuwählen. Dies umfasst neben der normalen Perspektive der Kamera einen transparenten Hintergrund, bei welchem das aufgenommene Objekt direkt ausgeschnitten wird. Aufbauend darauf soll der Hintergrund auch durch eine solide Farbe und eine ausgewählte Grafik ersetzbar sein. Da die Screenshots bereits in vordefinierter Auflösung und mit bearbeitetem Hintergrund erstellt werden, vereinfacht dies die Erstellung von Marketing-Aufnahmen.

Das zu entwickelnde Tool soll, zu der eben beschriebenen Funktion, eine weitere Hauptfunktion besitzen. Diese zweite Hauptfunktion ist das Aufnehmen eines ausgewählten Editor-Fensters in seiner nativen Größe. Beide Funktionen sollen unter Verwendung aller Renderpipelines ermöglicht werden.

Das Aufnehmen eines Editor-Fensters soll in einem dafür entwickelten eigenen Editor-Fenster möglich sein. Über ein Dropdown-Element kann der Nutzer das gewünschte Fenster, aus einer Liste aller offenen Editor-Fenster, auswählen. Anschließend soll der Nutzer die Schaltfläche zum Rendern des Fensters betätigen können, welches eine Vorschau der finalen Aufnahme zeigt. Daraufhin kann diese Aufnahme exportiert und abgespeichert werden. Für den Fall, dass ein Editor-Fenster an ein anderes gedockt ist und sich im Hintergrund befindet, soll dieses fokussiert werden, um es in den Vordergrund zu bringen.

Das Rendern einer Kamera soll für jedes Game Object möglich sein, welches eine Kamera Komponente besitzt. Hierzu muss nachträglich eine mit dem Tool gegebene Komponente auf das Objekt angewandt werden. Diese enthält alle Funktionen, welche zur Aufnahme der Kamera notwendig sind. Zunächst beinhaltet diese alle Einstellungen des Render Vorganges. Dies umfasst die Auflösung des zu tätigenden Screenshots, die Tiefe der Bilddatei sowie den gewählten Hintergrundtypen. Abhängig des gewählten Hintergrunds werden zudem weitere, für diesen notwendige, Einstellungen angezeigt. Nach erfolgreicher Einstellung aller Werte, kann der Nutzer über die Schaltfläche Render das Sichtfeld der Kamera, abbilden. Gleich der Abbildung eines Editor-Fensters erscheint anschließend eine Vorschau des erstellten Screenshots. Anschließend soll dieser über eine weitere Schaltfläche exportiert und abgespeichert werden. Um die Aufnahme der Kameraperspektiven effizienter zu gestalten, wird dies zusätzlich über ein Menu Item ermöglicht, welchem eine Tastenkombination zugewiesen werden kann. So kann der Nutzer einstellen, dass beispielsweise bei der Betätigung der Taste F12 alle ausgewählten Kameras ihre Perspektive aufzeichnen. Um bei einer Szene mit mehreren Kameras einzustellen, welche der Kameras bei Betätigung der Tastenkombination aufzeichnen soll, wird eine weitere Oberfläche erstellt. Dieses Editor-Fenster beinhaltet eine Liste aller Kameras, welche im Besitz der Render-Komponente sind. In jener Liste wird dem Nutzer ermöglicht, individuell den Kameras das Rendern zu verweigern oder zu erlauben. Bei erneutem Betätigen der F12 Taste würden lediglich Kameras, dessen Komponenten eingeschaltet sind, eine Aufnahme vornehmen.

3.4. Aufbau Evaluation

Anschließend an die erfolgreiche Implementation der konzipierten Tools sollen diese durch einen Probandentest evaluiert werden. Kernziel dieser Evaluation soll sein, Aussagen über die Effizienzsteigerung treffen zu können. Diese Steigerung bezieht sich auf den Vergleich zwischen der Bearbeitung einer repetitiven Aufgabe, unter Verwendung der implementierten Tools, sowie ohne jene Tools. Die zu erhebenden Daten sollen zeitlich sowie aufwandstechnisch auf ihre Effizienz überprüft werden.

In den bisherigen Kapiteln wurde die Kollektion der zu entwickelnden Tools lediglich als solche bezeichnet. Für die Probanden hingegen soll die Kollektion als ein einheitliches Tool erscheinen, bei welchem zusätzliche Funktionen hinzugefügt oder entfernt werden können. Um dies zu bewerkstelligen, soll die Kollektion den Namen “MegaPint of Code“ tragen. Im verbleibenden Teil der Arbeit fungiert MegaPint als Bezeichnung für die erstellte Package Collection.

Um den Probanden die Bearbeitung der Evaluation zu vereinfachen, sollen für diese möglich wenig Voraussetzungen bestehen. Um dies zu erreichen, soll zunächst die Anzahl der benötigten Programme reduziert werden. Aus diesem Grund soll bis auf die Beantwortung der auszufüllenden Fragebögen, die gesamte Bearbeitung innerhalb der Unity Engine stattfinden. Desweiteren werden den Probanden Hinweise sowie Lösungsmöglichkeiten für die meisten Aufgaben bereitgestellt, welche verwendet werden können, sollte die Bearbeitung nicht ohne Hilfsmittel möglich sein. Da die Evaluation einen größeren Umfang und dadurch einen hohen zeitlichen Aufwand mit sich bringt, soll es dem Probanden ermöglicht werden, die Bearbeitung jederzeit zu pausieren und zu einem späteren Zeitpunkt fortzusetzen. Um das Speichern des Fortschrittes sowie das Bearbeiten der Aufgaben ohne Beisein des Entwicklers zu ermöglichen, müssen neue Funktionen entwickelt werden. Diese Funktionen sollen in einem neuen individuellen Package implementiert werden.

Die Nutzung jenes Evaluations Packages ist nur Probanden möglich. Um dies zu gewährleisten, soll der Import des Packages sowie die Nutzung aller Funktionen dessen nur durch Eintragen eines validen Tester-Tokens freigeschaltet werden. Jener Tester-Token soll den Probanden in ihrem individuellen Einführungsdokument überreicht werden. Ab dem Zeitpunkt des Imports jenes Packages werden Aufzeichnungen über das Verhalten des Probanden erstellt. Diese Aufzeichnungen (im folgenden Logs genannt) enthalten zunächst Informationen über den Start sowie das Beenden der Unity Sitzung, um die zeitliche Einordnung zu ermöglichen. Zudem wird jede Verwendung der MegaPint Funktionen sowie deren Unity Äquivalente, zusammen mit einem Zeitstempel erfasst. Zusätzlich sollen Basisfunktionen der Unity Engine wie beispielsweise der Domain Reload, bei welchem die erstellten Skripts neu geladen werden, ebenfalls dokumentiert werden. Im Beispiel des Domain Reloads ermöglicht dies, die aufgezeichneten Nutzungen nach einem solchen Event auszuschließen, da diese mit hoher Wahrscheinlichkeit nicht durch den Nutzer, sondern automatisch ausgeführt wurden. Die aufgezeichneten Daten werden in lokalen Textdateien gespeichert. Diese Dateien sollen beim Schließen des Unity Editors in eine Datenbank hochgeladen werden.

Die Evaluation soll in zwei Teile geteilt werden. Zunächst der geführte Teil, bei welchem der Proband eine Reihe vorgegebener Aufgaben erfüllen muss. In diesem Teil der Evaluation soll der

Schwerpunkt auf den Tools für die Fehlerkorrektur von Assets und der vereinfachten Bildschirmaufnahme liegen. Der zweite Teil der Evaluation befasst sich mit den verbleibenden Tools für das automatische Speichern einer Szene sowie das Starten einer Hauptmenü Szene. Dieser Teil soll weitestgehend einer Feldstudie ähneln, in welcher die Probanden die zu evaluierenden Tools in eigenen Projekten verwenden und deren Nutzbarkeit testen.

3.4.1. Geführte Evaluation

Zunächst soll, um die Bearbeitung der Aufgaben zu ermöglichen, ein Editor-Fenster erstellt werden, welches die aktuelle Aufgabe anzeigt. Zu den angezeigten Informationen gehören der Name der Aufgabe sowie deren Beschreibung, welche kurz und prägnant erklärt, was der Proband tun muss. Zudem soll die Oberfläche dem Proband ermöglichen, die Aufgabe nach erfolgreichem Abschließen zu beenden und dadurch die Nächste zu starten. Einige Aufgaben benötigen Vorwissen über die Funktionen der MegaPint Kollektion oder der einzelnen Packages. In diesen Fällen verlinkt die Aufgabe auf Ressourcen innerhalb der Kollektion, welche der Proband vor Bearbeitung der Aufgabe verinnerlichen kann. Die für die Auswertung relevanten Aufgaben sind mit einer Zeitmessung versehen. Während dieser Aufgaben zeigt die Oberfläche die vergangene Zeit sowie die Teilaufgaben an, welche der Proband zu erledigen hat. Die Lösung einer Teilaufgabe wird automatisch von der Oberfläche erkannt. Nach Abschluss aller Teilaufgaben, kann der Proband in der Bearbeitung der Evaluation fortschreiten. Neben der Zeitmessung werden, während der Bearbeitung einer solchen Aufgabe, auch Informationen über die Eingaben des Nutzers erhoben. Hierbei sollen niemals die genauen Eingaben des Nutzers erhoben werden, um den Datenschutz des Probanden zu gewährleisten. Die aufgezeichneten Daten werden in Tastatur- und Mauseingaben unterteilt. Neben der benannten Oberfläche soll ein zweites Editor-Fenster erstellt werden, welches dem Probanden die Möglichkeit gibt, den Fortschritt der Aufgaben zurückzusetzen, falls dieser eine Aufgabe neustarten möchte.

Die folgenden Daten werden für die Auswertung der geführten Evaluation erhoben:

- Benötigte Zeit für das Absolvieren der einzelnen Aufgaben
- Benötigte Nutzereingaben für das Absolvieren der einzelnen Aufgaben
- Verwendung aller Funktionen des Tools zur vereinfachten Bildschirmaufnahme
- Verwendung aller Funktionen des Tools zur automatischen Fehlerbehebung

Der Proband soll aufeinander folgende Aufgaben zur manuellen Bearbeitung einer repetitiven Aufgabe und unter Verwendung der erstellten Tools absolvieren. Während der Proband zunächst die Aufgabe zur manuellen Erstellung von Screenshots bearbeitet, werden dabei seine benötigten Eingaben sowie die benötigte Zeit gemessen. Gleiches gilt für die nachfolgenden Aufgaben, welche unter Verwendung der Tools bearbeitet werden. Vergleicht man die gemessenen Daten zur benötigten Zeit, können Aussagen bezüglich der zeitlichen Effizienz getroffen werden. So ist es das Ziel der konzipierten Tools, dass die benötigte Zeit bei ihrer Nutzung geringer ausfällt als bei Nutzung der Unity Funktionen. Jenes Konzept gilt ebenfalls für die Eingaben des Nutzers.

Ergänzend zu den, während der Bearbeitung der Aufgaben erhobenen Daten, sollen zusätzliche Informationen und die Gefühlslage der Probanden mittels Fragebögen erhoben werden. Der Proband soll hierbei Angaben über die Nutzbarkeit der Tools sowie der nativen Unity Funktionen aus seiner Perspektive tätigen. Zusätzlich soll er angeben, ob er bereits Vorwissen in den getesteten Tätigkeiten besitzt. So unterscheidet sich beispielsweise die Arbeitsweise eines Entwicklers, welcher bereits maßgebliche Erfahrung im Erstellen von Screenshots hat, von der eines unerfahrenen Entwicklers.

3.4.2. Freie Evaluation

Der freie Teil der Evaluation erstreckt sich über eine Dauer von 2 Wochen. In dieser Zeit sollen die Probanden die zu testenden Tools in eigenen Projekten verwenden und diese auf ihre Nutzbarkeit sowie deren Effizienz prüfen. Ziel der freien Evaluation ist es, zu erarbeiten, ob eine Veränderung im Verhalten des Probanden feststellbar ist. Um Aussagen über jenes Verhalten tätigen zu können, müssen Daten über die Verwendung der Tool-Funktionen sowie deren Unity Äquivalente erhoben werden. Dies geschieht durch den Import des Packages, welches bereits für die geführte Evaluation verwendet wurde. Durch den Import jenes Packages wird dauerhaft die Verwendung einiger Funktionen dokumentiert.

Die folgenden Daten werden für die Auswertung der freien Evaluation erhoben:

- Verwendung aller Funktionen des Tools zum automatischen Speichern
- Verwendung aller Funktionen des Tools zum Starten einer Hauptmenü Szene
- Verwendung der Speicher Funktion der Unity Engine (oder Strg + S)
- Das manuelle Öffnen einer Szene
- Das Starten des Playmodes

Um Änderungen des Verhaltens des Probanden zu erkennen, sollen zunächst die erfassten Daten anhand der Entstehungszeit geordnet werden. Ob eine Änderung vorliegt, soll danach von der Häufigkeit der aufgetretenen Nutzung der Funktionen abhängig gemacht werden. So wird beispielsweise erfasst, wie oft der Nutzer die Szene manuell gespeichert hat. Im Gegensatz dazu wird ebenfalls gemessen, wie oft die Funktion zum automatischen Speichern dies übernommen hat. Ziel der entwickelten Tools ist es, das Verhalten des Nutzers weitgehend zu beeinflussen, um seine repetitiven Tätigkeiten zu reduzieren. Sollte ein solcher Trend anhand der ausgewerteten Daten erkennbar sein, können die entwickelten Tools als Verbesserung der repetitiven Aufgaben angesehen werden. Zusätzlich werden die Meinungen der Probanden durch Fragebögen einbezogen.

4. Implementierung

4.1. Aufbau der Package Collection

Wie bereits in der Konzeption festgehalten besteht MegaPint aus einem Haupt-Package, welches grundlegende Funktionen bereitstellt, welche von allen weiteren Packages genutzt werden. Hierbei ist anzumerken, dass keines der MegaPint Packages ohne das Haupt-Package funktioniert.

Der in Abbildung 4.1 dargestellte Grundaufbau repräsentiert die grundlegende Funktionsweise der Kollektion. Jedes Package wird über den internen Package Manager importiert, entfernt oder aktualisiert. Nach erfolgreicher Installation des Packages nutzt dieses die im folgenden Kapitel beschriebenen Kernfunktionen, um beispielsweise Informationen im Hauptfenster der MegaPint Kollektion anzuzeigen. Abseits dessen implementiert jedes Package individuelle Funktionalitäten und Oberflächen unter Verwendung bereitgestellter Hilfsfunktionen.

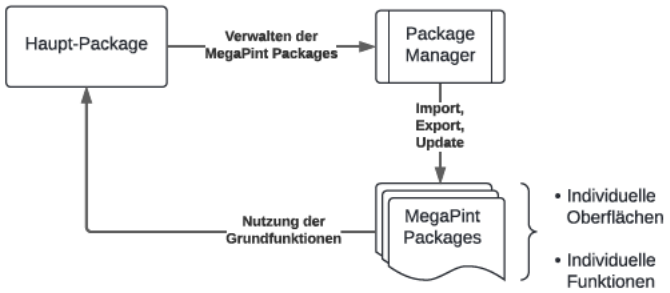

Abbildung 4.1.: Aufbau der Package Collection

Da jedes Package als eigenes UPM Package entwickelt wird, müssen von Unity festgelegte Regeln befolgt werden. Diese Regeln beziehen sich auf die zu erstellende package.json Datei sowie die generelle Struktur eines UPM Packages. Um jene Regeln einzuhalten und das gängige Format eines UPM Packages zu wahren, basiert die Struktur der entwickelten Packages auf der von Unity vorgegebenen. Jedoch bestehen einige Unterschiede zwischen der vorgegeben Struktur und der genutzten Struktur, wie in Abbildung A.0.1 zu erkennen ist.

Aus Unitys UPM Struktur übernommen werden die package.json, README.md und LICENSE.md Dateien. Diese enthalten essenzielle Informationen für den Nutzer und die Unity Engine, um die Packages zu verarbeiten. Zudem werden die entstehenden Assets sowie Skripts, wie im vorgegebenen Layout, in Editor und Runtime unterteilt. Alle weiteren Ordner wie Tests, Samples und Documentation werden nicht benötigt, da diese durch den internen Package Manager ausgelesen werden sollen. Der Editor Ordner enthält alle Objekte, welche lediglich im Unity Editor nutzbar sein sollen. Der Runtime Ordner hingegen enthält alle Objekte, welche möglicherweise auf Objekten

innerhalb von Szenen referenziert werden könnten. Dies umfasst alle Skripts, welche für Game Objects erstellt wurden sowie alle Assets, welche in diesen referenziert werden. Hierbei ist darauf zu achten, dass kein Skript Abhängigkeiten zu Objekten des Editor Ordners besitzen darf, da dies ein Ausführen des Programms außerhalb der Unity Engine unmöglich macht.

Ebenfalls aus dem von Unity bereitgestellten Layout übernommen werden die Benennung der Assembly Definition Dateien. Jedoch wird, wie in Abbildung A.0.1 ersichtlich ist, eine strikte Ordnerstruktur innerhalb des Editor Ordners gefordert. Dieser enthält zwei weitere Verzeichnisse. Das erste trägt den Namen Skripts und enthält alle, in diesem Package vorhandene Skripts, welche nicht zur Laufzeit referenziert werden. Das zweite Verzeichnis trägt den Namen Resources. Dies ist einer von mehreren besonderen Ordnernamen. Die Sonderheit von Ordnern mit diesem Namen ist, dass durch die Verwendung der Resources API jene Ordner nach Dateien durchsucht werden können. Alle Resources Ordner bilden hierbei ein Verzeichnis, welches nach dem gesuchten Pfad durchsucht wird. Diese Funktionalität ist essenziell für die Verwendung einiger Grundfunktionen der MegaPint Kollektion.

Um die Verwendung der Resources API zu verbessern, fordern einige Funktionen feste Pfade zu den jeweiligen Dateien. Um dies an einem kurzen Beispiel aufzuzeigen, soll dies anhand des Display Content Ordners erklärt werden. Jedes MegaPint Package besitzt eine UXML Datei, welche den Inhalt des Info Tabs des jeweiligen Packages beinhaltet. Im Hauptfenster der MegaPint Kollektion soll nach Auswahl des importierten Packages und der Betätigung der Info Schaltfläche jener Inhalt angezeigt werden. Um diesen Laden zu können, wird eine Referenz auf die zugrundeliegende UXML Datei benötigt. Diese befindet sich immer im Ordner Display Content mit dem Namen Info.uxml. Um die Datei finden zu können, setzt sich der Pfad wie folgt zusammen (Name des Packages/User Interface/Display Content/Info.uxml). Der Name des Packages ist hierbei variabel und abhängig von dem ausgewählten Package. Durch die Nutzung der Resources API ist der Pfad, in welchem die Resources Verzeichnisse liegen, nicht relevant und kann ignoriert werden. Um die Abfrage zu beschleunigen und eventuelle gleichnamige Verzeichnisse auszuschließen, beginnt jeder Resources Pfad in dem Grundverzeichnis MegaPint.

Wenngleich die genannte Struktur für jedes Package der MegaPint Kollektion gefordert wird, so ist der Aufbau des Haupt-Packages leicht abgeändert. Dieses benötigt beispielsweise kein Unterverzeichnis mit dessen Namen. Zudem besitzt das Haupt-Package ein zusätzliches Verzeichnis innerhalb des Editor Ordners. In jenem Verzeichnis werden Assembly Definition Dateien aufbewahrt. Trotz dessen, dass der Aufbau der Kollektion die Referenzen zwischen den gegebenen Funktionen einseitig limitiert, so dass lediglich die entwickelten Tools Zugriff auf das Haupt-Package haben, gibt es Ausnahmen. In Ausnahmefällen, in welchen es erforderlich ist, Funktionen der Packages in anderen Packages wie beispielsweise dem Haupt-Package zu verwenden, muss die Assembly Definition bereits vor dem Import des Packages vorhanden sein. Um mögliche Fehler, durch fehlende GUID Referenzen zu umgehen, werden die betroffenen Dateien nicht in den Packages sondern in dem genannten Verzeichnis erstellt. In den Packages wird nachträglich eine Assembly Reference Datei erstellt, welche die im Haupt-Package befindliche Assembly Definition Datei simuliert.

4.2. Kernfunktionen der Package Collection

4.2.1. Package Management

Die Verwaltung der einzelnen Packages der Kollektion ist eine der wichtigsten Hauptfunktionen. Um dem Nutzer diese Verwaltung zu ermöglichen, wurde ein interner Package Manager erstellt. Dieser basiert, betreffend seiner technischen sowie grafischen Umsetzung, auf dem Beispiel von Unitys Package Manager.

Grafische Umsetzung

Der interne Package Manager der MegaPint Kollektion besteht als individuelles Editor-Fenster. Dieses kann über das Menu Item “MegaPint/Package Manager“ oder eine alternativ angelegte Tastenkombination geöffnet werden. Der Package Manager enthält zunächst, wie auf Abbildung 4.2 ersichtlich, auf der linken Seite eine Auflistung aller existierender Packages der MegaPint Kollektion. Diese Auflistung ist ein von Unity vorgegebenes ListView Element. Jeder Eintrag dieser Liste besteht aus mehreren Textelementen, wobei das erste Element den Namen des Packages anzeigt. Das zweite Textelement zeigt die importierte Version des jeweiligen Packages. Ist das Package nicht importiert, so ist auch dieses Element nicht sichtbar. Durch das Auswählen eines Eintrages der Liste aktualisieren sich die Angaben auf der rechten Seite des Package Managers.

