Masterarbeit

Abschlussarbeit zur Erlangung des akademischen Grades Master of Arts

Game-Engines
get real with Unreal & Co.

Hochschule Ostwestfalen-Lippe
Fachbereich Medienproduktion

Vorgelegt von:
Marc Ottensmann
Matrikelnummer 15248096
Sommersemester 2016

Erstprüfer: Prof. Dr. rer. nat. Guido Falkemeier
Zweitprüfer: Prof. Dr. phil. Frank Lechtenberg

Abgabedatum: 10.08.2016

Hochschule Ostwestfalen-Lippe
University of Applied Sciences

medien**produktion**

Vorbemerkung

Diese Masterarbeit entstand im Rahmen des Studiengangs Medienproduktion an der Hochschule Ostwestfalen – Lippe am Standort Lemgo. Sie ist zugehörig zum Gemeinschaftsprojekt „Technische Visualisierung der neuen NVIDIA® VCA®“ von André Düchting und Marc Ottensmann und entstand im Sommersemester 2016.

Als leidenschaftlicher und regelmäßiger Konsument von Computer- und Videospielen ist mir schon oft der Begriff Spiele- oder Game-Engine begegnet. Die genaue Bedeutung des Begriffs war mir allerdings nie hinreichend bekannt.
Darüber hinaus musste ich zwei interessante Veränderungen innerhalb der Spielebranche feststellen. Zum einen bekamen Spiele von unabhängigen Entwicklern eine immer größere Relevanz, zum anderen wurden die Entwicklungsumgebungen der großen Spiele von den Unternehmen kostenlos jedem zur Verfügung gestellt. Damit werden aktuell entweder komplett neue Spiele entwickelt oder passionierte Fans nutzen die bessere Technik um alte Klassiker neu aufleben zu lassen.

Diese Umstände führten bei mir zwangsläufig zu der Frage, ob es heutzutage tatsächlich noch eine so komplexe Aufgabe ist, ein eigenes Spiel zu entwickeln, oder ob sich die dafür notwendigen Kompetenzen vielleicht verändert haben. Denn welcher faszinierte Spieler hegt nicht den Traum, sein eigenes Spiel nach seinen Vorstellungen zu entwerfen und tatsächlich umzusetzen.

Aus diesem Grund entschloss ich mich, diese in mir aufkommende Frage zum zentralen Aspekt meiner schriftlichen Abschlussarbeit zu machen. Darüber hinaus wollte ich ebenso einmal aufschlüsseln und festhalten, was sich alles hinter dem Begriff Game-Engine verbirgt.

Ich besaß keinerlei fundierte Vorkenntnisse zu diesen Themen und musste mich folglich selbstständig darin einarbeiten. Meine Ausarbeitung ist so gestaltet, dass sie einen allgemeinen Einblick in jedes Themengebiet gewährt. Ich habe mich bemüht den Ansatz so zu wählen, dass auch Leser ohne Hintergrund aus dem Computer- und Videospielbereich der Arbeit folgen können.

Marc Ottensmann, 03.08.2016

Inhaltsverzeichnis

1. Einleitung 1

2. Was ist eine Game-Engine? 3

3. Geschichte der Game-Engines 6
3.1 Die Quake Engines 8
3.2 Die Unreal Engines 9
3.3 Die Frostbite Engine 10
3.4 Die CryENGINE 11
3.5 Die Unity Engine 12

4. Allgemeiner Aufbau einer Game-Engine 14
4.1 Die Hardware-Ebene 16
4.2 Die Geräte Treiber 16
4.3 Das Betriebssystem 17
4.4 Fremdanbieter Anwendungen und Middleware 18
4.5 Plattformabhängige Ebene 20
4.6 Kern-Systeme 20
4.7 Der Ressourcen Manager 21
4.8 Die Rendering-Engine 21
4.8.1 Der low-level renderer 22
4.8.2 Der Szenengraph / culling Optimierungen 23
4.8.3 Visuelle Effekte 25
4.8.4 Das Front End 26
4.9 Werkzeuge zur Leistungsanalyse und Fehlerbehebung 27
4.10 Kollisionserkennung und Physik-Engines 28
4.11 Animationssysteme 29
4.12 Die Mensch-Maschine-Schnittstelle bzw. Controller 31
4.13 Das Audio-System 32
4.14 Online Multiplayer und Vernetzung 33
4.15 Fundamente des Spielprinzips 34
4.16 Spielspezifische Bestandteile 36

5. Game-Engines in der Praxis 37
5.1 Die Unreal Engine 4 37
5.1.1 Die grafische Oberfläche 38
5.1.2 blueprints 39

5.1.3 Die Objekte in der Spielwelt 41
5.1.4 Physik in der Unreal Engine 44
5.1.5 Einen Level erschaffen 48
5.1.6 Charaktere und Animationen 52
5.1.7 KI – Künstliche Intelligenz 56
5.2 Die Unity 5 Engine 60
5.2.1 Die grafische Oberfläche 61
5.2.2 C# in Unity 62
5.2.3 Objekte in 3D und 2D 64
5.2.4 Physik in Unity 68
5.2.5 Landschaften gestalten 72
5.2.6 Animation 75
5.2.7 KI – Künstliche Intelligenz 80

6. Fazit 84

I. Quellenverzeichnis 87
Literaturverzeichnis 87
Internetverzeichnis 88
Abbildungsverzeichnis 91

II. Anhang 93
Eidesstattliche Erklärung: 93
Danksagung 94

1. Einleitung

Im Jahre 1972 veröffentlichte das bis dahin unbekannte Unternehmen Atari unter dem Namen Pong das erste Videospiel.[1] Es war eine simple Tischtennis-Simulation mit einem einfachen Punkt als Ball und zwei Strichen links und rechts, die als Schläger fungierten. War es dem Spieler nicht möglich, den sich bewegenden Ball mit dem eigenen Schläger rechtzeitig zu erreichen, und ihn so zurückzuschlagen, bekam der Gegner einen Punkt. Trotz dieses simplen Spielprinzips erreichte Pong weltweite Bekannt- und Beliebtheit.
Heute und knapp 50 Jahre später ist die Computer- und Videospielindustrie zu einem Multi-Milliarden-Dollar Geschäft geworden.[2] Eine Industrie, die sich stets weiterentwickelt und intern immer wieder neue Maßstäbe setzt.

Zudem sind Computerspiele mittlerweile längst gesellschaftsfähig geworden. Das Klischeebehaftete Bild des bebrillten einsamen Typs der auf seinen PC-Monitor starrt, welches der eine oder andere vielleicht noch hat, ist heutzutage nicht mehr zeitgemäß. Auch in Folge der raschen Entwicklung des Smartphone-Marktes in den letzten Jahren gehören Computerspiele inzwischen zu den beliebtesten Methoden der Freizeitgestaltung. Sie sind ein stetiger Wegbegleiter.[3] Dabei spielt das Alter und das Geschlecht des Spielers keine allzu große Rolle mehr. Laut einer repräsentativen Umfrage aus dem Jahr 2015 spielen 42 Prozent der befragten deutschen Bundesbürger regelmäßig Computer- und Videospiele. Dabei war der Geschlechteranteil mit 43% männlichen zu 42% weiblichen Spielern erstmals beinahe gleich auf. Die meisten Spieler finden sich natürlich im Alterssegment der 14- bis 29-Jährigen (81%), doch auch in der Altersgruppe 30-49 Jahre gab ungefähr jeder Zweite (55%) an regelmäßig Computer- und Videospiele zu nutzen.[4]

Die Erstarkung der Spieleindustrie führte zu tiefen Veränderungen der Marktstruktur. Computerspiele werden weiterhin durch kapitalstarke Großunternehmen entwickelt, die kapitalintensiv umsatzstarke Titel und Serien produzieren (sogenannte A+ Spiele). Jedoch gewinnen sogenannte Indie-Spiele zusehends an Relevanz.[5] Dabei handelt es sich um

1 S. Pong- https://de.wikipedia.org/wiki/Pong (letzter Aufruf: 18.07.2016)

2 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. xiii

3 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 1

4 S. Gaming hat sich in allen Altersgruppen etabliert- https://www.bitkom.org/Presse/Presseinformation/Gaming-hat-sich-in-allen-Altersgruppen-etabliert.html (letzter Aufruf: 18.07.206)

5 Vgl. Kühl, Eike (13.10.2012): PC-Spiele erreichen das nächste Level- http://www.zeit.de/digital/games/2012-10/pc-konsole-videospiele-entwicklung (letzter Aufruf: 18.07.2016)
Und Vgl. Stampfl, Nora (2014): ZWERGENAUFSTAND. INDIE-GAMING-SZENE IN DEUTSCHLAND.- https://www.goethe.de/de/kul/mol/20445752.html (letzter Aufruf: 18.07.2016)

Spiele, die von Einzelpersonen oder kleineren Gruppen entwickelt und selbst veröffentlicht werden. Sie sind unabhängig von größeren Unternehmen oder Geldgebern.[6] Wenn der Bereich der unabhängigen Entwicklung also immer weiter wächst, könnte man fragen, ob sich die Kompetenzen verändert haben, die es braucht, um ein Spiel zu entwickeln.

Wer sich etwas im Bereich der Computer- und Videospiele auskennt, wird vermutlich bereits einmal den Begriff Game-Engine gehört haben. Damit wird oft vereinfacht die Software beschrieben, mit der das Spiel entwickelt wurde. Diese Art von Software hat sich in den letzten Jahren ebenfalls verändert so dass sich die namhaften Exemplare immer weiter öffnen und einer größeren Masse zur Verfügung stehen. Ein paar der wirklich großen Namen, die mitunter den Rang des Industriestandards innehaben, sind mittlerweile völlig kostenlos für Jedermann herunterzuladen. Dies lässt erneut die Frage aufkommen, ob die Spieleentwicklung nur für erfahrene Leute zugänglich ist, die sich gut in der Informationstechnik auskennen. Denn frei im Internet verfügbare Software ist in der Regel für jede Art von Benutzer gedacht, den Erfahrenen und den Unerfahrenen.

Daher soll es in dieser Arbeit um die Beantwortung eben jener Frage gehen: Kann heutzutage, mit dem aktuellen Stand der technischen Lösungen, jeder sein eigenes Spiel entwickeln?

Um diese Frage zu beantworten, wird zunächst einmal darauf eingegangen, was eine Game-Engine überhaupt ist, wozu diese gebraucht wird und wie sie im Allgemeinen funktioniert. Dazu werden der Aufbau und das generelle Konzept einer Engine betrachtet und Stück für Stück analysiert, um die Aufgaben der jeweiligen Einzelteile klar herauszustellen. Eine konkrete Betrachtung einer speziellen Engine wird an dieser Stelle noch nicht vorgenommen. Es geht erst einmal darum, einen allgemeinen Einblick in diesen Bereich der Softwareentwicklung zu bekommen. Zusätzlich erfolgt eine kleine historische Betrachtung der Entwicklung von Game-Engines. Anschließend werden anhand zweier konkreter Beispiele tiefere Einblicke in Funktionalität von Game-Engines gewährt. Vorab sei an dieser Stelle bereits gesagt, dass es für den hier gewählten Rahmen leider nicht möglich sein wird, jeden Teilbereich näher zu betrachten. Es wird daher eine Auswahl an essentiellen und interessanten Bereichen getroffen. Am Ende erfolgt ein abschließendes Fazit, das sich mit dem allgemeinen Prozess der Spieleentwicklung und der Beantwortung obiger einleitender Frage auseinandersetzt.

6 Daher die Bezeichnung Indie, vom englischen Begriff ***independent*** für unabhängig.

2. Was ist eine Game-Engine?

Ganz allgemein betrachtet ist eine Game-Engine eine Art von Software, die den Ablauf und die Darstellung von Computerspielen auf dem jeweiligen Medium bzw. der jeweiligen Plattform (PC, Konsole, mobile Endgeräte) steuert. Sie ist also dafür verantwortlich, dass die Spiele überhaupt funktionieren. Der Begriff bezeichnet heutzutage allerdings ebenfalls die komplette Entwicklungsumgebung, in der die Spiele entstehen. Also eine Sammlung von Werkzeugen und der technische Rahmen, mit denen Spieleentwickler arbeiten. Ohne eine Game-Engine könnte ein Computerspiel also weder funktionieren, noch entwickelt werden. Das Spiel ist abhängig von der jeweiligen Engine, aber deutlich von dieser abzugrenzen. Ein Spiel ist größer als seine Engine, auch wenn die Engine vorher gegeben sein muss, damit das Spiel entwickelt und ausgeführt werden kann.[7] „This is because an engine is only a part of a game, just as the heart is only a part of the body. But like the heart, the engine is an essential part.“[8]

Der Begriff Game-Engine wurde erstmals in den frühen 90er Jahren in Verbindung mit dem sehr beliebten Spiel DOOM von id Software verwendet.[9] DOOM war so strukturiert, dass es eine klare Trennung zwischen den Kernelementen der Software, z.B. die generelle Erzeugung dreidimensionaler Grafiken, der Kollisionserkennung oder dem Audiosystem und den spielspezifischen Inhalten, wie die Welt oder die Regeln des Spiels, gab. Dadurch war es den Entwicklern im Nachhinein möglich, durch verhältnismäßig einfaches Austauschen der Inhalte ein neues und anderes Spiel zu entwickeln, ohne viel an den Kernelementen der Software verändern zu müssen.[10] Eine Vorgehensweise, die bis heute Bestand hat und weiterhin angewendet wird. Zusätzlich entsteht den Entwicklern der Game-Engines durch die Lizensierung ihrer Software eine zweite valide Einkommensquelle.

Sich dessen bewusst, könnte man nun denken, dass eine Game-Engine mit einem DVD-Player vergleichbar ist. Egal welche Art von Film abgespielt werden soll, es funktioniert in der Regel. Für den technischen Rahmen macht der eigent-

---

7 Vgl. Thorn, Alan (2010): Game Engine Design and Implementation: Foundations of Game Development. Jones & Bartlett Publ Inc, S. 3-4

8 Thorn, Alan (2010): Game Engine Design and Implementation: Foundations of Game Development. Jones & Bartlett Publ Inc, S. 3

9 S. Doom- https://de.wikipedia.org/wiki/Doom (letzter Aufruf: 04.07.2016)
Und id Software- http://www.idsoftware.com/de-de (letzter Aufruf: 04.07.2016)

10 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 11

liche Inhalt also keinen Unterschied. Diese Vorstellung beschreibt zwar das Ideal, doch sie entspricht bei einer Game-Engine leider (noch) nicht der Realität. Eine bessere Analogie für den Fall der Game-Engine wäre ein Theater, bei dem das Bauwerk samt Bühne und den fest verbauten Elementen die Engine repräsentiert. Alles Veränderbare, wie das Theaterstück, die Schauspieler, die Kostüme usw. stehen für das Spiel und dessen Inhalt. Ist die Bühne also erst einmal vorhanden kann auf ihr prinzipiell alles gespielt werden, was dem Organisator beliebt. Auch dies wäre wieder die Idealvorstellung. In der Realität aber gibt es weitere Faktoren, die berücksichtigt werden müssen. Ist die Bühne überhaupt groß genug für das geplante Stück? Sind genügend Sitzplätze vorhanden für die zu erwartenden Zuschauer? Ist die Lichtanlage ausreichend?

Es gibt nicht die eine Bühne bzw. das eine Theater, in dem alles möglich ist. Genauso verhält es sich mit den Game-Engines. Auch hier gibt es nicht die eine Engine, die alles auf dem gleichen Niveau realisieren kann. Es handelt sich um teils hochgradig spezifizierte Software, die ursprünglich für ein einziges Spiel auf einer einzigen Plattform entwickelt und erst im Nachhinein weiter ausgebaut und erweitert wurde. Das geht mittlerweile so weit, dass sich bestimmte Engines auch bestimmten Genres zuordnen lassen, es aber nicht die eine Engine gibt, die das gleiche Ergebnis bei jedem Spiel in jedem Genre erzielen kann. Man kann sagen, dass je breiter oder unspezifischer eine Engine aufgebaut ist, desto suboptimaler ist sie, um ein bestimmtes Genre auf einer bestimmten Plattform zu realisieren.[11]

Diese Problematik ist in der Software-Entwicklung nichts Neues oder Ungewöhnliches. Um eine effiziente Software zu entwickeln, müssen Kompromisse gemacht und zwischen notwendigen und zusätzlichen Leistungskriterien unterschieden werden. Angenommen ein Autohersteller bringt ein neues Modell auf den Markt, das in der Lage sein soll, kurze Strecken zu schwimmen. Das wäre sicherlich eine interessante Angelegenheit aus Sicht der Marketingabteilung, aber für die Ingenieure und Konstrukteure auch eine enorme Herausforderung. Im normalen Straßenverkehr wäre die Schwimmfähigkeit darüber hinaus völlig nutzlos. Dies degradiert sie zu einer Funktion die aus Gründen der Effizienz verworfen werden sollte. Ist eine Game-Engine darauf spezialisiert enge Innenräume darzustellen besitzt sie verschiedene Möglichkeiten und Techniken, um zu bestimmen welche Gegenstände im jeweiligen Moment sichtbar sind und welche nicht. Folglich würde sie sich eher schwer damit tun weite Außenareale mit einer Vielzahl an Vegetation darstellen zu müssen. Die notwendigen Berechnungen, um die Sichtbarkeit der einzelnen Bäume festzustellen, wären einfach zu aufwendig. Andersherum fehlen einer Engine, die auf riesige Außenareale

11 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 11-12

spezialisiert ist, genau diese Techniken zur Bestimmung der Sichtbarkeit, weil sie in der Regel keine Überlagerung bzw. Verdeckung berücksichtigt. Dafür besitzt sie wiederum erweiterte Techniken zur Verwaltung der ***level-of-detail*** (***LOD***)[12] eben dieser Außenareale. Ein System, das dafür sorgt dass Objekte die weit entfernt sind mit weniger Detailgenauigkeit, d.h. wesentlich schneller, dargestellt werden, während Objekte die sich nahe zum Spieler befinden so detailgetreu wie möglich angezeigt werden.[13]

Natürlich ist diese ganze Zuordnung einer Engine zu einem Genre keine 100%-ige Festlegung. Man kann sich eine Skala der Wiederverwertung vorstellen, auf der jede Engine eingeordnet werden kann. Diese Skala reicht von „kann nur benutzt werden, um ein spezifisches Spiel zu kreieren" bis hin zu „kann benutzt werden, um jedes vorstellbare Spiel zu entwickeln." Letzteres ist zum gegenwärtigen Zeitpunkt allerdings keine Realität, sondern lediglich ein abgrenzender Extremwert. Darüber hinaus steht das Rad der Weiterentwicklung natürlich nicht still. Vom stetigen Fortschritt der Computer Hardware, spezialisierten Grafikkarten und besseren, effizienteren Algorithmen profitieren selbstverständlich auch die Entwickler der Game-Engines. Bis die beschriebene ideale Spiele-Engine existiert, weichen die harten Abgrenzungen von früher allerdings schon mal auf. So ist es jetzt schon möglich mit einer Engine, die beispielsweise ursprünglich für einen ***first-person shooter***[14] konzipiert wurde, ein absolut brauchbares Strategiespiel zu entwickeln. Dennoch bleiben die ebenfalls oben erwähnten Kompromisse bestehen. Ein Spiel kann nur davon profitieren, wenn die zugrundeliegende Engine auf das Spiel selbst und seine Plattform abgestimmt und angepasst ist.[15]

Dies war ein kurzer und allgemeiner Einstieg in das große Feld der Game-Engines, um zunächst einmal zu klären, was sich eigentlich hinter diesem Begriff verbirgt. Die hier ganz kurz angerissene Historie wird im nächsten Kapitel fortgeführt. Der detaillierte Aufbau einer Game-Engine folgt im übernächsten Kapitel.

---

12 S. Level of Detail- https://de.wikipedia.org/wiki/Level_of_Detail (letzter Aufruf: 04.07.2016)

13 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 12

14 Der ***first-person shooter*** bezeichnet eine Kategorie der Computerspiele, in der sich der Spieler frei agierend in der Egoperspektive durch eine dreidimensionale Spielwelt bewegt und Gegner mit Schusswaffen bekämpft. S. Ego-Shooter- https://de.wikipedia.org/wiki/Ego-Shooter (letzter Aufruf: 04.08.2016)

15 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 12-13

3. Geschichte der Game-Engines

Wie im vorherigen Kapitel bereits erwähnt, tauchte der Begriff Game-Engine erstmals ungefähr in den frühen 90er Jahren mit dem Erscheinen des Spiels DOOM auf. Die Doom-Engine war streng genommen keine Engine, die tatsächlich in der Lage war eine dreidimensionale Welt abzubilden (3D Engine). Sie schaffte es aber dem Spieler durch ihre Verfahren und Darstellungen das Gefühl zu geben, er würde sich im dreidimensionalen Raum bewegen. Zudem war sie der Grundstein für das Lizensieren von austauschbaren Software-Paketen in der Computerspielbranche.

Im Jahr 1992 entwickelte NovaLogic die Voxel Engine. Mit ihr war es erstmals möglich, volumetrische Objekte als 3D-Grafiken darzustellen. Davor wurden nur Vektorgrafiken benutzt, welche weniger detailreich waren und längere Berechnungszeiten erforderten.[16] Ein ***voxel*** meint dabei einen Bildpunkt im dreidimensionalen Raum. Sozusagen ein Pixel mit Volumen, daher auch die Bezeichnung ***voxel***.[17]

Später im Jahr 1993 wurde Duke Nukem 3D veröffentlicht, welches auf Grundlage der Build Engine entwickelt wurde. Ähnlich wie bei DOOM wurde auch hier eine eigentliche 2D-Welt als 3D-Welt verkauft. Die Entwickler erreichten dies, indem sie die Höhe der Objekte oder der Welt manipulierten. Dinge, die weiter vom Spieler entfernt waren, schienen kleiner als die die sich in seiner Nähe befanden.[18]

Die XnGine(1995) war die erste Engine, die in der Lage war, eine tatsächliche dreidimensionale Welt zu erzeugen. Sie war kompatibel mit hochauflösenden Grafiken und den Grafikkarten vom Hersteller 3dfx, welcher zu seiner Zeit den ersten brauchbaren 3D-Grafikchipsatz für den nichtprofessionellen Bereich entwickelte und herstellte.[19]

id Software, die Entwickler von DOOM, stellten 1996 die Quake-Engine als eine eigene Engine zur Darstellung einer dreidimensionaler Welt vor. Sie hatte die bis dahin einzigarte

---

16 Vgl. Sarathi Paul, Partha; Goon, Surajit; Bhattacharya, Abhishek (2012): HISTORY AND COMPARATIVE STUDY OF MODERN GAME ENGINES- http://bipublication.com/files/IJCMS-V3I2-2012-07.pdf S. 246 (letzter Aufruf: 04.07.2016)

17 S. Voxel (volume pixel)- http://www.itwissen.info/definition/lexikon/Voxel-volume-pixel-Volumenpixel.html (letzter Aufruf: 04.07.2016)

18 Vgl. Sarathi Paul, Partha; Goon, Surajit; Bhattacharya, Abhishek (2012): HISTORY AND COMPARATIVE STUDY OF MODERN GAME ENGINES- http://bipublication.com/files/IJCMS-V3I2-2012-07.pdf S. 246 (letzter Aufruf: 04.07.2016)

19 Vgl. Ebd.
Und 3dfx- https://de.wikipedia.org/wiki/3dfx (letzter Aufruf: 04.07.2016)

Fähigkeit, Objekte oder ganze Teile der Welt, die für den Spieler nicht sichtbar waren, aus dem Berechnungsprozess auszuschließen. Dies gelang indem die Tiefeninformationen der Objekte bei der Feststellung der Sichtbarkeit mit berücksichtigt wurden.[20]

Mit GoldSRC (1998) wurde eine neue Ära für PC Spiele eingeleitet. Diese Engine war erstmals in der Lage OpenGL und Direct3D zu unterstützen. Beides sind Programmierschnittstellen, die noch heute in der Computergrafik Anwendung finden und stetig weiterentwickelt werden.[21] OpenGL verfolgt dabei einen programmiersprachenübergreifenden Ansatz, während Direct3D ein Bestandteil von DirectX ist, also Microsofts Programmierschnittstelle für multimediale Anwendungen innerhalb des Windows Betriebssystem und der Spielekonsole Xbox.

Ebenfalls im Jahr 1998 wurde die erste Version einer der bis heute populärsten Engines veröffentlicht, die Unreal Engine. Sie legte das Fundament für die Unreal-Spieleserie und die Unreal Tournament-Spieleserie. Die Engine zeichnete sich durch ihre eigene Skriptsprache, UnrealScript genannt, und einen Welteneditor namens UnrealEd aus, also zwei eigenständige Systeme zur Programmierung von Spiellogiken bzw. dem Bau der Spielwelt.[22]

Der nächste Meilenstein der Entwicklung folgte 2001 mit der Geodmod Engine und dem Spiel Red Faction. Damit war es möglich, die Geometrie der Objekte oder der Welt nachträglich, also zur Laufzeit des Spiels, zu verändern. Traf eine Rakete eine Wand, wurde die Form der Rakete von der Wand abgezogen, um so ein adäquates Loch an dieser Stelle zu erschaffen.[23]

Die Serious Engine (ebenfalls 2001 erschienen) war dafür konstruiert worden, um große Plätze und eine hohe Anzahl an Charakteren gleichzeitig darzustellen. Damit wurde das erste Spiel der Serious Sam-Reihe entwickelt, welche sich durch immense Mengen an Gegnern auszeichnet die auf den Spieler zuströmen.[24]
Wie man sieht sind namentliche Ähnlichkeiten zwischen den Engines und dem jeweils ersten damit entwickelten Spiel nicht selten. Dies hat den einfachen Grund, dass es in

20 Vgl. Sarathi Paul, Partha; Goon, Surajit; Bhattacharya, Abhishek (2012): HISTORY AND COMPARATIVE STUDY OF MODERN GAME ENGINES- http://bipublication.com/files/IJCMS-V3I2-2012-07.pdf S. 246-247 (letzter Aufruf: 04.07.2016)

21 Vgl. Ebd. S. 247

22 Vgl. Ebd.

23 Vgl. Ebd.

24 Vgl. Ebd.

den Anfängen (d.h. vor dem Erscheinen von DOOM) so war, dass ein Entwicklerstudio bei jedem neuen Spiel bei null anfangen musste, das heißt ohne das bereits vorhandene Ressourcen genutzt werden konnten. Alle speziellen Eigenschaften, die das Spiel aufweisen sollte, mussten zunächst einmal auf Seiten der Software entwickelt werden. Vor der Einführung des Konzeptes einer Game-Engine wurden auch nachfolgende Teile zu bereits erhältlichen Spielen wieder komplett von Anfang an aufgebaut. Daran kann man den unglaublichen Mehrwert des Konzeptes erkennen, auch dann noch wenn die entwickelte Engine nur studiointern verwendet und nicht an andere Nutzer lizensiert wird.

Mit einem letzten kleinen Sprung ins Jahr 2006 wird dieser vereinfachte Überblick der Entwicklung der Game-Engines beendet. In diesem Jahr erschien die Rockstar Advanced Game Engine, welche eine Rahmenstruktur für die Darstellung der Spielwelt, eine Physiksimulation, eigene Engines für Audio, Netzwerk und Animationen und eine eigene Skriptsprache in sich vereinte (in späteren Kapiteln erfolgt mehr zu diesen einzelnen Bestandteilen einer Engine).[25] Damit war die Engine aus dem Hause Rockstar der Vorläufer der heutigen Engines, die eben diese Bestandteile und noch mehr heutzutage direkt mit sich bringen.

Damit ist dieser Teil des Kapitels beendet. Nun folgt eine etwas genauere Betrachtung der bekanntesten und aktuellsten Game-Engines.

3.1 Die Quake Engines[26]

Der erste ***first-person shooter*** (kurz: ***FPS***), der als solcher bekannt wurde und damit das Genre verbreitet hat, war das Spiel Castle Wolfenstein 3D (1992). Ebenfalls entwickelt von id Software war es der direkte Vorgänger vom bereits mehrfach erwähnten DOOM, bei dem erstmals eine Game-Engine verwendet wurde. Auf DOOM folgte die Spieleserie Quake, Quake II und Quake III, alle ebenfalls von id Software entwickelt und immer mit einer ähnlichen Struktur und Architektur. Dank dieser Strukturen kann genau das Verfahren eingesetzt werden, das im 2. Kapitel bereits beschrieben wurde. Wenn ein neues Spiel oder ein Nachfolger entstehen soll, wird nicht von Null angefangen, sondern es wird dasselbe Fundament (die Engine) wie beim vorherigen Spiel verwendet. Dieses wird dann den Bedürfnissen des neuen Titels entsprechend angepasst, verändert und erweitert. Die Technologie hinter den Quake Spielen findet auch deshalb hier Erwähnung, weil sie

25 Vgl. Sarathi Paul, Partha; Goon, Surajit; Bhattacharya, Abhishek (2012): HISTORY AND COMPARATIVE STUDY OF MODERN GAME ENGINES- http://bipublication.com/files/IJCMS-V3I2-2012-07.pdf S. 247 (letzter Aufruf: 04.07.2016)

26 Vgl. Ebd. S. 26-27

im Laufe der Jahre verwendet wurde um noch weitere Spiele und sogar weitere Engines zu entwickeln. Zum Beispiel die Source-Engine von Entwickler Valve[27], dem Schöpfer der beliebten Spieleserie Half-Life, hat entfernte Wurzeln in den Quake-Engines. Der Quellcode der Quake Spiele ist frei verfügbar und wurde unter der GNU General Public License (kurz: GNU GPL[28]) veröffentlicht. Er kann bei Bedarf heruntergeladen, verwendet, verändert und wieder neu verbreitet werden und dient daher heute noch als Lerninstrument.[29]

3.2 Die Unreal Engines[30]

Im Jahr 1998 brachte Epic Games[31] das für das ***FPS*** Genre legendäre und bahnbrechende Spiel Unreal auf den Markt. In der Fachpresse wurde es nicht nur für seine Grafik und Atmosphäre, sondern auch für die künstliche Intelligenz der Gegner und das Spielprinzip an sich häufig gelobt. Damit wurde die Unreal Engine sofort zum größten Konkurrenten für die Entwickler der Quake-Engine. Der direkte Nachfolger der Engine, die Unreal Engine 2 (UE2), war die Grundlage für das noch beliebtere Spiel Unreal Tournament 2004, welches vor allem in der sogenannten ***modding***-Szene beliebt war. Dabei werden Spiele in gewisser Weise von der jeweiligen Community modifiziert und dann den Spielern zur Verfügung gestellt. Ein naheliegendes Beispiel wäre das Austauschen der Gegnertypen. Wenn der Spieler im ursprünglichen Spiel gegen Aliens kämpfen muss so könnte er sich durch eine Modifizierung nun gegen Dinosaurier zur Wehr setzen müssen. Dies wäre natürlich eine verhältnismäßig simple Modifizierung und steht nicht stellvertretend für das Können und die Kreativität der ***modding***-Szene, vermittelt aber das grundlegende Prinzip. Auch wenn Unreal Tournament bereits im Jahr 2004 erschienen ist, werden dafür heute immer noch ***mods*** angefertigt und verbreitet.[32]
Doch die Engine wurde nicht nur für das eigentliche Hauptspiel oder Abwandlungen davon verwendet. Sie wurde auch für komplett andere Projekte benutzt. Darunter natürlich andere namhafte Computerspiele, aber auch Forschungsprojekte an Universitäten oder sogar Übungssoftware für das U.S.-Militär. Darüber hinaus gelang mit der

---

27 S. Valve- http://www.valvesoftware.com/ (letzter Aufruf: 05.07.2016)

28 S. GNU GPL- http://www.gnu.de/documents/gpl.de.html (letzter Aufruf: 05.07.2016)

29 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 27

30 Vgl. Ebd. S. 27-28

31 S. Epic Games- https://epicgames.com/ (letzter Aufruf: 05.07.2016)

32 S. MODDB- http://www.moddb.com/games/unreal-tournament-2004/mods (letzter Aufruf: 05.07.2016)

Unreal Engine 2 der erste erfolgreiche Versuch eine Engine, ursprünglich für Computerspiele konzipiert, auch auf die damals aktuelle Konsolen Generation zu exportieren. Die vierte Version (UE4) ist die momentan aktuelle Version der Engine. Sie bringt eine nie dagewesene Fülle an Werkzeugen und Eigenschaften mit, teils die besten in der derzeitigen Industrie.[33] Besonders erwähnenswert ist die grafische Benutzeroberfläche zur Erstellung von Spiellogik, genannt Kismet. Diese erleichtert die Entwicklung komplexer Spielabläufe. Eine der großen Stärken der Unreal Engine ist, dass sie sich gut eignet, um schnelle und funktionsfähige Prototypen zu erstellen, ohne sich mit nur einer Zeile Quellcode beschäftigen zu müssen. Für dieses „einfach zu benutzen“-Merkmal ist die Unreal Engine bekannt geworden. Dabei ist sie aber keineswegs perfekt oder trivial in der Benutzung. Jedes Studio, das mit der Engine arbeitet, wird sie genau seinen Bedürfnissen anpassen, so wie es generell üblich ist sei es durch Hinzukaufen von Software von anderen Anbietern oder der direkten Veränderungen auf Quellcode Ebene. Dennoch sei nochmals erwähnt dass die Unreal Engine ein mächtiges Werkzeug zur Entwicklung kommerzieller Spiele ist und von Epic Games darüber hinaus für den nicht-kommerziellen Bereich völlig kostenlos zur Verfügung gestellt wird.

3.3 Die Frostbite Engine[34]

Die Frostbite-Engine entstand aus den Bemühungen des Entwicklerstudios DICE eine Engine für das 2006 erschienene Battlefield: Bad Company zu erschaffen. Seitdem ist die Frostbite-Engine sozusagen die Hausmarke des weltweit tätigen und bekannten Herstellers und Verlegers (oder wie in der Branche eher geläufig der engl. Begriff ***publisher***) von Computer- und Konsolenspielen: Electronic Arts, kurz EA. Das börsennotierte Unternehmen ist vor allem bekannt für die große Reihe an Sportspieleserien wie Madden NFL, NBA Live oder FIFA, aber auch für andere Serien wie Battlefield, Mass Effect und Need for Speed. Die Frostbite-Engine beinhaltet drei wichtige Hauptkomponenten: FrostEd, Backend Services und Runtime.[35] FrostEd ist die Desktopanwendung zur Entwicklung der Spiele. Alle an der Entwicklung Beteiligten, ob Designer, Animatoren oder Programmierer arbeiten mit derselben Software, derselben grafischen Oberfläche. Dabei liegt ein großer Fokus auf der Echtzeitberechnung. Jeder Mitarbeiter soll die Änderungen, die er vornimmt, sofort in der aktuellen Spieleversion sehen können. Backend Services bezeichnet das System zur

---

33 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 27

34 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 28

35 S. This is Frostbite- http://www.frostbite.com/about/this-is-frostbite/ (letzter Aufruf: 05.07.2016)

Datenverwaltung hinter der Desktopanwendung. Diese gewährleistet die ganzheitliche und aktuelle Informationsversorgung aller Entwickler, auch bei sich stetig verändernden Projekten, Teams und Anforderungen. Runtime, als dritte Komponente der Frostbite-Engine, dient der Sicherstellung von Plattformkompatibilität. Vom Computer über Konsolen bis hin zu den mobilen Endgeräten soll nach Möglichkeit nur ein Spiel entwickelt werden müssen.

3.4 Die CryENGINE[36]

Die CryENGINE wurde vom deutschen Entwicklerstudio Crytek entwickelt und war ursprünglich als eine technische Demo für Grafikprozessorhersteller Nvidia gedacht. Sie sollte zeigen, was mit der damals aktuellen Hardware möglich war. Als das Potenzial erkannt wurde, entstand aus der Demo allerdings eine komplette Engine mit Hauptaugenmerk auf die grafische Darstellung der Spielwelt. Das erste Spiel, dass mit der CryENGINE entwickelt wurde, war das 2004 erschienene Far Cry. Far Cry stellte einen grafischen Meilenstein in der Geschichte der Computerspiele dar, auch aufgrund der für damalige Verhältnisse realistischen Darstellung von Wasser und der hohen Sichtweite innerhalb eines Levels.
Mit dem Erscheinen der CryENGINE 2 und dem dazugehörigen Spiel Crysis, setzte Crytek ihren Kurs fort. Crysis stach ebenfalls bei der grafischen Qualität hervor und brachte die damalige Hardwaregeneration an ihre Grenzen.
Nur zwei Jahre später folgte die dritte Generation der Engine, CryENGINE 3, die den Schritt auf den Konsolenmarkt wagte. Weiterhin lag das Hauptaugenmerk auf der grafischen Qualität der Spiele. Dies bedeutet jedoch nicht, dass es mit der CryENGINE nicht möglich ist auch inhaltlich überzeugende Spiele zu entwickeln.
Seit Mitte 2011 gibt es für die CryENGINE ebenfalls eine kostenlose Version für jeden Interessierten auf der offiziellen Webseite herunterzuladen, aktuell mit einem „Bezahl was du möchtest/was es dir wert ist“-Modell.[37] Auch wer nichts bezahlen möchte oder kann, bekommt vollen Zugriff auf den Quellcode, die Werkzeuge und Bestandteile der kompletten Engine. Das sich aktuell noch in der Entwicklung befindende Spiel Kingdom Come: Deliverance des tschechischen Entwicklerstudios Warhorse studios verwendet ebenfalls die CryENGINE. Dieses Spiel hat einen hochgradig detailorientierten und realitätsnahen Ansatz und möchte das mittelalterliche Böhmen so exakt wie möglich darstellen. Dafür sind viele

36 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 28-29

37 S. GET CRYENGINE- https://www.cryengine.com/get-cryengine (letzter Aufruf: 05.07.2016)

weite Außenareale gefüllt mit allerhand Vegetation nötig, eine weitere Stärke der CryENGINE.[38]

3.5 Die Unity Engine[39]

Unity ermöglicht die Entwicklung plattformunabhängiger 3D und 2D Spiele. Das beinhaltet sowohl mobile Endgeräte, aktuelle und vergangene Konsolengenerationen als auch Desktop Computer. Darüber hinaus existiert ein Webplayer um entsprechende Unity-basierte Erzeugnisse in den gängigen Webbrowsern abspielen zu können. Neben dieser Fülle an unterstützten Plattformen stützt sich die Unity-Engine auch auf ihren „einfach zu benutzen"-Charakter. Die Engine bringt einen einfach zu handhabenden Editor für die Spielwelt mit sich, in dem Objekte erschaffen und manipuliert werden können. Das Ergebnis ist dann entweder dort im Editor sichtbar oder kann direkt auf der Zielplattform betrachtet werden. Außerdem bietet Unity eine Reihe an Werkzeugen um die Leistung des jeweiligen Spiels auf den verschiedenen Plattformen zu analysieren und ggf. zu optimieren. Etwaige Änderungen und Kompromisse werden dabei plattformabhängig vorgenommen. Das heißt, angenommen Plattform A kann eine gewisse Spezifikation nicht ausführen, muss diese angepasst werden. Das führt unter Umständen zu einem anderen (vielleicht sogar schlechteren) Ergebnis. Plattform B ist allerdings in der Lage, das Spiel so auszuführen wie die Entwickler es ursprünglich beabsichtigt haben. Dann ist es in der Engine möglich den Code und die Spezifikationen so anzupassen, dass Plattform A mit anderen Parametern arbeitet als Plattform B. Ein ebenfalls interessanter Bestandteil der Unity-Engine ist ***Boo***, ein System zur Neuverknüpfung von Animationen. Dieses erlaubt die Übertragung der Animation eines spezifischen Charakters auf einen komplett anderen Charakter.

Mit den hier betrachteten Exemplaren ist die Liste aller Engines, die in der Industrie oder auch im privaten Bereich Anwendung finden, natürlich nicht vollständig. Es gibt darüber hinaus beispielsweise noch die PhyreEngine. Diese von Sony entwickelte Engine dient ausschließlich dazu Spiele für Sonys komplettes Konsolenpaket (Playstation 3, Playstation 4, Playstation Vita usw.) zu entwickeln, und

---

38 Vgl. Weiß, Stefan: Kingdom Come Deliverance. Auf zur Schnitzeljagd nach Böhmen: Wir haben die Beta ausführlich gespielt!, in: PC Games 283 (03/2016), S. 20-23

Und Vgl. Reuther, Philipp: Kingdom Come: Deliverance – Cryengine am Limit, in PC Games 283 (03/2016), S. 24-25

39 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 29-30

wird nur an von Sony selbst lizensierte Hersteller und Entwickler herausgegeben. Außerdem gibt es eine Vielzahl verschiedener Open Source Anwendungen (siehe z.B. OGRE[40]) oder 2D Spiele Engines für Benutzer, die keine Programmierkenntnisse besitzen (z.B. das vom MIT Media Lab entwickelte Scratch[41]). Diese erreichen natürlich nicht das gleiche Leistungsspektrum wie die Vertreter aus der Industrie, aber ein jeder der sich mit der Materie auseinandersetzen möchte, wird im Internet etwas finden das passend für die jeweiligen Ansprüche und Fähigkeiten ist.

Als letzten Punkt in diesem Kapitel sei der Vollständigkeit halber noch einmal das nicht zu unterschätzende Feld der Engines erwähnt, die der Öffentlichkeit nicht unbedingt zur Verfügung stehen. Gemeint sind all die technischen Umsetzungen, Modifikationen und Implementierungen, die ausschließlich intern in den Entwicklerstudios eingesetzt werden.[42] Dabei handelt es sich durchaus um Komplettpakete, die den in der Industrie weiter verbreiteten Engines in Nichts nachstehen, die aus verschiedenen Gründen aber ihren Entwicklungsort nie verlassen haben. Beispielsweise sind viele von EAs Strategiespielen mit einer Engine entstanden die Sage genannt wird. Letzten Endes begannen die jetzt großen und verbreiteten Engines auch erst mal als studiointerne Lösung bei der Entwicklung eines neuen Spiels.

Damit soll dieses Kapitel abgeschlossen sein. Es hat einen kleinen Einblick in die Historie der Game-Engines geliefert und sich anschließend noch einmal etwas ausführlicher mit den aktuellsten und bekanntesten Vertretern der Branche beschäftigt. Im Anschluss soll nun der Frage nachgegangen werden wie eine Game-Engine im Allgemeinen überhaupt aufgebaut ist.

---

40 S. OGRE3D- http://www.ogre3d.org/ (letzter Aufruf: 06.07.2016)

41 S. Scratch- https://scratch.mit.edu/ (letzter Aufruf: 05.07.2016)
Und Scratch (Programmiersprache)- https://de.wikipedia.org/wiki/Scratch_(Programmiersprache) (letzter Aufruf: 05.07.2016)

42 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 31

4. Allgemeiner Aufbau einer Game-Engine

In den letzten zwei Kapiteln wurde sich mit der Frage beschäftigt, was eine Game-Engine überhaupt ist und die Historie und Entwicklung einiger Engines wurde näher betrachtet. In diesem Kapitel steht nun der generelle Aufbau, die Architektur einer Engine im Fokus.

Wie bereits in Kapitel 2 erwähnt, besteht eine Game-Engine heutzutage in der Regel aus einer Umgebungsentwicklung, mit der die Entwickler das Spiel kreieren, und einer Komponente die zur Laufzeit des Spiels auf der jeweiligen Plattform ausgeführt wird. Da die Umgebungsentwicklungen ausgewählter Engines noch näher in späteren Kapiteln besprochen werden, soll es hier um die Laufzeitkomponente und deren Aufbau gehen.[43] Das bedeutet aber nicht, dass alle Systeme und Elemente einer Engine, die in diesem Kapitel vorgestellt werden, ausschließlich relevant sind für die Zeit in der das Spiel ausgeführt wird. Es gibt exklusive Teile, die entweder nur der Entwicklung oder nur dem Ausführen dienen, aber ebenso gibt es einige Überschneidungen die in beiden Stadien relevant sind.

Die nachfolgende Abbildung auf der nächsten Seite zeigt alle größeren Elemente, die in der Laufzeitkomponente einer Engine zusammen kommen. Wie man sehen kann, sind Game-Engines durchaus komplexe Softwaresysteme.

Wie bei vielen komplexen Systemen sind auch Engines in einer Ebenen-Architektur aufgebaut, in der normalerweise die oberen Ebenen abhängig sind von den darunterliegenden. Ist dies anders herum, also sind untere Ebenen abhängig von darüber befindlichen Ebenen, spricht man von einer zirkularen Abhängigkeit.[44] Solche Abhängigkeiten sollten im Softwaredesign vermieden werden, da sie zu problematischen Verknüpfungen führen. Einzelne Module einer Ebene können dann zu Testzwecken oder Problembehandlungen nicht ohne weiteres ausgetauscht oder wiederverwendet werden, da eine darunterliegende Ebene von ihnen abhängig sein könnte. Diese müsste dann ebenfalls ausgetauscht oder zumindest angepasst werden. Auch ein Domino-Effekt könnte bei dieser engen Verknüpfung auftreten, bei dem eine regionale Änderung auf einer Ebene sich zu einem globalen Problem für die gesamte Engine entwickeln kann. Zirkulare Abhängigkeiten begünstigen ebenfalls das Problem der unendlichen Rekursion. Dieses beschreibt den Zustand, in dem sich zu eng verknüpften Module immer wieder gegenseitig aufrufen. Dies würde unweigerlich dazu führen, dass die gesamte Engine in einer Endlosschleife feststeckt, und daraufhin vermutlich abstürzt. In keinem Fall ist dies ein wünschenswertes Szenario.

---

43 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 32-62

44 Vgl. Ebd. S. 34

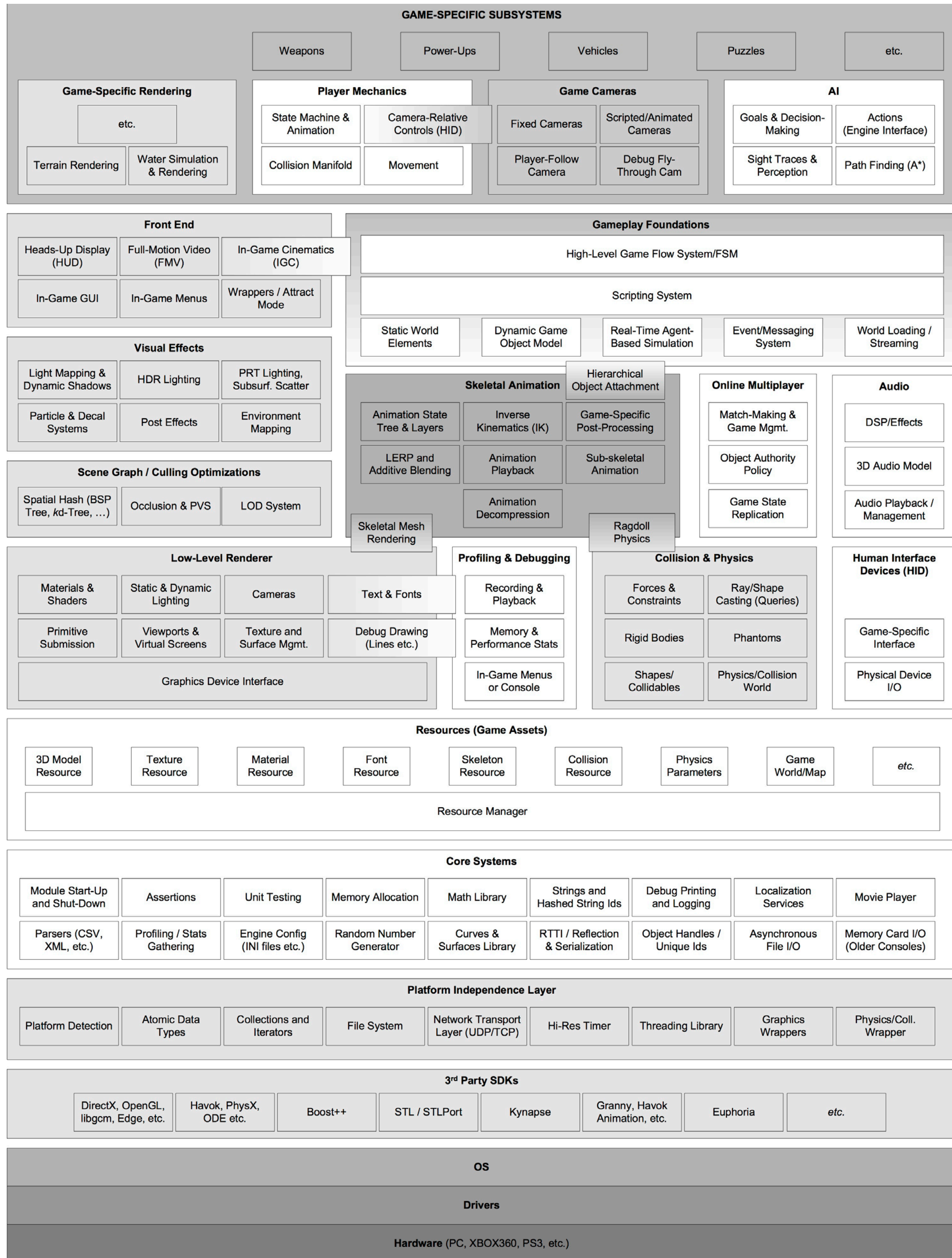

Abb. 1: Allgemeiner Aufbau bzw. die Architektur einer Game-Engine mit den verschiedenen Ebenen.

4.1 Die Hardware-Ebene[45]

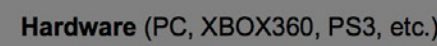

Abb. 2: Die Hardware-Ebene.

Die unterste oder auch erste Ebene repräsentiert die Hardware der Plattform, auf der das Spiel ausgeführt werden soll. Typische Plattformen sind natürlich Computer und die gängigen Videospielkonsolen, aber auch mobile Geräte wie Smartphones und Tablets. Die Hardware-Ebene wird hier größtenteils nur der Vollständigkeit halber aufgeführt, denn obwohl die Engine durch die Hardwarekapazität begrenz ist, hat sie keinen Einfluss auf eben diese.

4.2 Die Geräte Treiber[46]

Abb. 3: Die Treiber-Ebene.

Direkt über der Hardware-Ebene sind die Treiber der jeweiligen Hardware angeordnet. Diese werden direkt vom Hersteller des Gerätes oder der Plattform bereitgestellt und verwalten die vorhandenen Ressourcen innerhalb des Gerätes. Sie sind ebenfalls dafür zuständig mit den unzähligen Varianten von anderer (teils Fremd-)Hardware zu kommunizieren und so die oberen Ebenen vor diesem Aufwand abzuschirmen. Damit sind beispielsweise die verschiedenen Arten von Controllern („Normale“ von anderen Fremdherstellern, Joysticks, Lenkräder, Angeln, usw.) gemeint. Eine Engine erwartet zum Beispiel zu einem gegebenen Zeitpunkt eine Eingabe des Benutzers, ob dieser Knopf A oder Knopf B auf seinem Controller drückt. Ob dieser Knopf dann allerdings rot, blau, grün ist oder sich an der Seite eines Joysticks befindet, spielt für die Engine keine Rolle. Die Treiber übernehmen diese Aufgabe und sorgen dafür, dass alle Signale dorthin gelangen wo es beabsichtigt und relevant ist.

---

45 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 34

46 Vgl. Ebd.

4.3 Das Betriebssystem[47]

Abb. 4: Die Ebene auf der das Betriebssystem (engl.: *operating system*, *OS*) der jeweiligen Plattform installiert ist.

Das Betriebssystem eines Computers arbeitet zur Laufzeit konstant durch. Es organisiert die Ausführung mehrerer Programme und/oder Spiele zur selben Zeit. Betriebssysteme wie Mircrosoft Windows verfolgen dabei einen sogenannten ***time-sliced***[48] Ansatz. Dabei werden einem Prozess (wie das Ausführen des Spiels) immer nur eine gewisse Zeit lang die Ressourcen des Computers zur Verfügung gestellt werden. So kann das gleichzeitige Ausführen mehrerer Programme erfolgen. Endet die zugeteilte Zeitspanne, bewertet der Prozess-Scheduler (das Steuerprogramm[49]) die Situation neu und teilt dem Prozess entweder eine weitere Zeitspanne zu oder entzieht ihm den Zugang zu den Ressourcen des Computers. „This means that a PC game can never assume it has full control of the hardware – it must „play nice" with other programs in the system."[50] Für das Ausführen eines Spiels, das heißt für die Architektur einer Engine, bedeutet dies, dass sich das Spiel salopp formuliert benehmen und mit anderen Programmen koexistieren können muss. Ein Spiel wird niemals die gesamte Kontrolle über die Hardware eines Computers besitzen.

Auf einer Konsole sind die Machtverhältnisse anders gelagert. Hier ist das Betriebssystem nur eine verhältnismäßig einfache Sammlung von Befehlen und Algorithmen zur Steuerung und Kommunikation der Hardware und dem auszuführenden Spiel (oder einem anderen Medium, je nach Konsole). „On a console, the game typically „owns" the entire machine."[51] Bei einer Konsole hat das Spiel bzw. die Engine typischerweise die volle Kontrolle über die gesamte Hardware. Zumindest war dies der Fall bis zur Konsolengeneration der Playstation 3 und der Xbox 360. Die Betriebssysteme dieser Konsolen (und folglich auch die der nächsten Generation mit der Playstation 4 und der Xbox One) waren in der Lage, die Ausführung des Spiels

---

47 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 35

48 S. Time slice- https://en.wikipedia.org/wiki/Preemption_(computing)#Time_slice (letzter Aufruf: 06.07.2016)

49 S. Prozess-Scheduler- https://de.wikipedia.org/wiki/Prozess-Scheduler (letzter Aufruf: 06.07.2016)
Und Operating System- Process Scheduling- http://www.tutorialspoint.com/operating_system/os_process_scheduling.htm (letzter Aufruf: 06.07.2016)

50 Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 35

51 Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 35

zu unterbrechen oder gewisse Systemressourcen zu übernehmen, um beispielsweise Online-Nachrichten anzuzeigen oder dem Spieler die Möglichkeit zu geben, das Spiel zu pausieren und ins Hauptmenü der Konsole zurückzukehren, ohne erst den Umweg über das Hauptmenü des Spiels selbst nehmen zu müssen. Die Lücke zwischen der PC- Konsolenentwicklung schließt sich also immer weiter.

4.4 Fremdanbieter Anwendungen und Middleware[52] [53]

Abb. 5: Ebene der Fremdanbieter Anwendungen mit einigen Beispielen.

Bei der Entwicklung eines Spiels oder einer Engine ist es nicht unüblich zu einem gewissen Maße auf Softwarelösungen anderer Anbieter zurückzugreifen. Diese Bausteine, wie viele es auch sein mögen, befinden sich dann in der vierten Ebene über dem Betriebssystem. Diese Softwaresysteme bringen verschiedenste Befehle und Algorithmen mit sich und können unterschiedliche Schwerpunkte haben. Zum Beispiel definieren sie die Struktur und Verwaltung der erforderlichen Daten.

Wie jede andere Software, ist auch eine Game-Engine darauf angewiesen verschiedene Daten zu sammeln, sie zu manipulieren und sie an anderer Stelle wieder auszugeben. Wie genau das Sammeln erfolgt und unter welchen Bedingungen und Regeln dann die Manipulationen vorgenommen werden, muss im Vorfeld implementiert werden. Dazu können die Entwickler dann entweder selbst ein System entwickeln, das ihren Ansprüchen entspricht oder eben auf eine Lösung eines Fremdanbieters zurückgreifen. Dabei müssen dann gegebenenfalls Anpassungen durchgeführt werden.

Ein anderer Schwerpunkt dieser Fremdanbieter Anwendungen sind Grafikschnittstellen. Die bekanntesten in ihrem Bereich sind wohl OpenGL[54] und DirectX[55]. Beides sind Sammlungen von Befehlen und Algorithmen zur Darstellung dreidimensionaler Szenen. Dabei ist zu beachten, dass diese Sammlungen, auch Bibliotheken genannt, nur das Fundament bilden. Sie werden hinterlegt um darauf aufbauen zu können. Da die gra-

52 S. Middleware- http://www.itwissen.info/definition/lexikon/Middleware-middleware.html (letzter Aufruf: 06.07.2016)

53 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 35-38

54 S. OpenGL- https://www.khronos.org/opengl/ (letzter Aufruf: 06.07.2016)

55 S. DirectX- https://de.wikipedia.org/wiki/DirectX (letzter Aufruf: 06.07.2016)

fische Qualität eines jeden Spiels, das heißt auch der dahinterstehenden Engine, nach wie vor eines der wichtigsten Qualitätsmerkmale ist, misst ihr jedes Entwicklerstudio in der Regel immer einen großen Wert zu. Die Schnittstellen werden also noch weiter verbessert und angepasst, statt lediglich mit den grundlegenden Operationen zu arbeiten. Weitere Punkte sind Physiksysteme und die Charakteranimation. Beides sind sehr wichtige Punkte, denn ein Spiel mag grafisch noch so überzeugend sein, der Benutzer möchte sich auch darin bewegen können (was wiederum auch gut aussehen muss). Dafür gibt es verschiedene Lösungen in der Branche, die sich, wie alles andere auch, stets weiterentwickeln. In diesem Beispiel sogar in die Richtung, dass die beiden erwähnten Punkte sich immer mehr annähern und die Linie zwischen ihnen allmählich verschwimmt. Es existieren Systeme, welche die traditionelle Animation eines Charakters per Hand durch physikalische Simulationen unterstützen. Vereinfacht formuliert muss der Animator lediglich Start- und Zielpunkt einer Bewegung festlegen und die Software übernimmt alles dazwischen. Dies tut sie, indem sie die Bewegung, die der Charakter durchführen muss, anhand von Parametern, wie beispielsweise Gewicht, Zentrum der Schwerkraft und genaues Wissen darüber wie sich ein echter Mensch bewegt und ausbalanciert, errechnet und anwendet. Solche Systeme stellen ein gutes Beispiel dafür dar, wie die technische Weiterentwicklung auch in diesem Bereich die eigentliche Arbeit für den Menschen erleichtert.

Kurz zusammenfassend sei noch einmal erwähnt, dass diese Lösungen von Fremdanbietern immer nur ein Fundament legen. In der Regel werden sie im Laufe des Entwicklungsprozesses noch verändert oder erweitert und auf die jeweiligen Bedürfnisse des Spiels abgestimmt. Es soll an dieser Stelle nicht der Eindruck entstehen, man könnte eine Game-Engine aus verschiedenen Lösungen von Fremdanbietern „zusammenbasteln“. Es ist ebenso absolut möglich komplett auf externe Software zu verzichten und sämtlichen Quellcode selbst zu verfassen. Letzten Endes bleibt es eine Frage der Wirtschaftlichkeit und der eigenen Fähigkeiten.

4.5 Plattformabhängige Ebene[56]

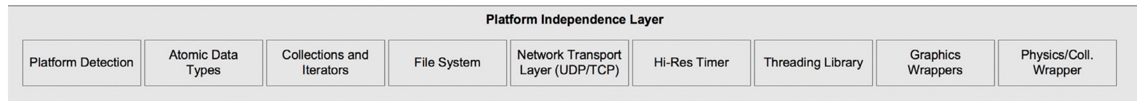

Abb. 6: Ebene der plattformabhängigen Elemente einer Engine.