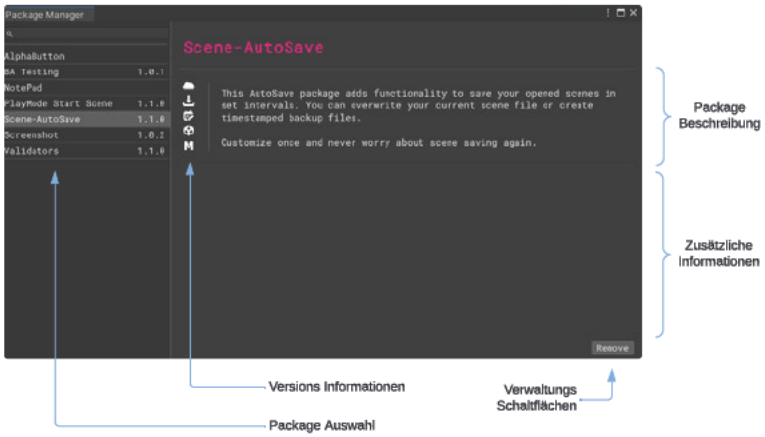

Abbildung 4.2.: Grafische Umsetzung des Package Managers

Neben der Namensanzeige des ausgewählten Packages wird darunter die Beschreibung des Packages wiedergegeben. Linkerhand der Beschreibung befinden sich Bildelemente, welche unterschiedliche Versionsangaben enthalten. Nachfolgend von oben nach unten geben diese die neueste Version des Packages, die installierte Version des Packages sowie das letzte Änderungsdatum an. Die beiden letzten Symbole geben die minimale Unity sowie MegaPint Version an, welche installiert sein muss, um das gewählte Package zu unterstützen. Jene Informationen sind für den Nutzer lesbar, sobald er mittels des Cursors diese Bildelemente berührt. Da es sich bei diesen Elementen um Elemente des Typen “VisualElement“ handelt, welche eine Hintergrundgrafik besitzen, unterstützen diese keine nativen Tooltips. Aus diesem Grund verwenden jene Elemente eine eigene Variation des Tooltips, auf dessen technische Umsetzung später eingegangen wird. Beim Auslösen der Tooltips ändert sich zudem die Farbe der Elemente, um dem Nutzer neben dem Tooltip eine visuelle Rückmeldung über

die Interaktion der Element zu bieten. Zudem färbt sich das Bildelement für die installierte Version orange, sobald eine neue Version des Packages verfügbar ist.

Unterhalb der eben genannten Elemente befinden sich zusätzliche Informationen über das gewählte Package. Dies umfasst mögliche Abhängigkeiten auf andere Packages, bestehende Samples sowie Informationen über alternative Package Variationen. So ist es beispielsweise möglich, zwei Variationen eines Packages anzubieten. Eine dieser Versionen integriert eines der MegaPint Packages, um einen Aspekt des Packages zu verbessern. So wird zunächst das Basis Package importiert. Danach kann der Nutzer durch das Auswählen jener Variation das Package abändern. Diese Variation zeigt ebenfalls ihre Abhängigkeiten auf das zu nutzende Package an und würde diese bei einem Austausch automatisch ebenfalls importieren.

Zuletzt enthält die Oberfläche am unteren rechten Rand die Schaltflächen zum Importieren, Entfernen sowie Aktualisieren des gewählten Packages. Die jeweiligen Schaltflächen werden nur angezeigt, wenn diese benötigt werden. So wird der Import Knopf nur gezeigt, sollte das Package noch nicht importiert sein und ebenfalls die Schaltfläche zum Aktualisieren des Packages nur, wenn dieses importiert und eine neue Version verfügbar ist. Nach Benutzung einer dieser Schaltflächen werden alle drei ausgegraut und können bis zur Vollendung des gestarteten Vorgangs nicht erneut verwendet werden. Um dem Nutzer die Arbeit mit der Oberfläche zusätzlich zu vereinfachen, wurden alle interaktiven Elemente mit prägnanten Tooltips ausgestattet. Außerdem wurde eine Suchleiste für die Einträge der Packages angelegt, mit welcher es dem Nutzer möglich ist, die angezeigten Einträge zu filtern.

Technische Umsetzung

Um die technische Umsetzung zu erläutern, wird zunächst auf die Verwendung der bereitgestellten API Methoden und deren Verwendung zum Importieren, Entfernen und Aktualisieren eines Packages eingegangen. Anschließend soll eine maßgebliche Einschränkung dieses Systems aufgezeigt und gelöst werden. Abschließend konzentriert sich dieses Kapitel auf die Speicherung der Package Informationen sowie den Abgleich der aktuellen mit der neuesten Version mittels einer Git Erweiterung.

Unitys API stellt drei Funktionen zur Verfügung, mit welchen die Verwaltung von UPM Packages innerhalb des Projektes ermöglicht wird. Diese sind “Client.Remove()“, “Client.Add()“ und “Client.Embed()“. Folgend wird die Implementation dieser Funktionen innerhalb des Package Managers der MegaPint Kollektion erläutert.

Zunächst wird die Methode “Client.Add()“ verwendet, um ein Package zu importieren. Jene Methode benötigt die URL des Packages als Parameter. Durch das Eingeben der Git Adresse importiert Unitys Package Manager das gefundene Package in das Projekt. Dies ist nur möglich, wenn das an der Adresse vorzufindende Projekt und die darin befindliche package.json dem Aufbau und den Auflagen eines UPM Package entsprechen.

Da die “Client.Add()“ Methode asynchron verläuft, wird diese ebenfalls in einem asynchronen Kontext verwendet. Hierfür wird, wie in Listing 4.1 gezeigt, ein asynchroner Task erstellt, welcher als Rückgabewert, den Erfolg des Vorgangs zurück liefert. Als Parameter verlangt diese Methode

die URL des zu importierenden Packages. Während der Bearbeitung des Prozesses wird, wie in Zeile fünf bis sechs in Listing 4.1 gezeigt, in regelmäßigem Intervalls der Fortschritt des Prozesses überprüft. Sobald dieser abgeschlossen ist, kann die Bearbeitung des Vorgangs mit der Rückgabe des Status beendet werden.

```
1  private static async Task <bool> Add(string packageUrl)
2  {
3      AddRequest request = Client.Add(packageUrl);
4
5      while (!request.IsCompleted)
6          await Task.Delay(RefreshRate);
7
8      if (request.Status >= StatusCode.Failure)
9          onFailure?.Invoke(request.Error.message);
10
11     return request.Status == StatusCode.Success;
12 }
```

Listing 4.1: Methode zum Importieren eines MegaPint Packages

Nach erfolgreichem Importieren des Packages kann dieses nun in das Projekt eingebettet werden. Hierzu wird die Methode "Client.Embed()" verwendet. Hierbei erstellt Unity eine Kopie der Package Dateien, welche, nun dem Packages Verzeichnis unterliegen und bei der Neuerstellung des Package Caches ebenfalls zurückgesetzt werden.

```
1  private static async Task Embed(string packageName)
2  {
3      EmbedRequest request = Client.Embed(packageName);
4
5      while (!request.IsCompleted)
6          await Task.Delay(RefreshRate);
7
8      if (request.Status >= StatusCode.Failure)
9          onFailure?.Invoke(request.Error.message);
10 }
```

Listing 4.2: Methode zum Einbetten eines MegaPint Packages

Da die Methode des Einbettens ebenfalls erneut asynchron verläuft, ist die, in Listing 4.2, gezeigte Methode ein asynchroner Task, welcher diesmal jedoch keinen Rückgabewert benötigt. Neben Verwendung der "Client.Embed()" Funktion in Zeile drei, funktioniert diese gleich der vorangegangenen. Um ein Package einzubetten, wird nicht dessen URL, sondern der Name des Packages benötigt. Hierbei handelt es sich nicht um den Anzeigenamen des Packages, sondern den Namen, welcher in der package.json Datei festgelegt ist.

Zuletzt wird die “Client.Remove()“ Funktion verwendet, um bereits importierte Packages zu entfernen. Hierzu wird, wie in Listing 4.3 erneut gezeigt, eine asynchrone Funktion geschaffen, welche die bereitgestellte Funktion in Zeile drei implementiert. Neben dem Namen des zu entfernenden Packages benötigt die Umsetzung des Package Managers für MegaPint einen zusätzlichen Parameter vom Typ Boolean. Dieser gibt an, ob der Package Cache nach erfolgreichem Entfernen erneuert werden soll. Dies ist von Vorteil, wenn mehrere Packages entfernt werden. Um eine Ladezeit nach jedem Entfernvorgang zu unterbinden, wird in diesem Fall der Cache einmalig nach Entfernen des letzten Packages erneuert. Abgesehen dieser Unterschiede, implementiert diese Methode die gleichen Funktionalitäten wie die vorangehenden.

```
1  public static async Task Remove(string packageName, bool
       suppressCacheRefresh = false)
2  {
3      RemoveRequest request = Client.Remove(packageName);
4
5       while (!request.IsCompleted)
6          await Task.Delay(RefreshRate);
7
8      if (request.Status >= StatusCode.Failure)
9          onFailure?.Invoke(request.Error.message);
10     else
11     {
12         onSuccess?.Invoke();
13
14         if (!suppressCacheRefresh)
15             PackageCache.Refresh();
16     }
17 }
```

Listing 4.3: Methode zum Entfernen eines MegaPint Packages

Durch das Betätigen der Schaltflächen auf der Oberfläche des Package Managers, wird mittels der in Listing 4.1, 4.2 und 4.3 aufgezeigten Methoden die Verwaltung des gewählten Packages möglich. Aufbauend auf diesen Grundfunktionen werden in der Umsetzung des internen Package Managers Erweiterungen zu diesen Methoden erstellt. So implementiert die Methode zum Aktualisieren eines Packages beispielsweise die in Listing 4.1 gezeigte Funktion. Wird ein, bereits im Projekt vorhandenes Package, erneut importiert, so bleiben die existierenden Dateien bestehen und werden lediglich gemäß den Änderungen aktualisiert. Jenes Prinzip wird auch bei der Installation einer Variation eines Packages verwendet. Hierbei wird die URL der Variation mittels der implementierten Funktion importiert. Da der Name des Packages in der package.json identisch mit dem bereits bestehenden Package ist, wird dieses mit den Daten der Variation überschrieben.

Bei der Nutzung des implementierten Systems tritt ein Problem auf, welches die Effizienz sowie die Nutzbarkeit dessen beeinträchtigt. Das besagte Problem beschränkt sich auf die Verwaltung mehrerer Packages zum gleichen Zeitpunkt. Hierbei kann es vorkommen, dass lediglich eine bis zwei der gewünschten Aktionen durchgeführt werden. So beispielsweise wenn der Nutzer mehrere

Packages gleichzeitig importieren möchte. Um dies zu verhindern, ist das Importieren von mehreren Packages über die grafische Oberfläche nicht möglich. Jedoch kann es vorkommen, dass ein Package mehrere Abhängigkeiten zu anderen Packages hat und diese beim Importieren ebenfalls installiert werden müssen. Daher ist eine Lösung hierfür zu implementieren.

Zunächst ist zu erläutern, wie dieser Sachverhalt entsteht. Nach dem Importieren eines Packages wird ein Domain Reload durchgeführt. Dies ist ein Prozess, in welchem die importierten Daten sowie alle im Projekt befindlichen überprüft werden und einige Caches sowie alle statischen Variablen von Skripts zurückgesetzt werden. Weist man den Package Manager nun an, ein Package zu importieren verläuft dieser Prozess ohne Probleme. Nach Beendigung des Prozesses beginnt der zweite Import. Jedoch möchte der Unity Editor nach der Installation des ersten Packages einen Domain Reload durchführen. Da dieser mit einer kleinen Verzögerung startet, läuft zu dieser Zeit bereits der zweite Importprozess. Nach Beendigung des zweiten Prozesses führt der Editor sofort den eingereihten Domain Reload durch, wodurch alle weiteren Anweisungen, welche an den Package Manager gestellt wurden, verloren gehen. Anschließend wird ein weiterer Domain Reload für den Import des zweiten Packages durchgeführt.

Um dieses Problem zu lösen, sollen die zu importierenden Packages mit einer Anweisung von Unitys Package Manager importiert werden. Hierzu ist es notwendig, die Adressen der Packages direkt in die Manifest Datei des Projektes zu schreiben. Da das direkte Bearbeiten der Manifest Datei zu ungeahnten Problemen führen kann, sollte dies nur angewandt werden, wenn dies notwendig ist. In dem genannten Fall wird dies notwendig, wenn drei Anweisungen zum Abschluss des Prozesses nötig wären. Durch die in Listing A.1 gezeigte Umsetzung des Einfügens der Package Adressen und einem anschließend ausgelösten Domain Reload, erkennt der Unity Editor alle fehlenden Packages und setzt die notwendigen Prozesse fort bis alle Packages importiert wurden.

Wie aus den implementierten Methoden hervorgeht kann kein Package importiert werden, ohne die zugehörige Git Adresse zu kennen. Desweiteren benötigen mehrere Schnittstellen wie beispielsweise der Package Manager oder das Hauptfenster weitere Informationen wie den Package Namen, dessen Beschreibung oder die Version des Packages. Diese benötigten Informationen sind über zwei Wege erreichbar. Zunächst enthält die package.json Datei die meisten Informationen, welche Unitys Package Manager zum Anzeigen des Packages benötigt. Zusätzlich dazu besteht für jedes Package eine eigene Klasse, welche weitere Daten beinhaltet. Diese Klassen können mittels des dafür erstellten Management Skripts ausgelesen werden. Dieses speichert nach jedem Domain Reload alle definierten “PackageData“ Klassen in einem Cache namens “DataCache“ ab. Wie in Abbildung 4.3 zu erkennen doppeln sich einige Informationen der beiden Quellen. Jedoch beinhalten beide Informationen, welche der Anderen nicht bekannt sind. Aus diesem Grund ist es notwendig, beide Quellen zu vereinen.

Um eine einzelne Schnittstelle zu schaffen, welche alle benötigten Informationen beinhaltet, wurde ein Cache geschaffen. Der Package Cache ist eine statische Klasse, welche nach jedem ausgeführten Domain Reload die Informationen beider Quellen zusammenfügt und als “CachedPackage“ bereitstellt. CachedPackage ist ebenfalls eine Klasse, welche als Informationsträger verwendet wird. Um die benötigten Informationen zusammenzufügen, wird zunächst der zuvor benannte DataCache mit den Informationen aller PackageData Klassen ausgelesen. Anschließend erstellt der Package

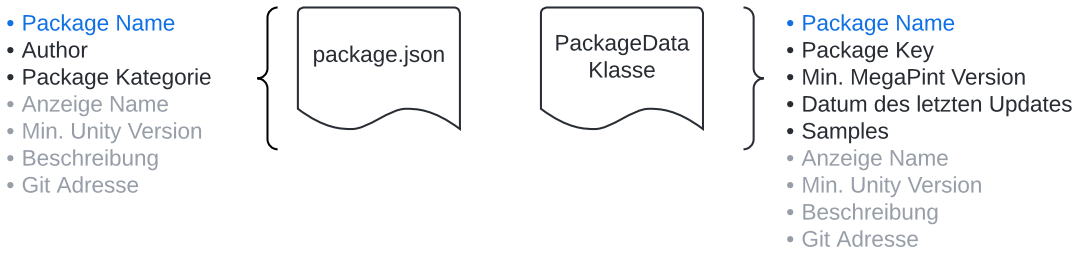

Abbildung 4.3.: Vergleich zwischen package.json und der PackageData Klassen

Manager, eine Liste aller importierten Packages. Da der Name des Packages in beiden Informationsquellen vorhanden ist, wird dieser verwendet um diese miteinander zu verbinden. Der PackageCache durchläuft die Liste aller MegaPint Packages und gleicht diese mit den installierten Packages ab. Ist ein Package importiert, so werden die Informationen mit denen der gefundenen package.json erweitert. Auf Grundlage dieses Systems ist allen Schnittstellen möglich, durch die Eingabe des Schlüssels des gewünschten Packages, alle Informationen auszulesen.

Abschließend soll nun aufgezeigt werden wie eine mögliche neue Version des Haupt-Package, unter Verwendung einer Git Erweiterung, erkannt werden kann. Diese Funktion ist maßgeblich hilfreich um dem Nutzer neue Funktionen und Fehlerbehebungen nahezubringen, da die Benachrichtigung über ein bereitstehendes Update innerhalb des Tools mehr Aufmerksamkeit erhält.

Bei der Veröffentlichung einer neuen Version wird mittels Git ein Release Tag erstellt. Diese Release Tags markieren einen Commit, welcher als die gewünschte Version gilt. Diese Tags können in der Zieladresse des Packages verwendet werden um diese zu spezifizieren. So kann beispielsweise nach der gewohnten URL ein "#development" folgen. Insofern ein Branch oder ein Tag mit dem Namen development existiert, wird dieser als Ziel des Import Prozesses verwendet. Da MegaPint dieses System bereits für den Import der Packages nutzt, muss lediglich der aktuell verwendete Tag mit dem neuesten Release Tag abgeglichen werden. Sollten diese nicht übereinstimmen, ist ein Update für das Package verfügbar.

Um den neuesten Release Tag eines Repositories auszulesen, kann man, wie in Listing A.4 gezeigt, mittels eines Git Befehls alle Tags auslesen und den neuesten zurückgeben. Um jedoch Git Befehle innerhalb der Unity Engine zu verwenden ist es notwendig, eine Extension Klasse zu schreiben, welche es erlaubt Git zu verwenden. Hierzu wird in Listing A.5 eine Methode erstellt, welche den auszuführenden Befehl als Parameter an einen Git Prozess übergibt. Die Erstellung eines solchen Git Prozesses innerhalb von Unity ist mittels der in Listing A.7 gezeigten Methode möglich.

4.2.2. Speichersystem

Um die Einstellungen der Tools durch den Nutzer, zu erhalten, müssen diese gespeichert werden. Das Speichern primitiver Datentypen wie Integer, Float, Boolean und String wird in Unity durch die "PlayerPrefs" ermöglicht. Dies ist vergleichbar mit einem Dictionary. Durch die Eingabe eines festgelegten Schlüssels können Werte abgespeichert und ausgelesen werden. Da alle zu speichernde

Daten aus primitiven Datentypen bestehen, liegt die Nutzung der PlayerPrefs nahe. Jedoch bestehen Nachteile an jenem System. So ist es für den Nutzer ein Leichtes, die Daten zu überschreiben, sobald dieser den jeweiligen Schlüssel kennt. Zudem wäre es möglich, dass der Nutzer in eigenen Tools die PlayerPrefs verwendet und sich die Schlüssel der einzelnen Werte überschreiben. Im Übrigen wird jenes System häufig zum Testen der Spieldateien verwendet, da mit nur wenigen Schritten alle angesammelten Daten zurückgesetzt und somit das Spiel von neuem getestet werden kann. Hierbei würden demnach auch alle gespeicherten Daten der MegaPint Kollektion verloren gehen.

Aus den eben genannten Gründen soll eine eigene Implementation eines Speichersystems, basierend auf dem Vorbild der PlayerPrefs, geschaffen werden. Das Speichern der Daten erfolgt innerhalb eines Scriptable Objects, welches der Nutzer bei Installation von MegaPint erstellen muss. Jenes Objekt enthält für jeden primitiven Datentypen ein serialisierbares Dictionary. Durch die Eingabe eines Schlüssels werden wie bei den PlayerPrefs Einträge in den jeweiligen Dictionaries zugewiesen oder ausgelesen. Die Klasse des Scriptable Objects enthält zudem statische Methoden, über welche diese Funktionalitäten aufgerufen werden können. Hierzu wird eine Referenz auf die im Projekt liegende Datei des Scriptable Objects benötigt. Jene Referenz wird im Kontext der “InitializeOnLoad“ Methode unter Verwendung der in Listing A.3 gezeigten Methode ausgelesen. InitializeOnLoad ist ein Klassenattribut, welches die Erstellung eines statischen Konstruktors erlaubt, welcher direkt nach jedem Domain Reload ausgeführt wird.

Um die in den Dictionaries liegende Daten zu serialisieren, ist es erforderlich, dass diese öffentlich zugänglich sind. Dies erlaubt es dem Nutzer die Werte der gespeicherten Einstellung nach Belieben zu ändern. Um dieses Problem teilweise zu lösen, wird, wie in Listing A.2 gezeigt, ein Editor für die Klasse des Scriptable Objects erstellt, welcher den Inspector des Objektes überschreibt. Hierdurch ist es nicht mehr möglich, die Werte der Daten über den Inspector zu bearbeiten. Jedoch ist diese Lösung nicht vollends sicher. Durch das Umstellen des Inspectors in den Debug Modus, können die Daten erneut bearbeitet werden.

4.2.3. Hilfsfunktionen

Für eine bessere Struktur innerhalb der Packages, werden einige Funktionalitäten durch partielle Klassen vereinfacht. Bei jenen Klassen handelt es sich um Klassen, welche in jedem Package vertreten sein müssen. In den partiellen Klassen, welche sich im Haupt-Package befinden, werden die Hilfsfunktionen implementiert, während diese in den verbleibenden verwendet werden.