Die meisten Spiele werden auf mehr als einer Plattform veröffentlicht. Große Verleger wie Electronic Arts beispielsweise streben sogar nach der größtmöglichen Anzahl an Plattformen, um den meisten Umsatz zu generieren. Ausgenommen davon sind lediglich Entwicklerstudios, die ausschließlich für eine einzige Plattform entwickeln. Diese Art Studios, wie zum Beispiel Sonys Naughty Dog[57], werden ***first-party studios*** genannt. Alle anderen Entwicklerstudios planen für gewöhnlich die Veröffentlichung auf mindestens zwei verschiedenen Plattformen. Das bedeutet, dass die eingesetzten Engines mit den verschiedenen Spezifikationen der Plattformen umgehen können müssen. Dazu werden sie mit einer eigenen Ebene ausgestattet, die genau dieses gewährleisten soll. Sie dient dazu den restlichen, das heißt den weiter oben gelegenen Teil der Engine vor dem abzuschirmen was darunter liegt, so dass die Engine im Grunde gar nicht registriert, ob sie gerade auf einem PC oder einer Playstation 4 ausgeführt wird. Indem sie eingehende Signale ersetzt oder anders verpackt schafft die Ebene eine standardisierte und konsequente Form der Kommunikation innerhalb der Engine über alle Plattformen hinweg.

4.6 Kern-Systeme[58]

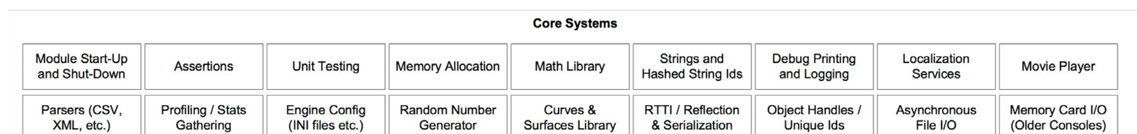

Abb. 7: Einige Beispiele zu Kernelementen einer Game-Engine.

Die Ebene der Kern-Systeme beinhaltet eine Reihe von nützlichen Hilfsmitteln, verschiedene Softwarelösungen die für unterschiedliche Zwecke konzipiert sind. Entscheidet man sich zum Beispiel dagegen die Datenstruktur eines Fremdanbieters (s. Kapitel 4.4) zu verwenden, wird die eigene Vorgehensweise in dieser Ebene implementiert. Darüber hinaus

---

56 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 38

57 S. Naughty Dog- http://www.naughtydog.com/ (letzter Aufruf: 06.07.2016)

58 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 39

befinden sich hier beispielsweise eine oder mehrere Bibliotheken mit mathematischen Formeln. Jedes Spiel ist auf eine Vielzahl dieser Formeln angewiesen, um alle Daten die erst zur Laufzeit entstehen korrekt manipulieren zu können. Ein Beispiel, ist die nötige Berechnung zur Flugbahn einer Handgranate in einem ***first-person shooter***, welche in der Regel physikalisch korrekt anmuten soll. Auch das System zur Verwaltung der Speichereinheiten befindet sich auf dieser Ebene, das auf die Bedürfnisse der Engine angepasst ist. Ein Teil dieser Ebene, der für gewöhnlich vor der Veröffentlichung wieder entfernt wird, sind Zeilen die der Problembehandlung dienen. Sie sollen logische Fehler und Störungen im Code finden, damit die Programmierer sie überarbeiten und beheben können.

4.7 Der Ressourcen Manager[59]

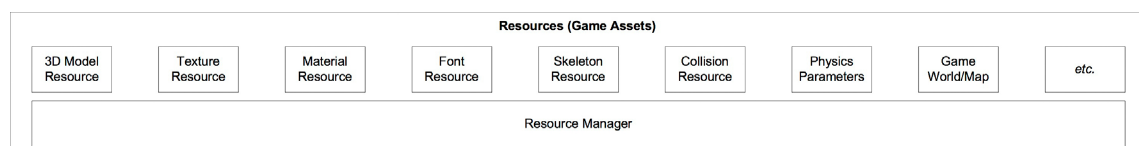

Abb. 8: Die Ebene auf der die Ressourcen (engl.: *assets*) eines Spiels und der dazugehörige Manager implementiert sind.

Der Ressourcen Manager hat von Engine zu Engine eine andere Form und Herangehensweise, seine Aufgabe ist allerdings immer gleich. Er bildet die Schnittstelle für alle Objekte und Elemente eines Spiels mit dem restlichen Teil der Engine. In der Regel werden diese Objekte und Elemente auch als ***assets*** (engl. für: Kapital, Gut, Aktivposten) bezeichnet. Das heißt, nochmal zurückkommend auf die Analogie mit dem Theater aus Kapitel 2, der Ressourcen Manager ist verantwortlich für die Bühne, die Kulissen, Kostüme und Requisiten. Ihm ist bekannt wo die jeweiligen Daten hinterlegt sind und übermittelt sie auf Anfrage an andere Komponenten der Engine.

4.8 Die Rendering-Engine[60]

Der Begriff ***rendering*** beschreibt in der Computergrafik ganz allgemein den Vorgang der Bildsynthese, das bedeutet die Berechnung eines Bildes aus gegebenen Rohdaten durch ein Computerprogramm.[61] Dabei kann es sich um ein einzelnes Bild handeln, eine Abfolge von mehreren Bildern und deren Verdichtung zu einem Videoclip oder Film, oder eben

---

59 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 40

60 Vgl. Ebd. S. 40-44

61 S. Bildsynthese- https://de.wikipedia.org/wiki/Bildsynthese (letzter Aufruf: 07.07.2016)
Und Rendering- http://www.itwissen.info/definition/lexikon/Rendering-rendering.html (letzter Aufruf: 07.07.2016)

wie in diesem Fall sehr viele schnell erzeugte Bilder, um ein Videospiel flüssig auf dem Bildschirm darzustellen.

Folglich gehört die Render-Engine zu den größten und komplexesten Komponenten einer Game-Engine. Eine Render-Engine kann auf viele verschiedene Arten aufgebaut werden, es gibt hierbei nicht den einen korrekten Weg. Jedoch ist zu beobachten, dass sich eine gewisse Grundphilosophie eingestellt hat, hauptsächlich beeinflusst durch die Grafikhardware der jeweiligen Plattform(en).

Eine daher übliche Vorgehensweise beim Aufbau der Render-Engine ist erneut ein Ebenen-Modell, das aussieht wie folgt:

4.8.1 Der low-level renderer[62]

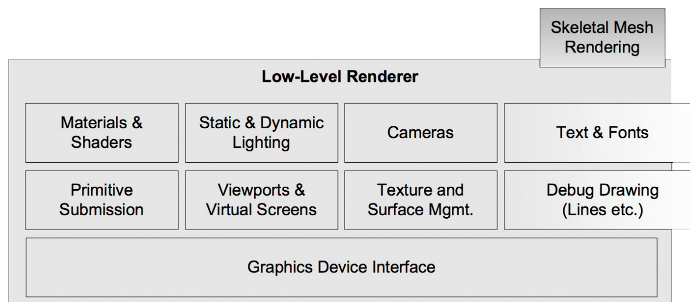

Abb. 9: Die unterste Ebene der Render-Engine, der ***low-level renderer***. Hier sind erstmals Verbindungen zu anderen Ebenen vorhanden, beispielsweise das ***skeletal mesh rendering***.

Der ***low-level renderer*** (engl. für: auf unterster Ebene) umfasst alle grundlegenden Bestandteile der Render-Engine. Auf dieser Ebene werden alle benötigten primitiven geometrischen Grundformen so schnell und so reichhaltig wie möglich berechnet und dargestellt, unabhängig davon ob sie zum gegebenen Zeitpunkt überhaupt sichtbar sind oder nicht. Mit primitiven geometrischen Grundformen sind die klassischen Elemente gemeint aus denen sich komplexe 3D-Modelle zusammensetzen: Punkte, Linien und Dreiecke. Diese Elemente, auch oft nur ***primitives*** genannt, können aufgrund ihrer Simplizität sehr schnell und ohne viel Aufwand berechnet werden.

Weitere Komponenten auf dieser Ebene sind beispielsweise für Materialien und Texturen, das Licht oder die Kameras, die den Bildausschnitt definieren, zuständig. Jedes primitive Element kommt mit einer Vielzahl an Daten daher. Diese erstrecken sich von der Position

---

62 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 40-42

des Elementes im dreidimensionalen Raum, über das Material, das darauf angewendet wurde, bis hin zu den einzelnen Lichtern, von denen es angestrahlt wird. Dies sind alles wichtige Faktoren die berücksichtigt werden müssen, ehe das finale Bild berechnet wird. Das Material definiert unter anderem mit welchem Teil der Hardware das primitive Element berechnet werden muss. Das Licht legt fest, welche Kalkulationen für eine dynamische Beleuchtung angewendet werden müssen. Das heißt der ***low-level renderer*** ist auch für die Verwaltung der jeweiligen Hardware zuständig. Er kontrolliert, welcher Teil der Hardware (z.B. in einem Computer ob die CPU oder die Grafikkarte verwendet werden muss, oder beide) wann und für welches Element verwendet wird.

4.8.2 Der Szenengraph / culling Optimierungen[63]

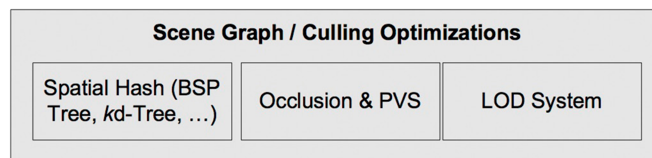

Abb. 10: Der Szenengraph und Elemente zur Bestimmung der Sichtbarkeit von Objekten bzw. deren Detailgenauigkeit.

Wie weiter oben bereits erwähnt, stellt der ***low-level renderer*** jegliche Form von Geometrie dar, die vorhanden ist, unabhängig davon ob der Spieler sie zum gegeben Zeitpunkt wahrnehmen kann. Dieses Vorgehen wäre allein überaus ineffizient. Daher wird eine höher gelegene Ebene benötigt, um festzulegen welche Elemente zum jeweiligen Zeitpunkt sichtbar sind und welche nicht. Dies geschieht auf dieser Ebene.

Für kleine Spielwelten genügt in der Regel ein einfacher sogenannter ***frustum cull*** (***to cull***, engl. für: auslesen, entnehmen, aussondern). Das bedeutet alles was nicht im Blickfeld der Kamera (d.h. des Spielers) liegt, wird aus dem Render-Prozess entfernt. Größere Spielwelten benötigen allerdings eine bessere räumliche Unterteilung, um die potenziell sichtbaren Elemente schnell und effizient festzulegen. Diese räumliche Unterteilung wird auch als Szenengraph[64] bezeichnet. Dabei handelt es sich Grunde um ein Diagramm, das die räumlichen Abhängigkeiten der verschiedenen Objekte und Elemente in einer Welt hierarchisch darstellt, beispielsweise als relativ simples Baumdiagramm. Dieser Graph dient aber nicht nur der Festlegung der Sichtbarkeit, sondern ist auch ein

---

63 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 42

64 S. What is a Scene Graph?- http://archive.gamedev.net/archive/reference/programming/features/scenegraph/index.html (letzter Aufruf: 07.07.2016)

gutes Hilfsmittel während der Entwicklung des Spiels. Damit können schnell Objekte hinzugefügt, entfernt, ausgetauscht oder mit diversen Operationen versehen werden.

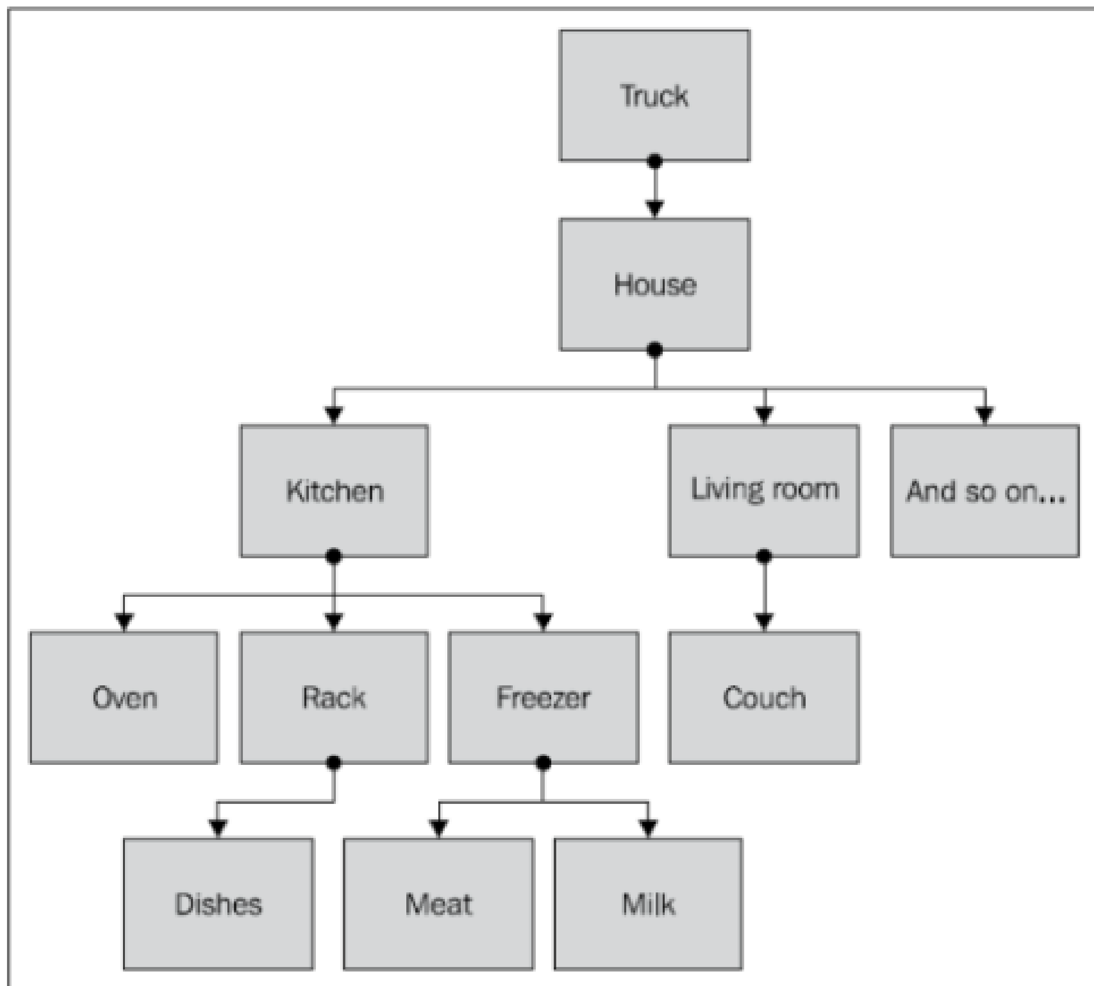

Abb. 11: Einfaches Beispiels eines Szenengraphs. Dargestellt werden die räumlichen Abhängigkeiten einiger Objekte der Spielwelt und ihre hierarchische Struktur. In diesem Fall ein komplettes Haus welches auf einem Transporter steht und so bewegt werden kann.

Neben der Entfernung etwaiger Objekte aufgrund der Sichtbarkeit können auf dieser Ebene auch Systeme zur Bestimmung des ***level-of-detail*** (*LOD*, s. Kapitel 2) implementiert werden, um den Render-Prozess noch weiter zu verbessern.

4.8.3 Visuelle Effekte[65]

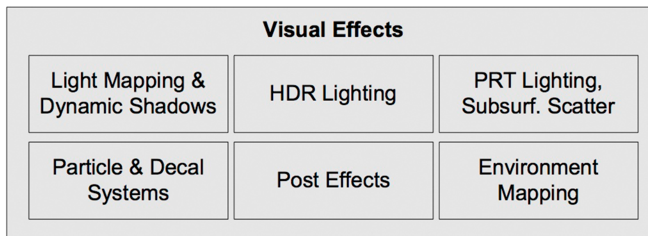

Abb. 12: Ein paar mögliche Implementierungen auf der Ebene der visuellen Effekte.

Aktuelle Game-Engines unterstützen mittlerweile eine Reihe von visuellen Effekten zur Aufwertung der grafischen Qualität der Spiele. Beispiele dafür sind:

- Partikelsysteme (Feuer, Rauch, Nebel, Wasserspritzer)
- sogenannte ***decal***[66] Systeme (Einschusslöcher, Fußspuren auf dem Untergrund, usw.)
- dynamische Schatten
- klassische Effekte der Postproduktion (Farbkorrektur, ***bloom*** Effekte, Blendenflecke, Farbsättigung)

Diese Effekte werden von einer speziellen Ebene innerhalb der Render-Engine gesteuert und an verschiedenen Stellen während des Render-Prozesses eingesetzt. So werden beispielsweise Partikel und ***decals*** schon während der Berechnung hinzugefügt, Effekte der Postproduktion aber erst hinterher auf das fertige Bild angewendet.

65 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 42-43

66 decal, engl. für: Aufkleber, Abziehbild, Klebebild

4.8.4 Das Front End[67]

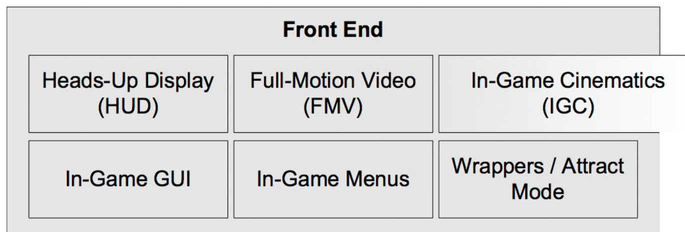

Abb. 13: Das Front End beschreibt die Ebene, auf der alle Elemente zu finden sind, die über die Spielwelt gelegt werden, wie beispielsweise Menüs. Für den Punkt der ***in-game cinematics*** besteht erneut eine Verbindung zu einer benachbarten Ebene.

Auf dieser Ebene werden die etwaigen 2D Elemente eines Spiels kontrolliert und erzeugt, die über die 3D Szene gelegt werden können. Klassische Beispiele sind Menüs, Karten, Anzeigen für Lebenspunkte und Munition oder das Inventar eines Charakters. Je nachdem um welches Gerne und Spiel es sich handelt, werden andere Elemente benötigt. Es ist auch nicht zwingend erforderlich, dass diese Elemente als 2D Komponenten über die 3D Szene gelegt werden. Sie können ebenfalls in die 3D Szene eingefügt und so ausgerichtet werden, dass sie immer zur Kamera zeigen. Unabhängig der Implementierung bestehen sie in der Regel aus texturieren Quadraten (zusammengesetzt aus zwei Dreiecken).

Ebenfalls auf dieser Ebene zu finden sind Systeme für die Steuerung von ***full-motion videos*** (*FMV*) oder ***in-game cinematics*** (*ICG*). Dabei handelt es sich allgemein um Zwischensequenzen (auch ***cutscenes***[68] genannt), die traditionell zwischen zwei Spielszenen eingespielt werden. ***FMVs*** bezeichnen dabei Sequenzen, die nicht in Echtzeit berechnet werden. Das heißt, sie heben sich von der Spielgrafik ab und sind qualitativ hochwertiger. ***ICGs*** werden ausschließlich innerhalb der Engine berechnet und ausschließlich zur Laufzeit des Spiels. Daher gleichen sich ***ICGs*** an die Spielgrafik an. Ob der Spieler während einer ***ICG*** die Kontrolle behält oder sogar eingreifen muss, oder ob er für die Dauer der Sequenz zum passiven Zuschauer wird, hängt dabei einzig und allein vom jeweiligen Spiel ab. Behält der Spieler die Kontrolle und geschieht die Handlung der Sequenz quasi neben ihm, egal ob er hinsieht oder nicht, spricht man von einer Skriptsequenz (oder auch ***scripted event***).

67 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 43-44

68 S. Zwischensequenz- https://de.wikipedia.org/wiki/Zwischensequenz (letzter Aufruf: 07.07.2016)

4.9 Werkzeuge zur Leistungsanalyse und Fehlerbehebung[69]

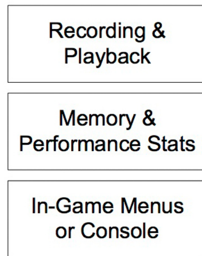

Abb. 14: Systeme zur Leistungsanalyse und Fehlerbehebung befinden sich auf dieser Ebene.

Spiele werden nicht wie Filme vorberechnet, sondern entstehen in Echtzeit. Dies bringt die Notwendigkeit diverser Systeme zur Leistungsanalyse und Fehlerbehebung mit sich. Sie umfassen Möglichkeiten zur Analyse der Speichernutzung, ein im Spiel befindliches Menü oder eine Konsole zur Fehlerbehebung und dem Aufnehmen und Wiedergeben von Spielinhalten. Etwas genauer gesagt ist es zum Beispiel möglich, ausgewählte Teile des Codes manuell auszuführen um ggf. Fehler zu finden, die Speicherauslastung (der Engine allgemein und der verschiedenen Untersysteme) und andere Kennzahlen der Spielleistung auf dem Bildschirm anzeigen zu lassen, oder Statistiken über die Leistung des Spiels als Textdatei oder Excel Tabelle auszugeben.

Die Playstation 4 besitzt darüber hinaus ein weiteres einzigartiges Merkmal zur Fehleranalyse bei kompletten Abstürzen des Spiels. Sie zeichnet standardmäßig die letzten 15 Sekunden des Spielinhalts auf. Gedacht ist diese Funktion dafür, dass der Spieler seine Erfolge durch einen einzigen Tastendruck mit der Gemeinschaft teilen kann. Diese Eigenschaft gewährt den Programmierern und Entwicklern im Falle eines Spielabsturz also nicht nur eine Anzahl an Datensätzen in Form von Textdateien, sondern ebenfalls einen Screenshot in genau dem Moment des Absturzes sowie Videomaterial über die letzten 15 Sekunden davor. Diese Daten können direkt an den Server des Entwicklers gesendet werden, selbst dann noch wenn das Spiel bereits beim Endverbraucher angekommen ist.

69 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 44-45

4.10 Kollisionserkennung und Physik-Engines[70]

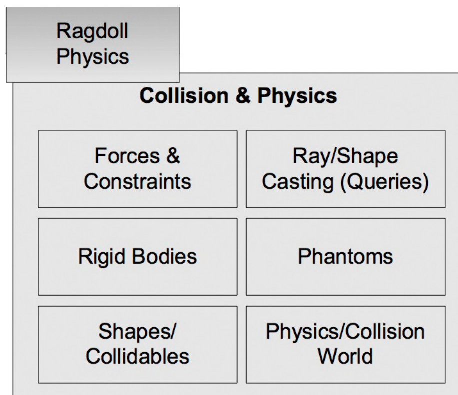

Abb. 15: Die Ebene zur Steuerung und Kontrolle aller physikalisch gestützten Elemente.

Die korrekte Erkennung von Kollisionen zwischen verschieden Objekten innerhalb eines Spiels ist unabdingbar für die Funktionalität des Spiels. Selbst der Urvater der Videospiele Pong[71] hätte ohne eine Kollisionserkennung nicht funktioniert. Ohne sie würden Objekte ineinander übergehen und eine Interaktion mit der Spielwelt wäre nicht machbar. Einige Spiele verfügen darüber hinaus über ein realistisches bis halb-realistisches Simulationssystem für dynamische Bewegungen. Oft werden diese Systeme unter dem Begriff Physik-Engine zusammengefasst. Die Kollisionserkennung und die Physik-Engine sind in der Regel sehr eng verknüpft. Denn wann immer eine Kollision erkannt wird, muss eine Reaktion erfolgen, die einer zufriedenstellenden Logik folgt. Angenommen man spielt eine Tennissimulation und schlägt den Ball mit dem Schläger, dann erwartet der Spieler, dass der Ball wieder von sich wegfliegt, statt auf den Boden zu fallen wie ein Stein. Die Physik-Engine übernimmt die hierfür erforderlichen Berechnungen.

Aufgrund der Komplexität dieser Physik-Engine ist es heutzutage üblich auf ein System eines Fremdanbieters zurückzugreifen, statt eine eigene Lösung zu erstellen. Die bekanntesten Beispiele sind Havok[72] und PhysX[73] von Nvidia. Havok ist heute der goldene Standard in der Industrie, da sie sehr umfangreich ist und gut auf allen Plattformen funktioniert. PhysX erzielt nicht weniger gute Resultate und wurde einerseits in die Unreal Engine 4 integriert, andererseits wird sie auch als unabhängige und kostenlose Engine vertrieben. Nvidia hat sie angepasst damit sie auf allen jüngsten Grafikprozessoren des Herstellers funktioniert.

---

70 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 46-47

71 S. Pong- https://de.wikipedia.org/wiki/Pong (letzter Aufruf: 07.07.2016)

72 S. Havok Physics- http://www.havok.com/physics/ (letzter Aufruf: 07.07.2016)

73 S. PhysX- http://www.geforce.com/hardware/technology/physx (letzter Aufruf: 07.07.2016)

4.11 Animationssysteme[74]

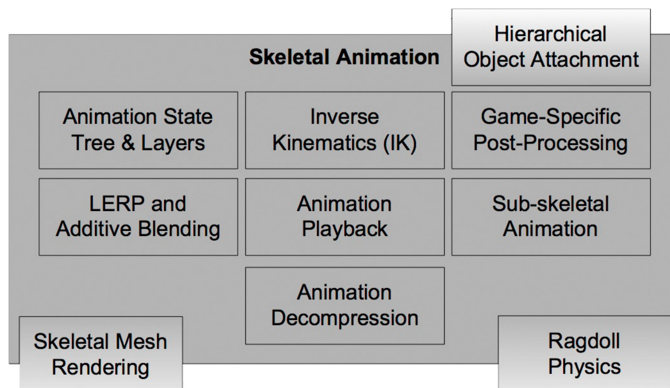

Abb. 16: Die ***skeletal animation*** Ebene steht im Zentrum der Render-Engine und weist die meisten Verbindungen zu anderen Ebene auf.

Jedes Spiel mit organischen oder halb-organischen Charakteren wie Menschen, Tiere, Cartoonfiguren oder auch Robotern benötigt ein System für die Animationen dieser Charaktere. Je nach Anspruch oder Prinzip des Spiels sind diese Animationen mal mehr und mal weniger realitätsgetreu. Man unterscheidet in fünf verschiedene Animationstypen, die in Spielen benutzt werden:

- ***sprite*** / ***texture*** Animation (2D-Bilder die immer zur Kamera gewandt sind)
- eine Hierarchie von starren Festkörpern, sogenannten ***rigid bodies***[75] (oft bei mechanischen Modellen eingesetzt, wie z.B. einem Zylinder in einem Verbrennungsmotor)
- Animation eines digitalen Skeletts
- Animation von Eckpunkten, sogenannten ***vertices*** (oder Singular: ***vertex***)
- und ***morph targets***

Die Verwendung von digitalen Skeletten erlaubt es dem Animator, jede benötigte Pose eines Charakters händisch herzustellen und abzuspeichern. Diese Methode ist heutzutage die in Spielen am häufigsten eingesetzte Methode. Sie stellt die Verbindung zwischen dem Animationssystem, welches die Posen zur Laufzeit abruft und mit den Skeletten erzeugt, und der Render-Engine dar, die die Punkte des eigentlichen Charaktermodells für jede Pose entsprechend anpasst und darstellt. Da das eigentliche Charaktermodell sozusagen die Haut ist, die über die Knochen gelegt wird, bezeichnet man diesem Vorgang auch als ***skinning***. ***morph targets*** bezeichnet eine Technik die sehr oft bei der Animation von Gesichtern angewendet wird. Die verschiedenen Gesichtsausdrücke werden vom Animator im Vorfeld erstellt und in einzelnen Stadien hinterlegt. Das heißt, es wird zum Beispiel ein Gesicht

74 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 47-48

75 S. Starrer Körper- https://de.wikipedia.org/wiki/Starrer_K%C3%B6rper (letzter Aufruf: 07.07.2016) Und Starrkörpersimulation- https://de.wikipedia.org/wiki/Starrk%C3%B6rpersimulation (letzter Aufruf: 07.07.2016)

mit normal weit geöffneten Augen, eines mit geschlossenen Augen und eines mit extrem weit aufgerissene Augen angefertigt. Zur Laufzeit können diese Stadien dann abgerufen und verschieden stark interpoliert werden. Für einen überraschten Gesichtsausdruck beispielsweise scheint ein Mittelmaß zwischen normal weit und extrem weit geöffneten Augen angebracht zu sein. Die Interpolation erfolgt dabei über einen gewissen Zeitraum hinweg, nicht abrupt von einem Bild auf das Nächste. Daher die Bezeichnung ***morph***, vom englischen ***to morph***: sich verwandeln. Da bei dieser Technik, ähnlich wie beim ***skinning***, hauptsächlich mit den Eckpunkten (des Gesichts in diesem Fall) gearbeitet wird, ist die Bezeichnung ***per-vertex animation***[76] ebenfalls geläufig.

Wenn sogenannte ***rag dolls*** (engl. für: Stoffpuppe) eingesetzt werden, existiert ebenfalls eine starke Verknüpfung zwischen dem Animationssystem und der Physik-Engine. Charaktere werden als ***rag doll*** bezeichnet, wenn sie nur schwach bis gar nicht animiert sind und ihre Bewegungen von der Physik-Engine berechnet werden. Diese Technik wird oft für tote Charaktere verwendet, die ab dem Zeitpunkt ihres Ablebens möglichst realitätsnah zu Boden fallen sollen. Intern werden sie dann als Hierarchie von starren Festkörpern behandelt. Insbesondere bei Spielen älterer Generationen, in denen die Physik-Engines noch nicht so leistungsstark waren, gibt es immer wieder Szene in denen solche ***rag dolls*** für unfreiwillig komische Einlagen sorgen. Da die Bewegungen eben erst zur Laufzeit erzeugt werden, haben die Animatoren bzw. Entwickler im Vorfeld keinen Einfluss darauf. Der Einsatz dieser Technik ist aber besonders für ***shooter***, in denen viele namenlose Nebencharaktere sterben, sehr effizient.

Abb. 17 und 18: Beispiele für misslungene Sterbeanimationen durch die Verwendung von ***rag dolls*** im Spiel Tom Clancy‘s Rainbow Six 3: Raven Shield von Red Storm Entertainment und Ubisoft.

76 S. Morph target animation- https://en.wikipedia.org/wiki/Morph_target_animation (letzter Aufruf: 07.07.2016)

4.12 Die Mensch-Maschine-Schnittstelle bzw. Controller[77]

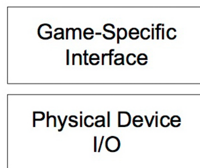

Abb. 19: das ***human interface device (HID)*** kontrolliert und reguliert die Eingaben des Benutzers.

Ohne die Eingaben des Benutzers würde ein Spiel zwar durchaus funktionieren, aber es würde sehr oft außer dem Hauptmenü nichts weiter passieren. Das heißt die Signale, die der Benutzer über die verschiedenen Eingabegeräte wie Maus und Tastatur, Gamepads o.ä. erzeugt, müssen an einer Stelle in der Engine erkannt, verarbeitet und weitergeleitet werden. Dafür ist diese Ebene zuständig. Da Gamepads seit einiger Zeit nicht nur in der Lage sind Signale zu empfangen, sondern durch beispielsweise eingebaute Vibrationsmotoren, auch zu senden, wird diese Ebene auch als ***player I/O*** bezeichnet (***I/O*** = ***input/output***, engl. für: Eingabe/Ausgabe). Sie erkennt Befehle wie das Drücken und Loslassen eines Knopfes, die Stellungen der Sticks auf einem Gamepad oder ist in der Lage den Beschleunigungsmesser in neueren Geräten auszulesen. Je nach Eingabe versendet die Ebene die Signale dann an die entsprechende Stelle der Engine, um die korrekten Abläufe in Gang zu bringen. Ebenfalls stellt sie dem Spieler oft die Möglichkeit bereit, die Tastenbelegungen der Eingabegeräte seinen Wünschen entsprechend zu verändern. Dementsprechend müssen Profile angelegt werden, um einkommende Signale dann auf neue aber weiterhin korrekte Weise auszulesen.

77 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 48

4.13 Das Audio-System[78]

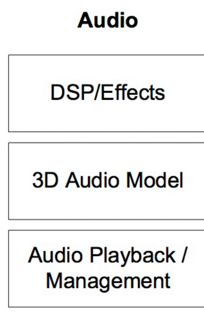

Abb. 20: Die Audio-Engine.

Der Sound in einem Spiel ist mindestens genauso wichtig wie die Grafik, dennoch bekommen Audio-Systeme oft weniger Aufmerksamkeit als Rendering- oder Physik-Engines. Nichtsdestotrotz bedarf ein gutes Spiel auch einer überzeugenden Sound-Engine. Die Rede ist dabei nicht nur von der Musik, die eingesetzt wird, um wichtige Momente in der Handlung zu begleiten. Ebenso wichtig sind die unzähligen Soundeffekte. Sie erzeugen erst den Grad an Immersion, den sich ein Spieler heutzutage wünscht. Durch den Schnee zu laufen, ohne das typische Stapf-Geräusch wahrzunehmen, fühlt sich falsch an. Eine Waffe abzufeuern, die keinen Ton von sich gibt, fühlt sich unecht an. Und in einem Rennauto zu sitzen, ohne die Geräusche des Motors zu hören, ist unvollständig. Allein diese drei simplen Beispiele zeigen deutlich, wie wichtig eine gute Sound-Engine ist und auch inwieweit sie in die anderen Systeme integriert werden muss. Denn nahezu jeder Charakter oder jedes Objekt innerhalb des Spiels gibt mindestens ein Geräusch von sich. Daher ist es auch keine Seltenheit, dass die Entwickler viel Zeit in das Audio-System investieren, um ein qualitativ hochwertiges und abgerundetes Endergebnis zu erhalten.

78 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 49

4.14 Online Multiplayer und Vernetzung[79]

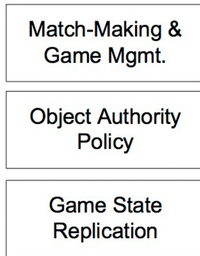

Abb. 21: Alle Elemente, die in den Bereichen Multiplayer und Netzwerkverbindung tätig sind, befinden sich auf dieser Ebene.

Viele Spiele, heutzutage mehr denn je, erlauben es dem Spieler zusammen mit anderen Spielern in einer einzelnen Welt zu interagieren. Dabei unterscheidet man in vier Kategorien:

1. ***single-screen multiplayer*** | Zwei oder mehr Eingabegeräte sind an einer Plattform angeschlossen und die Spieler bevölkern gleichzeitig eine einzelne Welt, die von einer Kamera begleitet wird.
2. ***split-screen multiplayer*** | Mehrere Spieler bevölkern eine einzelne virtuelle Welt mit mehreren Eingabegeräten, die an einer Plattform angeschlossen sind. Für jeden Spieler existiert eine Kamera. Der Bildschirm wird in Abschnitte unterteilt, so dass jeder Spieler seinen Charakter unabhängig der anderen verfolgen kann.
3. ***networked multiplayer*** | Mehrere Plattformen sind miteinander vernetzt (in der Regel baulich getrennt) und jede richtet das Spiel eines Spielers aus.
4. ***massively multiplayer online game (MMOG)*** | Buchstäblich hunderttausende von Spielern können online gleichzeitig eine riesige beständige Welt bevölkern, die von leistungsstarken Servern ausgerichtet wird.

Multiplayer Spiele sind in vieler Weise den Singleplayer Spielen ähnlich. Dennoch kann die Unterstützung eines Multiplayer Modus einen nicht zu unterschätzenden Einfluss auf gewisse Bestandteile einer Game-Engine haben. Systeme wie der Welt-Editor, die Eingabegeräte, die Render-Engine, das Animationssystem und weitere Elemente sind davon betroffen. Nachträglich einen Multiplayer Modus in ein bestehendes Singleplayer Spiel einzubauen ist nicht unmöglich, es kann allerdings eine sehr mühselige Aufgabe sein. Daher ist es weitaus sinnvoller etwaige Multiplayer Elemente direkt vom ersten Tag an mit einzuplanen. Interessanterweise ist es umso einfacher, den andersherum verlaufenden Weg zu gehen. Das heißt, aus einem bestehenden Multiplayer Spiel ein Singleplayer Spiel zu machen. Denn tatsächlich ist es so, dass viele Game-Engines den Singleplayer Modus als einen Son-

79 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 49-50

derfall des Multiplayer Modus betrachten, in dem es lediglich einen Spieler gibt. Die Quake-Engine ist bekannt für ihren sogenannten ***client-on-top-of-server*** Modus. Das heißt, der Computer auf dem das Spiel ausgeführt wird, dient im Singleplayer Modus gleichzeitig als Server der das Spiel ausrichtet, als auch als Nutzer dieses Servers.[80]

4.15 Fundamente des Spielprinzips[81]

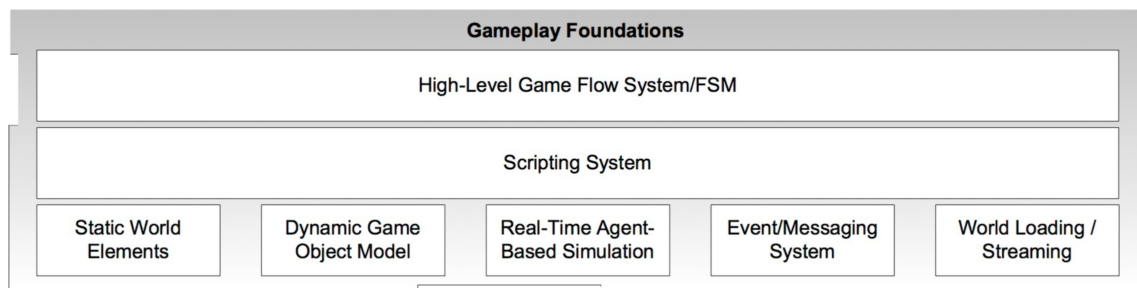

Abb. 22: Die ***gameplay foundations*** fassen alle Elemente zusammen, die quasi als Fundament für ein gewisses Genre dienen.

Der Begriff Spielprinzip, oder wie in der Branche eher üblich als ***gameplay*** bezeichnet, beschreibt alle Inhalte, Handlungen, Charakterfähigkeiten, Regeln und Ziele eines Spiels, also alle Elemente die neben den optischen Qualitäten zum Spielspaß beitragen. Diese Ebene stellt die Fundamente für das Spielprinzip bereit und schließt damit die Lücke zwischen dem tatsächlichen Quellcode des Spiels und den unteren Systemen der Engine, die bis hierher betrachtet wurden. Diese Fundamente werden üblicherweise in der gleichen Programmiersprache wie der Rest der Engine implementiert, können allerdings auch in einer hochrangigeren Skriptsprache verfasst sein.

Auf dieser Ebene befinden sich einige Kernbestandteile wie beispielsweise die eigentliche Spielwelt, sowohl mit ihren statischen als auch den dynamischen Objekten. Zu diesen Objekten gehören:

- statische Geometrie für Hintergründe, wie z.B. Gebäude, Straßen oder die Landschaft
- dynamische starre Festkörper, wie z.B. Steine, Möbel, Flaschen usw.
- Spieler-Charaktere, ***player characters*** (***PC***)
- Nicht-Spieler-Charaktere, ***non-player characters*** (***NPC***)
- Waffen
- Geschosse, Projektile

80 S. Client-Server Prinzip- http://www.fachadmin.de/index.php/Client-Server_Prinzip (letzter Aufruf: 07.07.2016)

81 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 50-53

- Fahrzeuge jeder Art
- Lichter
- Kameras
- und Weitere

Die Inhalte der Welt sind in der Regel in einer objektorientierten Weise angelegt, das heißt für gewöhnlich auch in einer objektorientierten Programmiersprache wie C++ oder Java.

Um die Entwicklung eines Spiels zu vereinfachen, wird in der Regel auf dieser Ebene auch ein Skriptsystem implementiert. Das Nutzen einer Skriptsprache erlaubt es den Entwicklern schnell und einfach spielspezifische Regeln zu verändern. Ohne die Verwendung von Skripts müsste bei jeder Änderung das gesamte Spiel erst neu kompiliert, das heißt einmal komplett in Maschinensprache übersetzt werden, damit der Computer es ausführen kann.[82] Bei der enormen Menge an Quellcode, die ein Spiel mit sich bringt, würde dieser Vorgang für kleine Änderungen unverhältnismäßig lange dauern. Ein Skript dagegen kann mitunter während des laufenden Betriebs des Spiels (in der Entwicklung) geladen und ausgeführt werden, um die gewünschten Änderungen vorzunehmen.

Ebenfalls wird auf dieser Ebene der Engine ein grundlegendes System für die künstliche Intelligenz (KI)[83] implementiert. Dieses System übernimmt die Kontrolle über alle Nicht-Spieler-Charaktere, sofern diese nicht festen Regeln unterliegen und frei agieren und reagieren sollen. Üblicherweise werden die grundlegenden Eigenschaften, die eine KI innehaben soll, durch eine Fremdanbieterlösung wie beispielsweise Gameware Navigation von Autodesk[84] umgesetzt. Dieses System beinhaltet Funktionsblöcke wie die Generierung von Navigationsgittern, Wegfindung, Vermeidung von statischen und dynamischen Objekten und die Identifizierung von Schwachstellen in der Spielwelt (z.B. um den Spieler durch ein Fenster hindurch anzugreifen). Außerdem besitzt es eine gut durchdachte Schnittstelle zwischen der KI selbst und den Animatoren. Mit diesen grundlegenden Elementen können die Entwickler dann eine eigene, spielspezifische künstliche Intelligenz erschaffen.

---

82 S. Compiler – https://de.wikipedia.org/wiki/Compiler (letzter Aufruf: 08.07.2016)

83 S. Kern, Sabine; Neumayer, Ingo (07.07.2016): Computer und Roboter. Künstliche Intelligenz- http://www.planet-wissen.de/technik/computer_und_roboter/kuenstliche_intelligenz/ (letzter Aufruf: 08.07.2016)

Und Künstliche Intelligenz- https://de.wikipedia.org/wiki/K%C3%BCnstliche_Intelligenz (letzter Aufruf: 08.07.2016)

84 S. NAVIGATION- http://www.autodesk.com/products/navigation/overview (letzter Aufruf: 08.07.2016)

4.16 Spielspezifische Bestandteile[85]

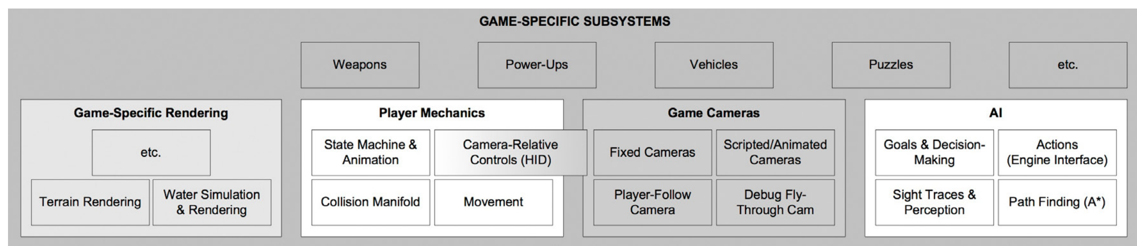

Abb. 23: Letztendlich befinden sich auf dieser Ebene nun alle Inhalte eines spezifischen Spiels.

Wenn es in irgendeiner Art und Weise möglich ist eine ansatzweise klare Linie zwischen der Game-Engine und dem Spiel selbst zu ziehen, dann würde man sie an dieser Stelle ziehen. Zwischen der Ebene mit den Fundamenten des Spielprinzips und den spielspezifischen Bestandteilen.

Auf dieser Ebene befinden sich nun schlussendlich alle Elemente, die direkt mit dem spezifischen Spiel zusammenhängen. Also nicht allgemeine Elemente, die für einen ***first-person shooter*** relevant sind, sondern Elemente die tatsächlich nur für ein DOOM, ein Unreal 2 oder ein Battlefield 4 wichtig sind. Wäre es möglich eine „perfekte“ Engine zu programmieren, wäre ein Modulsystem denkbar, bei dem dann lediglich diese spielspezifische Ebene ausgetauscht oder angepasst werden müsste, um ein neues Spiel oder einen Nachfolger zu kreieren.

Diese Ebene enthält Spezifika wie die Modelle der genutzten Waffen und Fahrzeuge, sogenannte ***power-ups*** (z.B. +15% Rüstung für X Sekunden), Puzzle und Rätsel Elemente, usw. Ebenso befinden sich hier allerdings auch Systeme wie die höherrangigen Funktionen der KI, spielspezifische Teile der Render-Engine, das Kamerasystem oder die Eigenschaften der Charaktere.

Damit soll dieses Kapitel über den allgemeinen Aufbau und die einzelnen Bestandteile einer Game-Engine abgeschlossen sein. Das Ziel war es, einen groben Überblick zu gewähren. Viele Teile würden natürlich einen wesentlich tieferen Ansatz zulassen, welcher für ein genaueres Verständnis auch erforderlich wäre. Für den hier gewählten Rahmen soll dies an dieser Stelle allerdings ausreichend sein. In den folgenden Kapiteln sollen ein paar der hier bereits erwähnten Bestandteile und Systeme anhand von ausgewählten Game-Engines noch auf ihre Merkmale und Besonderheiten im praktischen Einsatz untersucht werden.

85 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 53-54

5. Game-Engines in der Praxis

Dieses Kapitel widmet sich dem praktischen Aspekt ausgewählter Game-Engines. Genauer gesagt werden hier die Engines Unreal 4 und Unity 5 noch einmal näher betrachtet und auf ihre generellen Merkmale und Besonderheiten hin untersucht, sowie in speziellen Teilbereichen wie beispielsweise das Erschaffen der Spielwelt. Vollständig alle Teilbereiche und Bestandteile können in dieser Arbeit allerdings nicht zur Gänze erfasst und behandelt werden. Dieses Vorhaben wäre zu umfangreich.

Den Anfang macht die Unreal Engine.

5.1 Die Unreal Engine 4

Die Unreal Engine 4 von Epic Games[86] ist eine topaktuelle und moderne Game-Engine, die zum einen in der Spieleindustrie, aber auch von Selbstständigen und Hobbyentwicklern eingesetzt wird. Mit der Engine ist es möglich sowohl aufwendige 3D-Spiele als auch verhältnismäßig einfachere 2D-Spiele zu erschaffen. Wobei erwähnt werden muss, dass der 3D-Bereich der ist, auf den die Engine spezialisiert ist.[87] Ebenfalls werden alle gängigen Spiele-Plattformen unterstützt, d.h. Computer, Konsolen und mobile Endgeräte. Ein besonderes Merkmal der Unreal Engine: Sie ist seit März 2015 völlig kostenfrei auf dem Markt, auch für den kommerziellen Bereich. Epic Games erhebt lediglich ab einem Quartalsumsatz von mehr als 3.000 $ eine Umsatzbeteiligung von 5%. Der komplette Quellcode steht den Entwicklern in jedem Fall zur Verfügung und kann nach Belieben verändert und angepasst werden.

Die interessanteste Besonderheit der Unreal Engine 4, vor allem für den semiprofessionellen- und Hobbybereich, ist allerdings die Tatsache, dass es möglich ist ein Spiel zu entwickeln, ohne größere Programmierkenntnisse zu besitzen. Das System der sogenannten ***blueprints*** ermöglicht es Spielinhalte zu kreieren „[…] ohne eine einzige Zeile Code zu schreiben […]“.[88] Sind Programmierkenntnisse vorhanden, können diese den Entwicklungsprozess natürlich gegebenenfalls vereinfachen oder beschleunigen. Eine Grundvoraussetzung sind sie in der Unreal Engine allerdings nicht. Im übernächsten Kapitel werden die ***blueprints*** noch näher betrachtet und erläutert.

---

86 S. Epic Game- https://www.epicgames.com/ (letzter Aufruf: 11.07.2016)

87 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 2

88 Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 2

5.1.1 Die grafische Oberfläche[89]

Wie jede handelsübliche Software besitzt auch eine Game-Engine eine grafische Oberfläche. Schließlich handelt es sich hierbei um eine komplette Entwicklungsumgebung für Spiele. Hier finden sich alle Werkzeuge und Informationen, die für den Entwicklungsprozess nötig sind. Im Wesentlichen beinhaltet die Oberfläche sechs Hauptbereiche:

1. ***game/editor view***: In diesem Bereich wird der jeweilige Level des Spiels erbaut. Die Welt wird erschaffen und Gegenstände in ihr platziert. Dem Entwickler ist es möglich, sich frei hindurch zu bewegen, Anpassungen zu machen und aus verschieden Betrachtungsweisen zu wählen. Am oberen Rand befindet sich zudem der Play-Knopf um das Spiel auszuführen.
2. ***content browser***: Der ***content browser*** ist die Dateiverwaltung für alle Spielinhalte (***assets***) und die ***blueprints***. Elemente können hier importiert, verwaltet, neu angelegt oder editiert werden.
3. ***world outliner***: Hier werden alle Elemente angezeigt, die sich in dem momentan geladenen Level befinden. Eine hierarchische Ordnerstruktur gewährt dabei Informationen über die Abhängigkeiten der Elemente bzw. können diese hier hergestellt und angepasst werden.
4. ***details***: Das ***details***-Fenster beinhaltet alle Informationen und Einstellungsmöglichkeiten des aktuell ausgewählten Elementes oder Objekts. Dies geht von simplen Einstellungen wie der Position des Objekts, über sein Material bis hin zu physikalischen Eigenschaften.
5. ***world settings***: Hier können globale Einstellungen für jeden Level vorgenommen werden. Zum Beispiel wie tief ein Gegenstand fallen darf ehe er automatisch gelöscht wird, wenn dieser aus bestimmten Gründen einmal die Grenzen der Karte verlässt.
6. ***modes***: Der Bereich ***modes*** ist sozusagen der Werkzeugkasten der Engine. Hier befinden sich verschiedene hilfreiche Gegenstände und Werkzeuge, wie beispielsweise einfache geometrische Objekte, Lichter oder Werkzeuge zur Erschaffung von Landschaften und Vegetation.

89 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 14-28

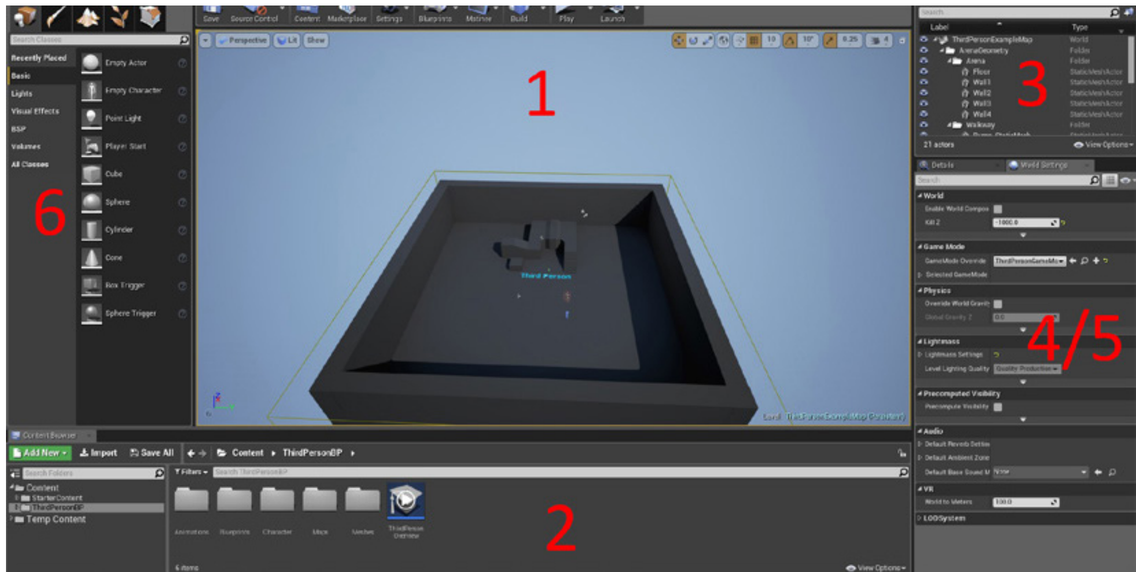

Abb. 24: Grafische Benutzeroberfläche der Unreal Engine mit den oben erwähnten sechs Hauptbereichen.

Nähere Informationen folgen je nach Bereich noch in späteren Kapiteln oder werden für den hier gewählten Rahmen als nicht notwendig betrachtet. Die Komplexität einer Game-Engine legt ohnehin nahe, sich bei Interesse selbst damit auseinanderzusetzen, um die vielen Bestandteile zu entdecken.

5.1.2 blueprints[90]

Wer grundlegende Kenntnisse von objektorientierter Programmierung besitzt für den ist der Begriff der Klasse in diesem Zusammenhang kein Fremdwort. ***blueprints*** sind im Grunde genommen nichts anderes als solche Klassen, in diesem Fall in der Programmiersprache C++. Innerhalb einer solchen Klasse befinden sich kleine, bausteinähnliche Logikelemente, die meist nur eine Aufgabe besitzen und mittels Knotenpunkten miteinander verbunden werden. Werden sie angesteuert oder aufgerufen entweder weil sie in der Kette an der Reihe sind oder explizit angesprochen werden, führen sie ihre Aufgabe aus. Mit Bausteinen und natürlich ***blueprints*** an sich, die miteinander kommunizieren können, können in der Unreal Engine dann komplexe Logikketten entwickelt werden um Spielinhalte zu erschaffen.

90 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 29-70

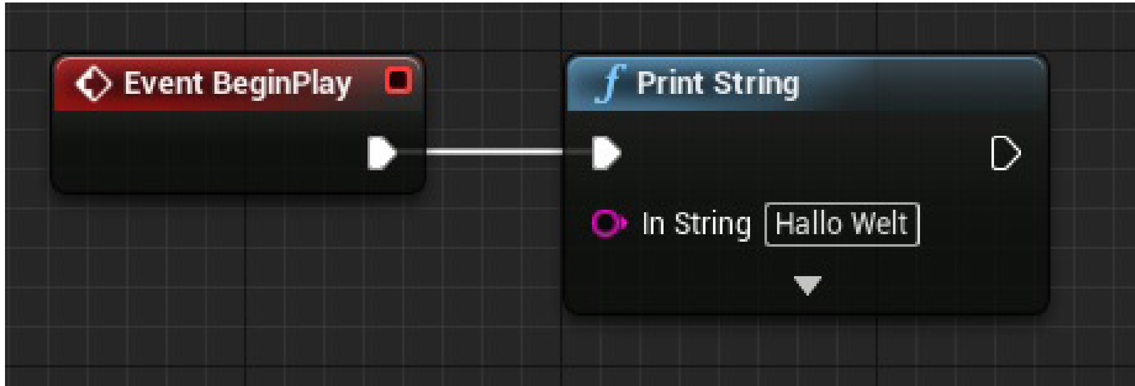

Abb. 25: Zwei Logikbausteine innerhalb eines ***blueprints,*** die eine einfache Logikkette bilden. Das Event links wird beim Ausführen des Spiels aufgerufen und erzeugt dadurch die Anzeige des Textes „Hallo Welt“ auf dem Bildschirm.

Ein einfaches Beispiel: Drückt der Spieler auf einen Knopf, soll das Licht an- bzw. ausgeschaltet werden. Das bedeutet, dass drei Bausteine benötigt werden (lässt man den Spieler selbst der Einfachheit halber erst mal außen vor). Der erste Baustein registriert das Drücken des Knopfes, der zweite Baustein überprüft den gegenwärtigen Zustand des Lichtes und der dritte Baustein schaltet das Licht dann ein oder aus.[91] Diese drei Bausteine bilden damit bereits eine Logikkette und stellen einen Spielinhalt (***gameplay***) dar. Zugegeben wäre dies kein sehr aufregendes Spiel, doch per Definition wäre es bereits eines.

In dem obigen Beispiel mit dem Lichtschalter würde das ***blueprint*** in seiner Gesamtheit dann verschiedene Elemente beinhalten: Der Schalter selbst (vermutlich als 3D-Modell), eventuell eine optische Darstellung des Lichtes in Form einer Glühbirne und die im Hintergrund agierenden Logikbausteine um die Funktion auszuführen, d.h. um das Licht an- und auszuschalten. Alles zusammengefasst ergäbe dies ein sogenanntes ***blueprint***, welches dann benannt und beliebig oft in der Spielwelt platziert werden kann. Man kann sagen, dass der Entwickler für jedes „Ding“ in der Spielwelt, das selbst mit anderen interagiert oder mit dem interagiert werden soll, ein solches ***blueprint*** anlegt.

Dabei gibt es zum einen verschiedene Grundformen von ***blueprints***, zum anderen sind diese allerdings auch völlig frei definierbar. Alle benötigten Komponenten, seien es 3D-Modelle, Lichter, Variablen, Funktionen oder Events können jederzeit zum ***blueprint*** hinzugefügt werden. Dabei reichen die Logikelemente von allgemeinen Funktionen bis hin zu sehr spezialisierten. Für den oben bereits angeführten Spielinhalt mit dem Lichtschalter stünde beispielsweise die Funktion ***toggle visibility*** bereit, die einfach die Sichtbarkeit eines beliebigen Objektes ein- und ausschalten kann. So müsste man für das Licht nicht erst extra zwei Helligkeiten definieren zwischen denen hin und her geschaltet wird.

91 Vgl. Ebd. S. 29

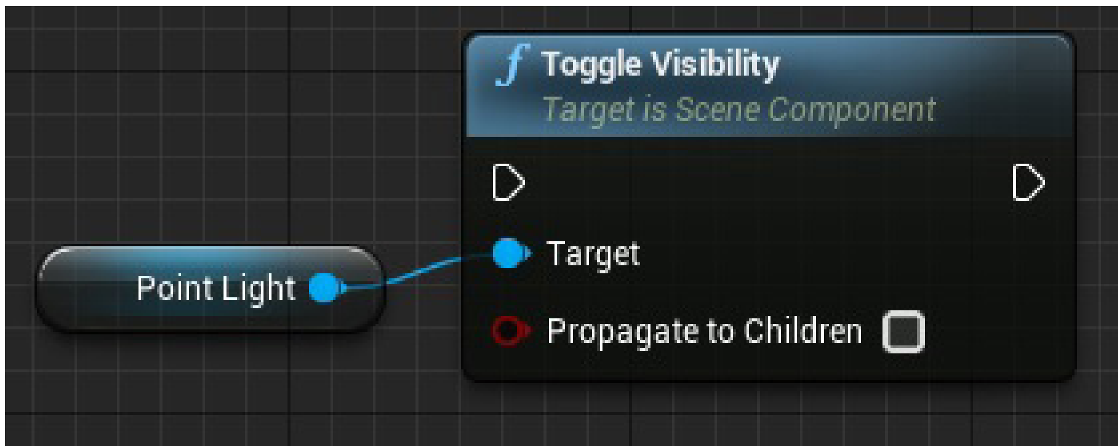

Abb. 26: Die erwähnte Funktion ***toggle visibility,*** die mit einer einfachen Lichtquelle (links) verbunden ist und diese steuern kann.

Da es sich bei den ***blueprints*** zwar um eine optische Programmierschnittstelle handelt, aber dies letztendlich nur eine Vereinfachung darstellt, können Entwickler natürlich auch direkt auf Ebene des Quellcodes einsteigen und ihre eigenen Funktionen, Events usw. verfassen.