Speichern von Pfaden
Alle Pfade, welche andernfalls als konstante Strings in Klassen definiert werden würden, werden in der partiellen Klasse “Constants“ festgehalten. Hierbei wird eine Unterklasse mit dem Namen des jeweiligen Packages zur besseren Strukturierung angelegt. Diese enthält alle konstanten Pfade und beugt damit Dopplungen innerhalb des Codes vor und bietet eine einheitliche Schnittstelle für Pfadänderungen.

Aufrufen von gespeicherten Werten
Ähnlich wie das Abfragen bestehender Pfade, nutzen mehrere Schnittstellen die gleichen Werte des

Speichersystems. Anstelle den Aufruf des Speichersystems mit dem jeweiligen Schlüssel in jeder Klasse neu zu definieren, geschieht dies in einer partiellen Klasse. Diese Klasse enthält Properties für jeden Wert des Packages, welcher im Speichersystem vorhanden ist. Diese Funktionalität erspart erneut die Duplikation bereits geschriebenen Codes und ermöglicht es, die Änderungen möglicher Schlüssel Strings in einer Datei vorzunehmen.

Öffnen von Editor Fenstern
Die meisten Editor-Fenster der MegaPint Kollektion benötigen ausgelesene Werte des Speichersystems. Sollte die Integrität der Speicherdaten nicht gegeben oder das Asset nicht im Projekt vorhanden sein, treten Fehler auf. Um dies zu verhindern, soll vor dem Öffnen eines Editor-Fensters überprüft werden, ob das Settings Asset existiert und gelesen werden kann. Um diese Abfrage nicht in die Methode eines jeden Menu Items übertragen zu müssen, wurde eine neue Methode zum Öffnen der Fenstern erstellt. Diese agiert wie die bisherige, jedoch öffnet sie, sofern das Settings Asset nicht gefunden wird, ein separates Fenster, mit welchem dieses erstellt wird. Wie in Listing A.8 gezeigt wird das gegebene Editor-Fenster nur nach erfolgreicher Überprüfung geöffnet. Jene Funktionalität ist erneut in einer partiellen Klasse implementiert, in welcher ebenfalls alle Menu Items untergebracht werden.

Anzeigen der Package Oberflächen im Hauptfenster
Mit der Auswahl eines Packages im Hauptfenster der Kollektion werden die in den Packages definierten Oberflächen angezeigt. Wie bereits im Aufbau der Kollektion erwähnt, werden die UXML Dateien der Oberflächen über ihren festen Pfad gefunden. Anschließend werden diese mittels der dafür bereitgestellten API initialisiert und in die Oberfläche des Hauptfensters integriert. Da das UI Toolkit im Gegensatz zum IMGUI System mit Callbacks arbeitet, müssen diese bei Erstellung der Oberfläche definiert werden. Die Implementation dieser geschieht ebenfalls in einer partiellen Klasse der jeweiligen Packages. Der Name der Methode ist hierbei immer der ausgeschriebene Schlüssel des Packages. Anschließend werden bei der Erstellung der Oberflächen die Methoden mit der in Listing 4.4 gezeigten Funktion aufgerufen. Um die gesuchte Methode innerhalb der partiellen Klasse zu finden, wird Reflection verwendet. Dabei handelt es sich um ein Prinzip, welches es ermöglicht Variablen sowie auch Methoden anhand ihres Namens zu finden und auszuführen. Der übergebene Name ist hierbei der Schlüssel des gewählten Packages, um die gleichnamige Methode zu finden. Die übergebenen Parameter enthalten das oberste visuelle Element der erstellten Oberfläche sowie weitere benötigte Referenzen.

```
1  private static void CallMethodByName(string name, object[] parameters)
2  {
3      MethodInfo method = typeof(DisplayContent).GetMethod(name,
           BindingFlags.NonPublic | BindingFlags.Static);
4
5      if (method != null)
6          method.Invoke(typeof(DisplayContent), parameters);
7  }
```

Listing 4.4: Methode zum öffnen eines Editor-Fensters

4.3. User Interface

4.3.1. Anwendung des UI Toolkits

Die Erstellung der Oberflächen selbst wird unter Verwendung des UI Builders vorgenommen. Anschließend werden für die interaktiven Elemente innerhalb der Skripts, in welchen jene Oberflächen verwendet werden, Funktionalitäten mittels Callback Methoden erschaffen. Bei Callback Methoden handelt es sich um überwachende Methoden, welche ihre Funktionalität auslösen sobald die erzielte Interaktion auftritt. So kann ein Listenelement beispielsweise eine Callback Methode für das Auswählen eines Elements und das Betätigen der rechten Maustaste besitzen. Um die Oberflächen mittels des UI Builders zu erstellen, benötigt es bereits vorgefertigte visuelle Elemente. Um die Kompatibilität mit zukünftigen Unity Versionen sowie die Einheitlichkeit mit dem bestehenden Aussehen des Editors zu wahren, werden maßgeblich nur die von Unity bereits gegebenen Elemente verwendet.

Um die Vorteile des UI Toolkits auszuschöpfen und um einige der im Konzept genannten Funktionen umzusetzen, werden innerhalb der Oberflächen USS Dateien verwendet. Um eine einfachere Strukturierung des Inhalts der USS Dateien zu ermöglichen, werden drei von ihnen erstellt. Die erste USS Datei enthält zunächst Klassen für eigens erstellte visuelle Elemente, die Klasse des Rootelements und die Klassen für das helle sowie das dunkle Farbschema. Die zweite Datei enthält alle Klassen, welche bei Anwendung auf ein Element ein bestimmtes Attribut dessen überschreiben. Die letzte Datei besteht aus allen überschriebenen Unity-Klassen. Jede UXML Datei enthält alle drei dieser USS Dateien.

Die Anwendung der von Unity abweichenden Farbpalette wurde mittels der USS Klassen sowie der Verwendung von Pseudo States realisiert. Die Implementation derer soll nachfolgend am Beispiel eines Knopfes aufgezeigt werden. Zunächst wird, wie in Listing 4.5 gezeigt, die von Unity definierte Klasse in einer USS Datei überschrieben. Anschließend werden die gewünschten Parameter, wie die Hintergrundfarbe, Textfarbe und Randfarbe, mit den im Rootelement definierten Farbvariablen überschrieben. Durch das Anhängen der USS Datei, in welcher diese Klasse eingetragen ist, wird fortan in allen Knopf-Elementen der Oberfläche Unitys Design überschrieben.

```
1  .unity-button {
2      background-color: var(--button-color);
3      color: var(--text-primary-color);
4      -unity-background-image-tint-color: var(--button-image-tint);
5
6      border-top-color: var(--button-border-color-top);
7      border-bottom-color: var(--button-border-color-bottom);
8      border-left-color: var(--button-border-color-left);
9      border-right-color: var(--button-border-color-right);
10 }
```

Listing 4.5: Überschriebene USS Button Klasse

```
1  .unity-button:hover {
2      border-color: var(--identity-color);
3  }
4
5  .unity-button:active {
6      color: var(--button-active-text-color);
7      -unity-background-image-tint-color: var(--button-active-text-color);
8
9      border-color: var(--identity-color);
10     background-color: var(--identity-interacted-color);
11 }
```

Listing 4.6: Überschriebene USS Button PseudoStates

Bei einer Interaktion des überschriebenen Knopfes würde es, trotz der überschriebenen Werte, zur Anzeige von Unitys Design kommen, da keiner der angelegten Pseudo States überschrieben wurde. Hierzu soll festgelegt werden, dass während des Hover Status der Knopf seine Randfarbe und während des Drückens durch den Nutzer seine Randfarbe sowie Hintergrundfarbe ändert. Hierzu müssen lediglich jene Pseudo States, wie in Listing 4.6 gezeigt, überschrieben werden.

Das Überschreiben der bestehenden Klassen ist nur möglich durch die Angabe der genauen Klassennamen. Bei genesteten Elementen und unter Einbeziehung des Status des Elternelements können diese Namen zunehmend komplexer werden. Die Klasse ".unity-toggle:active>.unity-toggle__input> #unity-checkmark" überschreibt beispielsweise das Aussehen des "checkmark" Elements innerhalb des "unity-toggle__input" Elements eines von Unity definierten Toggles, welcher gerade den Pseudo State active besitzt. Um den Klassennamen eines Elements herauszufinden wurde der UI Debugger des UI Toolkits verwendet. Hierbei handelt es sich um ein Werkzeug, welches vergleichbar mit der Entwickleransicht einer Website ist. Durch das Auswählen eines Elements innerhalb des UI Debuggers, werden dessen Attribute sowie auch alle auf dieses Element angewandte Klassen angezeigt. So muss lediglich die verwendete Klasse des gewünschten Element mittels des Debuggers ausgelesen und im Anschluss in der dafür vorgesehenen USS Datei überschrieben werden, um dessen Design zu bearbeiten.

4.3.2. Verwendung von UXML Factories

Um die Verwendung des UI Builders zu verbessern und die Nutzung komplexerer Elemente innerhalb dessen zu ermöglichen, besteht die Möglichkeit, eigene visuelle Elemente als "UXMLFactory" zu erstellen. Ein, mittels einer UXMLFactory, erstelltes Element kann anschließend im UI Builder wie die von Unity definierten Elemente verwendet werden. UXMLFactories legen bereits bei der Erschaffung des Elements dessen Werte, Klassen und Kinder auf Basis eines zugrunde liegenden Elements fest. So ist es beispielsweise möglich eine UXMLFactory zu erstellen, welche auf dem Textelement basiert. Jene Factory soll sekundäre Textelemente erschaffen. Daher wird bei der Erstellung des Elements die Farbe des Textes auf die Sekundärfarbe festgelegt.

Ebenfalls ist es möglich, durch die Verwendung des Basistypen VisualElement als Grundlage der Factory vollkommen neue Elemente zu erschaffen, wie in Listing A.9 gezeigt. Hierbei handelt es sich um eine Factory, welche eine horizontale Linie über die zur Verfügung stehende Länge zeichnet. Da dieses Element im Design der Oberflächen häufig auftritt, steigert die einmalige Erstellung einer UXMLFactory die Effizienz im Umgang mit dem UI Builder maßgeblich, da nicht in jeder Oberfläche das gewünschte Element neu erstellt werden muss.

4.3.3. Umsetzung des Rootelement Prinzips

Die Umsetzung des Rootelements basiert auf dem zuvor genannten Prinzip der UXMLFactories. Eine solche Factory wurde für das Rootelement angelegt. Während der Initialisierung eines jeden Rootelements, wird die in Listing 4.7 gezeigte Methode ausgeführt. Hierbei wird zunächst festgelegt, dass nach Auslösen des eingetragenen statischen Events jene Methode erneut ausgeführt wird. Dies ermöglicht die Änderung des Farbschemas ohne die Notwendigkeit, die Oberfläche neu zu öffnen. Anschließend wird, sofern vorhanden, die bestehende Design-Klasse des Rootelements entfernt und durch eine neue ersetzt.

In Ausnahmefällen, in welchen die Umsetzung der Oberfläche nur durch das IMGUI System stattfinden kann, ist die Verwendung der USS Klassen und damit des Designs der restlichen Oberflächen ist nicht möglich. Um dennoch die Farbwerte der USS Datei auslesen zu können werden diese in einer statischen Klasse gespeichert. Durch das nach Ausführen des Events, am Ende der in Listing 4.7 gezeigten Methode, werden jene Werte abgespeichert. Da ein direktes Auslesen der Werte aus der USS Datei nicht unterstützt wird, wird nach jeder Änderung des Farbschemas einmalig für jede relevante Farbe ein visuelles Element erstellt, welches die zur Farbe korrespondierende Klasse besitzt. Aus diesem Element wird nun der Farbwert ausgelesen und im Cache abgespeichert. Diese abgespeicherten Farben können dann in den IMGUI Oberflächen zur Erstellung der Elemente verwendet werden. Um die USS Klassen innerhalb der Skripts verwenden zu können, benötigt es deren Namen. Daher wurde zusätzlich ein Skript angelegt, in welchem alle USS Klassen mit deren Namen abgespeichert werden. Ein Beispiel der Verwendung dieser Klasse ist in Listing 4.7 in den Zeilen Neun und Zehn zu erkennen.

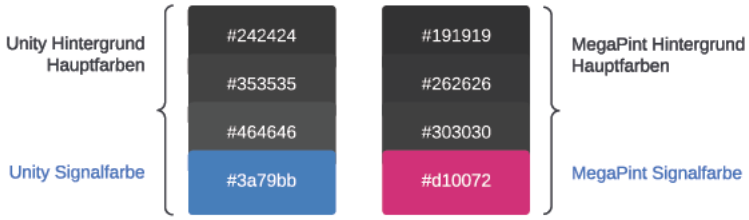

Abbildung 4.4.: Vergleich der Farbschemas von Unity und der MegaPint Kollektion

Das durch das Rootelement durchgesetzte Farbschema basiert jeweils stark auf dem Äquivalent des Unity Editors. So ist in Abbildung 4.4 der Vergleich zwischen den beiden dunklen Schemata abgebildet. Dort lässt sich deutlich erkennen, dass die Hintergrundfarben ebenfalls wie bei Unity aus mehreren dezenten Graustufen bestehen. Jede Hintergrundfarbe der MegaPint Kollektion ist dunkler als sein Unity Äquivalent. Dies ermöglicht dem Nutzer die leichte Differenzierung zwischen den Oberflächen des Editors und

der entwickelten Tools. Um dem Nutzer eine weitere offensichtliche Differenzierung bereitzustellen, wird die Signalfarbe ausgetauscht. Diese Farbe findet sich in allen interaktiven Elementen sowie Auswahlen des Nutzers. Durch das Austauschen der Signalfarbe wird beispielsweise bei der Verwendung eines Buttons eindeutig, dass es sich hierbei um eine zusätzliche, durch ein Tool zur Verfügung gestellte, Funktion handelt.

```
1  public static void ApplyRootElementTheme(this VisualElement element,
       Action <VisualElement> onRepaintAction = null, bool subscribeToEvent
       = true)
2  {
3      if (subscribeToEvent)
4          s_onForceRepaint += () => ApplyRootElementTheme(element,
               onRepaintAction, false);
5
6      element.RemoveFromClassList(StyleSheetClasses.Theme.Dark);
7      element.RemoveFromClassList(StyleSheetClasses.Theme.Light);
8
9      element.AddToClassList("mp");
10     element.AddToClassList(StyleSheetClasses.Theme.Current);
11
12     onRepaintAction?.Invoke(element);
13 }
```

Listing 4.7: Anwendung des Rootelement

Ebenfalls wird durch die Umsetzung des Rootelements eine eigene Implementation für Tooltips möglich. Hierbei werden allen Elementen, welche den eigenen Tooltip implementieren sollen über den UI Builder eine Klasse zugewiesen. Bei der Initialisierung des Rootelements führt dies für all seine Kinder-Elemente mit der definierten Klasse die notwendigen Vorbereitungen durch. Hierdurch ist es nicht notwendig, eigene Factories für jedes betroffene Element zu erstellen. Jene Initialisierung besteht daraus, Callback Methoden für das “PointerEnterEvent“ und das “PointerLeaveEvent“ zu erstellen. In diesen wird eine Methode zum Anzeigen oder Verstecken der eigens dafür erstellten Oberfläche aufgerufen. Während der Tooltip angezeigt wird, reagiert dieser zudem auf die Änderungen der Mausposition, welche ebenfalls als Callback des Elements zurückgegeben wird.

4.3.4. IMGUI Ausnahmen

Wie bereits im vorangehenden Kapitel besagt, bestehen einige Ausnahmen, in welchen das IMGUI System zur Implementation der Oberflächen verwendet wurde. Bei diesen Ausnahmen handelt es sich um Oberflächen, in welchen die Verwendung des angestrebten Systems nicht möglich ist. Die in den implementierten Tools vorkommenden Ausnahmen sind größtenteils “PropertyDrawers“. Dies sind Klassen, mit welchen man das Aussehen einer serialisierten Klasse innerhalb des Inspectors verändern kann. Da die Anwendung der PropertyDrawer Klassen in diesem Fall innerhalb eines IMGUI Kontextes stattfindet, kann das UI Elements System, aufgrund seiner Inkompatibilität,

nicht verwendet werden. Neben den PropertyDrawers besteht das eben genannte Problem zudem bei Klassen, welche von "UnityEditor.Editor" erben und die "OnInspectorGUI" Methode implementieren. Da diese Methode ebenfalls in einem IMGUI Kontext agiert, muss hier dessen API verwendet werden. Ein Beispiel für die Umsetzung einer Oberfläche mittels IMGUI lässt sich in Listing A.10 finden.

4.4. Lösung der repetitiven Aufgaben

4.4.1. Automatisches Speichern

Die Umsetzung der grafischen Oberfläche besteht wie im Konzept erarbeitet und in Abbildung 4.5 gezeigt aus zwei Teilen. Zunächst wurde ein neues Editor-Fenster erstellt, welches alle Informationen betreffend der aktiven Funktion enthalten. Durch das Betätigen der Start sowie Stopp Schaltfläche innerhalb des Editor-Fensters ändert sich auch der Status der in Unitys Toolbar befindlichen Schaltfläche.

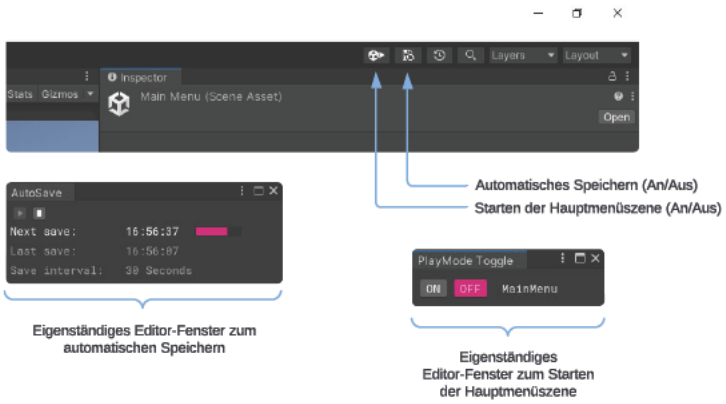

Abbildung 4.5.: Benutzeroberflächen zum automatischen Speichern und Starten der Hauptmenü Szene

Das Hinzufügen von Elementen in Unitys Toolbar ist standardmäßig nicht möglich. Realisiert wird dies durch die Verwendung von Reflection. Die API der Toolbar ist versiegelt. Jedoch enthält die interne Klasse der Toolbar eine Variable mit dem Namen "m_root". Durch Reflection wird diese, wie in Listing 4.8 gezeigt, ausgelesen. Mit Erhalt dieses visuellen Elements lassen sich alle Elemente innerhalb der Toolbar bearbeiten, entfernen und es ist möglich neue Elemente hinzuzufügen.

```
1 Type toolbarType = typeof(UnityEditor.Editor).Assembly.GetType("
      UnityEditor.Toolbar");
2 object toolbar = Resources.FindObjectsOfTypeAll(toolbarType)[0];
3
4 FieldInfo root = toolbar.GetType().GetField("m_Root", BindingFlags.
      NonPublic | BindingFlags.Instance);
```

Listing 4.8: Erhalt des visuellen Elements der Unity Toolbar

Die technische Umsetzung der Funktion beruht auf einer asynchronen Methode welche als Timer agiert. Nach Ablauf des festgelegten Intervalls wird die Speicher Methode der “EditorSceneManager“ Klasse aufgerufen. Da das Speichern einer Szene nur im Kontext des Unity Editors möglich ist, führt das Aufrufen der Methode während des Playmodes zu Problemen. Um diese zu umgehen, wird das Speichern während des Playmodes pausiert. Nach Beendigung dessen, wird der Timer sofern er zuvor lief, neu gestartet. Zudem wird der Zustand des Timers (An/Aus) in den Einstellungen gespeichert, sodass bei Neustart des Editors oder einem Domain Reload, unter Verwendung des “InitializeOnLoad“ Attributs der Timer in seinen ursprünglichen Zustand zurückkehrt.

4.4.2. Automatisches Starten der Hauptmenü Szene

Wie in der Konzeption des Tools beschrieben und in der Abbildung 4.5 gezeigt, besteht die grafische Umsetzung aus zwei Teilen. Wie schon zuvor bei der Implementierung des Tools zum automatischen Speichern besitzt dieses Tool eine eigene Oberfläche, welche zur Bedienung und dem Anzeigen weiterer Informationen dient. Die Umsetzung der Schaltfläche innerhalb Unitys Toolbar entspricht der des vorangegangenen Kapitels.

Die technische Implementation der Funktion beruht auf bereits vorhandenen Funktionen der Unity API. Die Klasse “EditorSceneManager“ enthält eine Variable mit dem Namen “playModeStartScene“. Durch das Zuweisen jener Variable mit dem gewünschten Szenen Asset, startet der Playmode in der ausgewählten Szene. Durch das Zuweisen des Wertes “null“ lässt sich die Einstellung aufheben und der Playmode startet wie gewohnt in der geöffneten Szene.