Das System der ***blueprints*** stellt dem Programmierlaien zwar ein gutes Fundament zur Verfügung, um einen weniger komplexen Einstieg zu erhalten, gänzlich unvertraut sollte man mit der Materie allerdings nicht sein. Das System wirkt zwar wie ein simples „Plug & Play“, erfordert aber doch ein paar Grundkenntnisse der Programmierung. Man sollte vertraut sein mit Begriffen wie Variablen und deren Typen, Arrays, Vektoren, Schleifen und diversen mathematischen und programmiertechnischen Grundkonzepten. Wer beispielsweise eine Highscore-Liste am Ende seines Spiels anzeigen möchte, der sollte wissen, dass er dies am sinnvollsten mit einem Array macht, welches er mit einer Schleife befüllt und ausliest. Natürlich hängt dies auch von den Ansprüchen des jeweiligen Benutzers ab bzw. welche erklärten Ziele erreicht werden sollen. In der Programmierung gibt es niemals nur den einen richtigen Weg um eine Aufgabe auszuführen. Was hier nur festgehalten werden soll ist, dass die ***blueprints*** dem Unerfahrenen definitiv helfen können, eine gewisse Affinität zum Thema allerdings von Vorteil ist.

5.1.3 Die Objekte in der Spielwelt[92]

Die Spielwelt ist jedem Fall ein komplexes Thema, umso mehr in aufwendig produzierten aktuellen Spieletiteln großer Studios. Die Unreal Engine bringt eine sehr hohe Anzahl an Werkzeugen, Optionen und Einstellungsmöglichkeiten mit, um die Spielwelt nach den ex-

---

92 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 71-118

akten Wünschen des Entwicklers zu gestalten. An dieser Stelle ist dabei noch nicht einmal die Rede von riesigen Landschaften. Hier soll es erst einmal nur um Objekte in der Spielwelt gehen.

Das Platzieren eines Objektes in der Spielwelt ist recht simpel. In der Regel werden die 3D-Modelle in einem externen Programm wie Autodesk Maya oder 3ds Max angefertigt und exportiert. Die Unreal Engine selbst, so wie es für eine Game-Engine üblich ist, bietet keine großen Möglichkeiten, um direkt in der Engine komplexe 3D-Modelle zu erstellen. Nachdem das Objekt in das Projekt importiert wurde, kann es via Drag & Drop einfach aus dem ***content browser*** in den ***viewer*** gezogen werden. Im Fenster mit den Details kann dann die Position näher eingestellt werden. Dabei ist zu beachten, dass die Engine zwischen globalen und lokalen Transformationen unterscheidet. Das bedeutet jedes Objekt besitzt eine Weltkoordinate und eine relative Koordinate. Die Weltkoordinate ist selbsterklärend, sie bestimmt die Position des Objekts im gegenwärtigen Level. Eine relative Koordinate kommt dann zum Einsatz, wenn sich das Objekt innerhalb eines ***blueprints*** befindet. Die Position ist dann abhängig vom absoluten Nullpunkt des jeweiligen ***blueprints***, während die Position des ***blueprints*** an sich durch die Weltkoordinaten beschrieben wird. Welt- und relative Koordinaten stehen dabei in einer Wechselwirkung zueinander, das heißt sie können miteinander kollidieren. Befindet sich beispielsweise ein ***blueprint*** auf einer Höhe von 30, ein gewisses Objekt innerhalb des ***blueprints*** aber auf der Höhe -50 dann könnte dies dazu führen, dass das Objekt in dem Level (der Welt) gar nicht sichtbar ist, weil es sich mit einer Höhe von -20 unter dem Untergrund des Levels befindet. Dies sollte bei der Platzierung von Objekten stets beachtet werden. Das Prinzip von Welt- und relativen Koordinaten findet auch bei den anderen Formen der Transformationen Anwendung, also bei der Rotation und der Skalierung von Objekten.

Für die Beeinflussung von Position, Rotation und Skalierung stehen erneut spezielle Logikelemente bereit, wie beispielsweise das Element ***addRelativeRotation,*** welches bei Aufruf die relative Rotation eines gegebenen Objektes um einen gegebenen Wert verändert. Hierbei handelt es sich allerdings nur um eine Veränderung, die innerhalb eines Frames ausgeführt wird. Soll eine dynamische Veränderung auftreten, z.B. beim Öffnen einer Tür, ist schon eine aufwendigere Kette von Logikelementen notwendig, die mit einer Verzögerung und einer Abfrage, ob die Tür bereits geöffnet ist, arbeitet.

Ein Objekt innerhalb des Levels oder eines ***blueprints*** zu platzieren ist allerdings nur der erste Schritt. Mit einem Doppelklick auf das Objekt lässt sich der ***static-mesh-editor*** öffnen. In dieser neuen Umgebung können eine Vielzahl an Einstellungen rund um dieses eine Objekt getätigt werden. Es ist zum Beispiel möglich, den Objekten einen Sockel zu geben. Dies sind spezielle Punkte, die frei auf dem Objekt platziert und transformiert werden können. Sie dienen dann wiederum anderen Objekten als Ankerpunkt. Angenommen

das Hauptobjekt ist ein einfacher Stein. Diesem Stein kann nun ein Sockel hinzugefügt werden, und dem Sockel daraufhin ein 3D-Modell eines mittelalterlichen Schwerts. Fertig ist Excalibur aus der Legende um König Artus. Der Sockel erlaubt es dem Entwickler darüber hinaus nun durch einfaches Klicken und Auswählen aus dem Schwertmodell beispielsweise eine Lanze zu machen oder ein Banner, ausschließlich durch Austauschen des dem Sockel zugeordneten Objekts. Anpassungen bezüglich der Ausrichtung des neuen Objekts müssen dabei nicht vorgenommen werden, da diese sich nach dem vorher eingerichteten Sockel richten.[93]

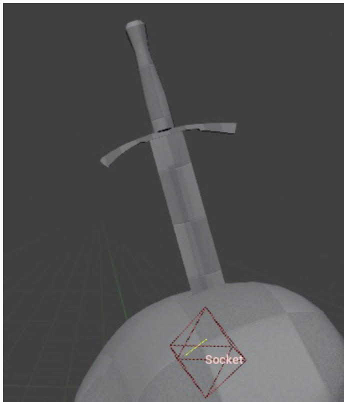

Abb. 27: Das Modell eines Steins an dem ein Sockel (hier in Rot dargestellt) angebracht wurde, der das Modell eines Schwertes hält.

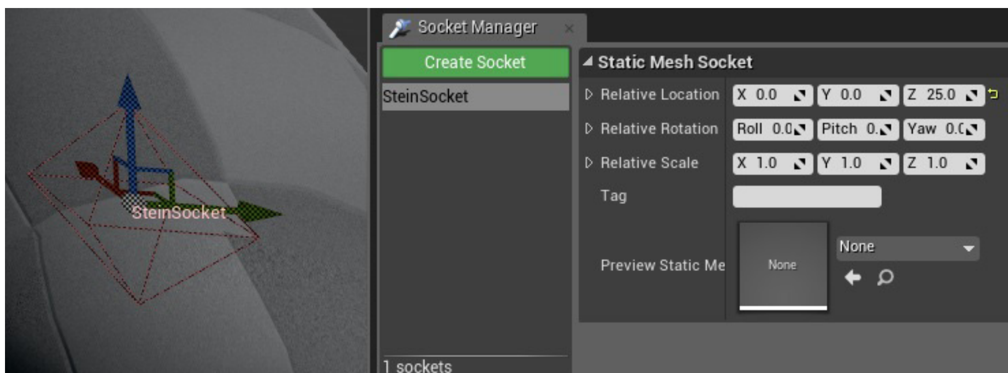

Abb. 28: Verfügbare Optionen des erstellten Sockels zur Steuerung der Position, der Rotation, usw.

Ebenfalls befinden sich im ***static-mesh-editor*** Optionen für die Kollisionserkennung des Objekts. Von der automatischen Berechnung anhand der äußeren Form, über einfache geometrische Formen, bis hin zu manuellen Einstellungsmöglichkeiten für ein höchstpräzises Ergebnis ist hier alles möglich. Präzise Ergebnisse sind dabei nur sinnvoll bei wichtigen Objekten, da die Berechnung der Kollision zur Laufzeit rechenintensiver wird, desto genauer sie sein muss. Für Wände und Türen reichen zum Beispiel schon simple Formen aus, um eine realistische Kollision zu gewährleisten.[94]

---

93 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 83-84

94 Vgl. Ebd. S. 95

Ein Objekt benötigt natürlich auch ein Material, das zum einen die Farbe oder die Textur des Objekts angibt, als auch gewisse andere Eigenschaften definiert. Bei diesem Punkt gibt es sehr viele Überschneidungen zu generellen Themen der Computergrafik, auf die in dieser Arbeit nicht näher eingegangen werden soll. Wie auch ein 3D-Modellierungprogramm bringt die Unreal Engine einen Material-Editor mit den üblichen Funktionen mit. Materialien und Texturen können erstellt (wenn nicht schon vorhanden bei importierten Objekten), zugewiesen und angepasst werden. Beispiele für übliche Einstellungsmöglichkeiten sind Basisfarbe, Leuchteffekte, Reflexion, Rauheit oder Transparenz. Besonders ist dabei allerdings, dass alle Eigenschaften eines Materials durch Logikelemente zur Laufzeit manipuliert werden können. Gibt es in einem Spiel zum Beispiel einen dynamischen Wechsel der Tageszeit, müssen gegebenenfalls manche Materialen verändert werden.

Außerdem können Materialien physikalische Eigenschaften zugewiesen werden. Da, wie an vorheriger Stelle bereits erwähnt, Spiele heutzutage oft Elemente enthalten, die einer physikalischen Simulation bedürfen, kann es notwendig sein, dass verschiedene Materialen verschieden reagieren, wenn Objekte miteinander interagieren. Eine Oberfläche aus Eis sollte beispielsweise eine andere Reibungseigenschaft aufweisen als Beton. Wenn Objekte zerstörbar sein sollen ist zu erwarten dass eine Kiste aus Holz weniger aushält als eine aus Metall. Des Weiteren können verschiedene Typen von Oberflächen definiert werden. Diese Typen sind sehr hilfreich, wenn z.B. verschiedene Geräusche abgespielt werden sollen, wenn der Spieler auf verschiedenen Oberflächen läuft (Beton, Holz, Gras, usw.) Dazu wäre es dann notwendig, wann immer der Spieler sich bewegt, im Hintergrund den Oberflächentyp abzufragen und den entsprechend Sound abzuspielen.[95]

5.1.4 Physik in der Unreal Engine[96]

Viele Spiele haben heutzutage den Anspruch ein gewisses Maß an Realismus zu bieten. Mitunter ist dies sogar ein Qualitätsmerkmal mit dem sich die Entwickler brüsten können. Um diesen Realismus zu erreichen, bedarf es einer physikalischen Simulation für gewisse Teile der Spielwelt. Welche Teile oder Objekte dies sind, hängt vom jeweiligen Spiel ab. Das Aktivieren der Physik-Engine ist in der Unreal Engine relativ simpel. Bei jedem Objekt, egal wo es sich befindet, kann in den Einstellungen durch das einfache Setzen eines Hakens aktiviert werden, dass für dieses Objekt physikalische Kräfte simuliert werden sollen.

---

95 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 117-118

96 Vgl. Ebd. S. 139-163

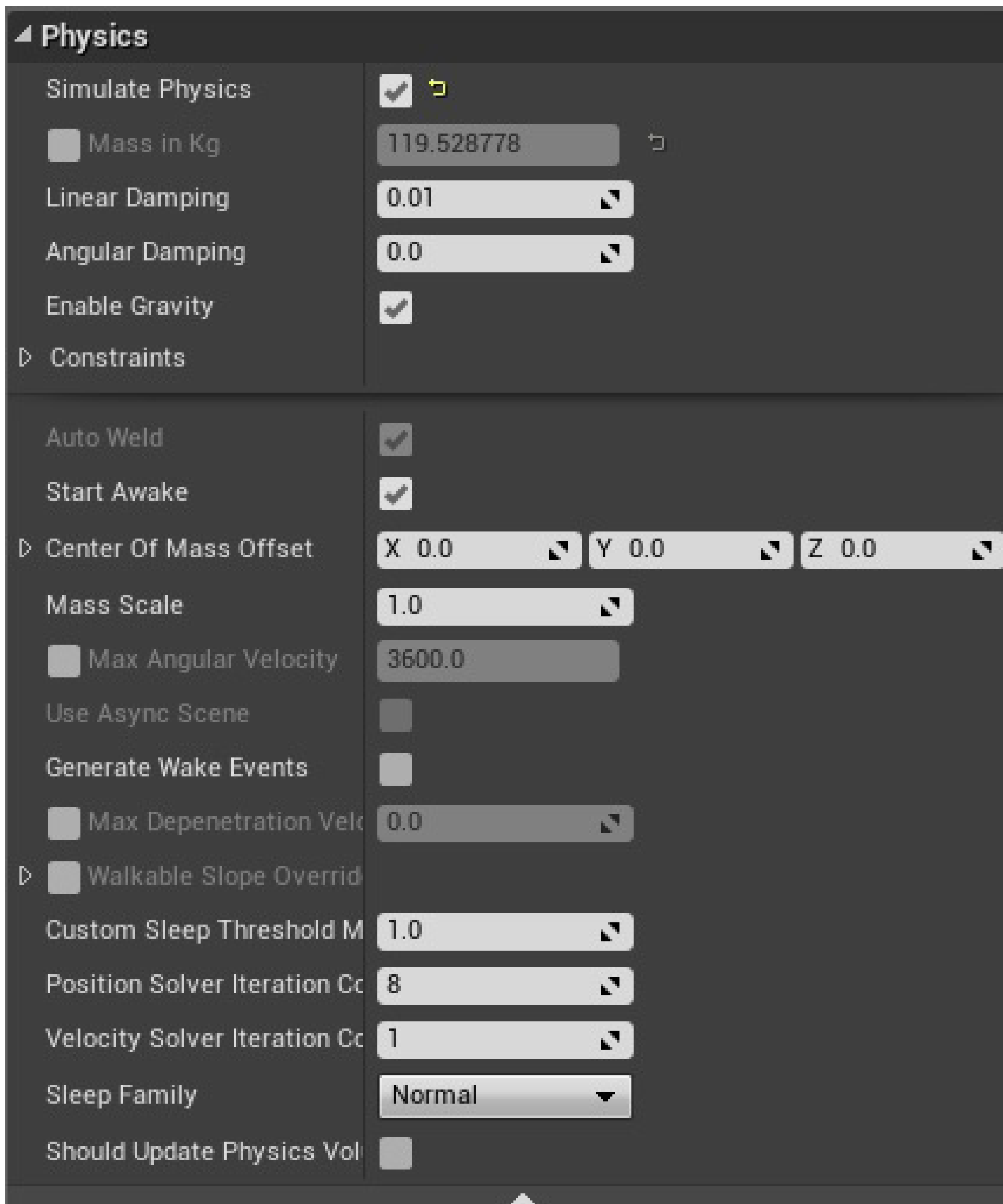

Abb. 29: Übersicht der standardmäßigen Einstellungen für physikalische Simulationen.

Selbstredend gibt es auch hier wieder eine Vielzahl von Einstellungsmöglichkeiten. Beispielsweise kann ganz naheliegend das Gewicht eingestellt werden, das das Objekt haben soll. Dies kann aber auch automatisch auf Basis der Abmessungen berechnet werden. Der Einfluss der Gravitation auf das Objekt kann ein- und ausgeschaltet werden, letzteres wäre nötig, wenn das Objekt wie im Weltall schweben soll. Bestimmte Achsen können deaktiviert werden, um zu erreichen, dass sich Objekte nur in bestimmte Richtungen bewegen können bzw. sie sich nicht in bestimmte Richtungen

bewegen können. Bei einem sogenannten ***sidescroller***[97] (berühmtester Vertreter dieses Genres ist wohl die Super Mario Reihe von Nintendo[98]), bei dem sich der Spieler nur nach links / rechts bewegen kann, kann es Sinn ergeben die Achse zu deaktivieren, auf der prinzipiell Bewegungen in die Tiefe möglich sind. Sie wird ganz einfach nicht gebraucht. Dadurch kann Rechenleistung gespart und eine potenzielle Fehlerquelle vermieden werden.

Darüber hinaus gibt es noch weitere sinnvolle Optionen die, wie in der Unreal Engine üblich, alle über die Logikbausteine ansprech- und manipulierbar sind. Das ist zum Beispiel dann sehr wichtig, wenn der Spieler in der Lage sein soll, einen Gegenstand aufzuheben. In diesem Fall muss ab dem Zeitpunkt, in dem der Spieler den Gegenstand in die Hand nimmt, die Simulation physikalischer Eigenschaften für diesen Gegenstand deaktiviert werden. Ansonsten ist die Gefahr sehr groß, dass unschöne und vor allem physikalisch völlig inkorrekte Dinge auftreten. Genauso müssen die Eigenschaften allerdings wieder aktiviert werden, sobald der Spieler den Gegenstand loslässt, sonst würde dieser einfach in der Luft stehen bleiben, was ebenfalls kein korrektes Verhalten.[99]

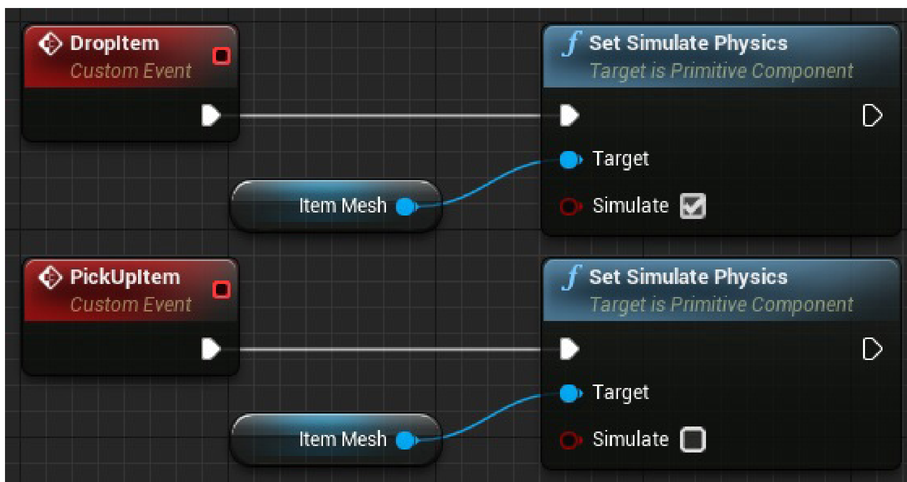

Abb. 30: Vereinfachtes Beispiel zur Realisierung des oben beschriebenen Ein- und Ausschalten der physikalischen Eigenschaften eines Objektes, wenn es aufgehoben bzw. fallen gelassen wird.

Der letzte Teil dieses Kapitels soll dem gewidmet sein, was viele Spieler sehr gerne machen: Dinge zerstören. Das mag kindisch klingen, ist aber ebenfalls ein Qualitätsmerkmal moderner Spiele und gehört zum eingangs erwähnten Maß an Realismus. Der Spieler erwartet einfach, dass etwas passiert wenn er beispielsweise mit seiner Waffe auf eine Holzkiste feuert. Schließlich würde in der Realität auch etwas passieren. Um Objekte zerstörbar zu machen, müssen sie zunächst Teil eines ***blueprints*** sein, und diesem muss die Eigenschaft ***destructible*** zugewiesen werden. Dann muss das betreffen-

---

97 S. Side-Scroller- https://de.wikipedia.org/wiki/Side-Scroller (letzter Aufruf: 12.07.2016)

98 S. S. Super Mario Bros.- https://de.wikipedia.org/wiki/Super_Mario_Bros. (letzter Aufruf: 12.07.2016)

99 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 145-146

de Objekt in ein zerstörbares Objekt umgewandelt werden. Dies geschieht im ***destructible-mesh-editor*** von Unreal. In diesem Editor, der optisch den anderen sehr ähnelt, können erneut eine Vielzahl an Einstellungen getätigt werden. Es muss festgelegt werden, in wie viele Teile das Objekt zerlegt werden soll. Das bedeutet jegliche Zerstörung, die man hinterher im Spiel sehen oder erfahren kann, wurde im Vorfeld bereits festgelegt. Zum Zeitpunkt der Entstehung dieser Arbeit ist es nicht möglich, die Zerstörung eines Objektes dynamisch zur Laufzeit zu berechnen.[100] Mit einem Knopfdruck wird dann die Unterteilung des Objektes in die eingestellte Anzahl an Stücken durchgeführt. Um das Ergebnis besser beurteilen zu können, bietet der Editor die Möglichkeit die einzelnen Stücke etwas voneinander zu entfernen, so als hätte man die Zerstörung nach ein paar Millisekunden eingefroren. Den Innenseiten der Bruchstücke kann dann noch ein neues Material zugewiesen werden, um sie realistischer zu gestalten. Dies ist relativ wichtig, denn je nachdem um welches Objekt es sich handelt, kann der Spieler später in der Lage sein, sich die Trümmer genau anzuschauen.

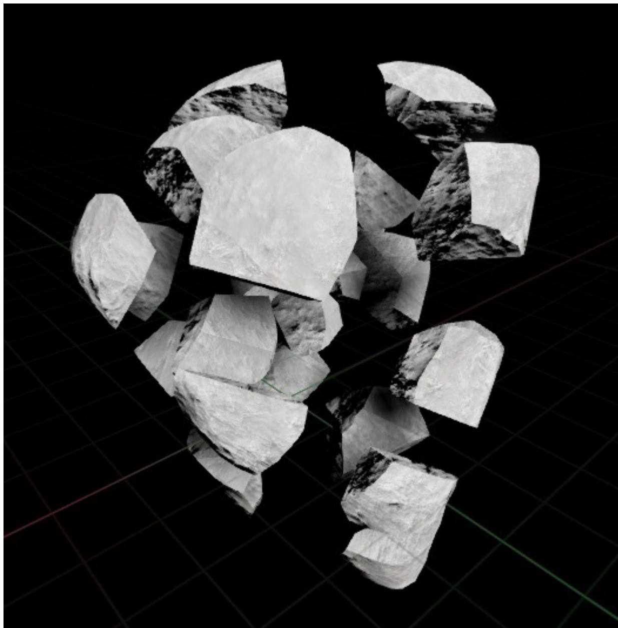

Abb. 31: Beispiel eines zerstörten Steinmodells, dem zwei verschiedene Texturen zugewiesen wurden.

Ferner ist es möglich, einen Schwellenwert für erlittenen Schaden anzugeben, der zunächst überschritten werden muss, ehe die Zerstörung des Objektes ausgeführt wird. Es können Einstellungen zu Partikeleffekten und Sounds erfolgen, so dass die Zerstörung diese automatisch mit auslöst, um abermals realistischer zu wirken. Selbst für jedes einzelne Teilstück eines zerstörten Objektes ist es möglich, Parameter festzulegen. So kann beispielsweise für bestimmte Stücke angegeben werden, dass sie nicht Teil der Zerstörung sind, sie also an ihrem Platz bleiben und keine physikalischen Eigenschaften aufweisen. Dies wäre dann nützlich, wenn der Spieler in der Lage sein soll mittels einer Explosion ein

100 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 148

Loch in eine Wand zu reißen, um weiterzukommen. Würde er die ganze Wand zerstören, statt nur einem kleinen Teil, könnte es dem Maß an Realismus schaden. Der Raum hätte vielleicht nur noch drei Wände, aber das Haus würde nicht komplett einstürzen.

Die Physik-Engine umfasst noch viele weitere, teils sehr komplexe Einstellungen und Möglichkeiten eine Simulation ganz nach den Wünschen des Entwicklers zu gestalten. Für einen groben Einblick sollen die hier erwähnten Themen aber ausreichend sein.

5.1.5 Einen Level erschaffen

Einen Level oder eine ***map*** zu erschaffen ist ein aufwendiger, aber wichtiger Prozess. Die Qualität der Spielwelt ist oft ein entscheidender Faktor für den Erfolg eines Spiels. In diesem Kapitel soll daher auf ein paar Punkte des Entstehungsprozesses eingegangen werden. Erneut muss erwähnt werden, dass in diesem Rahmen leider nicht genug Platz herrscht, um jeden Schritt im Detail zu beleuchten.

Wie bei jedem Projekt empfiehlt es sich am Anfang eine grobe Idee oder einen ersten Plan zu haben, wie die gewünschte Spielwelt oder der Level aussehen soll. Idealerweise wird diese grobe Idee durch eine ungefähre Skizze unterstützt. Anhand dieser Skizze erschafft man dann zunächst einen Prototyp des Levels mit einfachen geometrischen Formen, dieser Schritt wird ***whiteboxing*** oder ***grayboxing*** genannt.[101] Es geht lediglich darum einen spielbaren Level zu erschaffen, das Aussehen ist gänzlich nebensächlich. In der Unreal Engine findet man diese geometrischen Formen unter dem Punkt ***BSP (binary space partitioning)***. ***BSP*** beschreibt dabei eine Technik aus der Informatik, um geometrische Objekte räumlich zu unterteilen. Im Falle eines Computerspiels entsteht so eine hierarchische Struktur, die definiert welche Objekte wichtig und/oder sichtbar sind.[102] Die Technik wird vor allem dann für Teile der Welt verwendet, wenn diese sich in ihren geometrischen Eigenschaften nicht mehr verändern. Mittels der erwähnten einfachen geometrischen Formen wird Stück für Stück der Level aufgebaut. Würfel, Zylinder, Kugeln und andere Formen imitieren dabei die später eingesetzten richtigen und detaillierten Modelle. Die Unreal Engine erlaubt es allerdings bereits mit diesen einfachen Objekten komplexere Strukturen abzubilden. So kann man beispielsweise einen Quader in einem größeren Würfel platzieren, den Quader von dem

101 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 165

102 Vgl. Göth, Christoph (21.02.2001): Binary Space Patitioning Trees.- https://www.uni-koblenz.de/~cg/veranst/ws0001/sem/Goeth.pdf (letzter Aufruf: 12.07.2016)

Würfel subtrahieren und erhält so einen Durchgang, als Platzhalter für eine spätere Tür.[103] Eine solche komplexere Struktur kann dabei sehr einfach in ein tatsächliches 3D-Modell umgewandelt und dadurch beliebig oft in dem Level platziert werden. Ebenfalls können die Platzhalter zur besseren Visualisierung mit Materialien und Texturen versehen werden. Dies erlaubt eine genauere Vorstellung davon, wie der Level später aussehen wird.

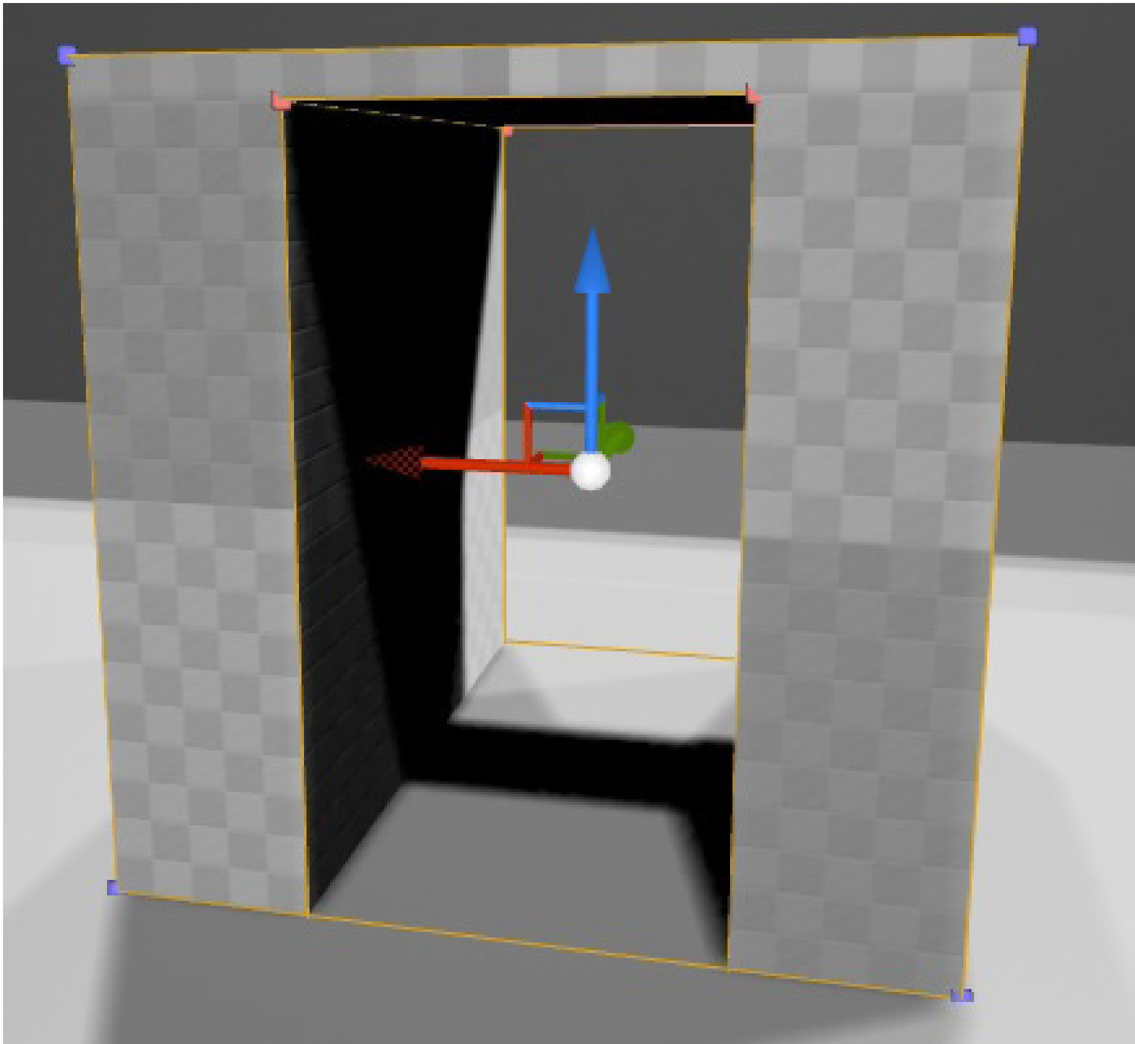

Abb. 32: Zwei voneinander abgezogene geometrische Grundformen zur Darstellung eines einfachen Durchgangs für den Prototypen eines Levels.