4.4.3. Fehlerkorrektur von Assets

Die Erstellung eines Regelsatzes sowie die Behebung aufgetretener Fehler erfolgt zunächst wie in Abbildung 4.6 gezeigt, über den Inspector des Objektes. Jede von “ValidatableMonoBehaviour“ (fortan mit VMB abgekürzt) erbende Klasse erzeugt automatisch eine Status Komponente auf dem Objekt. Hierbei handelt es sich um eine Komponente, welche den aktuellen Status aller VMBs des Objektes zusammenfasst und die auftretenden Fehler anzeigt. Zusätzlich enthält die Oberfläche neben dem Listenelement weitere Schaltflächen, über welche die auftretenden Fehler behoben werden. Die Umsetzung der Oberfläche eines jeden VMB erfolgt über die Verwendung eines eigenen “CustomEditor“. Jener Editor zeichnet zunächst die geforderten Elemente eines VMB. Anschließend werden mittels der in Listing 4.9 gezeigten Zeilen die verbleibenden serialisierten Elemente der Klasse gezeichnet. Somit ist es möglich, trotz des Überschreiben des Inspectors, in der zu validierenden Klasse Variablen zu serialisieren und im Inspector zuzuweisen.

```
1 private static readonly string[] s_exclusion = {"m_Script", "
      _importedSettings"};
2
3 DrawPropertiesExcluding(serializedObject, s_exclusion);
```

Listing 4.9: Anzeigen des ursprünglichen Inspectors

Die Erstellung eines “Single Point Of Entry“ für die Fehlerbehebung wird über die geschaffene Oberfläche, welche in Abbildung 4.7 zu erkennen ist, geschaffen. Dieser sammelt, basierend auf den Einstellungen des Nutzers, alle Objekte mit Status-Skripten. Anschließend werden die aufgetretenen Fehler sowie der allgemeine Status des Objektes ausgelesen. Anhand dessen werden die Objekte in die Kategorien “Ok“, “Warning“ und “Error“ eingeteilt. Durch den Gebrauch mehrerer Listen in der Oberfläche wird es ermöglicht, jedes einzelne gefundene Objekt der jeweiligen Kategorie auszuwählen. Hierbei wird die Suche nach bestehender VMBs einmalig ausgeführt. Die gefundenen Daten werden gespeichert und erst auf Anweisung des Nutzers erneuert. Um Statusänderungen der Objekte trotzdem dauerhaft zu erkennen, werden Callback Methoden für die einzelnen Elemente erstellt. Bei der Statusänderung eines Objektes wird lediglich dieses Element in die korrespondierende Kategorie verschoben. Durch die Auswahl der Elemente in einer der Listen erhält der Nutzer, vollen Zugriff auf die bekannten Funktionen des VMBs. Um die weitere Fehlerbehebung zu vereinfachen, wird durch zweifaches Auswählen eines Elements, dieses im Inspector ausgewählt.

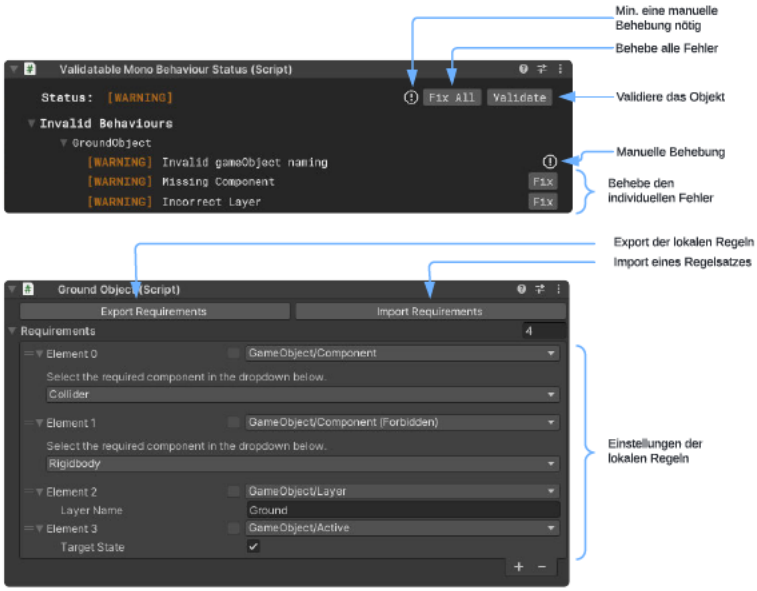

Abbildung 4.6.: Fehlerbehebung über die Komponenten des Objektes

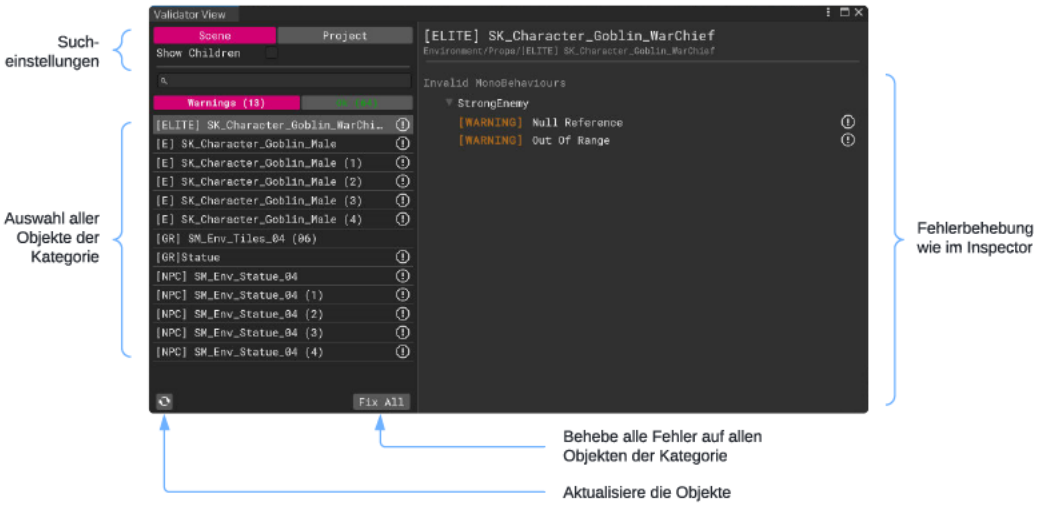

Abbildung 4.7.: Fehlerbehebung über das Editor-Fenster “Validator View“

Die technische Implementation der Validatoren basiert auf der “OnValidate“ Methode eines Game Objects. Innerhalb jener Methode werden die Werte und Gegebenheiten des Objektes überprüft. Um nicht für jedes Objekt eine neue OnValidate Methode schreiben zu müssen, wird die Definition des Regelwerkes auf die Verwendung dafür vorgefertigter Skripts ausgelagert. Hierdurch muss die Regel einmalig in einem Skript niedergeschrieben sein. Nachträglich kann diese in jedem VMB verwendet werden. Um eine solche Regel zu erstellen, muss eine neues Skript erstellt werden, dessen Klasse von “ScriptableValidationRequirement“ erbt. Jene Klasse fordert die Implementation einer Methode zur Validierung sowie der Initialisierung. In der Initialisierung werden den möglichen Einstellungen der Regel Grundwerte zugewiesen. Der Inhalt der Validierungsmethode ist von der Umsetzung sowie der zu validierenden Sachlage abhängig. Wurde ein Fehler gefunden, so ist dieser mittels der in Listing 4.10 gezeigten Methode dem Kontext der Validierung hinzuzufügen. Jeder hinzugefügte Fehler wird nach Validierung des Objektes in dessen Status angezeigt. Der Fehler mit dem höchsten Status prägt hierbei den Status des gesamten Objektes. Wird bei der Übergabe des gefundenen Fehlers eine “FixAction“ Methode übergeben, so wird diese bei automatischer Behebung des Fehlers ausgeführt.

```
1 AddError("Incorrect␣active␣state", errorMessage, ValidationState.Warning
      , FixAction);
```

Listing 4.10: Hinzufügen eines Fehlers

Um eine erstellte Regel im Inspector des VMB auswählen zu können, muss diese vorgegebene Attribute besitzen. Diese Attribute werden bei der Erstellung des Dropdowns zur Auswahl der Regel ausgelesen. Zu den, in den Attributen geforderten, Daten gehören zunächst ein optionaler Tooltip und der Anzeigename der Regel. Zusätzlich dazu muss der Type der Klasse, in welchem die Regel niedergeschrieben ist mitgegeben werden. Dieser Type dient zum Vergleich mit anderen bereits vorhandenen Regeln. Hierdurch kann sichergestellt werden, dass keine inkompatiblen Regeln auf dem gleichen Objekt vorhanden sind. Zuletzt fordern die Attribute Angaben zur Anzeige-Reihenfolge der Regel im Dropdown, eine Liste inkompatibler Regeln sowie, ob die Regel mehrfach auf ein Objekt angewandt werden darf.

Aufbauend auf der Liste der lokalen Regeln besteht die Möglichkeit externe Regelsätze zu importieren. Diese werden in Scriptable Objects gespeichert. Bei Import eines solchen, wird dessen Referenz gespeichert, wodurch bei der Validierung des Objektes jene Regeln ebenfalls berücksichtigt werden. Um die Anwendung inkompatibler Regeln nach der Nutzung externer Regelsätze zu unterbinden, wurde ein Prioritätssystem implementiert. Die in der lokalen Liste definierten Regeln haben immer die höchste Priorität. Anschließend kann der Nutzer die Priorität der importierten Regelsätze beliebig wählen. Bei der Validierung des Objektes werden nur Regeln angewandt, welche mit allen Regeln höherer Priorität kompatibel sind.

4.4.4. Vereinfachte Bildschirmaufnahmen

Das Rendern eines Editor-Fensters wird zunächst innerhalb eines eigenen Editor-Fensters, wie in Abbildung 4.9 gezeigt, realisiert. Bei Betätigen der Render-Schaltfläche wird das Editor-Fenster

des im Dropdown gewählten Typen ausgelesen. Hierbei wird die Breite, Höhe sowie die Position des Fensters ausgelesen. Anschließend werden ab der gegebenen Position alle Pixel innerhalb der gegebenen Breite und Höhe ausgelesen und in einer Textur gespeichert, welche anschließend als Vorschau angezeigt wird. Hierbei kann es vorkommen, dass das gewünschte Editor-Fenster sich im Hintergrund befindet. Um nicht die Fenster im Vordergrund abzubilden, wird mittels der dafür vorgegeben Methode das Fenster vor dem Auslesen der Pixel fokussiert.

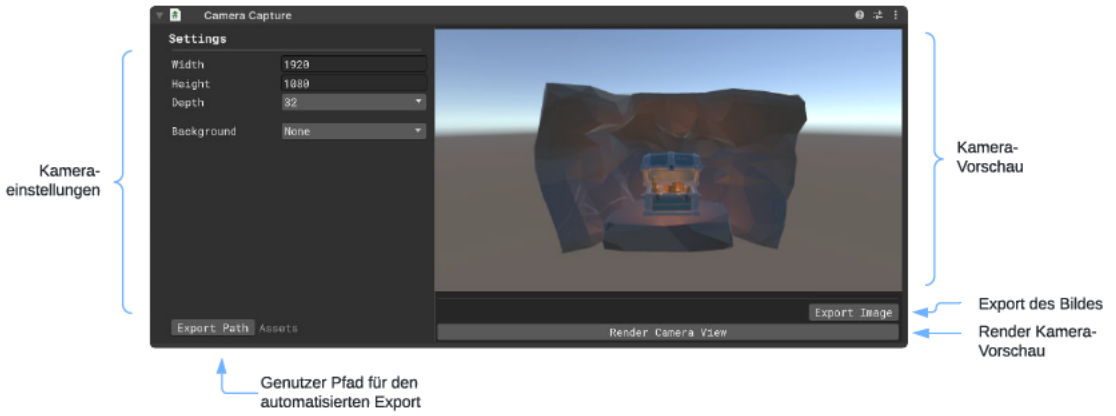

Abbildung 4.8.: Rendern einer Kamera Komponente

Das Rendern einer Kamera Komponente findet, wie in Abbildung 4.8 gezeigt, innerhalb einer weiteren Komponente auf dem Objekt statt. Der Render Prozess besteht hierbei maßgeblich aus drei Teilen. Zunächst wird nach Betätigen der Render-Schaltfläche die Kamera auf das Rendern vorbereitet. Dazu gehört unter anderem die Anwendung der getätigten Einstellungen innerhalb der Kamera Komponente, wie beispielsweise der Hintergrundtyp oder die Hintergrundfarbe. Hierbei werden die ursprünglichen Einstellungen abgespeichert. Nach Abschluss der Vorbereitungen wird ein Bild der Kameraansicht erstellt. Hierzu wird die Kamera, wie in Listing A.6 beschrieben, angewiesen einen einzelnen Frame auf eine Textur auszugeben, welche als Output der Komponente zugewiesen wurde. Jene Textur wird anschließend als Ergebnis des Prozesses an die Oberfläche zurückgegeben, wo diese als Vorschau angezeigt wird. Nach

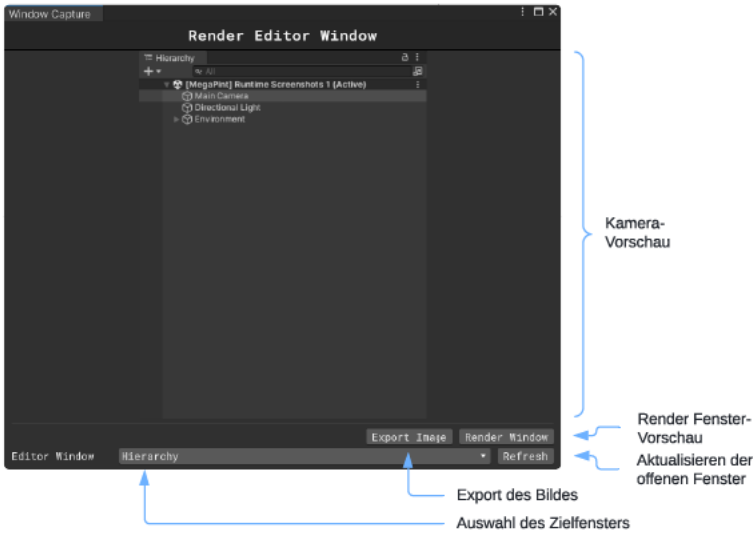

Abbildung 4.9.: Rendern eines Editor-Fensters

dem Prozess des Renderns, wird die Kamera zurückgesetzt, indem ihr die gespeicherten Werte zugewiesen werden.

Die soeben genannte Implementation agiert fehlerfrei unter der Verwendung der Lightweight Render Pipeline, welche in der genutzten Version die Standard-Pipeline der Engine ist. Unter Verwendung der URP (Universal Render Pipeline) treten ebenfalls keine Probleme auf. Jedoch verlieren, unter Verwendung von Postprocessing, die erstellten Texturen jegliche Transparenz. Dieses Problem basiert auf einer Änderung innerhalb eines Shaders, welcher erst bei der Nutzung von Postprocessing verwendet wird. In diesem wird die Alpha der ausgegebenen Textur immer auf den Wert 1 gesetzt, womit keine Transparenz zustande kommt. Um dieses Problem zu beheben, wurde eine weitere Oberfläche implementiert. Dies ist ein Setup, welches nach Ausführung die Nutzung von Transparenz erlaubt.

Bei dessen Ausführung wird zunächst eine Kopie des genutzten “UniversalRendererData“ Asset gemacht. Anschließend wird eine Kopie des ursprünglichen “PostProcessingData“ Assets erstellt und in die Kopie der ersten Datei eingefügt. In dem neuen “PostProcessingData“ Asset wird der betroffene Shader mit einer überarbeiteten Version dessen ausgetauscht. Abschließend wird das erstellte “UniversalRendererData“ Asset mit seiner GUID in das “RenderPipelineAsset“ des Projektes eingefügt. Hierbei wird die ID des Assets abgespeichert. Sobald die ID eines betreffenden Assets in den Einstellungen der Kollektion vorhanden ist, wird bei der Vorbereitung der Kamera, der Renderer, in dessen “UniversalAdditionalCameraData“ Komponente, mit dem gespeicherten ausgetauscht. Hierdurch nutzt die Kamera nur während der Erstellung einer Aufnahme den abgeänderten Shader.

Ebenfalls unter Verwendung der HDRP (High Definition Render Pipeline) tritt ein Problem spezifisch zu jener Pipeline auf. Aufgrund der realistischen Lichtgebung der HDRP sind abgelichtete Objekte meist sehr dunkel oder hell. Dies ist auf die Überbelichtung oder Unterbelichtung der Aufnahme zurückzuführen, welche dadurch entsteht, dass beim Rendern der Aufnahme nur ein Frame zur Verfügung steht. Um diesem Problem entgegen zu wirken, kann der Nutzer eine Belichtungszeit festlegen. Bei Betätigung der Render-Schaltfläche wird nun zunächst die Belichtungszeit abgewartet, bevor die Aufnahme getätigt wird.

4.5. Evaluations Package

4.5.1. Validierung des Tester Tokens

Da das Evaluations Package als gewöhnliches Package der MegaPint Kollektion implementiert wurde, jedoch nur für Probanden der Evaluation zugänglich sein soll, müssen zunächst zwei Gegebenheiten berücksichtigt werden. Zunächst darf das Package, für nicht zugelassene Nutzer, nicht über den internen Package Manager importierbar sein. Zudem müssen die Funktionen des Packages, nach einem manuellen Import unter Verwendung der Git Adresse, ebenfalls nicht verwendbar sein. Um dies zu realisieren, wurde allen Probanden ein Token übergeben, welchen sie im Hauptfenster der Kollektion eintragen müssen. Nach erfolgreicher Validierung des Tokens können alle Funktionen rundum das Evaluations Package verwendet werden.

Die Validierung des Tokens erfolgt über ein, auf einer Website liegendes, PHP Skript. Jenes Skript überprüft ob der mitgegebene Token in der Datenbank hinterlegt ist. Ist dies der Fall, so gibt das Skript eine positive Bestätigung zurück. Die Implementation auf Seiten der Unity Engine nutzt die "UnityWebRequest" Klasse. Mit den in Listing 4.11 gezeigten Zeilen, wird eine Anfrage an das PHP Skript gestellt. Anschließend wird in einem asynchronen Kontext auf den Rückgabewert der Anfrage gewartet. Jener Rückgabewert gibt anschließend an, ob der Token als gültig validiert wurde.

```
1 UnityWebRequest request = UnityWebRequest.Get($"{ValidateTokenURL}?token
      =UnityWebRequest.EscapeURL(SaveValues.BasePackage.TesterToken)}");
2
3 request.SendWebRequest();
```

Listing 4.11: Token Validierungs Anfrage

Da das Stellen der Anfrage und Warten auf die Rückmeldung einige Millisekunden in Anspruch nimmt, sollte dies nicht häufig ausgeführt werden. Aus diesem Grund wird die Überprüfung einmalig nach jedem Domain Reload durchgeführt und der Wert anschließend abgespeichert. Das Speichern innerhalb der MegaPint Einstellungen ist nicht möglich, da diese unter Verwendung des Debug Modus bearbeitetet werden können.

4.5.2. Speichern des Fortschrittes

Um den Fortschritt innerhalb der Evaluationsaufgaben sowie die benötigten Zeiten und Eingaben des Probanden festzuhalten, müssen diese Angaben abgesichert werden. Da die erstellten Aufgaben in Form von Scriptable Objects in den Dateien des Packages liegen, wurde zunächst eine sehr simple Speichermethode implementiert. Hierbei wurden die benötigten Zeiten sowie Angaben über den Fortschritt innerhalb der Aufgabe direkt in den Objekten gesichert. Nach erstmaligem Testen außerhalb des Projektes zur Erstellung der Packages, wurde jedoch bekannt, dass dies nicht möglich ist. Durch das Importieren der Packages und das Einbetten in das Packages Verzeichnis, werden die darin befindlichen Daten bei Neuerstellung des Verzeichnisses zurückgesetzt. Ebenfalls ist es anschließend nicht mehr möglich, die im Package befindlichen Scriptable Objects zu bearbeiten, was die Zuweisung des Fortschritts unmöglich macht.

Um dieses Problem zu umgehen, werden die notwendigen Daten innerhalb der MegaPint Einstellungen gespeichert. Da dieses Asset innerhalb des Projektes liegt, bestehen keinerlei Restriktionen. Weil der Nutzer unter Verwendung des Debug Modus die Werte abändern kann, werden die Schlüssel derer durch die GUIDs der korrespondierenden Scriptable Objects ersetzt. Hierdurch werden die Einstellungen für den Nutzer unleserlich, was die Bearbeitung erschwert.

4.5.3. Aufzeichnung der benötigten Daten

Die Daten über die Nutzung von MegaPint sowie Unity Funktionen, welche für die Auswertung der freien und geführten Evaluation relevant sind, werden innerhalb des Unity Editors erfasst. Hierzu werden durch die Verwendung der InitializeOnLoad Methode, nach jedem Domain Reload, Callback Methoden für alle relevanten Funktionen erstellt. Nach Ausführen einer der überwachten Methoden wird dessen Verwendung mit einem Zeitstempel in einer Log Klasse abgespeichert. Nach einer festgelegten Anzahl an Logs sowie vor dem Beenden des Editors wird die Log Klasse serialisiert und lokal als JSON Datei abgespeichert.

Unter Verwendung der Unity API ist es möglich, ein Callback für das Schließen des Editors zu implementieren. Diese Methode erfordert die Rückgabe eines Booleans. Jener Boolean gibt an, ob nach Ausführung der Methode der Editor geschlossen werden darf. Sobald eine der implementierten Callback Methoden das Schließen untersagt, wird dies gestoppt. Auf Grundlage dieser Methode wird ein automatischer Upload der aufgezeichneten Daten implementiert. Bei erstmaligem Auslösen der Methode wird überprüft, ob lokal abgespeicherte Log Dateien vorhanden sind. Ist dies der Fall, wird das Schließen abgebrochen und ein Upload durchgeführt. Jener Upload basiert erneut auf einem, auf einer Website liegendem, PHP Skript, welches als Übergabewert einen Text akzeptiert. Durch erneute Verwendung der UnityWebRequest Klasse wird eine Anfrage an das Skript gestellt, in welcher der Text der gefundenen Dateien enthalten ist. Sollte der Upload aufgrund eines Fehlers oder fehlenden Internetzugriffs fehlschlagen, verbleiben die Dateien lokal und werden nicht gelöscht. Nach dem Upload wird dem Editor erneut der Befehl zum Schließen gegeben, welcher nun nicht von der Callback Methode gestoppt wird.

5. Evaluation

Das nun folgende Kapitel umfasst die Auswertung der durchgeführten Evaluation. Die Anzahl der Probanden der Evaluation beläuft sich auf zwölf Personen. Drei der Probanden besitzen, nach eigenen Angaben, wenig Vorerfahrung im Umgang mit der Unity Engine. Zudem hat einer der Probanden Unity zuvor noch überhaupt nicht verwendet. Die verbleibenden neun Probanden haben nach eigenen Angaben ausreichend bis viel Erfahrung mit der Handhabung der Engine.

Aufgrund der geringen Probandenzahl und der geringen Vielfältigkeit der Teilnehmenden, sind die Ergebnisse der Evaluation nicht auf die breite Nutzervielfalt der Unity Engine anwendbar. Ebenfalls kann bei den Probanden nicht von einer Expertengruppe gesprochen werden, da der überwiegende Teil die eigenen Fähigkeiten nicht als solche einschätzt. Daher ist zu berücksichtigen, dass die in den folgenden Absätzen erarbeiteten Aussagen für die vorliegende Zielgruppe erarbeitet wurde. Jene beinhaltet Personen, welche aktiv mit der Unity Engine arbeiten und daher bereits Erfahrungen mit dieser besitzen, sich jedoch nicht als Experte einschätzen.

5.1. Auswertung aufgezeichneter Daten

Wie bereits in der Konzeption der Evaluation beschrieben, beinhaltete sowohl die geführte als auch die freie Evaluation die Aufzeichnung von Messdaten. In den folgenden Abschnitten werden zunächst die erhobenen Daten ausgewertet. Hierbei werden die Daten zunächst anonym behandelt um Aussagen über die Allgemeinheit der Probanden treffen zu können. Anschließend an die Auswertung der Umfragen, werden die Messdaten mit den Kenntnissen der Probanden interpretiert.

5.1.1. Geführte Evaluation

Um die Steigerung der Effizienz messbar nachweisen zu können, wurden bei der Erfüllung der Aufgaben jeweils die benötigte Zeit sowie die benötigten Tastatur- und Mauseingaben gemessen. Für die zwei, in der geführten Evaluation getesteten Funktionen, wurden jeweils drei Aufgaben erstellt. Die erste Aufgabe umfasst immer die Ausführung der Tätigkeit ohne die Nutzung des Tools. Die anfolgenden Aufgabe hingegen fordern die Verwendung des entwickelten Tools. Hieraus ergibt sich die Möglichkeit eines Vergleichs zwischen ohne und mit Nutzung der entwickelten Funktionen.

Vereinfachte Bildschirmaufnahmen
Die erste Aufgabe der Evaluation des Tools für Screenshots beinhaltet die Erstellung von vier Aufnahmen mit einem beliebigen Werkzeug. Unter Verwendung des gewählten Werkzeugs mussten die Probanden Aufnahmen der gegebenen Stillleben erstellen, welche nacheinander durch eine sich

rotierende Kamera gezeigt wurden. Die Schwierigkeit hierbei bestand darin, die Stillleben möglichst zentriert aufzunehmen. Die zweite Aufgabe der Evaluation ist grundlegend identisch, jedoch wurden die Funktionen des Tools verwendet. Die letzte Aufgabe sah vor, die gleichen vier Stillleben mit vier unterschiedlichen Hintergrundeinstellungen aufzunehmen.

Vergleicht man die aufgezeichneten Daten in Abbildung 5.1, so ist zu erkennen, dass die benötigte Zeit zur Bearbeitung einer Aufgabe mit gleichbleibenden Aufwand in den meisten Fällen durch die Verwendung des Tools abnimmt. Im Schnitt wird die nötige Zeit der Probanden um bis zu zehn Sekunden reduziert, wie sich aus Abbildung 5.2 erlesen lässt. Die größte erzielte Zeitersparnis gegenüber der manuellen Erstellung beträgt 124 Sekunden. Vier weitere Probanden erzielten eine Steigerung um mindestens eine Minute unter Verwendung des entwickelten Tools. Entgegen dieser positiven Ergebnisse waren drei Probanden erheblich langsamer im Vergleich zur ersten Aufgabe. Dies ist wahrscheinlich auf die Nutzung eines ihnen unbekannten Tools zurückzuführen und muss bei der Auswertung der Fragebögen berücksichtigt werden.

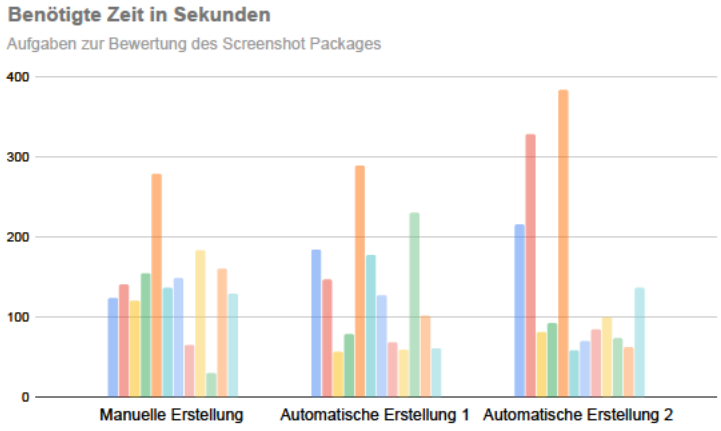

Abbildung 5.1.: Benötigte Zeit in Sekunden (Screenshots)

Zusätzlich zu der benötigten Zeit konnten ebenfalls die notwendigen Nutzereingaben, wie in Abbildung 5.3 zu erkennen ist, erheblich reduziert werden. Die größte gemessene Reduzierung liegt bei 105 Eingaben. Weitere drei Probanden reduzierten ihre Eingaben um mehr als 50. Wie in Abbildung 5.2 erkennbar, wurde eine durchschnittliche Verbesserung von 20 Eingaben erreicht. Die Probanden, welche sich bereits zeitlich verschlechtert hatten, benötigten ebenfalls mehr Eingaben.

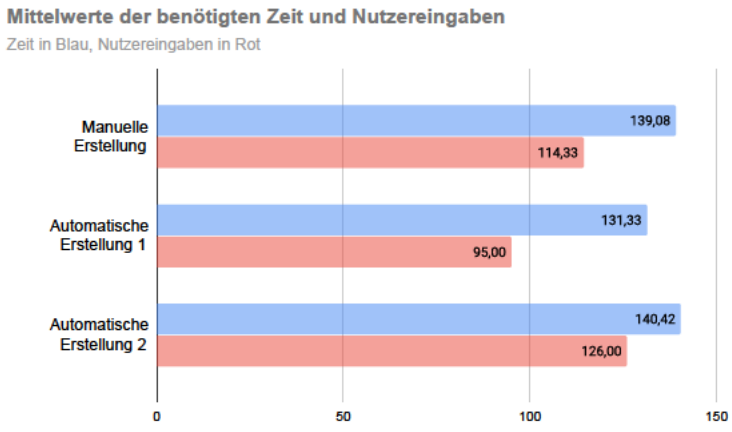

Abbildung 5.2.: Vergleich des Nutzeraufwands (Screenshots)

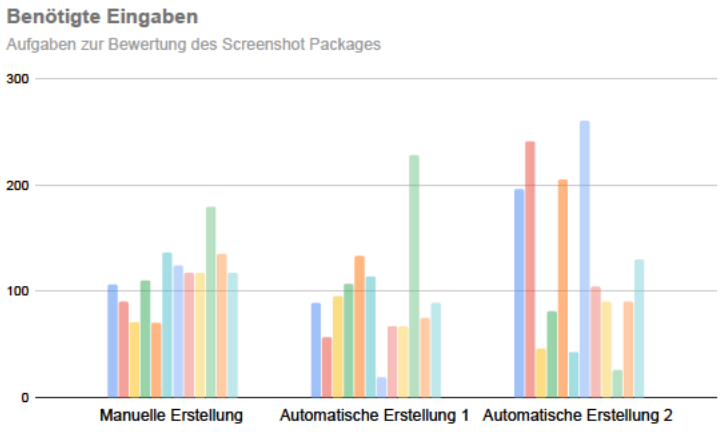

Abbildung 5.3.: Benötigte Nutzereingaben (Screenshots)

Die dritte Aufgabe der Evaluation erforderte, anders als die erste und zweite Aufgabe, die Erstellung von 16 Aufnahmen. Aufgrund der Notwendigkeit zur Erstellung von drei weiteren Kameras sowie der Aufforderung zur kreativen Einstellung derer, ist ein genereller zeitlicher Zuwachs sowie ein starker Anstieg in den Nutzereingaben zu verzeichnen. Trotz dieser Anstiege konnten die Probanden, unter dem zusätzlichen Aufwand von durchschnittlich 30 Eingaben und zehn Sekunden, das vierfache Ergebnis erzielen. Jener zusätzliche Aufwand ist hierbei nur einmalig notwendig, um die Kameraobjekte zu erschaffen und einzustellen. Nach der initialen Erstellung wird lediglich ein vergleichbarer Aufwand wie in Aufgabe zwei benötigt.

Mehr als die Hälfte der Probanden konnte ohne Vorwissen über das getestete Tool ihre Effizienz hinsichtlich der benötigten Zeit sowie der Eingaben verbessern. Mit wachsender Vertrautheit in die gegebenen Funktionen ist zudem eine starke Effizienzsteigerung möglich, wie einige der Probanden erwiesen haben. Auf Grundlage dieser Daten, kann das getestete Tool zur vereinfachten Bildschirmaufnahme als effizienzsteigernd eingestuft werden.

Fehlerkorrektur von Assets

Die Evaluation des Tools zur automatischen Fehlerkorrektur von Assets beginnt erneut mit der manuellen Bearbeitung ohne Verwendung der entwickelten Funktionen. Hierbei werden die Probanden aufgefordert, eine Szene zu reparieren. Jene Szene enthält eine Vielzahl an Assets verschiedener Kategorien, welche bestimmte Regeln befolgen müssen. So beispielsweise ist gefordert, dass alle Elemente des Bodens, namentlich mit “[GR]“ gekennzeichnet, keinen Rigidbody besitzen dürfen, eine Collider Komponente besitzen müssen und dem Layer “Ground“ zugewiesen sind. Die geforderten Regeln wurden den Probanden durch die, in Abbildung B.1 gezeigten Tabelle bereitgestellt. Bei Starten des Playmodes erscheint nach kurzer Zeit die Bestätigung über Erfolg oder Fehlschlag der Aufgabe. Bei einem Fehlschlag werden dem Nutzer die fehlerhaften Objekte innerhalb der Konsole mit den aufgetretenen Problemen angezeigt. Anschließend ist es an dem Nutzer, diese Probleme zu beheben. Im Kontrast zur ersten Aufgabe nutzen die Probanden in der zweiten Aufgabe das erstellte Tool zur Fehlerkorrektur. Zudem hat die Szene einen erhöhten Umfang von großzügig anderthalbfacher Menge der vorangegangenen Aufgabe. Für die Lösung dieser Aufgabe wurden bereits alle notwendigen Einstellungen auf den Objekten getätigt, so dass alle auftretende Fehler erkannt werden.

Die dritte Aufgabe umfasst den gleichen Umfang der zweiten Aufgabe. Jedoch wurden in Jener keine Vorbereitungen getroffen. Um den vollen Aufwand zu messen, wird hier neben dem in Aufgabe

zwei gemessenen Aufwand für die Anwendung des Tools, der Aufwand für die Vorbereitung gemessen. Da der Nutzer durch das Tool in der Lage sein soll, eigene Regeln zu erstellen, wird zudem in Aufgabe vier der Aufwand zur Erstellung einer Regel per Skript gemessen. Der Proband soll eine Regel erschaffen, welche den “useGravity“ Boolean einer Rigidbody Komponente überprüft.

Vergleicht man zunächst die in Abbildung 5.4 aufgezeichneten Daten, lässt sich eine deutliche Reduzierung der benötigten Zeit zwischen der Aufgabe eins und zwei feststellen. Die höchste erzielte Reduzierung beträgt über zehn Minuten. Im Kontext aller Probanden lässt sich, wie in Abbildung 5.5 gezeigt, belegen, dass die Anwendung der vorbereiteten Regeln, trotz des erhöhten Aufwands die notwendige Zeit auf ein Drittel reduzierte.

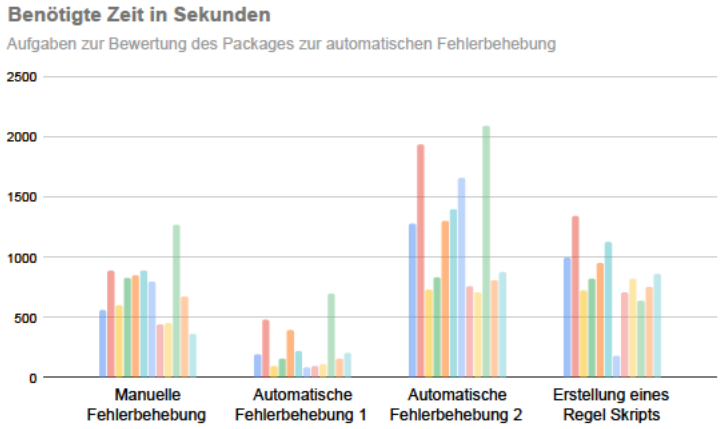

Abbildung 5.4.: Benötigte Zeit in Sekunden (Fehlerkorrekturen)

In Abbildung 5.6 ist zu erkennen, dass ebenfalls die notwendigen Nutzereingaben stark verringert wurden. Lediglich ein Proband verschlechterte sich hinsichtlich der benötigten Eingaben. Der Proband mit der höchsten Ersparnis konnte seine Eingaben um über 600 Stück senken. Den verbleibenden Probanden gelang es ebenfalls, jene Eingaben um meist die Hälfte zu reduzieren, wie sich in Abbildung 5.5 erkennen lässt.

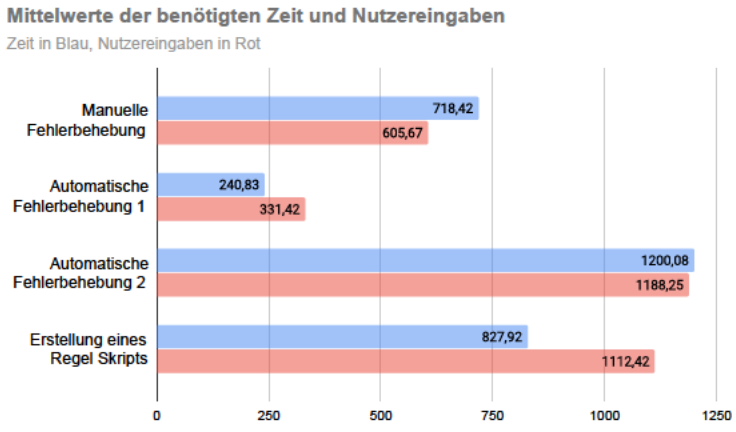

Abbildung 5.5.: Vergleich Nutzeraufwand (Fehlerkorrekturen)

Bei der Bearbeitung der Aufgabe drei hingegen lässt sich, wie in Abbildung 5.4 und 5.6 gezeigt, ein starker Anstieg in sowohl der benötigten Zeit sowie den Nutzereingaben feststellen. Da in dieser Aufgabe die Vorbereitung der Szene sowie die Anwendung nach dem Prinzip der zweiten Aufgabe gefordert ist, muss dies berücksichtigt werden. Selbst durch einen Abzug der in Aufgabe zwei gemessenen Werte bleibt ein starker Anstieg in den gemessenen Datensätzen erhalten. Daraus ist zu schließen, dass die Vorbereitung der Szene für die anschließende Anwendung einen immensen Aufwand benötigt. Ebenfalls lässt sich aus den gezeigten Daten erlesen, dass das Erstellen eines Skriptes für eine neue

Regel einen hohen Aufwand mit sich bringt. Der gemessene Aufwand zur Erstellung der Skripts ist ohne Kontext nicht aussagekräftig und muss in den folgenden Abschnitten mit den Vorkenntnissen der Probanden in Verbindung gebracht werden.

Die Einordnung der Effizienzsteigerung des Tools ist, basierend auf den ausgewerteten Daten, abhängig von dem vorliegenden Projekt. Die Szenen der Aufgaben enthielten zunächst ungefähr 50, nachfolgend 75 fehlerhafte Objekte. Anhand dieser Werte soll nun an einem kurzen Beispiel die Effizienz beurteilt werden.

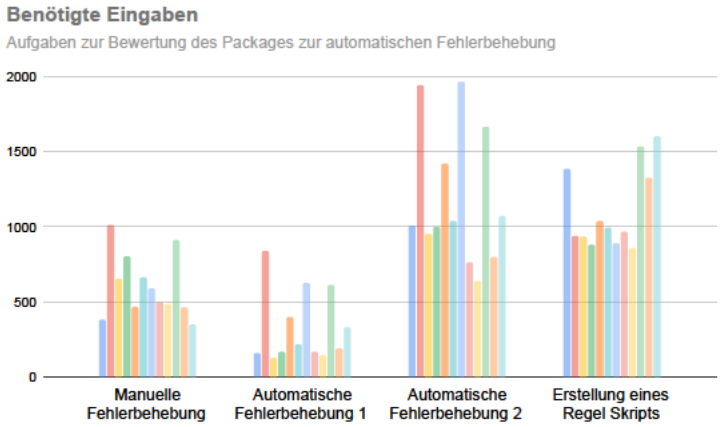

Abbildung 5.6.: Benötigte Nutzereingaben (Fehlerkorrekturen)

Als Ausgangspunkt dient ein Projekt mit einer Szene, welche 75 Assets enthält. Aufgrund häufiger Überarbeitungen des Designs sowie der zugrundeliegenden Skripts soll jene Szene einmal wöchentlich auf fehlerhafte Objekte überprüft werden. Zur Berechnung soll der gemessene Wert zur Überprüfung von rund 50 Objekten verwendet werden, da der Entwickler mit seinem Projekt vertrauter ist. Davon ausgehend, dass der Entwickler jedes Objekt überprüft und wie in der Aufgabe der Großteil jener fehlerhaft ist, dauert eine wöchentliche Überprüfung im Schnitt 700 Sekunden. Über eine Zeitspanne von fünf Wochen wäre der Entwickler eine Stunde mit der Überprüfung und Korrektur der 50 Objekte beschäftigt. Nutzt der Entwickler die erstellten neuen Funktionen würde die Überprüfung sowie Behebung der fehlerhaften Objekte basierend auf den Messwerten rund 240 Sekunden dauern. In den fünf Wochen wäre der Entwickler somit 20 Minuten beschäftigt. Berücksichtigt man zusätzlich die gemessene Zeit um eine Szene der gegebenen Größenordnung vorzubereiten ist der Entwickler weitere 20 Minuten beschäftigt. In jenem Beispiel würde durch die Nutzung des Tools ein Zeitersparnis von rund 20 Minuten in den ersten fünf Wochen entstehen. Basierend auf dieser Tendenz ist davon auszugehen, dass das Tool den Arbeitsaufwand des Entwicklers ab einer bestimmten Größenordnung auf bis zu 30 Prozent reduziert.

Als Fazit der ausgewerteten Daten ist zu ziehen, dass die Verwendung des erstellten Tools in Projekten mit hoher Assetanzahl sowie regelmäßigen Wartungen die Effizienz steigert. In Projekten mit wenigen Assets und wenigen Szenen, welche einmalig aufgesetzt werden, ist die Anwendung jedoch nicht ratsam, da die benötigte Vorbereitungszeit die gewonnene Effizienz überschreitet.

5.1.2. Freie Evaluation

Die freie Evaluation befasst sich mit der Messung der Verhaltensänderung der Probanden über einen längeren Zeitraum. Da sich der Zeitraum der freien Evaluation über zwei Wochen erstreckte und die Anforderungen an die Probanden höher ausfielen, nahm lediglich rund die Hälfte der Probanden an dieser teil. Das folgende Kapitel wird die prägnantesten Beispiele auswerten und resultierend Aussagen über die Effizienz der getesteten Tools treffen.

Automatisches Speichern

Um Änderungen im Nutzungsverhalten zu erkennen, wurde die Nutzung der Speicherfunktion von Unity sowie der Funktion des Tools aufgezeichnet. Erkennbar bei dem Verhalten von Proband eins in Abbildung 5.7 und Proband zwei in Abbildung 5.8 ist, dass zunächst nur die Speicherfunktion der Unity Engine verwendet wurde. Nach den ersten Stunden jedoch verwendeten die Probanden die Tool-Funktion ausgiebig. Beide aufgezeigten Studienteilnehmer nutzen weiterhin die Speicherfunktion der Engine, während die Funktion des Tools jedoch den Hauptteil der Aufrufe ausmacht. Nach wenigen Stunden bereits ist bei Proband eins die Verwendung der Unity-Funktion kaum bis gar nicht mehr vorhanden. Hingegen übernimmt die Tool-Funktion das vollkommene Speichern der Szenen. Allerdings ist bei Proband zwei zu erkennen, dass die Funktion für einige Zeit deaktiviert wurde. Danach jedoch ist, ähnlich wie bei Proband eins, ein klarer Trend, zur Verringerung des manuellen Speichern erkennbar. Die fortbestehende, vereinzelte Nutzung der Unity-Funktion kann auf das antrainierte Verhalten der Probanden zurückgeführt werden und wird bei der Auswertung der Vorkenntnisse berücksichtigt.