Dieser Prototyp eignet sich hervorragend, um erste Spielmechaniken zu testen. Zum Beispiel können bereits erste ***trigger*** Objekte platziert werden. Dabei handelt es sich um einfache Formen, wie Würfel oder ähnliches, die für den Spieler später unsichtbar sind. Sie sind an eine ***overlap***-Funktion gebunden und lösen ein Event aus, wenn beispielsweise der Spieler sie berührt. Das ausgelöste Event startet wiederum eine Logikkette, die einen definierten Effekt hat, wie zum Beispiel das automatische Öffnen einer Tür oder das Laden des nächsten Levels, wenn der Spieler einen bestimmten Bereich betritt. Hierbei sind keine Grenzen gesetzt.

103 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 166-169

Ist der Prototyp soweit fertig und erfüllt seinen Zweck, besteht der nächste Schritt darin, die einfachen geometrischen Formen gegen ihre eigentlichen Modelle auszutauschen. Diese werden dann weiter angepasst und es werden Details hinzugefügt, bis das gewünschte Ziel erreicht und der Level fertig ist.

Die detailreichen einzelnen 3D-Modelle werden wie eingangs schon einmal erwähnt in der Regel in einem anderen Programm erstellt und dann in die Game-Engine importiert. Landschaften allerdings werden traditionell innerhalb der Engine erschaffen. Dafür besitzt die Unreal Engine ein paar wirkungsvolle Werkzeuge.

Man beginnt mit dem Untergrund der Landschaft. Dieser kann dann entweder aus einer anpassbaren Grundfläche selbst per Hand geformt oder mittels einer hinterlegten ***heightmap*** automatisch erzeugt werden. Eine ***heightmap*** meint ein einfaches schwarz/weiß-Bild, das die Höheninformationen angibt. Je heller eine Stelle auf der ***heightmap***, desto höher wird das Terrain an dieser Stelle. Möchte man das Terrain selbst formen, stehen dazu verschiedene Werkzeuge bereit: Erhöhen, Senken, Glätten, eine Rampe zwischen zwei Punkten erzeugen, und es ist sogar möglich zwei verschiedene Arten von Erosion nachzuahmen, um eine natürliche Landschaft zu erhalten. Die Anwendung dieser Werkzeuge ist nicht ganz einfach, ähnlich wie bei anderen sogenannten ***sculpting***[104] (engl. für: Bildhauerei) Programmen. Damit vertraue Künstler sind allerdings in der Lage, täuschend echte Landschaften zu erschaffen.

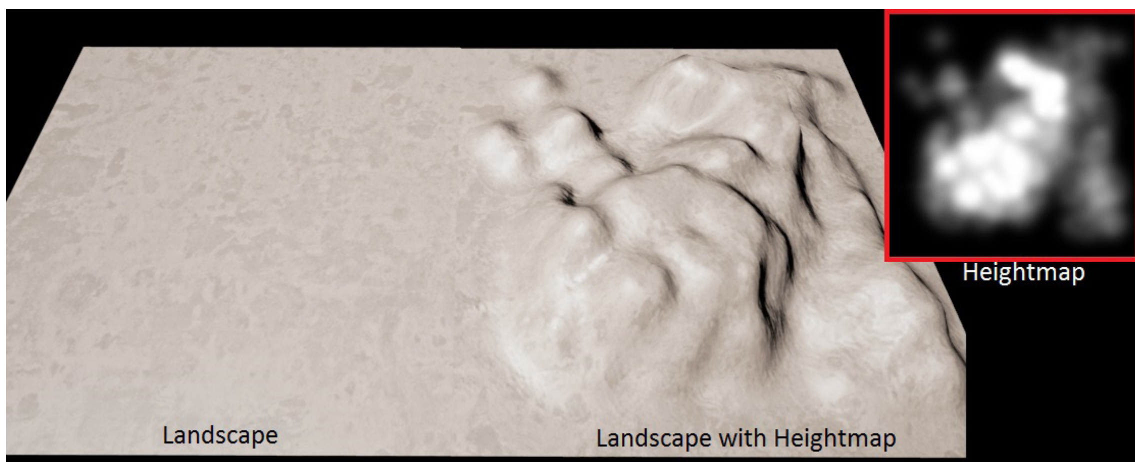

Abb. 33: Beispiel für die Anwendung einer ***heightmap*** auf eine Landschaft. Je heller die Stelle auf der ***map,*** desto höher wird das Gelände dargestellt.

Mit einem Linien-Werkzeug, bekannt aus Programmen wie Photoshop und Illustrator, können direkt auf der Landschaft Punkte und Linien eingezeichnet werden. Schließt man die Linie, kann ihr beispielsweise ein Straßenmodell zugewiesen werden, um sehr einfach

104 S. Digital sculpting- https://en.wikipedia.org/wiki/Digital_sculpting (letzter Aufruf: 12.07.2016)

eine Rennstrecke zu erschaffen.[105] Auch den Eckpunkten können direkt Objekte zugewiesen werden. Auf diese Weise erübrigt es sich, in der Welt oft vorkommende Objekte einzeln platzieren zu müssen. Eine Straßenlaterne wäre in diesem Fall ein Beispiel für ein solches Objekt.

Eine Landschaft mit einem Material zu versehen ist an sich nicht komplexer als es bei einem anderen Objekt der Fall ist. Um ein optisch überzeugendes Ergebnis zu erzielen, ist allerdings etwas mehr Liebe zum Detail notwendig. Eine Rasenfläche ist schließlich auch nicht einfach nur grün. Unter dem Rasen befindet sich der Erdboden der eine eigene, zum Rasen völlig unterschiedliche Struktur und Farbgebung aufweist. Aus diesem Grund können für Landschaftsmaterialien Ebenen von verschiedenen Texturen definiert werden. Bis zu sechs verschiedene Ebenen können in einem Material hinterlegt werden. Dazu gibt es erneut viele verschiedene Optionen, Bausteine und Mischformen. Ein solches Ebenen beinhaltendes ***landscape-material*** kann dann der Landschaft zugewiesen werden. Mit einem Pinsel-Werkzeug ist es dem Künstler möglich, die jeweilige Textur direkt an den korrekten Stellen aufzumalen. Für flachere Stellen wird dann beispielsweise die Ebene gewählt, die Rasen darstellen soll, für steilere Bereiche eine Ebene mit einem etwas mehr felsenartigen Aussehen. Die Weichheit des Pinsels reguliert dabei die Übergänge einer Textur zur anderen.

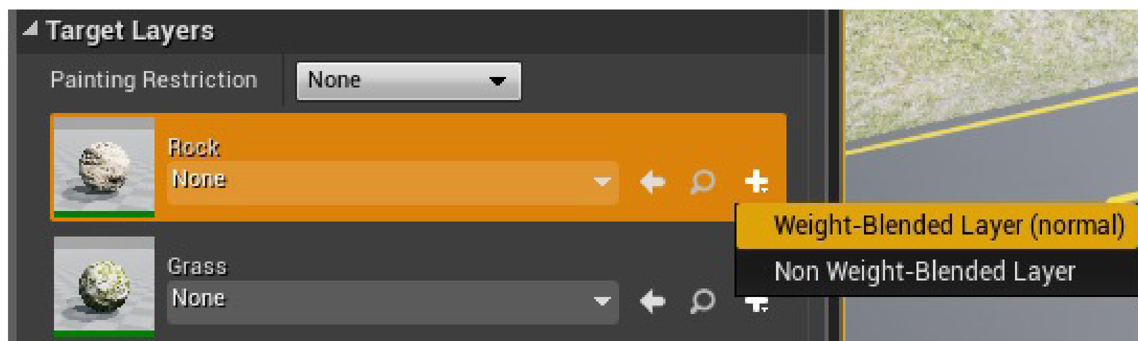

Abb. 34: Beispiel eines Landschaftsmaterials, das zwei verschiedene Ebenen besitzt. Vor dem Auftragen kann die jeweilige Ebene einfach ausgewählt werden.

Zu einer überzeugenden Landschaft fehlt jetzt nur noch etwas Vegetation. Das Hinzufügen dieser gestaltet sich ähnlich wie das Auftragen des Landschafts-Materials auf die Landschaft selbst: mit einem Pinsel-Werkzeug. Die gewünschten Arten der Vegetation werden als 3D-Modelle hinterlegt und können dann mit dem Pinsel einfach aufgemalt werden. Unterstützende Einstellungen sind beispielsweise die Möglichkeit konkrete Objekte, wie die Straße oder die Laternen aus obigem Beispiel, auszuschließen. Das heißt es ist nicht nötig, allzu vorsichtig zu sein, wenn man beim Malen in die Nähe von anderen

105 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 184-187

Objekten kommt, diese Stellen werden schlicht ignoriert. Des Weiteren kann eine Mindest- und Maximalgröße, z.B. für Bäume, definiert werden. Die erzeugten Bäume werden dann zufällig innerhalb des angegeben Bereiches erzeugt. Man kann aber auch eine zufällige Rotation eintragen, damit nicht alle Bäume identisch wachsen. Außerdem kann eine minimale und maximale Steigung definiert werden, innerhalb derer die Vegetation aufgetragen werden soll. Das bedeutet, der Benutzer muss sich keine Sorgen darum machen, versehentlich einen Baum an einer Steilwand oder einem Berghang zu platzieren. Dies vereinfacht den Prozess erneut.[106]

5.1.6 Charaktere und Animationen[107]

Es ist zwar abhängig vom Genre des Spiels, aber in der Regel erwartet der Spieler irgendeine Form von Charakter, durch den er das Spiel erlebt. Dieser Charakter bewegt sich gewöhnlicherweise in irgendeiner Form, das bedeutet seine Bewegungen müssen durch Animationen auch dargestellt werden. Im folgenden Kapitel wird näher darauf eigegangen, wie diese Animationen in der Unreal Engine auf ein Charaktermodell angewandt werden.

Man unterscheidet zwischen dem eigentlichen Modell (auch als ***mesh*** bezeichnet) eines Charakters, dem Skelett und dem ***PhysicsAsset***. Das Modell ist das Aussehen des Charakters. Das Skelett, bestehend aus einzelnen Knochen, kontrolliert mithilfe der Animationen das Modell. Im ***PhysicsAsset*** sind physikalische Eigenschaften des Charakters hinterlegt, die erforderlich sind, wenn gewisse Animationen durch physikalische Simulationen unterstützt werden sollen (s. Kapitel 4.4).

Wird der Charakter in die Engine importiert und geöffnet, wird dies erneut in einem speziellen Editor getan. Hier hat der Entwickler unter anderem Zugriff auf den Skelett-Baum des Charakters, das heißt auf eine hierarchische Auflistung der einzelnen Knochen, die den Charakter später einmal steuern sollen. An jeden einzelnen Knochen können, wenn erforderlich, Sockel platziert werden. Diese haben die gleiche Funktion wie die in Kapitel 5.1.3 bereits Erwähnten. An ihnen kann ein Objekt angebracht werden, das dann dem Sockel folgt. So wird zum Beispiel gewährleistet, dass sich die Waffe des Charakters in

---

106 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 198-201

107 Vgl. Ebd. S. 303-327

einem ***third-person shooter***[108] immer korrekt mitbewegt. Zur Laufzeit des Spiels können dann die Objekte durch ***blueprints*** ausgetauscht werden. Wurden die Modelle im Vorfeld richtig angelegt, das heißt richtig orientiert, ist das visuelle Ergebnis bei jedem Objekt stets korrekt.[109]

Neben dem Skelett-Baum werden auch alle importierten Animationen in diesem Editor angezeigt. Durch einen Doppelklick können sie zu Vorschauzwecken abgespielt werden. Wie bereits im Zusammenhang mit den 3D-Modell erwähnt, ist es auch bei den Animationen üblich, diese in einem anderen Programm anzufertigen und dann von dort zu importieren. Dabei kann es vorkommen, dass der Animator zum Beispiel alle Animationen, in denen sich ein Charakter umsieht, in einer Datei abspeichert. Dies ist oft einfacher als immer wieder von Null an neun verschiedene Animationen anzufertigen. Diese Animation, in der der Charakter in alle Richtungen (vorne, links, rechts, linksunten, linksoben, usw.) sieht, kann innerhalb der Engine dann zunächst in die Extremposen zerlegt und als 1-Frame Animationen gespeichert werden. Mit einer speziellen Funktion, der ***aim offset***, können diese Extremposen dann ihrer Richtung entsprechend in einem Graphen angeordnet werden. Die Engine interpoliert daraus die korrekte Animation auf Basis der Eingabe des Spielers. Steht der Charakter also teilnahmslos herum, sieht er gerade aus nach vorn. Erfolgt dann eine Eingabe, er soll nach rechts sehen, springt der Charakter nicht sofort in die Extrempose in der er nach rechts sieht, sondern die Engine spielt die gesamte und richtige Animation ab, wie der Charakter seinen Oberkörper und Kopf dreht und dann letztlich in der Extrempose ankommt. Unglücklicherweise handelt es sich hier um eine Technik, die ohne Bewegte Bilder nur sehr schwer nachvollziehbar ist. Um die Möglichkeiten der Unreal Engine aufzuzeigen, findet sie hier dennoch Erwähnung.[110]

---

108 Ähnlich den ***first-person shootern*** bezeichnet auch der ***third-person shooter*** eine Kategorie von Computerspielen. Hierbei befindet sich die Kamera allerdings hinter dem Spieler-Charakter, anders als die Egoperspektive beim ***first-person shooter***. Nimmt die Kamera eine Schulterperspektive ein fällt das Spiel ebenfalls in die Kategorie der ***third-person shooter***.

Vgl. Third-Person-Shooter- https://de.wikipedia.org/wiki/Third-Person-Shooter (letzter Aufruf: 04.08.2016)

109 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 305-307

110 Vgl. Ebd. S. 309-310

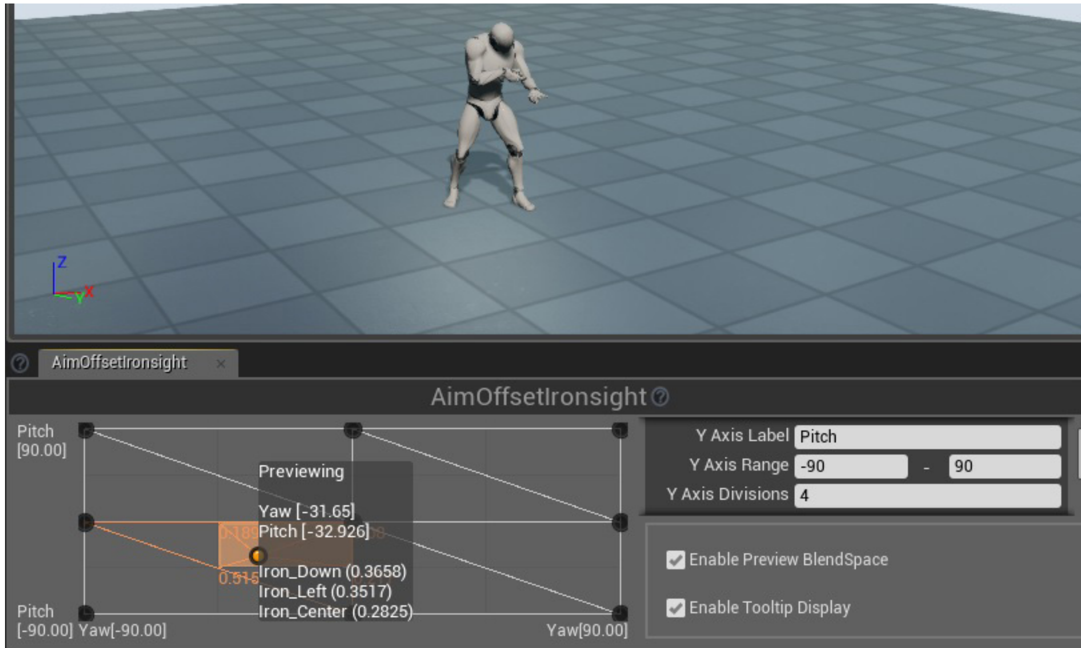

Abb. 35: Darstellung des ***aim offset*** Graphen. Die Extremposen der Animation (dargestellt als schwarze Punkte) werden ihrer Richtung entsprechend angeordnet. Das heißt „nach rechts schauen“ auf die rechte Seite des Graphen usw. Die Engine interpoliert alle nötigen Werte, um eine flüssige Animation zu erzeugen. Soll der Charakter beispielsweise weiter nach oben schauen können als nach unten, wäre es hier möglich, die unteren Punkte etwas nach oben zu verschieben. Dadurch wird der Charakter in seinen Extremwerten eingeschränkt.

Bei längeren Animationen ist es möglich sogenannte ***notifies***, also Benachrichtigungen, zu erstellen. Dabei handelt es sich um präzise, auf die Animation abgestimmte Events. Wenn der Charakter beispielsweise sein Schwert aus der Scheide ziehen soll, kann ab dem Moment, ab dem er die Hand am Griff des Schwertes hat, ein Event gestartet werden, welches dann eine Logikkette in Gang setzt. Denkbar hier wäre zum Beispiel das Abspielen eines passenden Soundeffekts, solange das Schwert aus der Scheide gezogen wird.[111]

Die zentrale Steuereinheit für alle Animationen eines Charakters ist das ***animation blueprint***.[112] Hier sind alle essentiellen Logikelemente enthalten, um die Animationen zu kontrollieren und um von außen mit dem Charakter zu kommunizieren. Hauptelement ist zum einen der Baustein ***event blueprint update animation***. Dieser ist dafür zuständig, mit jedem Aktualisieren des Bildes Informationen über den gegenwärtigen Zustand des Charakters zu sammeln und bei Bedarf zu übermitteln. Beispiele für diese Informationen sind die derzeitige Bewegungsgeschwindigkeit des Charakters oder

111 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 308-309

112 Vgl. Ebd. S. 312-323

seine Position im Level.[113] Ein weiteres Hauptelement des ***blueprints*** ist der ***anim graph***.[114] Hier wird der logische Ablauf der Animationen dargestellt und anhand der jeweiligen Informationen bestimmt, welche Animation abgespielt werden soll. Ein simples Beispiel wäre folgende Logikkette: Wenn Geschwindigkeit = 0, spiele Animation „Rumstehen“ ab. So könnte für jede vorhandene Animation bzw. für jede Tätigkeit eine Abfrage erstellt werden. Vereinfacht formuliert z.B.: Wenn der Spieler die W-Taste drückt, spiele die Animation „nach vorne Laufen ab“, usw. Eine solche Vorgehensweise ist zwar nachvollziehbar und wäre funktional, führt allerdings sehr schnell zu Unübersichtlichkeit im Graphen.

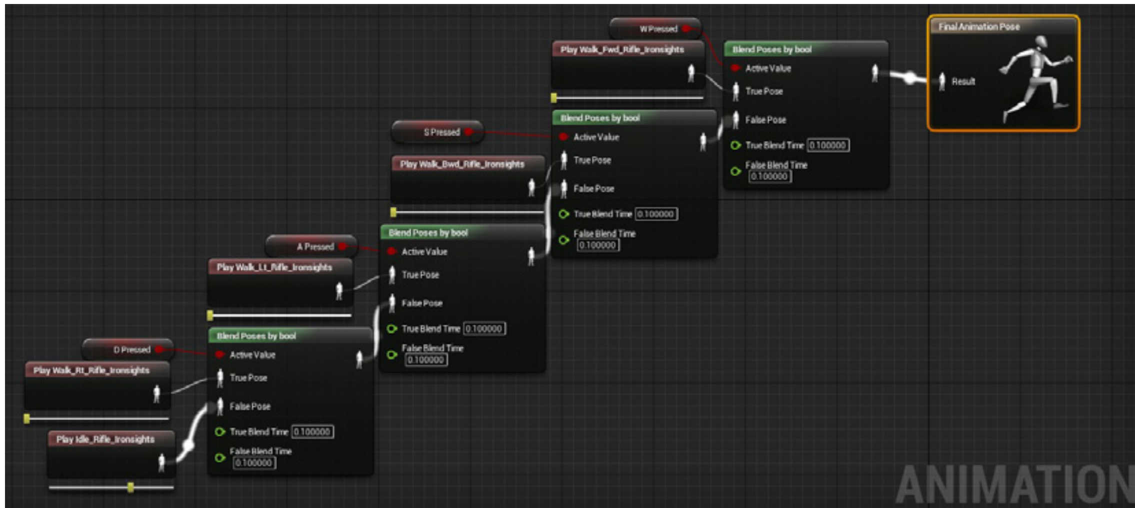

Abb. 36: Beispiel eines ***anim graphs*** in dem logische Abfragen genutzt werden, um die verschiedenen Animationen zu steuern. Wird „D“ gedrückt, spiele Animation „nach rechts drehen“ ab, usw. Eine solche Verfahrensweise ergibt schnell einen sehr komplexen und unübersichtlichen Graphen.

Aus diesem Grund können in der Unreal Engine sogenannte ***state machines*** angelegt werden. Dies sind, wenn man so möchte, eine Art Unterordner für die Animationen. Es kann für jeden Zustand des Charakters (stehen, gehen, kriechen, springen) eine ***state machine*** definiert werden, die sich dann wiederum nur um die Parameter und Bedingungen einer einzelnen Animation kümmert. Dieses Container Prinzip schafft eine wesentlich bessere Übersicht. Mit den ***states*** kann aber nicht nur eine bessere Übersicht erreicht werden, es können ebenfalls Bedingungen definiert werden, um von einem Zustand in einen anderen zu gelangen. Angenommen der Charakter steht teilnahmslos herum, dann drückt der Spieler die Leertaste und der Charakter signalisiert dem ***animation blueprint,*** dass er jetzt springt. Dann muss die Sprung-Animation ausgeführt werden. Der Charakter wechselt also vom Stehen- in den Sprung-Zustand. Dafür ist zunächst einmal keine Bedingung notwendig. Um jetzt aus dem Sprung-Zustand wieder herauszukommen, wäre es sinnvoll erst einmal die Frage zu klären, ob der Charakter wieder gelandet ist. Am einfachsten ist da

113 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 312

114 Vgl. Ebd. S. 316-323

eine Abfrage, ob die Sprung-Animation komplett abgespielt wurde, so lange müsste der Sprung ja mindestens dauern. Ist dies der Fall, kann weiter überprüft werden, ob sich der Charakter im Fall befindet, also ob er von seinem Sprung aus nun auf dem Rückweg gen Boden ist, oder ob er ein Objekt hinuntergesprungen ist. Dieses könnte mit einer Variablen, in der die Bewegungsgeschwindigkeit gespeichert wird, abgefragt werden. Befindet sich der Charakter im Fall, muss, sofern vorhanden, die Fall-Animation abgespielt werden. Befindet er sich nicht mehr im Fall, kann der Charakter wieder in den Stehen-Zustand gesetzt werden.

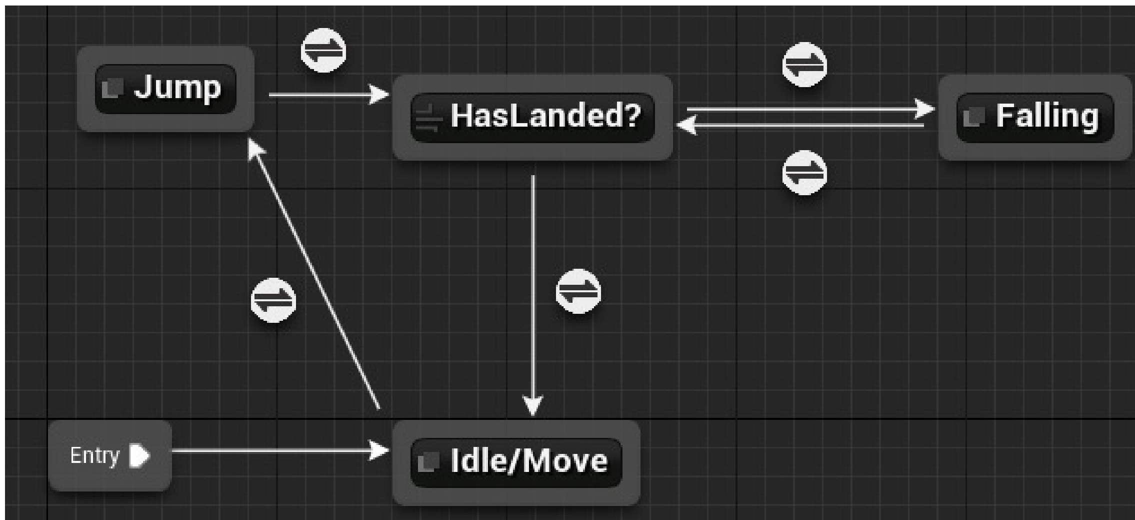

Abb. 37: Die ungefähre ***state machine*** für obiges Beispiel.

Dies ist jetzt nur eine vereinfachte Beschreibung der Logik, die hinter dem System steckt, aber sie soll auch nur der Veranschaulichung dienen. Der große Vorteil der ***state machines*** ist die Übersicht, die ermöglicht wird. Die Anzahl an Animationen die ein einzelner Charakter mit sich bringt, sollte nicht unterschätzt werden und die Zustände erlauben, es jede Animation in ihrem eigenen kleinen Umfeld zu handhaben.

5.1.7 KI – Künstliche Intelligenz[115]

Die künstliche Intelligenz der Nicht-Spieler-Charaktere in einem Spiel ist eines der komplexesten Elemente in der Spieleentwicklung. Sie grundlegend einzurichten, damit sie rudimentäre Funktionen übernimmt, ist dabei nicht das Problem, wie dieses Kapitel zeigen soll. Die Schwierigkeit stellt sich im ausbalancieren der KI dar. Sind die Gegenspieler zu intelligent oder unterfordern sie den Spieler, schlägt sich dies direkt negativ auf den Spielspaß nieder und führt damit zu einem Qualitätsverlust des gesamten Spiels.

115 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 343-353

In der Unreal Engine unterscheidet sich ein Nicht-Spieler-Charakter (kurz: NSC) nicht allzu sehr von einem Spieler-Charakter. Statt auf Eingaben des Spielers zu reagieren, wie der Charakter es tut, wird der NSC von einem ***ai controller***[116] gesteuert, welcher erneut einem ***blueprint*** hinzugefügt wird. In den Optionen des NSC kann dann eingestellt werden, dass dieser Charakter von eben diesem ***ai controller*** gesteuert wird. Zusätzlich benötigt der NSC ebenfalls ein ***animation blueprint,*** welches die Animationen steuert, wie im vorherigen Kapitel erläutert. Handelt es sich bei dem NSC um einen Charakter, der dem Spieler-Charakter ähnlich ist, ist es möglich die Animationen beider Charaktere von einem einzelnen ***blueprint*** kontrollieren zu lassen. Wurde dem NSC eine Steuereinheit zugewiesen, fehlt für das Fundament nur noch ein sogenanntes ***nav mesh,*** also ein Navigationsnetz. Dabei handelt es sich, ähnlich wie bei den ***triggern***, um eine unsichtbare Box, die um den Level gespannt wird in denen sich der NSC bewegen können soll. Auf diese Weise können genau die Bereiche markiert werden, zu denen der NSC Zutritt haben soll und zu welchen nicht. In den Projekteinstellungen können die Parameter dieses Netzes noch detaillierter eingestellt werden. Zum Beispiel können Winkel definiert werden, die die KI überwinden können soll. Je nach Einstellung kann dies schon verhindern, dass ein NSC eine simple Treppe erklimmen kann.

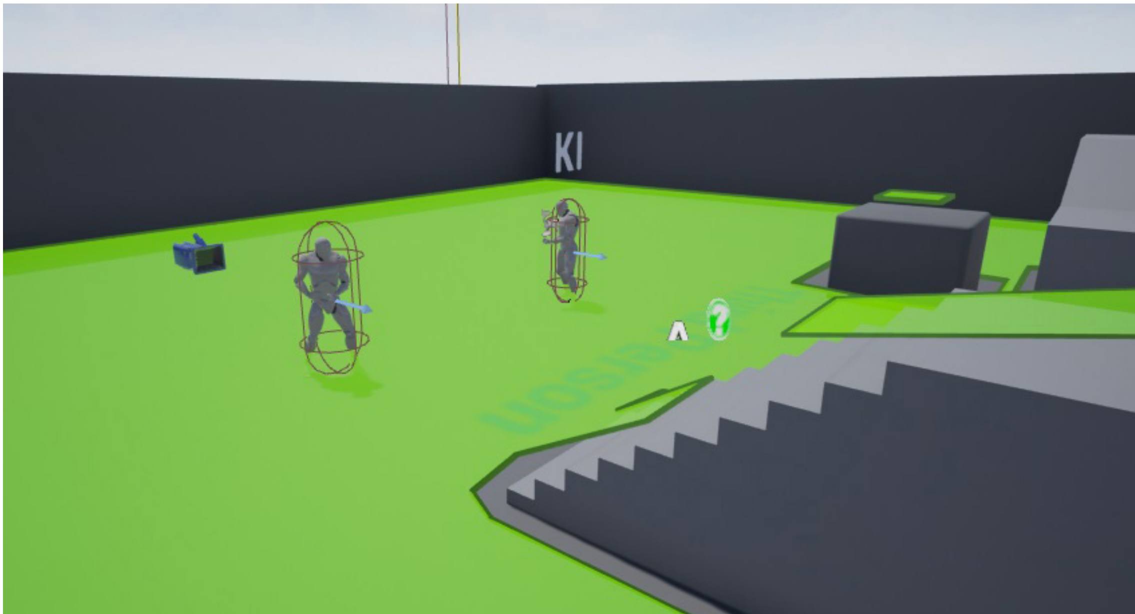

Abb. 38: Beispiel für ein ***nav mesh***, hier in grün eingezeichnet.