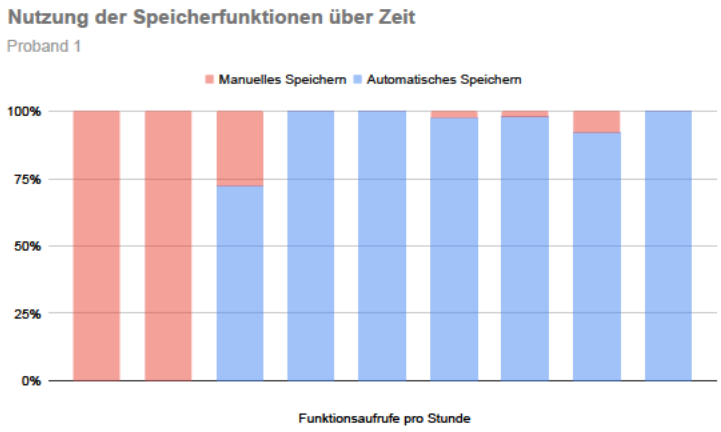

Abbildung 5.7.: Nutzungsverhalten Proband 1

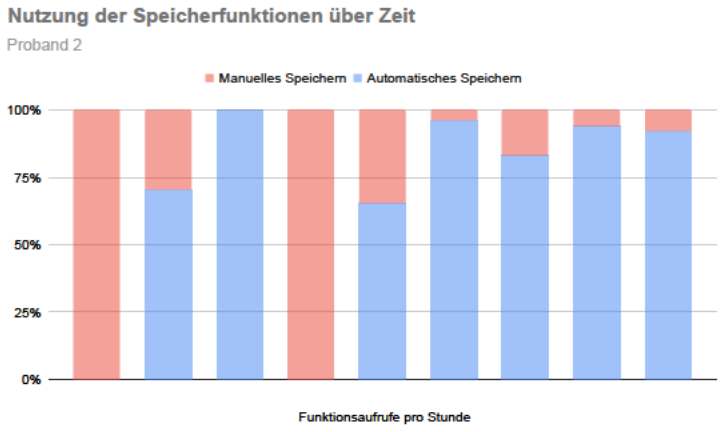

Abbildung 5.8.: Nutzungsverhalten Proband 2

Anhand der präsentierten Daten ist auswertend zu sagen, dass die Nutzung der Funktion zum automatischen Speichern von den Probanden in ihr Nutzungsverhalten aufgenommen wurde. Durch den erkennbaren Trend verringert sich die Notwendigkeit die Speicherfunktion händisch auszuführen. Dies wiederum verringert die notwendigen Nutzereingaben und steigert die Effizienz geringfügig. Die größere Steigerung der Effizienz rührt daher, dass der Nutzer während der Bedienung des Programms sorgenfrei arbeiten kann ohne den Verlust großer Fortschritte zu riskieren.

Automatisches Starten der Hauptmenü Szene

Zur Erarbeitung der Effizienzsteigerung wurde das Verhalten eines Probanden ausgewählt, welches repräsentativ für die erhobenen Daten steht. Die Abbildung 5.9 zeigt zum einen die Nutzung der erstellten Tool-Funktion sowie das Verhalten, welches das Tool zu verringern vermag. Das zu vermeidende Verhalten sieht vor, nach der Bearbeitung einer Szene, in die Hauptmenü-Szene zu wechseln, von dort aus den Playmode zu starten und anschließend, nach ausgiebigem Testen, wieder in die ursprüngliche Szene zu wechseln. Dieses Verhalten wird erkannt indem die aufgezeichneten Logs nach Szenenwechseln durchsucht werden. Zielt ein Szenenwechsel auf die zum Starten des Spiels genutzte Szene und wird der Playmode innerhalb eines Zeitfenster gestartet, so wird dies als ein solches Verhalten aufgefasst.

Die gemessenen Daten des Probanden zeigen ähnlich wie die Auswertung der Speicherfunktion zunächst keine Nutzung der Tool-Funktion. Nach wenigen Stunden tritt ebenfalls die erste Nutzung ein. Nach initialer Nutzung der Funktion, ist ein starker Anstieg der Verwendungen zu erkennen. Trotz der durchgängig hohen Anwendung der Tool-Funktion kommt es vor, dass der Proband in sein altes Verhalten verfällt. Dies kann ebenfalls, wie in den vorangehenden Abschnitten benannt, auf das antrainierte Verhalten der Probanden zurückgeführt werden und muss bei der Auswertung bedacht werden.

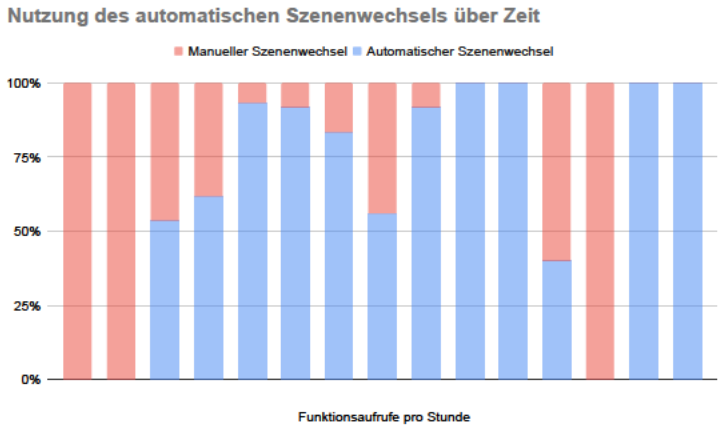

Abbildung 5.9.: Nutzungsverhalten der Szenenwechsel Funktion

Basierend auf den vorliegenden Messdaten ist zu erkennen, dass nach der initialen Nutzung der Funktion die Probanden ihr Verhalten ändern und die Anwendung der neuen Funktion dominiert. Trotz des verbleibenden Vorkommens des ineffizienteren Szenenwechsels steigerte das Tool die Effizienz der Probanden, da diese durch jede Verwendung der Funktion wertvolle Zeit sparten.

5.2. Auswertung Fragebögen

Nach der Auswertung der erhobenen Daten, sollen nun die Ergebnisse der Umfragen interpretiert und mit den bereits bestehenden Informationen verknüpft werden. Um eine statistische Auswertung der Umfrageergebnisse zu ermöglichen, sowie um die Einheitlichkeit und Verständlichkeit der Fragen zu verbessern, wurde die Likert Skala verwendet. Hierbei reichen die Antwortmöglichkeiten der Probanden von 1 (stimme überhaupt nicht zu) bis 5 (stimme voll und ganz zu). Unter Verwendung dieser Bewertung soll die Intensität der Meinung des jeweiligen Probanden zu der gestellten Frage bewertet werden. Um fortan Aussagen in Bezug auf die Vorkenntnisse der Probanden treffen zu können, sollen zunächst die Einschätzungen der eigenen Fähigkeiten aller Probanden analysiert werden.

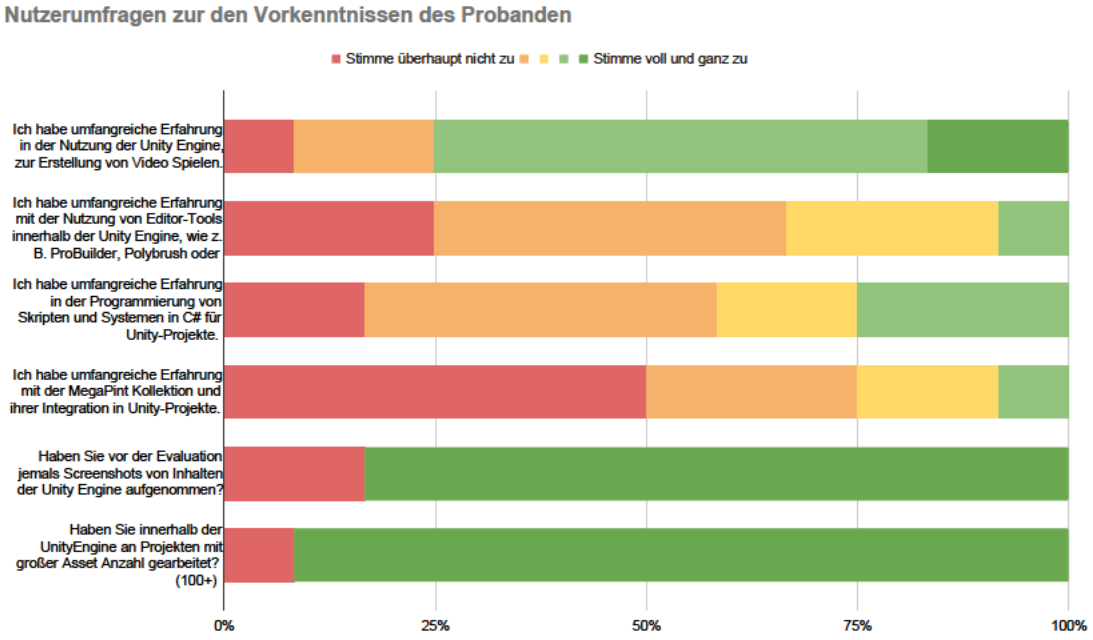

Abbildung 5.10.: Vorkenntnisse der Probanden

Zunächst sollen die Probanden ihre Erfahrung im Umgang mit der Unity Engine zur Erstellung von Videospielen einschätzen. Hierbei ist der überwiegende Teil von seinen Fähigkeiten überzeugt, wovon zwei Probanden voll und ganz zustimmten, umfangreiche Erfahrungen zu besitzen. Eine Person hingegen stimmt dieser Aussage überhaupt nicht zu und zwei weitere schätzen ihre Erfahrungen ebenfalls als mangelnd ein. Verglichen mit der überwiegend positiven Beantwortung der ersten Frage fallen die Antworten auf die Frage nach der Erfahrung im Umgang mit Editor-Tools schlecht aus. Lediglich ein Proband stimmt zu, Erfahrungen mit Editor-Tools zu besitzen. Der überwiegende Teil der Probanden stimmt dieser Aussage nicht zu, wobei drei Probanden keinerlei Erfahrung mit Editor-Tools besitzen. Ein ähnliches Verhältnis ist, wie die Abbildung 5.10 zeigt, bei der Frage nach den Erfahrungen im Umgang mit C# der Fall.

An der Evaluation teilnehmend gibt es einen Probanden der zustimmt, bereits Erfahrungen mit den Packages der MegaPint Kollektion zu besitzen. Fünf weitere geben an, eher weniger Erfahrung mit diesen zu besitzen. Die restlichen Probanden haben zuvor noch nicht mit der MegaPint Kollektion und deren Packages gearbeitet. Zwei der zwölf Probanden geben an, noch keine Screenshots innerhalb der Engine aufgenommen zu haben, während die verbleibenden Probanden die Frage bejahten. Ein ähnliches Ergebnis erzielte die letzte Frage nach den Vorkenntnissen, wo lediglich eine Person angab noch nicht innerhalb eines Projektes mit mehr als 100 Assets gearbeitet zu haben.

Basierend auf den erhobenen Vorkenntnissen der Probanden sollen diese nun genutzt werden, um die Frage zu beantworten, ob die im vorherigen Kapitel gemessene Effizienzsteigerung von den Vorkenntnissen der Probanden abhängig ist. Die Abbildungen 5.11 und 5.12 sind die addierten Werte der Abbildungen 5.1 mit 5.3 und 5.4 mit 5.6. Sie repräsentieren den Aufwand, bestehend aus notwendiger Zeit und Nutzereingaben, der einzelnen Probanden für die Aufgaben der geführten Evaluation. Die Farbgebung der Werte spiegelt das angegebene Vorwissen des jeweiligen Probanden wieder. Rot steht hierbei für die niedrigsten und ein dunkles Grün für die bestmöglichen Angaben aller Antworten.

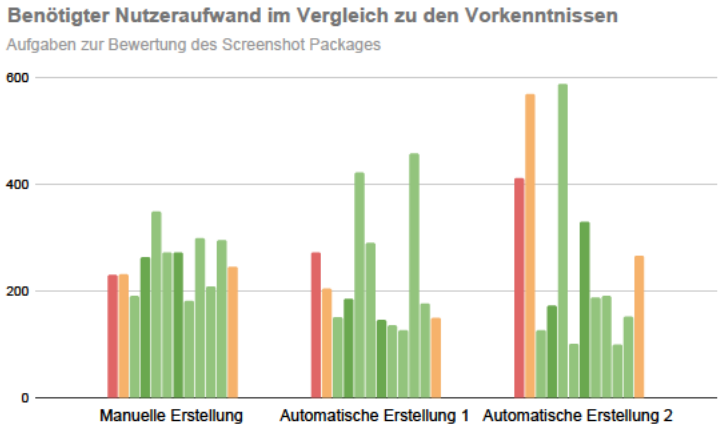

Abbildung 5.11.: Benötigter Aufwand mit Bezug auf das Vorwissen der Probanden (Screenshots)

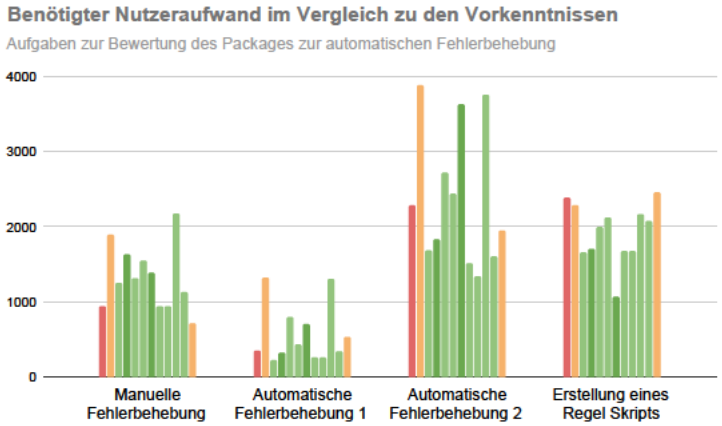

Abbildung 5.12.: Benötigter Aufwand mit Bezug auf das Vorwissen der Probanden (Fehlerkorrekturen)

Wie aus den Abbildungen 5.11 und 5.12 zu erkennen ist, benötigen die Probanden mit geringeren Vorkenntnissen häufig mehr Aufwand als die erfahreneren Probanden. Jedoch lässt sich daraus nicht darauf schließen, dass mit zunehmender Erfahrung der notwendige Aufwand sinkt, da ebenfalls Probanden mit sehr hohem Vorwissen

erheblich größeren Aufwand benötigten. Was jedoch eindeutig erkennbar ist, ist, dass die Probanden mit wenig Unity Erfahrung in Aufgaben, welche viel Arbeit innerhalb der Unity Oberflächen beinhalten, im Schnitt deutlich länger brauchen. Im Vergleich hierzu unterscheidet sich die Abweichung im notwendigen Aufwand zur Bearbeitung der manuellen Erstellung von Screenshots nur wenig.

Auf Basis dieser Daten lässt sich festhalten, dass die fehlende Erfahrung mit der Unity Engine den notwendigen Aufwand erhöht. Eine hohe Erfahrung mit der Unity Engine jedoch ist nicht gleich eine Garantie auf eine effizientere Vorgehensweise, da dies abhängig davon ist, ob der Nutzer die zu testenden Tätigkeiten häufig ausübt und die zu nutzenden Tools intuitiv verwenden kann.

5.2.1. Geführte Evaluation

Vereinfachte Bildschirmaufnahmen

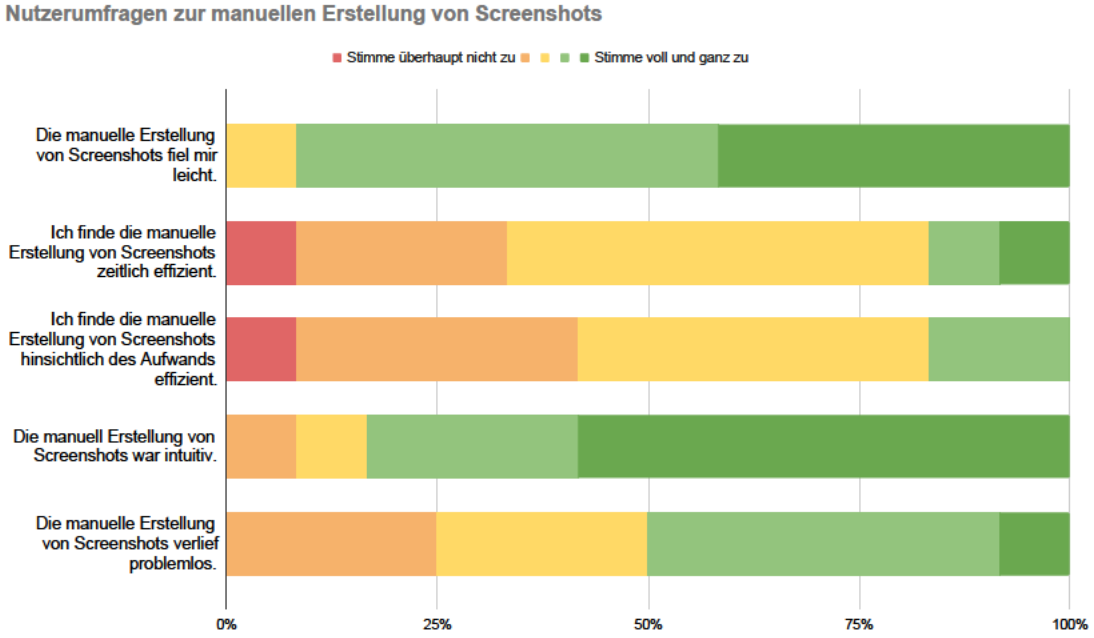

Abbildung 5.13.: Umfrageergebnisse zur manuellen Erstellung von Screenshots

Wie aus Abbildung 5.13 hervorgeht, fiel dem Großteil der Probanden, nach eigener Aussage, die manuelle Erstellung von Screenshots leicht. Ebenfalls berichtet der Großteil von einem intuitiven Umgang mit dem selbst gewählten Programm. Die Einschätzung der zeitlichen sowie aufwandstechnischen Effizienz hingegen zeigt eindeutig, dass die Mehrheit die manuelle Erstellung als ineffizient bewertet. Lediglich zwei der zwölf Probanden bewerteten die manuelle Erstellung zeitlich sowie hinsichtlich des Aufwands als effizient. Bei der Hälfte der Probanden sind, basierend auf den Umfrageergebnissen, während der manuellen Erstellung Probleme aufgetreten. Nach Auswertung der von den Probanden angegebenen Problemen, beschreiben alle den gleichen Sachverhalt. Die

Aufgabe beinhaltete eine sich drehende Kamera während das Ziel der Aufgabe, das Aufnehmen der Stillleben im Zentrum des Bildschirmes war. Durch die Verwendung der gewählten Tools wurde zunächst, laut Probandenaussagen, die Anwendung im Hintergrund nicht pausiert, wodurch sich die Kamera weiter drehte und einige Stillleben nicht mit einer Rotation erfasst werden konnten. Zudem besaßen die Programme eine kleine Verzögerung, welche das Zentrieren der Stillleben erschwerte.

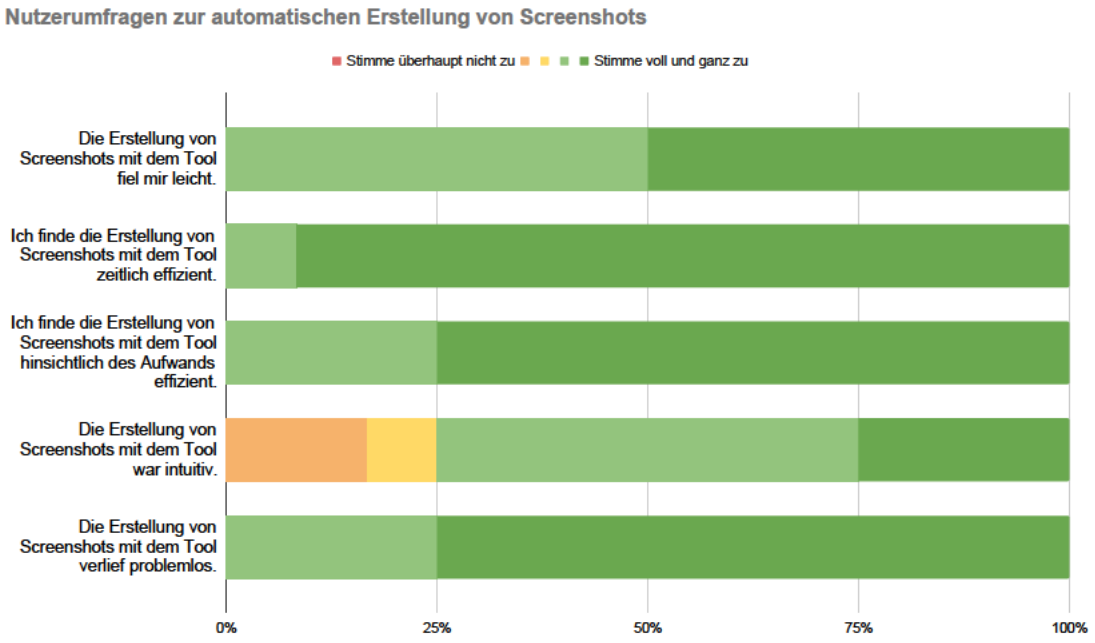

Abbildung 5.14.: Umfrageergebnisse zur automatischen Erstellung von Screenshots

Entgegen der Umfrageergebnisse zur manuellen Erstellung von Screenshots, wurde die automatische Erstellung sehr positiv bewertet. Ebenfalls wurde die Effizienz des Tools als überwiegend sehr hoch eingeschätzt. Lediglich die intuitive Nutzung des Tools wurde von den Probanden niedriger bewertet. Drei der zwölf Probanden sagen aus, dass die automatische Erstellung von Screenshots eher nicht intuitiv war. Wie bereits im vorangegangenen Kapitel vermutet, hat dies Auswirkungen auf den benötigten Aufwand genommen. Die Probanden, welche einen Anstieg des Aufwands zwischen der manuellen und der automatischen Erstellung aufwiesen, sind jene Probanden, welche die Intuitivität als eher schlecht bewerteten. Abschließend lässt sich jedoch eindeutig belegen, dass die Umfrageergebnisse für eine verbesserte Effizienz durch das genutzte Tool sprechen. Dies wurde ebenfalls durch zusätzliche Bemerkungen der Probanden in den Umfragen ersichtlich.