Das ***nav mesh*** reguliert den Bereich in dem sich ein NSC bewegen kann. Damit er sich tatsächlich bewegt, bedarf es aber noch einiger Logikelemente. Um beispielsweise einen NSC von A nach B laufen zu lassen, also eine einfache Patrouille, müssen zunächst Wegpunkte im Level definiert werden. Dafür gibt es in der Unreal Engine einfache Punkte, genannt ***billboards.*** Ein solcher Punkt wird in einem weiteren ***blueprint*** platziert und dieser kann dann

116 „ai" steht für den englischen Ausdruck ***artificial intelligence*** und bedeutet künstliche Intelligenz.

im Level verteilt werden. In diesem Beispiel werden zwei Wegpunkte/***blueprints*** benötigt. Dem NSC wird daraufhin eine Variable hinzugefügt, in der die Wegpunkte gespeichert werden können. Genauer gesagt handelt es sich dabei um ein Array. Das heißt, der NSC kennt jetzt bereits alle Wegpunkte, die für ihn gedacht sind und weiß wo sie sich im Level befinden. Damit er sich nun bewegen kann, müssen die entsprechenden Logikbausteine im ***ai controller*** angelegt werden. Es wird eine Variable benötigt, um auch innerhalb des ***ai controllers*** die Wegpunkte speichern zu können. Der ***ai controller*** ist dann in der Lage mit dem NSC über einen ***cast to***-Befehl zu kommunizieren, die dort hinterlegten Wegpunktdaten auszulesen und lokal in der angelegten Variablen zu speichern. Durch einen ***move to***-Befehl wird der NSC dann zum ersten Wegpunkt geschickt.

Mit dem Baustein ***get move status*** kann abgefragt werden, ob der NSC sich noch in der Bewegung befindet. Ist dies nicht der Fall ist erst einmal davon auszugehen, dass er den ersten Wegpunkt erreicht hat. Daraufhin wird das nächste Element innerhalb des Arrays, also der zweite Wegpunkt, ausgelesen und erneut der ***move to***-Befehl ausgeführt. Ist der Algorithmus am letzten Element des Arrays angekommen, also sind keine weiteren Wegpunkte mehr hinterlegt, kann wieder zum ersten Element gesprungen, werden damit der Charakter zum ersten Wegpunkt zurückläuft. Alternativ wird nur ein Element zurückgesprungen, damit der NSC zum vorherigen statt zum ersten Wegpunkt läuft. Da sind dem Entwickler keine Grenzen gesetzt. Bei insgesamt nur zwei Wegpunkten spielt dies natürlich keine Rolle. Doch sobald mindestens drei Punkte definiert sind, entstehen zwei verschiedene Bewegungsmuster. Im ersten Fall wäre es A>B>C>A>B>C, im zweiten Fall A>B>C>B>A.

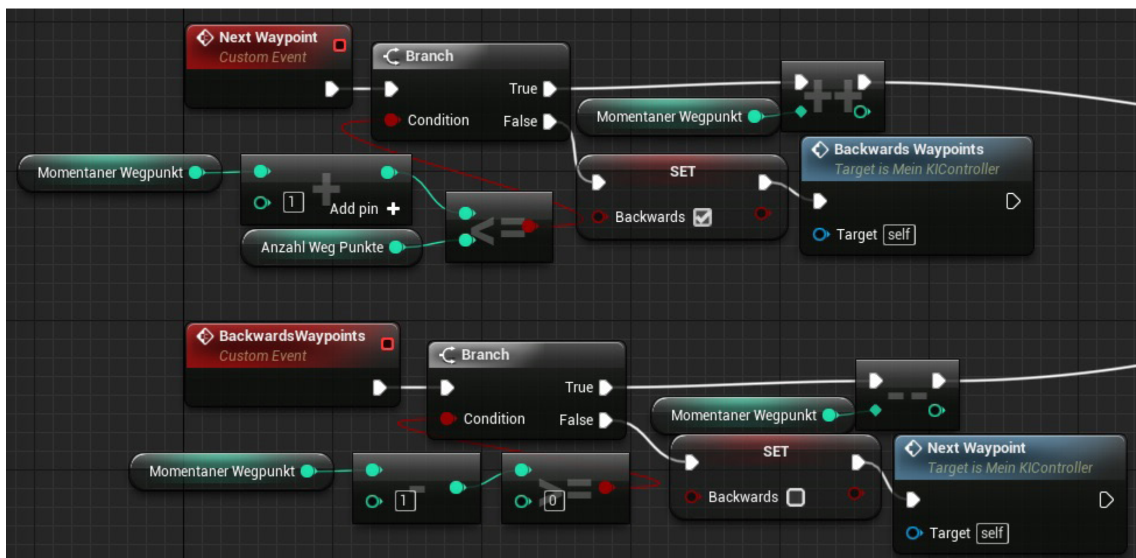

Abb. 39: Beispiel für eine Logikkette um einen NSC entweder zum nächsten oder vorherigen Wegpunkt laufen zu lassen. Eine vermeintlich simple Aufgabe erfordert schon einige Bausteine, um sie durchzuführen.

Ein Gegenspieler, der nur von A nach B läuft, würde natürlich in die Kategorie fallen, dass er den Spieler unterfordert. Also muss noch eine Interaktion zwischen Spieler-Charakter und NSC erfolgen.
Dem NSC kann beispielsweise ein kegelförmiges, später im Spiel unsichtbares Volumen gegeben werden, das als Sichtfeld des Charakters dient. Mit dem Logikbaustein ***onComponentBeginOverlap*** kann dann abgerufen werden, ob irgendein anderes Objekt mit diesem kegelförmigen Volumen kollidiert. Handelt es sich dabei um den Spieler-Charakter, was zunächst eine Kommunikation mit dem Spieler-Charakter via dem ***cast to***-Befehl bedarf, kann ein beliebiges Event ausgeführt werden. Ganz gleich ob sich der NSC nur auf den Spieler-Charakter zubewegen, seine Waffe ziehen oder direkt auf ihn feuern soll. Dabei ist zu beachten, dass die Kollision des Spielers mit dem Sichtfeld gleichzeitig die Patrouille abbrechen sollte, um logische Fehler zu vermeiden. Ist die Abfrage, ob sich der Spieler-Charakter im Sichtfeld des NSC befindet, nicht mehr korrekt, weil dieser sich bewegt hat, können spezielle Events wie z.B. ***searchLastKnownPosition*** definiert werden, um den Spieler verfolgen zu lassen. In diesem Feld gibt es buchstäblich Milliarden Möglichkeiten, wie eine KI handeln und reagieren soll.[117]

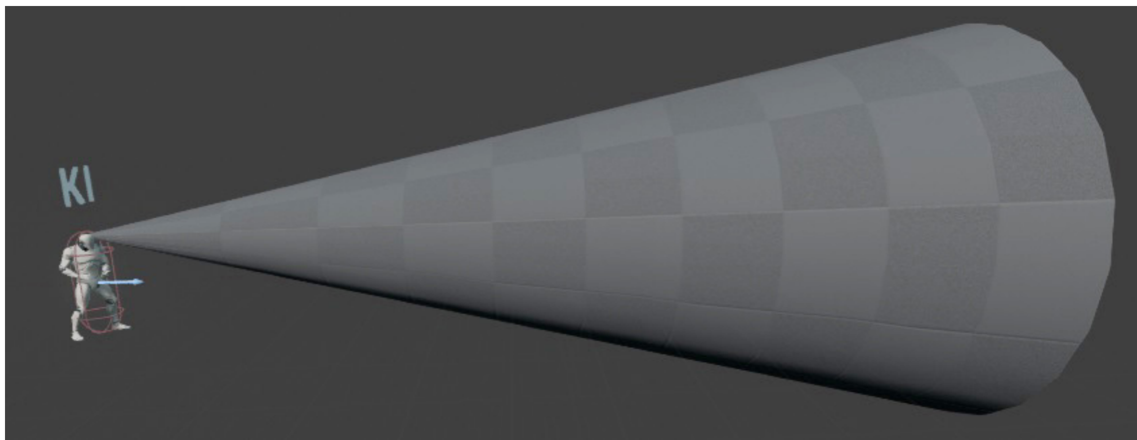

Abb. 40: Visualisierung des kegelförmigen Objektes das als Sichtfeld des NSCs fungiert. Im späteren Spiel wäre es natürlich für den Spieler nicht sichtbar.

Damit ist dieses Kapitel über die grundlegenden Funktionen und Prinzipien zur Entwicklung einer künstlichen Intelligenz in der Unreal Engine 4 auch schon abgeschlossen. Wie eingangs erwähnt, ist dies ein sehr breites Feld mit ebenso großer Tiefe. Ob und wie die Gegenspieler in einem Spiel reagieren, war schon immer ein wichtiges Kriterium für die Qualität eines Spiels. Entsprechend wird diesem Feld viel Aufmerksamkeit zuteil.

Das Kapitel über die Unreal Engine kommt damit ebenfalls zu seinem Ende. Es sollte dazu dienen, einen ersten Einblick in die Software und den Entwicklungsprozess eines Spieles

117 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. 352

zu gewähren. Im nun folgenden Kapitel steht die Unity Engine im Fokus. Diese unterscheidet sich zwar in vielen Punkten von der Unreal Engine, doch gibt es naturgemäß auch viele Überschneidungen. Aus diesem Grund werden gewisse Bereiche oder Aspekte gegebenfalls ausgespart, um dafür etwaigen Unterschieden mehr Platz zu gewähren.

5.2 Die Unity 5 Engine

Die Unity Engine wurde von der dänischen Firma Unity Technologies[118] entwickelt und liegt zum gegenwärtigen Zeitpunkt in der 5. Generation vor, daher die übliche Bezeichnung Unity 5. Die Engine startete, so wie viele Vertreter ihres Bereiches, als Entwicklungsumgebung für 3D Computerspiele. Inzwischen wird sie aber auch für Projekte in anderen Bereichen eingesetzt. Dazu gehören ebenfalls interaktive 2D Spiele sowie Architekturvisualisierungen, der E-Learning-Bereich und die Digital-Signage-Branche[119] in der Inhalte für Werbe- und Informationssysteme erstellt werden.[120]

Wie auch die Unreal Engine, ist die Unity Engine für alle semi-professionellen und Hobby Entwickler in einer kostenlosen Personal-Edition verfügbar. Dabei steht dem Anwender alles Nötige zur Verfügung, um das Spiel seiner Wahl zu entwickeln. Eine Nutzung zu kommerziellen Zwecken ist ebenfalls erlaubt, allerdings nur bis zu einem Maximalumsatz von 100.000 US-Dollar pro Geschäftsjahr. Für alles, was darüber hinausgeht existiert eine Professional-Edition mit ein paar zusätzlichen Eigenschaften. So gut wie alle Elemente aus früheren professionellen Versionen sind allerdings seit 2015 in der Personal-Edition bereits vorhanden und können genutzt werden.[121] Egal ob Personal- oder Professional-Edition, mit der Unity Engine können Spiele und Inhalte für alle gängigen Plattformen entwickelt werden (PC, Konsolen, mobile Endgeräte). Darüber hinaus ist es außerdem möglich, Anwendungen für Webbrowser zu kreieren, die über ein von Unity eigens entwickeltes Plug-In abgespielt werden können.[122]

Die Engine verfügt allerdings nicht über viele Werkzeuge oder komplexe Mechaniken um aufwendige 3D-Modelle zu erschaffen. Auch hier gehen die internen Möglichkeiten nicht über ein paar einfache geometrische Objekte hinaus. Für eine Game-Engine der heutigen Zeit allerdings nichts Ungewöhnliches.

---

118 S. Unity- https://unity3d.com/public-relations (letzter Aufruf: 13.07.2016)

119 S. Digital Signage- https://de.wikipedia.org/wiki/Digital_Signage (letzter Aufruf: 13.07.2016)

120 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 1

121 Vgl. Ebd. S. 4

122 Vgl. Ebd. S. 2

5.2.1 Die grafische Oberfläche[123]

Die grafische Benutzeroberfläche der Unity Engine folgt den üblichen Gestaltungsprinzipien in der Softwareentwicklung. Sie weist ebenfalls mehrere verschiedene Fenster auf, die dem Benutzer Informationen, Einstellungsmöglichkeiten und Werkzeuge zur Verfügung stellen, die für die Spieleentwicklung erforderlich sind. Im Genaueren sind das:

1. ***scene view***: Dieser Bereich dient der interaktiven Gestaltung der 3D-Szenen für die Spielwelt.
2. ***game view***: Dieses Fenster ist der Vorschaubereich für das fertige Spiel und wird immer dann automatisch geöffnet/hervorgeholt, wenn das Spiel zu Testzwecken gestartet wird.
3. ***hierarchy***: Die hierarchy stellt eine Auflistung aller Objekte, die gegenwärtig in der jeweiligen Szene existent sind, dar. Ihre hierarchische Struktur wird ebenfalls abgebildet.
4. ***inspector***: Wird ein Objekt ausgewählt, zeigt der ***inspector*** alle zugehörigen Komponenten und Parameter des Objektes an.
5. ***project browser***: Im ***project browser*** werden alle digitalen Inhalte (***assets***) angezeigt und verwaltet. Von dort aus können sie via Drag & Drop in die Spielszene gebracht werden.
6. ***console***: Die ***console*** ist die Schnittstelle der Engine mit dem Benutzer und dient dazu Fehler- und Hinweismeldungen auszugeben.

Wie üblich können die Fenster und Bereiche der Oberfläche frei den Wünschen des Benutzers entsprechend angepasst werden. Weitere Optionen wie beispielsweise verschiedene Darstellungen der Szene (mit oder ohne Licht, Drahtgittermodelle, usw.) sind in die jeweiligen Bereiche mit integriert und können dort vorgefunden werden.

123 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 2-22

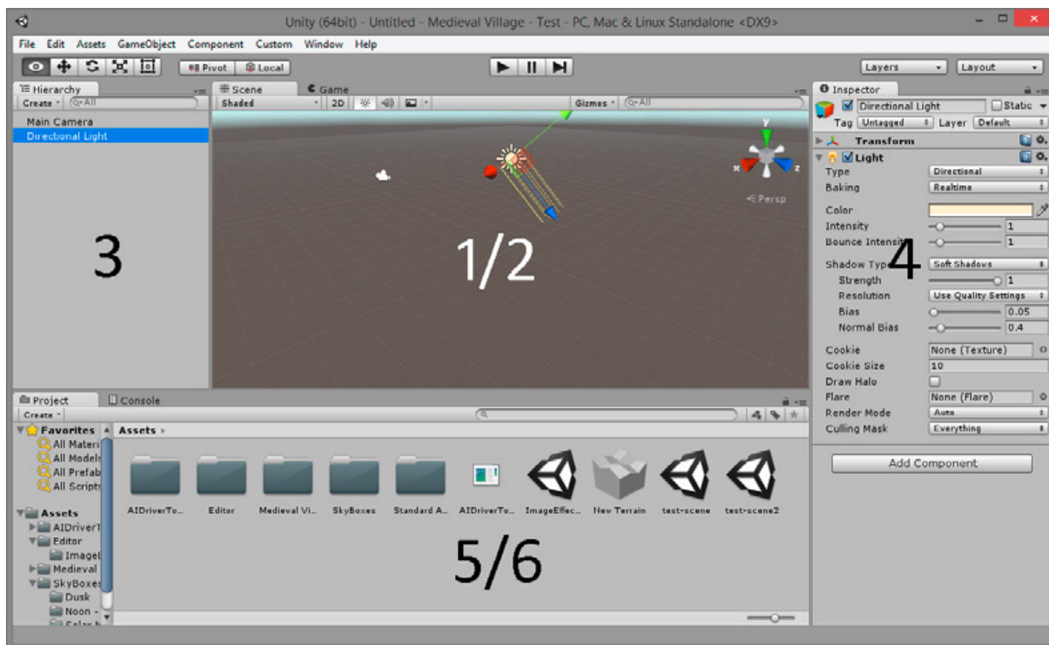

Abb. 41: Die grafische Benutzeroberfläche der Unity Engine mit den sechs oben erwähnten Hauptbereichen.

5.2.2 C# in Unity

C# (oder auch C-Sharp[124]) ist die Programmiersprache auf der die Unity Engine basiert. In dieser objektorientierten Sprache werden alle individuellen Logikelemente verfasst, die für das jeweilige Spiel notwendig sind. Die Engine beinhaltet zwar bereits ein paar vorgefertigte Komponenten in den Bereichen Physik, Audio oder Partikeleffekte, die eigentlichen Kernelemente müssen allerdings vom Entwickler selbst geschrieben werden.[125]

Dies erfolgt über Skripte, also über Textdateien, die beim Ausführen des Spiels in Maschinensprache übersetzt werden, damit der Computer sie direkt anwenden kann. Dabei stehen dem Entwickler alle üblichen Konzepte und Elemente einer objektorientierten Programmiersprache zur Verfügung: Es können Variablen definiert, mathematische Operationen ausgeführt, Abfragen und Methoden erstellt oder ganze Klassen konstruiert werden. Eine Klasse ist so gesehen eine Bauanleitung oder eine Inventarliste für ein programmiertechnisches Objekt. In der Klasse wird festgelegt welche Eigenschaften und Funktionen ein Objekt, das dieser Klasse zugeordnet wird, aufweist. Ein sehr einfaches Beispiel wären die Klassen ***mensch*** und ***katze.*** Jedes Objekt der Klasse ***mensch*** verfügt über die Funkti-

124 S. C#- https://msdn.microsoft.com/de-de/library/kx37x362.aspx (letzter Aufruf: 14.07.2016)
Und C-Sharp- https://de.wikipedia.org/wiki/C-Sharp (letzter Aufruf: 14.07.2016)

125 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 35

onen ***sprechen***, während ein Objekt der Klasse ***katze*** die Funktion ***miauen*** besitzt. Dies ist wirklich ein sehr rudimentäres Beispiel und soll an dieser Stelle nur der Veranschaulichung des Prinzips einer Klasse in einer objektorientierten Programmiersprache dienen, für diejenigen, die damit noch nicht in Berührung kamen.[126]

Als in Kapitel 5.1.2 das Konzept der ***blueprints*** in der Unreal Engine erläutert wurde, wurde ebenfalls bereits einmal auf Klassen in einer objektorientierten Programmiersprache verwiesen. Ähnlich wie die ***blueprints*** stellen auch die Klassen hier den Rahmen für meist nur ein einziges Logikelement dar. Da jedes auszuführende Skript meist einer Klasse entspricht, werden diese Begriffe im Zusammenhang mit der Unity Engine auch oft als Synonym verwendet.[127] Eine visuelle Skriptsprache wie die Unreal Engine mit den ***blueprints*** weist die Unity Engine allerdings nicht auf. Alle erforderlichen Logikelemente, bis auf ein paar wenige Ausnahmen, müssen wie bereits oben erwähnt selbst per Hand verfasst werden.

Ein detaillierter Einblick in die Programmiersprache C# ist nicht der beabsichtigte Zweck dieser Arbeit und würde ebenfalls den Rahmen sprengen. Wichtig ist nur festzuhalten, dass wenn im Zusammenhang mit der Unity Engine von Skripten oder Logikelementen/-Bausteinen die Rede ist, das oben geschilderte Konzept einer Klasse gemeint ist. Sie führen ihre jeweiligen Aufgaben und Funktionen aus, um letztendlich den gewünschten Effekt im Spiel auszulösen.

Für das abschließende Verständnis muss allerdings noch auf die sogenannten ***game objects*** eingegangen werden. Jedes Objekt in einer Szene ist zunächst einmal solch ein ***game object***. Dabei handelt es sich in erster Linie um einen leeren Container, der alle einzelnen Komponenten eines Objektes zusammenfasst. Vereinfacht ausgedrückt: Wenn also ein simples Würfel-Objekt in der Szene platziert wird, dann handelt es sich dabei um ein ***game object*** mit der Komponente „geometrische Eigenschaften eines Würfels“. Möchte man den Würfel in der Szene bewegen muss dem ***game object*** zunächst die Komponente ***transform*** hinzugefügt werden. Diese ist standardmäßig bereits bei jedem Objekt vorhanden, aber das Prinzip wird verdeutlicht.[128] Die Komponenten eines Objektes definieren seine Eigenschaften und Funktionen. Verfasste Skripte können ebenso als Komponenten einem Objekt hinzugefügt werden. Das bedeutet also, um zum Beispiel jedem Spieler-Charakter und jedem Nicht-Spieler-Cha-

---

126 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 50-53

127 Vgl. Ebd. S. 71

128 Vgl. Ebd. S. 26-28

rakter die Eigenschaft zu verleihen, eine Lebensanzeige zu besitzen, wird dazu einmalig ein Skript verfasst mit dem dies ermöglicht wird. Dieses Skript, beispielsweise ***health bar*** genannt, wird dann jedem Objekt in der Szene zugewiesen, das eine Lebensanzeige bekommen soll. Ganz so einfach und global gültig ist diese Vorgehensweise in komplexeren Spielen natürlich nicht, da dort verschiedene Gegnertypen in der Regel auch verschieden viel Leben besitzen. Das grundlegende Prinzip wird so aber deutlich.[129]

Damit wird dieses Kapitel über die zugrundeliegende Programmiersprache der Unity Engine abgeschlossen. Dabei sollte erwähnt werden, dass der Ansatz der Unreal Engine, eine visuelle Skriptsprache verwenden zu können, weder besser noch schlechter ist als die manuelle Herangehensweise der Unity Engine. Letzten Endes geht es hier um Programmiersprachen, die prinzipiell fähig sind alles zu ermöglichen was gewünscht und benötigt wird, sofern der Entwickler das erforderliche Können im Umgang mit der Sprache aufweist.

5.2.3 Objekte in 3D und 2D

Um ein Objekt in der Spielwelt bzw. der Szene zu platzieren, muss es zunächst einmal einfach via Drag & Drop aus dem ***project browser*** in die ***scene view*** gezogen werden. Das Objekt wird automatisch als ***game object*** angelegt und so gehandhabt. Wie im vorherigen Kapitel bereits erwähnt, müssen dem Objekt nun noch die entsprechenden Komponenten zugewiesen werden. Um ein 3D-Modell im Spiel darstellen zu können, werden die Komponenten ***mesh filter*** und ***mesh renderer*** benötigt. Der ***mesh filter*** sorgt dafür, dass das ***game object*** die Geometrie, oder auch das ***mesh***, des Objektes überhaupt aufnehmen kann. Mit anderen Worten sorgt der Filter dafür, dass das ***game object*** „weiß", dass es ein 3D-Modell sein soll. Damit das Objekt dann auch wirklich dargestellt werden kann, wird der ***mesh renderer*** benötigt, der das Objekt mit einem Material versieht, das das Oberflächenaussehen mittels Texturen und Farben beschreibt.[130] Zusätzlich wird noch unterschieden zwischen dem ***mesh renderer*** für starre Objekte und dem ***skinned mesh renderer*** für animierte Objekte, bei denen sich die äußere Form durch die Animationen verändert.[131] Sollte das Objekt dann noch in der Welt frei platziert werden können, wird ebenfalls noch die ***transform*** Komponente benötigt.

---

129 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 81

130 Vgl. Ebd. S. 112-114

131 Vgl. Ebd. S. 112

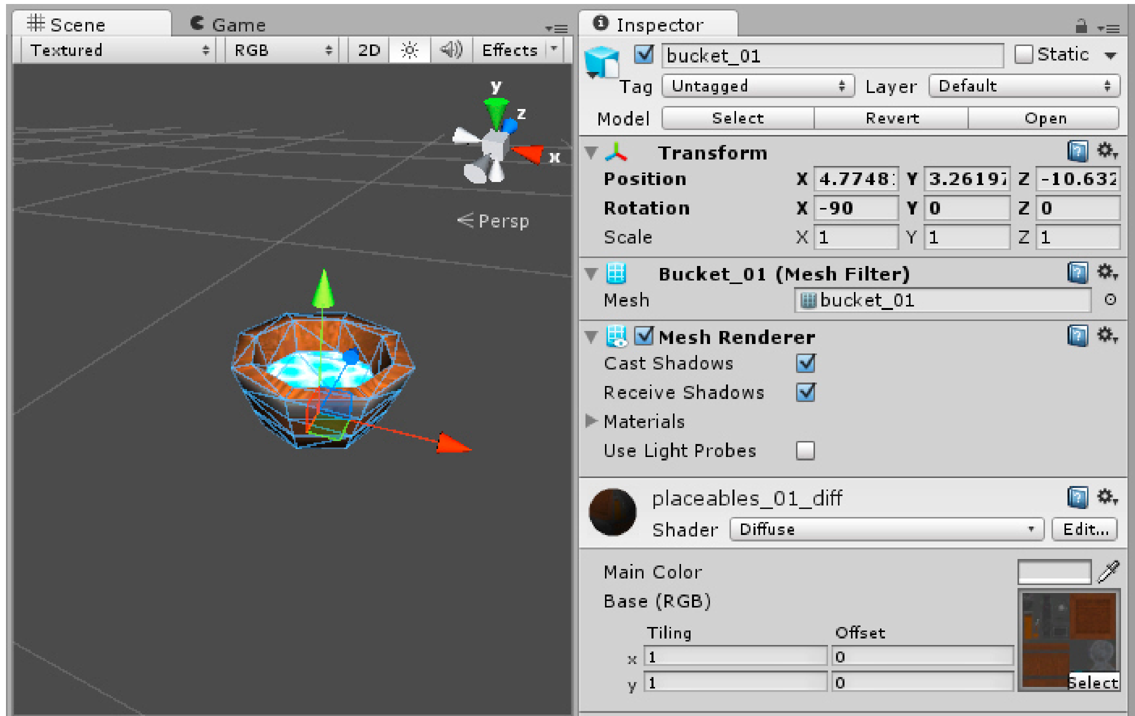

Abb. 42: Beispiel eines ***game objects*** dass über die Komponenten ***mesh filter*** und ***mesh renderer*** verfügt. Unter ***mesh*** findet sich das tatsächliche Modell, es besitzt einen einfachen diffusen ***shader*** und unten rechts ist die entsprechende Textur zu sehen die dargestellt werden soll.

Wird ein 3D-Modell via Drag & Drop aus dem ***Project Browser*** in die Szene gezogen werden die hier erwähnten drei Kernkomponenten automatisch dem ***game object*** hinzugefügt, um es dem Benutzer zu vereinfachen. Warum diese Komponenten gebraucht werden, sollte an dieser Stelle allerdings verdeutlich werden. Auf alle Komponenten eines Objektes kann durch ein Skript direkt zugegriffen werden. So kann beispielsweise das Aussehen eines Objektes oder seine Position während des laufenden Spiels verändert werden.

Die Materialien, die ein Objekt benötigt, können direkt in der Engine erzeugt werden. Das Material legt dabei nicht nur die Farbe des Objektes fest, sondern auch dessen optische Eigenschaften, z.B. ob es transparent ist. Zur Berechnung dieser Eigenschaften werden sogenannte ***shader*** eingesetzt. Sie berechnen in einer speziellen Programmiersprache wie die Materialien eines Objekts unter gegebenen Einflüssen dargestellt werden.[132]

Unity besitzt eine ganze Reihe an verschiedenen vorgefertigten ***shadern,*** die problemlos benutzt werden können. Das Herzstück ist allerdings der ***standard shader,*** der sich besonders gut dafür eignet realistische Oberflächen darzustellen und flexibel einsetzbar

---

132 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 117

ist. Er nutzt ein Verfahren namens ***physically based shading*** bei dem zur Berechnung der Oberfläche physikalische Gesetzmäßigkeiten berücksichtigt werden. wie zum Beispiel die Energieerhaltung, um ein konstantes Erscheinungsbild der Oberfläche zu erreichen. Außerdem ist das Verfahren unabhängig von der Umgebungsbeleuchtung. Kommt es bei anderen ***shadern*** durchaus vor, dass sich die Wirkung einer Oberfläche verändert wenn sich die Beleuchtung der Szene verändert, bleibt das Ergebnis beim ***physically based shading*** konstant.[133]

Der ***standard shader*** besitzt eine Vielzahl an Einstellungsmöglichkeiten und Eigenschaften, um jedes gewünschte Ergebnis erreichen zu können. Unter der Option ***rendering Mode*** kann beispielsweise die Haupteigenschaft des Materials eingestellt werden, sprich ob das Material deckend oder blickdicht ist (Holz, Metall, Kunststoff), ob es transparent ist, ob es später ein- und ausgeblendet werden können soll oder ob es nur stellenweise transparent ist. Im gewählten Modus können dann weitere Einstellungen, zum Beispiel zur Farbe oder Leuchtkraft des Objektes, getätigt werden. Jede Option hier im Detail zu erwähnen würde zu weit führen. Darüber hinaus gibt es auch hier wieder eine große Überschneidung zu den üblichen Verfahren und Möglichkeiten einer 3D-Modellierungssoftware zur äußeren Gestaltung eines Objektes. Abschließend sei nur noch erwähnt, dass es unter Umständen von Vorteil sein kann, eben nicht jede Eigenschaft und jede Option des ***shaders*** zu verwenden. Denn jede nicht genutzte Eigenschaft stellt für das fertige Spiel weniger Rechenaufwand und damit einen Leistungsgewinn dar.[134]

Abb. 43: Beispielszene aus dem Demo-Projekt Viking Village von Unity Technologies, bei der hauptsächlich der ***standard shader*** verwendet wird.

133 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 119

134 Vgl. Ebd. S. 119

Wie eingangs erwähnt ist die Unity Engine ebenfalls sehr gut geeignet um zweidimensionale Spiele zu entwickeln. Diese unterscheiden sich natürlich von ihren 3D Kollegen und weisen entsprechende Merkmale und Besonderheiten in der Entwicklung auf.