Fehlerkorrektur von Assets

Wie in Abbildung 5.15 zu erkennen ist, fiel den meisten Probanden die manuelle Fehlerbehebung schwer. Ebenfalls wurde diese mehrheitlich als kontraintuitiv bewertet. Die Probanden, welche jene Fragen hingegen positiv beantworteten, besaßen bereits Vorwissen im Umgang mit Unity und der Bearbeitung von Assets. Dieses Vorwissen erklärt die intuitive Nutzung der Unity Funktionen. Die

zeitliche Effizienz und der notwendige Aufwand wurden als überwiegend negativ bewertet. Hierbei stimmten drei bis vier der Probanden den jeweiligen Aussagen überhaupt nicht zu.

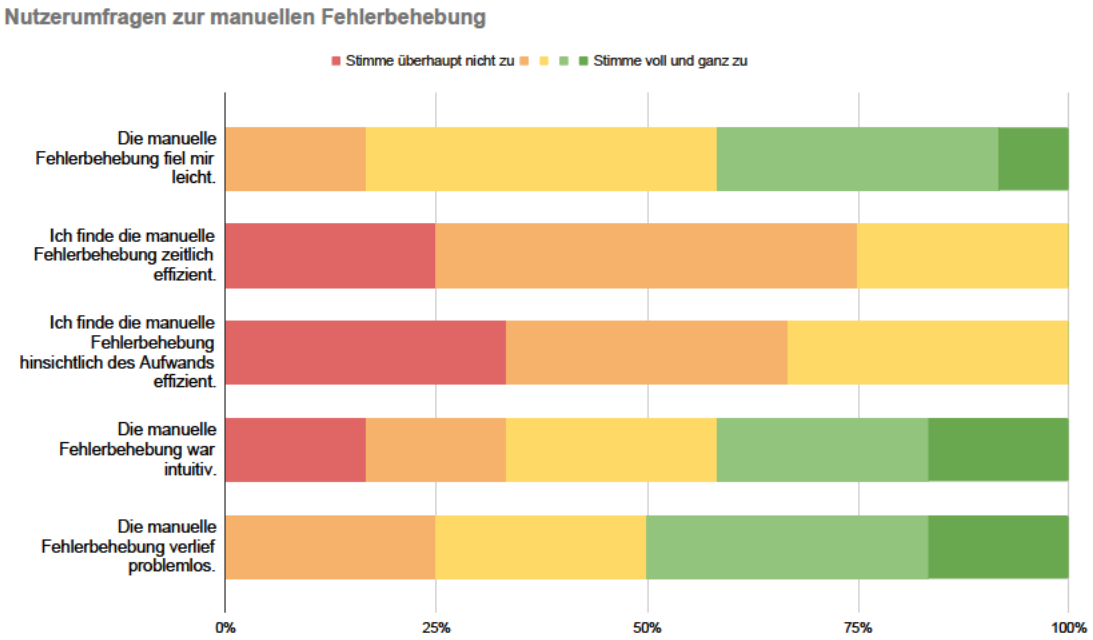

Abbildung 5.15.: Umfrageergebnisse zur manuellen Fehlerbehebung

Verglichen mit den bisherigen Ergebnissen der manuellen Fehlerbehebung ist eine deutliche Verbesserung in der Umfrage zur automatischen Fehlerbehebung, welche in Abbildung 5.16 zu sehen ist, zu erkennen. Die Probanden geben an, das Tool leicht und intuitiv genutzt zu haben. Bei der intuitiven Verwendung gaben lediglich drei Probanden an, dass sie dieser Aussage eher weniger zustimmen. Ebenfalls, wie bei der Auswertung der Bildschirmaufnahmen, ist zu erkennen, dass diese Probanden ein Anstieg in dem benötigten Aufwand zur Bearbeitung der Aufgabe aufweisen.

Die Aussagen hinsichtlich der Effizienz wurden überwiegend mit voller Zustimmung beantwortet. Diese Tendenz wurde zusätzlich durch Anmerkungen der Probanden in den Umfragen bestärkt. Hierbei bestärken die Probanden den Sachverhalt, dass das Tool die Korrektur von Fehlern auf Assets deutlich vereinfacht und effizienter gestaltet. Aus diesem Grund ist zusammenfassend das Tool zur automatischen Fehlerkorrektur anhand der Umfrageergebnissen als effizient einzustufen.

Als meist einzige Schwierigkeit der Probanden wurde die Erstellung eigener Regeln durch Code genannt. Dies kann mittels der festgestellten Vorkenntnisse jedoch darauf zurückgeführt werden, dass der überwiegende Teil der Probanden wenig Vorwissen in diesem Bereich besitzt.

Abschließend an die geführte Evaluation wurden den Probanden die in Abbildung B.2 gezeigten Fragen gestellt. Auf die Frage, ob die getesteten Tools ihre Effizienz innerhalb der Engine gesteigert haben, stimmten neun Probanden voll und ganz zu, zwei Probanden stimmten überwiegend und ein Proband teilweise zu. Desweiteren stimmten fünf Probanden voll und ganz zu, dass sie in Zukunft

die Nutzung von Editor Tools in Betracht ziehen werden. Weitere sieben Probanden stimmten der Aussage ebenfalls überwiegend zu. Aus diesen abschließenden Ergebnissen lässt sich erkennen, dass die getesteten Tools der Bildschirmaufnahme und Fehlerkorrektur die Effizienz der Probanden nach eigenem Gefühl steigerten.

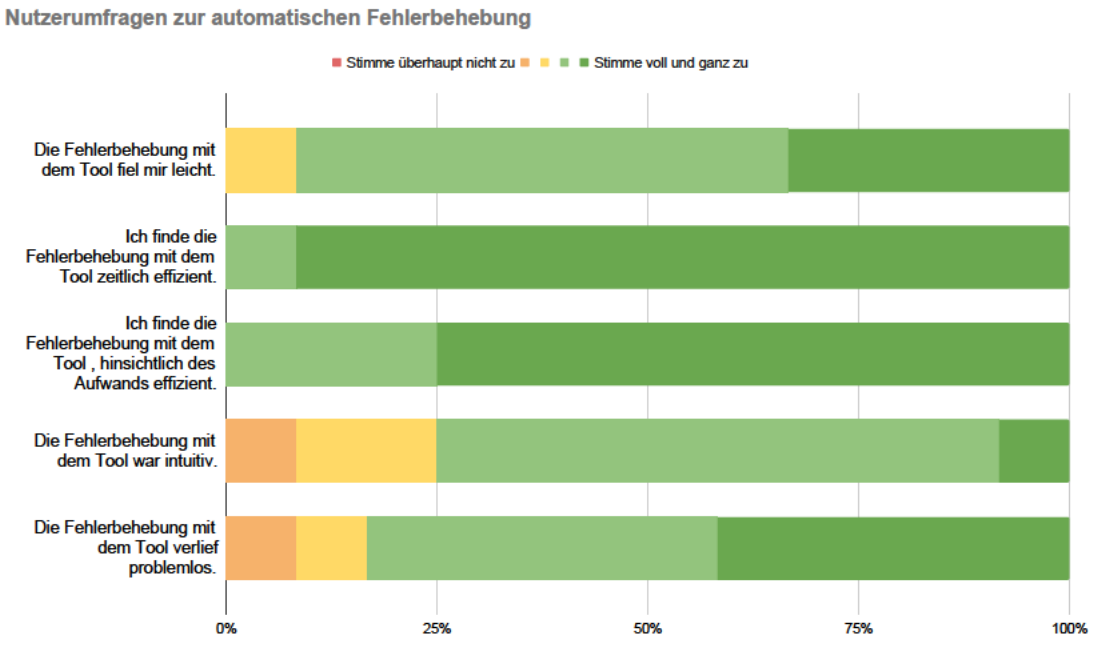

Abbildung 5.16.: Umfrageergebnisse zur automatischen Fehlerbehebung

5.2.2. Freie Evaluation

Automatisches Speichern
Wie in Abbildung B.3 zu erkennen, gibt rund ein Drittel der Probanden an, die Funktion intensiv verwendet zu haben. Die verbleibenden Probanden stimmen dieser Aussage ebenfalls überwiegend zu. Alle Probanden geben zudem an, dass ihnen die Nutzung der Funktion leicht gefallen ist, sie das Tool intuitiv benutzten und keine Probleme aufgetreten sind. Alle Nutzer haben eine leichte Änderung ihres Verhaltens bekundet, was die, im vorangegangenen Kapitel ausgewerteten Daten ebenfalls belegen. Ein Drittel der Probanden stimmt der Aussage, dass durch das Tool die Effizienz gesteigert wird, nur in Maßen zu, während die verbleibenden Probanden überwiegend zustimmen.

Die Funktion des automatischen Speichern wurde, basierend auf den vorliegenden Ergebnissen, überwiegend positiv aufgenommen und der in den aufgezeichneten Daten erfasste Trend wurde durch die Probanden bestätigt. Ebenfalls lässt sich durch die Verbindung der Vorkenntnisse belegen, dass Nutzer mit hohem Vorwissen im Bereich von Unity, eine langsamere Verhaltensänderung aufweisen. Dies ist auf die bestehenden antrainierten Verhaltensmuster zurückzuführen.

Automatisches Starten der Hauptmenü Szene
Das Tool zum automatischen Starten einer Hauptmenü Szene wurde weniger intensiv von den Probanden verwendet, da dies meist nur in speziellen Fällen, wie dem Leveldesign, Anwendung findet. Trotz dessen wurde das Tool, wie in Abbildung B.4 erkenntlich, durch die teilnehmenden Probanden als intuitiv und die Nutzung als leicht bewertet. Ebenfalls verzeichnete kein Proband Probleme mit der Funktion. Im Vergleich zur Speicherfunktion wurde die Verhaltensänderung als weniger vorhanden eingestuft. Dies ist auf die geringe Nutzung der Funktion durch einige Probanden zurückzuführen. Hinsichtlich der Steigerung der Effizienz wurde die Funktion ebenfalls durch ein Drittel als eher gering eingestuft, während zwei Drittel der teilnehmenden Probanden eine überwiegende bis starke Steigerung ihrer Effizienz feststellten.

Trotz der geringeren Beteiligung kann das Tool zum automatischen Starten einer Hauptmenü Szene als effizienzsteigernd bewertet werden. Diese Bewertung der Probanden ist übereinstimmend mit der Auswertung der aufgezeichneten Daten.

5.3. Zusammenfassung

Aus den Ergebnissen der ausgewerteten Daten sowie der Umfrageergebnisse beider Teile der Evaluation, lassen sich folgende Ergebnisse zusammenfassen. Die erhobenen Daten belegen evidentisch, dass die getesteten Funktionen zur Verbesserung der Effizienz beitragen. Jene Steigerung ist zeitlich sowie anhand der Reduzierung des notwendigen Aufwands belegbar. Die Probanden bestätigen die gemessene Effizienzsteigerung ebenfalls durch ihre Angaben innerhalb der Umfragen. Durch die Einschätzung der Probanden sowie der evidentischen Daten können die entworfenen Funktionen eindeutig als effizienzsteigernd klassifiziert werden.

Unter Berücksichtigung der Vorkenntnisse der Probanden wurden zusätzliche Sachverhalte aufgedeckt. Probanden mit wenig Vorwissen über die Unity Engine benötigen im Durchschnitt mehr Aufwand in der Bearbeitung der gestellten Aufgaben als ein Proband mit weitreichender Unity Erfahrung. Jedoch garantiert ein solides Vorwissen keineswegs die effiziente und intuitive Nutzung neuer Editor-Tools. Ebenfalls erfahrene Entwickler müssen sich an die Bedienung neuer Funktionen und Werkzeuge gewöhnen, wodurch der anfängliche Aufwand hoch ausfallen kann.

Final lässt sich zudem feststellen, dass Nutzer durch die Bereitstellung eines Tools, welches eine alltägliche, häufig genutzte Funktion verbessert, ihr Verhalten selbstständig ändern. Diese Erkenntnis steht in Einklang mit der Aussage von Tim Cochran, *"Developers, if empowered, will naturally optimize, but often I find the micro-feedback loops have been neglected[...]"* [Coc21]. Die einfache und intuitive Bedienung des Tools ist, wie in der Evaluation gezeigt, ein integraler Bestandteil um das Verhalten des Nutzers zu verändern. Dieser Prozess geschieht über eine Zeitspanne, welche von der bereits gesammelten Erfahrung abhängig ist. Entwickler mit langjähriger Erfahrung ändern ihr Verhaltensmuster langsamer als Entwickler, welche keine solche Muster verinnerlicht haben. Diese These wird bestärkt durch die Aussage, dass die Zufriedenheit und somit der Wille zur Nutzung eines neuen Tools, von der Frustration eines Entwicklers mit dem Tool abhängig ist [GSN23].

6. Fazit und Ausblick

Zu Beginn der Arbeit wurden zunächst vier repetitive Aufgaben ausgewählt, welche innerhalb der Unity Engine auftreten. Hierbei handelt es sich zunächst um das Erstellen von Bildschirmaufnahmen und die Korrektur von aufgetretenen Fehlern auf Assets. Zusätzlich wurden das manuelle Speichern und das Wechseln in die Hauptmenü Szene zum Starten des Spiels als repetitive Aufgaben ausgewählt. Anschließend wurden für jene Aufgaben Lösungen konzipiert, welche die Arbeit in Unity effizienter gestalten sollen.

Um dem Nutzer den Import und die Verwendung der konzipierten Lösungen zu vereinfachen, wurde ein Package Collection System erstellt. Hierbei handelt es sich um eine Ansammlung von Packages, welche über das zentrale Haupt-Package direkt in Unity importiert werden können. Die Umsetzung der grafischen Oberfläche wurde durch die Verwendung des UI Toolkits und webbasierter Programmierprinzipien ermöglicht.

Um die Steigerung der Effizienz messbar zu machen, wurde eine Evaluation konzipiert. Die Durchführung der Evaluation fand innerhalb der Unity Engine statt. Die Probanden mussten hierbei durch eine Anzahl an Aufgaben navigieren. Während der Bearbeitung der Aufgaben wurden die benötigten Zeiten und Eingaben aufgezeichnet. Durch die Auswertung dieser konnten nachträglich eindeutige Aussagen zur Steigerung der Effizienz getroffen werden. Zusätzlich wurde in einem Teil der Evaluation die Änderung des Nutzerverhaltens über den Zeitraum von zwei Wochen gemessen. Hierbei wurde die Häufigkeit der verwendeten Funktionen verglichen. Durch die Verbindung der erhaltenen Daten mit den Vorkenntnissen und Umfrageergebnissen der Probanden, konnten individuelle Aussagen über die Effizienz getroffen werden.

Als Ergebnis der Evaluation zeigte sich deutlich, dass die entworfenen Editor-Tools die Effizienz der Nutzer steigern. Dies wurde anhand der aufgezeichneten Daten sowie den Aussagen der Probanden belegt. Hierbei konnte festgestellt werden, dass die Vorkenntnisse eines Entwicklers Einfluss auf den notwendigen Aufwand zur Lösung der gestellten Aufgaben haben. Entwickler mit geringem Vorwissen brauchten im Schnitt länger als erfahrene Entwickler. Jedoch ist aufzuzeigen, dass einige erfahrene Probanden durch die Nutzung von, für sie neuer Werkzeuge einen starken Anstieg im Aufwand erfuhren. Daraus ist zu schließen, dass eine intuitive Bedienung von Tools ein integraler Bestandteil sein muss. Ein weiteres Ergebnis der Evaluation zeigt, dass Entwickler ihr Verhalten über die Zeit durchaus ändern, wenn eine effizientere Funktion zur Verfügung steht. Die notwendige Zeit, um die neue Funktion vollständig zu integrieren, ist hierbei von den bereits bestehenden Angewohnheiten des Entwicklers abhängig. Erfahrene Nutzer tendieren zu einem längeren Umgewöhnungszeitraum.

Um die Ergebnisse dieser Arbeit auf die Allgemeinheit der Unity Nutzer zu pauschalisieren, sind weitere Evaluationen notwendig. So wäre es in aufbauenden Arbeiten möglich, die durchgeführte

Evaluation um weitere Zielgruppen zu erweitern. Relevant hierfür ist die Bearbeitung durch eine Expertengruppe sowie weitere Probanden welche keine Erfahrung in der Verwendung der Unity Engine besitzen. Eine Erweiterung durch diese beiden Zielgruppen würde aufschlussreiche Informationen liefern, welche auf die Allgemeinheit anwendbar wären.

Um in Zukunft repetitive Aufgaben der Unity Engine durch die Erstellung von Editor Tools zu lösen, können einige Ergebnisse dieser Arbeit hilfreich sein. Die intuitive Bedienung der Funktionen der entwickelten Tools bedarf anhand der Nutzerumfragen Verbesserung. Um die intuitive Bedienung in künftigen Tools besser umzusetzen, wäre die Umsetzung dynamischer Hinweise in Betracht zu ziehen. Einige nützliche Funktionen der komplexeren Tools wurden von den Nutzern nicht wahrgenommen. Um dies zu verbessern, wäre eine Möglichkeit, die Handlungen des Nutzers aufzuzeichnen und zu interpretieren. Anhand dieser Daten würden dynamische Hinweise eingeblendet, welche den Nutzer darauf hinweisen, dass er durch die Verwendung einer bestimmten Funktion seinen Arbeitsfluss deutlich vereinfachen kann.

Die Evaluation zeigt, dass das umgesetzte Prinzip einer Kollektion von Packages das Importieren von UPM Packages stark vereinfacht. Der Nutzer muss hierbei keine Git Adressen kennen oder diese über den Package Manager importieren. Ebenfalls erleichtert jenes Konzept die Entwicklung eines neuen Tools, da eine große Menge an Hilfsfunktionen bereits zur Verfügung gestellt wird. Hierdurch wird der Aufwand zur Implementation sowie die Menge an notwendigem Code drastisch reduziert. Für die Erstellung zukünftiger Tools wäre die Implementation einer Kollektion möglicherweise nicht dem Aufwand entsprechend. Für wenige Packages wäre es jedoch ratsam, ein Package als Abhängigkeit anzugeben, welches alle notwendigen Hilfsfunktionen beinhaltet, um diese nicht in jedem Tool neu zu definieren.

Um den Unity internen Import der Packages ohne Wissen über die Git Adressen zu ermöglichen, wäre ein durchsuchbares Register vorteilhaft. Zukünftig könnte die Implementation eines solchen Registers die Nutzung von Packages, welche auf Git Hub liegen stark vereinfachen. Das Register würde, ähnlich der Website OpenUPM, die Namen der Packages mit den zugrundeliegenden Git Adressen verknüpfen und somit den Import ohne weitere notwendige Eingaben direkt in Unity ermöglichen.

Literaturverzeichnis

[Abb18] D. Abbott: *Using FMOD Studio with Unity*, Bachelorarbeit, Kajaani University of Applied Sciences, 2018.

[AC98] M. Abadi und L. Cardelli: *A theory of objects*, Springer, 1998.

[AT24] N. Ahmad und N. Tripathi: *Benefits, challenges, and implications of open-source software for health-tech startups: An empirical study*, in *Lecture Notes in Business Information Processing*, S. 265–282, Springer Nature Switzerland, 2024.

[BR23] A. Boutellier und P. Raptis: *Teaching virtual production: the challenges of developing a formal curriculum*, *Film Educ. J.*, Bd. 6(2), 2023.

[Coc21] T. Cochran: *Maximizing Developer Effectiveness*, 1 2021, besucht am 03.10.2024.
URL https://martinfowler.com/articles/developer-effectiveness.html

[ddGBe] Verband der deutschen Games-Branche e.V.: *Spielgeschichte*, besucht am 13.08.2024.
URL https://www.game.de/themen/kulturgut-digitale-spiele-uebersicht/spielgeschichte/

[ddGBe21] Verband der deutschen Games-Branche e.V.: *Indie Games werden immer beliebter*, 2 2021, besucht am 14.08.2024.
URL https://www.game.de/indie-games-werden-immer-beliebter/

[GH89] J. Grundy und J. Hosking: *Software Tools*, *Bio/technology*, Bd. 7:S. 386–386, 4 1989.

[Gra18] K. Grant: *CSS in Depth*, Manning Publications, 2018.

[Gre18] J. Gregory: *Game Engine Architecture, Third Edition, 3rd Edition*, A K Peters/CRC Press, 2018.

[GSN23] M. Greiler, M.A. Storey und A. Noda: *An actionable framework for understanding and improving developer experience*, *IEEE Transactions on Software Engineering*, Bd. 49(4):S. 1411–1425, 2023.

[Har23] C. Hardman: *Game Programming with Unity and C: A Complete Beginner's Guide*, Apress, 2023.

[IVG24] Z. Irfan, S.N. Vaughn und L. Ganti: *History and Evolution of the Reflex Hammer*, *Cureus*, Bd. 16(8):S. e66333, 8 2024.

[Kok21] B. Kok: *Beginning Unity Editor Scripting: Create and Publish Your Game Tools*, Apress, Berkeley, CA, 2021.

[PGB12] P.S. Paul, S. Goon und A. Bhattacharya: *History and comparative study of modern game engines*, *International Journal of Advanced Computer and Mathematical Sciences*, Bd. 3(2):S. 245–249, 6 2012.

[Pra22] H.K. Prasad: *Training and Development Programs: Evaluating Effectiveness and ROI*, *IJFANS International Journal of Food and Nutritional Sciences*, Bd. 11(8):S. 4718–4722, 2022.

[RF12] W.E. Riddle und R.E. Fairley: *Software Development Tools*, Springer Berlin Heidelberg, 2012.

[RM24] M. Ranaweera und Q.H. Mahmoud: *Deep Reinforcement Learning with Godot game engine*, *Electronics (Basel)*, Bd. 13(5):S. 985, 2024.

[Sal22] T. Salmela: *Game Development Using the Open-Source Godot Game Engine*, Bachelorarbeit, Tampere University of Applied Sciences, 2022.

[SB11] B. Schade-Bünsow: *Werkzeug, Revolution und Evolution*, *Bauwelt Bauverlag BV*, Bd. 23:S. 17–21, 6 2011.

[Sch24] M. Schlosser: *Spieleentwicklung mit Unity, Das umfassende Handbuch*, Rheinwerk Verlag, 2024.

[SF21] M. Smith und S. Ferns: *Unity 2021 Cookbook - Fourth Edition*, Packt Publishing, 2021.

[SR22] F. Sapio und R. Ratini: *Developing and testing a new reinforcement learning toolkit with unreal engine*, in *Artificial Intelligence in HCI*, S. 317–334, Springer International Publishing, 2022.

[SS14] J.T. Streib und T. Soma: *Guide to java*, Springer, 2014.

[Ste] SteamDB: *Steam Indie Game Releases by Year*, besucht am 14.08.2024.
URL https://steamdb.info/stats/releases/?tagid=492

[Tec24] Unity Technologies: *Create and export asset packages*, 8 2024, besucht am 16.08.2024.
URL https://docs.unity3d.com/Manual/AssetPackagesCreate.html

[Ver18] J. Verret: *Implementing Agile Methodology: Challenges and Best Practices*, Diplomarbeit, University of Oregon, 2018.

Anhang

A. Dokumente Zur Implementation

Aufbau der Package Collection

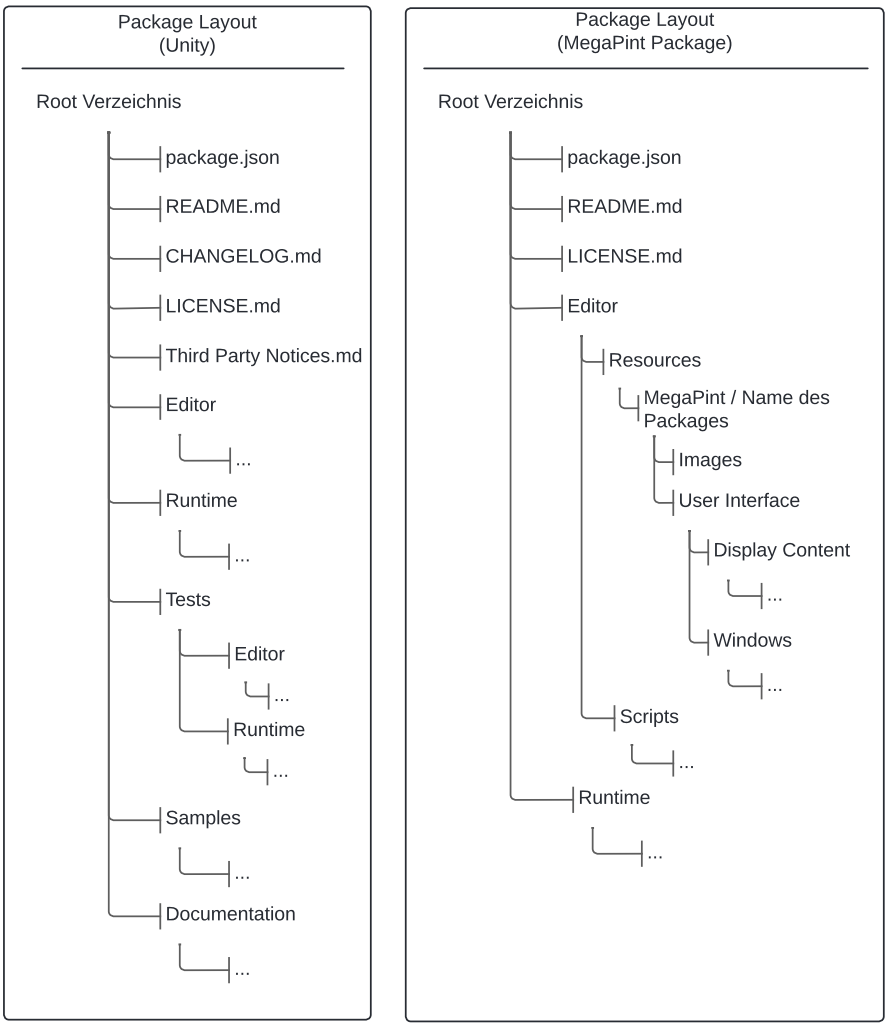

Abbildung A.0.1.: Vergleich Unitys UPM Layout und MegaPint Package Layout

Implementation

```
1  private static void ImportBulk(List <CachedPackage> packages)
2  {
3      if (packages is not {Count: > 0})
4          return;
5
6      var path = Application.dataPath[..^7];
7      path = Path.Combine(path, "Packages", "manifest.json");
8
9      var manifestText = File.ReadAllText(path);
10
11     const string Divider = ""dependencies": {";
12
13     var parts = manifestText.Split(Divider);
14
15     var part1 = $"{parts[0]}{Divider}";
16     var part2 = parts[1];
17
18     foreach (CachedPackage package in packages)
19     {
20         var name = $""{package.Name}":";
21
22         if (part2.Contains(name))
23             continue;
24
25         var url = $""{PackageManagerUtility.GetPackageUrl(package)}"";
26
27         part2 = $"{name}{url},{part2}";
28     }
29
30     var newManifest = $"{part1}{part2}";
31     File.WriteAllText(path, newManifest);
32 }
```

Listing A.1: Methode zum Importieren mehrerer Packages

```
1  [CustomEditor(typeof(MegaPintMainSettings))]
2  public class MegaPintSettingsDrawer : UnityEditor.Editor
3  {
4      public override void OnInspectorGUI()
5      {
6          EditorGUILayout.LabelField("This asset is used to store all
               MegaPint settings.");
7      }
8  }
```

Listing A.2: Settings Asset Inspector

```
1  public static bool Exists()
2  {
3      if (instance != null)
4          return true;
5
6      var search = AssetDatabase.FindAssets($"t:{nameof(
           MegaPintMainSettings)}", new[] {"Assets"});
7
8      if (search.Length == 0)
9          return false;
10
11     instance = AssetDatabase.LoadAssetAtPath <MegaPintMainSettings>(
           AssetDatabase.GUIDToAssetPath(search[0]));
12
13     return instance != null;
14 }
```

Listing A.3: Methode zum Finden des Settings Assets

```
1  public static string LatestGitTag(string repository)
2  {
3      return GitTags(repository)[^1];
4  }
5
6  private static string[] GitTags(string repository)
7  {
8      List <string> tags = new();
9
10     var outPut = RunGit($"ls-remote␣--tags␣{repository}");
11
12     var ses = outPut.Split("tags/");
13
14     for (var i = 1; i < ses.Length; i++)
15     {
16         var s = ses[i];
17         tags.Add(s.Split("\n")[0]);
18     }
19
20     return tags.ToArray();
21 }
```

Listing A.4: Auslesen des neuesten Tags eines Git Repositories

```
1  private static string RunGit(string arguments)
2  {
3      using var process = new Process();
4
5      var exitCode = process.Run(
6          "git",
7          arguments,
8          out var output,
9          out var errors);
10
11     if (exitCode == 0)
12         return output;
13
14     throw new Exception($"Git Exit Code: {exitCode} - {errors}");
15 }
```

Listing A.5: Ausführen eines Git Befehls

```
1  public static Texture2D RenderCamera(Camera camera, int width, int
       height, int depth)
2  {
3      RenderTexture cameraTarget = camera.targetTexture;
4
5      var myRenderTarget = new RenderTexture(width, height, depth);
6
7      camera.targetTexture = myRenderTarget;
8
9      RenderTexture activeRenderTexture = RenderTexture.active;
10     RenderTexture.active = myRenderTarget;
11
12     camera.Render();
13
14     var image = new Texture2D(myRenderTarget.width, myRenderTarget.
           height);
15     image.ReadPixels(new Rect(0, 0, myRenderTarget.width, myRenderTarget
           .height), 0, 0);
16     image.Apply();
17
18     RenderTexture.active = activeRenderTexture;
19     camera.targetTexture = cameraTarget;
20
21     return image;
22 }
```

Listing A.6: Rendern einer Kamera

```
1  public static int Run(
2      this Process process,
3      string application,
4      string arguments,
5      out string output,
6      out string errors)
7  {
8      process.StartInfo = new ProcessStartInfo
9      {
10         CreateNoWindow = true,
11         UseShellExecute = false,
12         RedirectStandardError = true,
13         RedirectStandardOutput = true,
14         FileName = application,
15         Arguments = arguments
16     };
17
18     var outputBuilder = new StringBuilder();
19     var errorsBuilder = new StringBuilder();
20     process.OutputDataReceived += (_, args) => outputBuilder.AppendLine(
           args.Data);
21     process.ErrorDataReceived += (_, args) => errorsBuilder.AppendLine(
           args.Data);
22
23     process.Start();
24     process.BeginOutputReadLine();
25     process.BeginErrorReadLine();
26     process.WaitForExit();
27
28     output = outputBuilder.ToString().TrimEnd();
29     errors = errorsBuilder.ToString().TrimEnd();
30
31     return process.ExitCode;
32 }
```

Listing A.7: Erstellen eines möglichen Git Prozesses

```
1  public static EditorWindowBase TryOpen <T>(bool utility, string title =
       "") where T : EditorWindowBase
2  {
3      if (typeof(T) == typeof(FirstSteps))
4          return EditorWindow.GetWindow <T>(true, title).ShowWindow();
5
6      if (!MegaPintMainSettings.Exists())
7          return EditorWindow.GetWindow <FirstSteps>(true, title).
               ShowWindow();
8
9      return EditorWindow.GetWindow <T>(utility, title).ShowWindow();
10 }
```

Listing A.8: Methode zum öffnen eines Editor-Fensters

```
1  internal class Separator : VisualElement
2  {
3      public new class UxmlFactory : UxmlFactory <Separator, UxmlTraits>
4      {
5      }
6
7      public new class UxmlTraits : VisualElement.UxmlTraits
8      {
9          private readonly UxmlFloatAttributeDescription _space = new() {
               name = "Space", defaultValue = 1.5f};
10
11         public override void Init(
12             VisualElement element,
13             IUxmlAttributes attributes,
14             CreationContext context)
15         {
16             base.Init(element, attributes, context);
17
18             element.style.height = _space.GetValueFromBag(attributes,
                   context);
19
20             element.AddToClassList(StyleSheetClasses.Background.Color.
                   Separator);
21
22             element.SetMargin(4f);
23             element.SetBorderRadius(1f);
24         }
25     }
26 }
```

Listing A.9: UXMLFactory zur Erstellung einer horizontalen Linie

```
1  [CustomPropertyDrawer(typeof(ClassValidation))]
2  internal class ClassValidationDrawer : PropertyDrawer
3  {
4      public override float GetPropertyHeight(SerializedProperty property,
           GUIContent label)
5      {
6          return property.FindPropertyRelative("propertyHeight").
               floatValue;
7      }
8
9      public override void OnGUI(Rect position, SerializedProperty
           property, GUIContent label)
10     {
11         var height = 0f;
12
13         if (!property.FindPropertyRelative("foundClass").boolValue)
14         {
15             var rect = new Rect(position.x, position.y, position.width,
                   EditorGUIUtility.singleLineHeight);
16             height += EditorGUIUtility.singleLineHeight;
17
18             EditorGUI.LabelField(
19                 rect,
20                 "No class could be found with the specified parameters!"
                       ,
21                 new GUIStyle {normal = {textColor = Color.red}});
22         }
23
24         property.FindPropertyRelative("propertyHeight").floatValue =
               height;
25         property.serializedObject.ApplyModifiedProperties();
26     }
27 }
```

Listing A.10: IMGUI Anwendung innerhalb eines PropertyDrawers

B. Dokumente Zur Evaluation

Die folgenden Regeln sind auf alle Szenen Objekte des links genannten Typen anzuwenden.

Item	- Name startet mit **[ITEM]** - Besitzt eine Rigidbody Komponente - Besitzt eine Collider Komponente - Layer ist "**Item**" (Muss innerhalb von 3 Sekunden mit einem Objekt mit dem Layer "**Ground**" kollidieren.)
Ground Object	- Name startet mit **[GR]** - Besitzt keine Rigidbody Komponente - Besitzt eine Collider Komponente - Layer ist "**Ground**"
Debug Object	- Name startet mit **[DEBUG]** - Ist inaktiv
NPC	- Name startet mit **[NPC]** - Hat einen Namen - Leben 50 - 99
Enemy	- Name startet mit **[E]** - Leben 10 - 30
Strong Enemy	- Name startet mit **[ELITE]** - Hat einen Namen - Leben 30 - 60

Abbildung B.1.: Regeln für Szenen-Objekte in den Aufgaben der Fehlerkorrektur von Assets innerhalb der geführten Evaluation

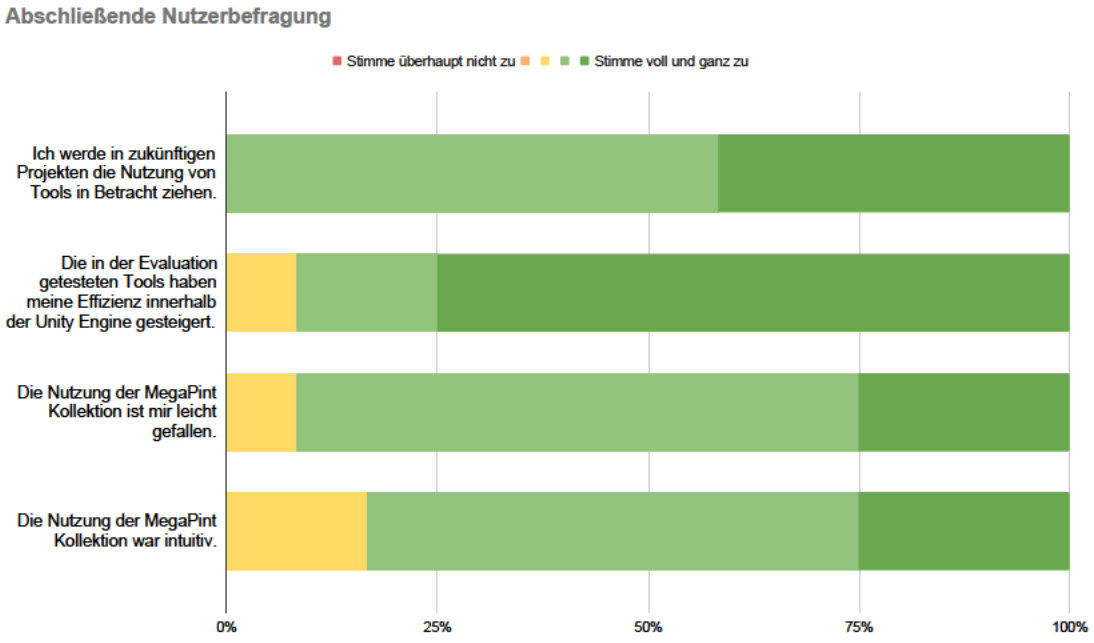

Abbildung B.2.: Ergebnisse der abschließenden Nutzerumfrage

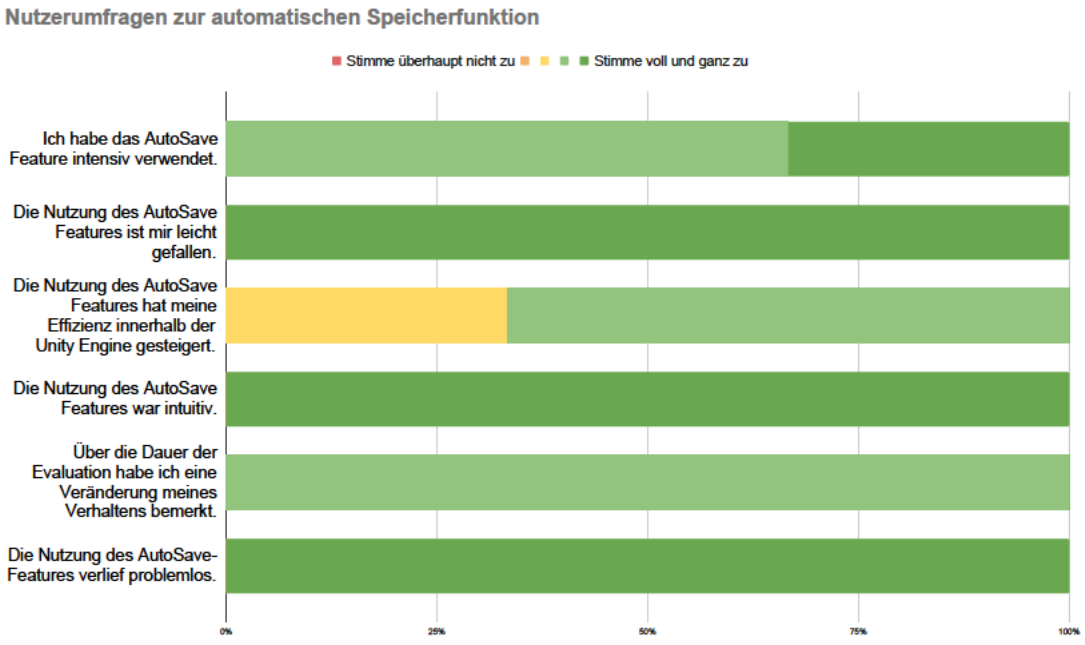

Abbildung B.3.: Umfrageergebnisse zum automatischen Speichern

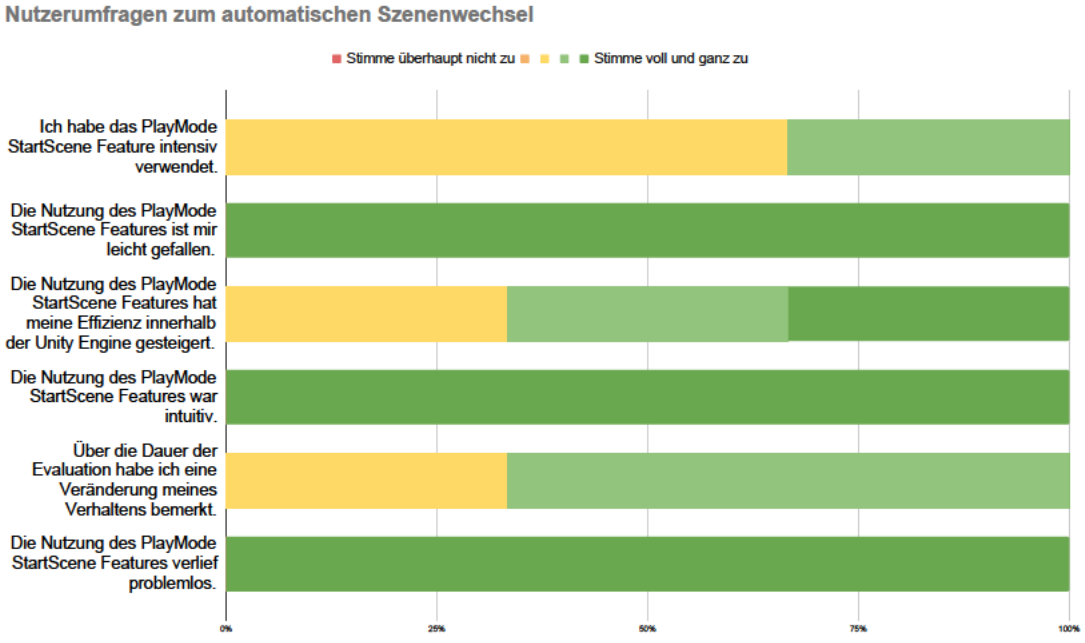

Abbildung B.4.: Umfrageergebnisse zum automatischen Starten der Hauptmenü Szene

Selbstständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Mittweida, den 6. Oktober 2024

Niklas Räder