Zuallererst gibt es in einem 2D Spiel keine 3D-Modelle sondern lediglich Grafikobjekte, sogenannte ***sprites***. In Unity werden alle Texturen als ***sprite*** bezeichnet, die dem gleichnamigen ***texture type*** zugeordnet sind. Wird das Projekt direkt zu Beginn als 2D Spiel angelegt, wird der Typ automatisch bei allen Grafiken auf ***sprite*** gesetzt. Neben dem Typ existieren allerdings noch weitere Eigenschaften. So kann unter ***sprite mode*** festgelegt werden, ob es sich um eine einfache (***single***) oder eine mehrfache (***multiple***) Textur handelt. Eine einfache Textur besteht aus einem Element und einer Datei. Bei mehrfachen Texturen werden mehrere Elemente zwar in einer Datei angelegt und gespeichert, bestehen aber aus verschiedenen Teilen, zum Beispiel wenn bei der Grafik eines Autos die Reifen und die Karosserie voneinander getrennt behandelt werden sollen. Die einzelnen Teile können dann im ***sprite editor*** festgelegt und weiter benannt werden. Dafür gibt es auch eine automatische Funktion, bei der Unity selbst versucht die verschiedenen Teile zu bestimmen. Des Weiteren kann sowohl bei einfachen als auch bei multiplen ***sprites*** der ***pivot***-Punkt festgelegt werden. Dieser Punkt definiert die Rotations- und Bewegungsachse der Textur. Bei einem Autoreifen läge er also genau zentral, bei der Heckklappe des Wagens eher am oberen Ende.[135]

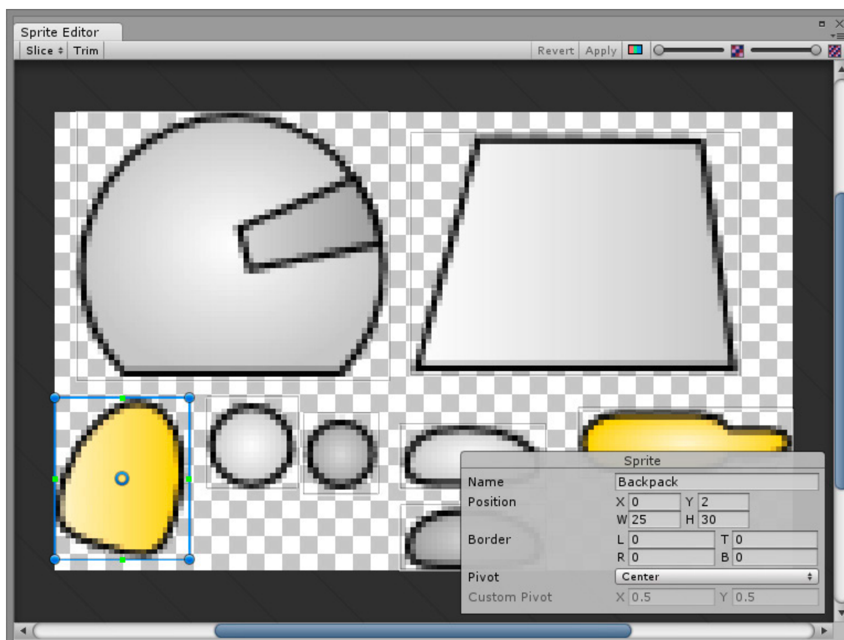

Abb. 44: Multiple ***sprite*** Textur im ***sprite editor***. Die einzelnen Elemente die sich innerhalb einer Datei befinden können hier definiert werden, sowie der jeweilige ***pivot***-Punkt (blauer Kreis unten links) verändert werden kann.

135 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 143-148

Ähnlich dem ***mesh renderer*** bei einem 3D-Modell benötigt auch ein ***game object*** das ein Grafikobjekt enthält eine spezielle Komponente um das Objekt darzustellen, den ***sprite renderer***. In dieser Komponente wird das Grafikobjekt an sich festgelegt, ob es mit einer zusätzlichen Farbe eingefärbt werden soll oder wie die Reihenfolge aussieht, sprich in welcher Tiefe oder auf welcher Ebene (Vorne, Mitte, Hinten, usw.) das ***sprite*** schlussendlich dargestellt werden soll. Dies entscheidet darüber, wie es sich mit anderen Objekten überschneidet. Bei vielen Objekten in einer Szene ist das eine nicht zu unterschätzende Aufgabe. Denn wenn der Spieler-Charakter plötzlich hinter einem Objekt das im Hintergrund liegt verschwindet, stellt dies in der Regel wohl einen Fehler dar.[136]

Da die Grafikobjekte üblicherweise auch aus einem externen Programm importiert werden, besitzen sie bereits eine äußere Erscheinung. Dennoch kann dem Objekt ein Material zugewiesen werden. Dabei stehen zwei verschiedene ***sprite shader*** zur Auswahl. Der ***standard shader*** sorgt dafür, dass das Objekt nicht mit Lichtquellen interagiert und immer gleich dargestellt wird, wohin gegen der ***diffuse shader*** die Interaktion mit und die Beeinflussung durch Lichtquellen ermöglicht.[137]

5.2.4 Physik in Unity

Der teilweise gewünschte Grad an Realismus in einem Spiel ist noch aus Kapitel 5.1.4 Physik in der Unreal Engine bekannt. Spiele, die mit der Unity Engine entwickelt werden, haben eventuell einen ebenso hohen Anspruch an den Realismus. Um diesem nachzukommen, simuliert Unity physikalische Gesetzmäßigkeiten mit Hilfe von Nvidias Physik-Engine PhysX. Es stehen erneut verschiedene Komponenten zur Verfügung, die den ***game objects*** hinzugefügt werden können, damit diese den physikalischen Gesetzen folgen und miteinander interagieren können. Dabei unterscheidet Unity zwischen Komponenten für 3D- und für 2D-Modelle. Da bei letzteren aber oft nur eine Bewegungsachse weniger beachtet wird und die allgemeinen Parameter sich nahezu gleichen, soll an dieser Stelle der Fokus lediglich auf die 3D-Modelle gelegt werden, um einen generellen Überblick über das Thema zu erhalten.

Ganz allgemein finden die Berechnungen der physikalischen Eigenschaften in regelmäßigen Zeitabständen statt. Das Intervall wird mit dem ***fixed timestep*** Parameter festgelegt und gilt projektweit. Je kleiner der dort eingetragene Wert, desto öfter werden die Berechnun-

---

136 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 149-151

137 Vgl. Ebd. S. 151

gen durchgeführt. Befinden sich Objekte in der Szene, die sich schnell bewegen, kann es förderlich oder sogar notwendig sein, diesen Wert zu verringern. Andernfalls kann es passieren, dass es zwischen zwei sich schnell bewegenden Obejkten zwar eine Kollision gibt, diese allerdings ohne Auswirkung bleibt weil zum Zeitpunkt der Kollision keine Berechnung der physikalischen Eigenschaften stattgefunden hat. Andersherum gilt folglich, dass bei langsamen Objekten der Wert erhöht werden kann. Denn jede Berechnung, besonders die der Physik-Engine, stellt einen Rechenaufwand dar, den der Prozessor bewältigen muss.[138] Wurden die Berechnungen abgeschlossen, wird in allen Skripten die ***fixed update***-Methode, sofern vorhanden, aufgerufen und die entsprechenden Werte aktualisiert damit die Objekte sich korrekt verhalten.

Das Kernelement bzw. die Kernkomponente der Physik-Engine in Unity ist die ***rigidbody***-Komponente.[139] Jedes Objekt, das sich in der Szene bewegt oder zu irgendeinem Zeitpunkt einmal bewegt werden soll, benötigt diese Komponente. Sie definiert die für die physikalischen Berechnungen erforderlichen Werte wie Masse und Schwerpunkt des Objektes, den Luftwiderstand, den Drehmoment oder den Einfluss der Gravitation auf das Objekt. Ebenfalls ist es hier auch wieder möglich, bestimmte Bewegungsachsen zu deaktivieren, wenn diese nicht berücksichtigt werden sollen. Durch die ***interpolate*** Eigenschaft kann ein weicherer Verlauf der physikalisch gestützten Bewegungen erreicht werden. Dies ist mitunter nötig da die Berechnungen der Physik-Engine wie oben erwähnt in festen Zeitabständen stattfinden. Der Render-Prozess arbeitet allerdings so schnell wie es ihm leistungstechnisch möglich ist. Die Rate mit der das Bild erneuert wird variiert also. Dadurch könnte es vorkommen, dass manche Bewegungen ruckeln. Durch die Interpolation wird dieser Umstand vermieden oder zumindest abgemildert.[140]

Die Kollisionserkennung fällt ebenfalls in den Aufgabenbereich der Physik-Engine. Dazu gibt es die ***collider*** Komponente. Wird diese einem Objekt hinzugefügt, wird in der Szenenansicht die jeweils ausgewählte Form des ***colliders*** angezeigt. Es gibt simple Formen wie Würfel, Kugel und Kapsel oder die Geometrie des Objektes selbst kann als Form ausgewählt werden. Dabei ist zu bedenken, dass die einfacheren Formen leistungsfreundlicher sind als eine genaue Abtastung des Objektes. Aus diesem Grund ist es nicht unüblich,

---

138 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 209-210

139 Vgl. Ebd. S. 210

140 Vgl. Ebd. S. 211

selbst komplexere Modelle mit verhältnismäßig einfachen ***collidern*** auszustatten. Eine Kombination aus verschiedene Formen ist ebenfalls möglich.[141]

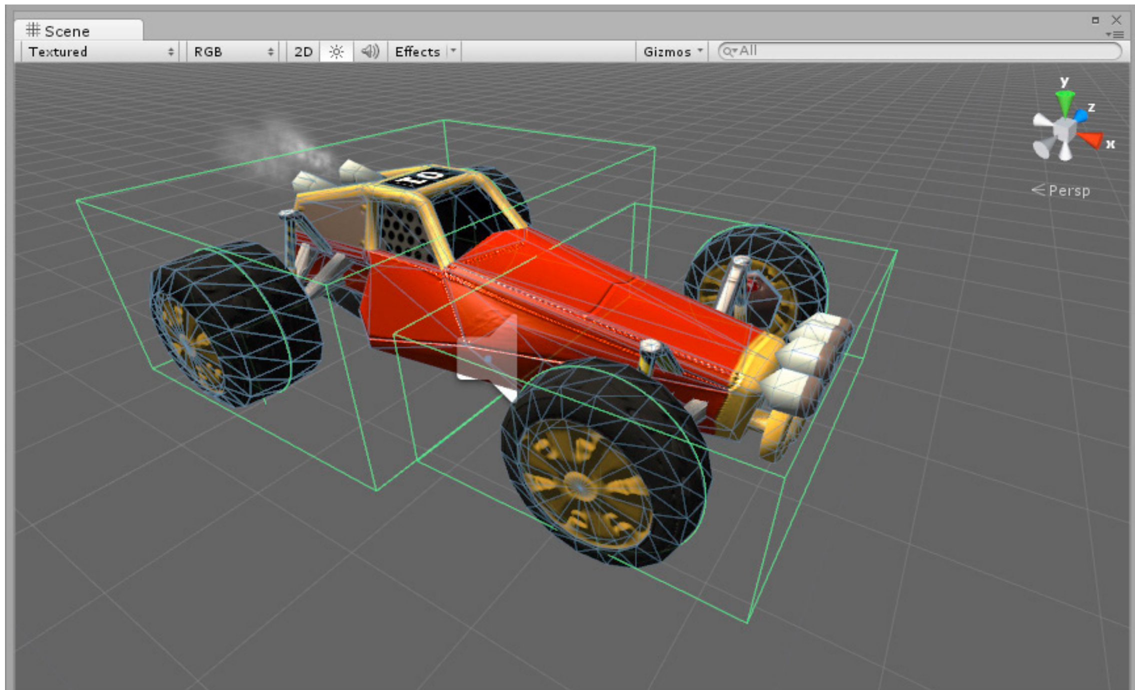

Abb. 45: Leistungsfreundliche ***collider***-Anordnung bei einem komplexeren Modell. Diese Methode liefert nur ein sehr grobes Ergebnis, verringert dafür allerdings den Aufwand bei den Berechnungen von Kollisionen erheblich.

Um eine Kollision auszuwerten stehen in Unity standardmäßig drei Methoden bereit:

- ***on collision enter*** wird ausgelöst in dem Moment, in dem zwei Objekte miteinander kollidieren
- ***on collision stay*** wird regelmäßig ausgeführt solange die Kollision zweier Objekte anhält
- ***on collision exit*** wird folglich beim Beenden einer Kollision ausgelöst

Mittels dieser Methoden ist es möglich Informationen über die Kollision zweier Objekte auszulesen, wie zum Beispiel der Aufprallpunkt oder die –Geschwindigkeit. Darüber hinaus ermöglichen sie ebenfalls die Kommunikation der Objekte untereinander. Folgendes Beispiel: Objekt A ist eine gerade Fläche, Objekt B ein darüber schwebender Würfel. Wird die Simulation gestartet, fällt Objekt B aufgrund der Gravitation und seiner physikalischen Eigenschaften hinab auf Objekt A. Für den Moment der Kollision (***on collision enter***) ist definiert, dass Objekt A dem Objekt B mitteilen soll, sich wieder um den Wert X nach oben

141 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 216-217

zu bewegen. Verbildlicht ausgedrückt ist Objekt A ein Trampolin und Objekt B der darauf hüpfende Mensch.[142]

collider können durch das Aktivieren der Eigenschaft ***is trigger*** in einen ***trigger*** umgewandelt werden, das bedeutet sie werden zu Signalgebern umfunktioniert. Dabei werden sie von der Physik-Engine insofern ignoriert, dass sie anderen ***collidern*** das Passieren ihres Bereiches erlauben. Daraufhin wird dann eine von drei ***trigger***-Methoden ausgeführt, die identisch sind zu den weiter oben erläuterten ***collider***-Methoden. Es gibt also eine Methode für das Eindringen, das Aufhalten in und das Verlassen des ***trigger***-Bereiches. Die Möglichkeit zur Informationsgewinnung und Kommunikation mit dem kollidierenden ***game objects*** ist bei ***triggern*** ebenfalls vorhanden.[143]

Mit den Materialeigenschaften eines ***colliders*** ist es ferner möglich, einem Objekt gewisse Oberflächeneigenschaften hinzuzufügen, die bei den Berechnungen der Physik-Engine berücksichtigt werden. Beispiele für solche Eigenschaften sind die Reibung und das Federverhalten. Die Reibung wird hierbei noch einmal in Gleitreibung und Haftreibung unterteilt. Die Gleitreibung beschreibt wie sich das Objekt verhält, wenn es über eine andere Oberfläche gleitet. Die Haftreibung meint den Anfangswiderstand eines Objektes, der überwunden werden muss, ehe es beginnt über die unterliegende Oberfläche zu gleiten.[144]

Damit soll dieser kurze und allgemeine Einblick in die Physik-Engine von Unity abgeschlossen sein. Selbstverständlich gibt es auch in diesem Bereich noch viele weitere Optionen und Möglichkeiten für komplexe und aufwendige Simulationen. Beispielsweise gibt es eine spezielle Form eines ***colliders,*** um das Rad eines Wagens realitätsnah zu erfassen oder die Möglichkeit Objekte nur mit simulierten Kräften, wie Wind- oder Strömungseffekte, zu bewegen.

---

142 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 219-220

143 Vgl. Ebd. S. 220-221

144 Vgl. Ebd. S. 238

5.2.5 Landschaften gestalten

Um Landschaften in Unity zu gestalten, bringt die Engine einen ganz eigenen Objekttyp mit, die ***terrains***. Mit diesem Objekt können komplexe Landschaften erschaffen, mit detaillierten Texturen versehen und mit Vegetation perfektioniert werden. Dafür steht eine Reihe an Werkzeugen bereit.

Wie aus anderen Programmen oder Engines bekannt, arbeitet auch Unity dazu hauptsächlich mit einem Pinsel-Werkzeug. Damit lässt sich die Höhe der Landschaft bearbeiten und es können Hügel und Berge geformt werden. Durch das Abrunden der Kanten und/oder dem Abflachen der Spitzen, kann eine natürlichere Optik erzielt werden. Um eine Vertiefung in der Landschaft zu erzeugen, also z.B. eine Schlucht oder ein Flussbett, ist allerdings ein kleiner Trick notwendig. Denn eine Landschaft in Unity kann nicht unter ihren absoluten Nullpunkt herabgesenkt werden. Soll das Terrain eine Vertiefung enthalten muss zunächst einmal die gesamte Grundfläche angehoben werden. Anschließend kann die Oberfläche mit dem Pinsel-Werkzeug auch nach unten verschoben werden. Eine Verschiebung nach links oder rechts ist ebenfalls nicht möglich. Das bedeutet um zum Beispiel eine Höhle in die Landschaft zu integrieren, müsste diese in einem externen Programm zuerst modelliert und dann in die Engine importiert werden.[145]

Unity bietet darüber hinaus ebenfalls die Möglichkeit ***heightmaps*** als Basis für Landschaften zu verwenden. So können Karten realer Landschaften als Schablone für ein digitales Terrain verwendet werden. Dies ist eine deutlich schnellere Methode als jede Erhebung manuell durchzuführen. Das Ergebnis der ***heightmap*** kann hinterher natürlich noch mit dem Pinsel-Werkzeug verfeinert werden, wenn dies gewünscht oder notwendig ist.

Das Pinsel-Werkzeug kommt ebenfalls beim texturieren der erzeugten Landschaften zum Einsatz. Wie auch die Unreal Engine besitzt Unity dazu ein Ebenen-System um komplexe Texturen mit verschiedenen sich überlagernden Elementen zu ermöglichen. Dabei sollte darauf geachtet werden, dass die einzelnen Texturen so angelegt sind, dass sie nahtlos aneinander gefügt werden können. Diese Art von Textur wird auch ***tiling texture*** genannt, vom englischen ***tile*** für Kachel. Andernfalls wäre ein unschöner Übergang zwischen, aber auch innerhalb der einzelnen Texturen zu erkennen. Ferner empfiehlt es sich mit einer Textur zu beginnen, die als Grundierung dient. Darauf können dann weitere Texturen aufgetragen werden, die Akzente und Abwechslung in die Landschaft bringen.[146]

---

145 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 319-323

146 Vgl. Ebd. S. 325-328

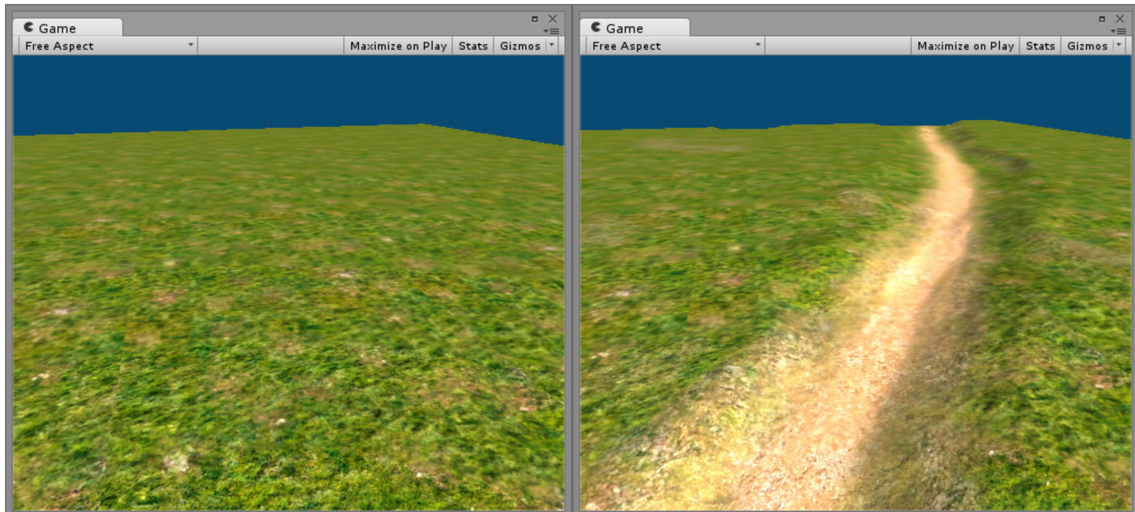

Abb. 46: Gegenüberstellung einer einfach texturieren Landschaft (links) und einer Landschaft der durch verschiedene Texturen Akzente verliehen wurden, um den Realitätsgrad zu erhöhen (rechts).

Ist die Gestaltung des Untergrunds abgeschlossen, können mit dem ***place tree***-Werkzeug Bäume und Sträucher hinzugefügt werden. Eine Auswahl an verschiedenen Modellen kann dazu angelegt und dann erneut mit einem Pinsel-Werkzeug aufgetragen werden. Dafür gibt es verschiedene Optionen wie die Größe des Pinsels, die Dichte in der Bäume erschaffen werden, eine Farbvariation oder die zufällig generierte Höhe der Bäume. Mit der ***mass place trees*** Funktion kann eine vorher angegebene Anzahl an Bäumen zufällig auf der ganzen Landschaft verteilt werden. Die ist nützlich um ganze Waldareale zu erzeugen oder ein gewisses Fundament an Vegetation zu legen, das dann in einem zweiten Schritt manuell noch angepasst werden kann. Wenn gewünscht oder notwendig können Bäume auch mit ***collidern*** ausgestattet werden, die dann von der Physik-Engine berücksichtigt werden. Dabei ist zu beachten, dass zum gegenwärtigen Zeitpunkt eine Obergrenze von 65.536 für mit ***collidern*** versehenen Bäumen existiert. Mehr ***collider*** für Bäume werden aus Leistungstechnischen Gründen nicht unterstützt.[147]

147 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 328-330

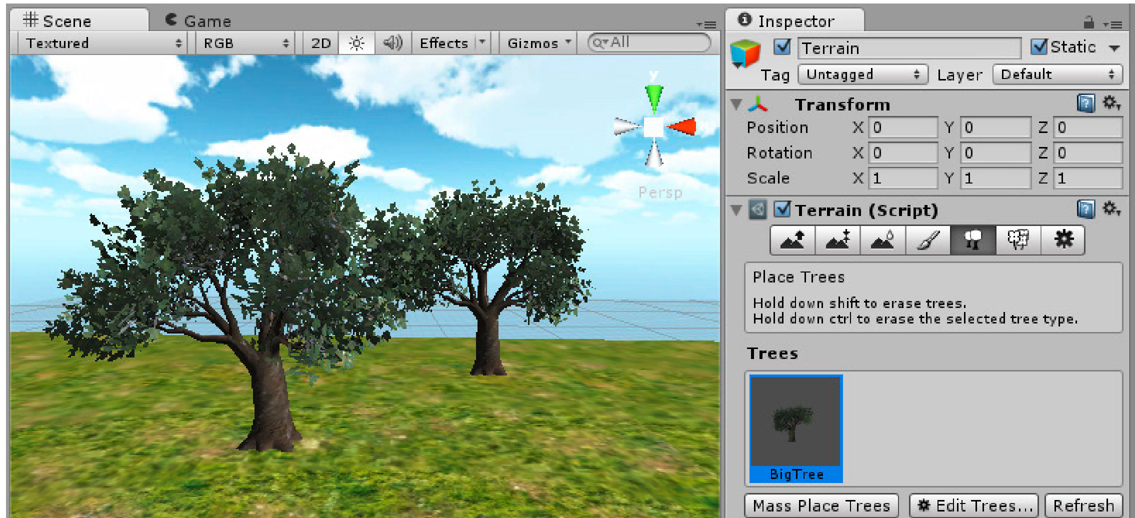

Abb. 47: Platzierung von Bäumen in der Landschaft. Unten rechts können verschiedene Modelle ausgewählt werden die dann entweder einzeln, mit einem Pinsel-Werkzeug oder der Funktion ***mass place trees*** der Landschaft hinzugefügt werden.

Neben Bäumen und größeren Sträuchern können allerdings auch kleinere Details der Landschaft hinzugefügt werden. Dazu steht dem Entwickler das ***paint details***-Werkzeug zur Verfügung. Dieses ist entweder für sehr einfache Modelle, z.B. kleine Steine, oder auch nur für simple Texturen, z.B. Gras, gedacht. Die einfachen Modelle werden dabei in der Szene verteilt, während die Texturen auf kleinen, unsichtbaren zweidimensionalen Flächen dargestellt werden. In beiden Fällen erfolgt die Verteilung in der Szene in sogenannten ***clustern***, also Anhäufungen. Die optische Darstellung dieser ***cluster*** kann dabei erneut durch verschiedene Einstellungsmöglichkeiten verfeinert werden. Neben der Anzahl an Einzelelementen kann die Höhe, Breite und Farbe definiert werden. In der Regel ist es so dass die Elemente in der Mitte des ***clusters*** am größten sind und eine gesunde Farbe aufweisen (im Falle von Rasen also grün), sie zum Rand hin aber kleiner und kränklicher (farbtechnisch gesehen) dargestellt werden. Im Falle der simplen zweidimensionalen Texturen kann darüber hinaus noch mit der Option ***billboard*** (engl. für Plakatwand, Anschlagtafel) festgelegt werden, ob die Frontseite der Textur immer zur Kamera ausgerichtet sein soll. Dadurch fällt der Fakt, dass es sich nur um zweidimensionale Objekte handelt, im laufenden Spiel nicht so sehr auf und die für den Spieler erzeugte Immersion wirkt stärker.[148]

148 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 330-332

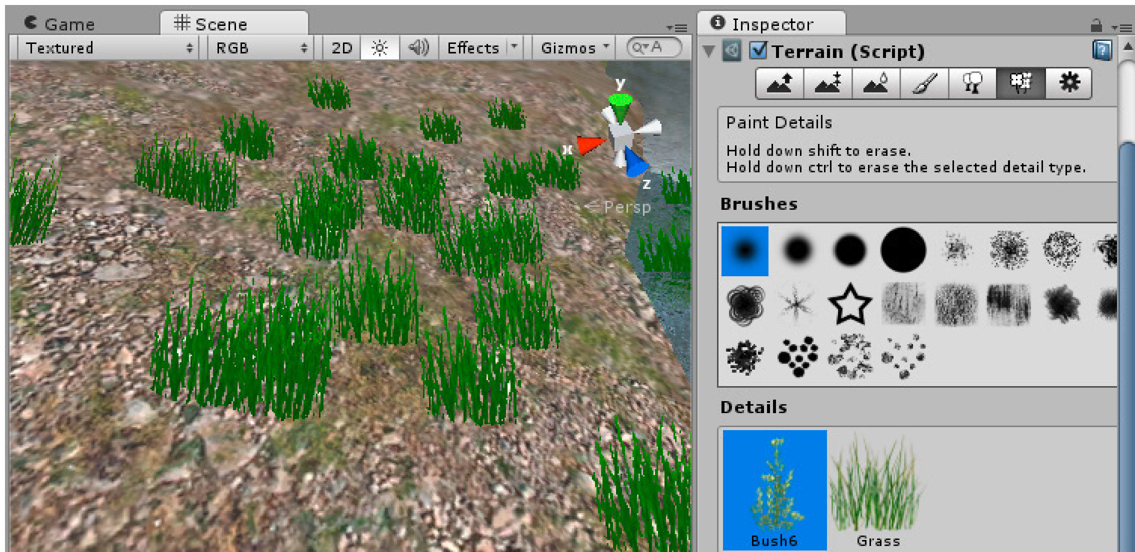

Abb. 48: Platzierung von kleineren Details in der Landschaft. Diese könne ebenfalls mit zum Beispiel dem Pinsel-Werkzeug der Landschaft hinzugefügt werden.

Um den Grad an Realismus weiter positiv zu beeinflussen, können der Landschaft noch Windzonen hinzugefügt werden. Diese wirken sich zwar nicht auf alle ***game objects*** aus, allerdings werden sensible Objekte wie Bäume und Sträucher davon beeinflusst. Eine Windzone kann einen gerichteten Wind, wie er draußen in der Natur vorkommt oder einen sphärischen Wind, der einer industriellen Turbine gleicht, erzeugen. Selbstverständlich können auch hier die Stärke, der Radius, Turbulenzen und weitere Optionen angepasst werden, um das gewünschte Ergebnis zu erreichen.[149]

5.2.6 Animation

Die benötigten Animationen für Objekte und Charaktere können direkt in Unity angefertigt werden. Die Engine stellt dazu dem Entwickler die entsprechenden Mittel zur Verfügung. Für 2D-Animationen ist dies auch durchaus praktikabel, da es hierbei oft nur um das verhältnismäßig simple Verschieben und Rotieren von Grafiken geht. Die Animationen komplexer 3D-Modelle werden allerdings üblicherweise ebenfalls in externen Programmen erstellt und dann nach Unity exportiert.

Beim Importieren von Animationen müssen gewisse Einstellungen vorgenommen werden. Diese befinden sich in den ***import settings*** unter den Reitern ***model, rig*** und ***animations***. Die interessantesten Optionen hält dabei der Reiter ***rig*** bereit. Anders als in üblichen 3D-Programmen arbeitet Unity nicht zwangsläufig mit der Skelett-

149 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 339-342

struktur des importierten Modells. Die Engine nutzt dafür ein System, das ***avatar*** genannt wird. ***avatar*** bezeichnet ein internes Format zur Speicherung der Skelettinformationen und zur Abbildung dieser auf dem Modell. Dabei kann zwischen allgemeinen und humanoiden Charakteren unterschieden werden. Bei allgemeinen Charakteren muss der Entwickler keine tiefgehenden Einstellungen tätigen, da der ***avatar*** automatisch erstellt wird. Für dieses spezifische Modell vorhandene Animationen können einfach importiert und abgespielt werden. Bei humanoiden Charakteren erstellt Unity ebenfalls automatisch den passenden ***avatar***. Dieser kann bzw. muss im Falle eines Fehlers allerdings auch manuell erstellt werden. Dabei werden die tatsächlichen Knochen des Modells den jeweiligen Entsprechungen des ***avatars*** zugeordnet, damit die Engine die richtigen Transformationen durchführen kann. Der große Vorteil dieses Systems ist, dass der Entwickler bei der Verwendung von mehreren humanoiden Charakteren lediglich einen einzelnen ***avatar*** anlegen muss. Beim Import eines weiteren Charakters steht ab diesem Zeitpunkt fortan die Option ***copy from other avatar*** zu Verfügung. Damit können alle Einstellungen zur Skelettstruktur, Gewichtung und den Muskeln einfach übernommen werden.[150]

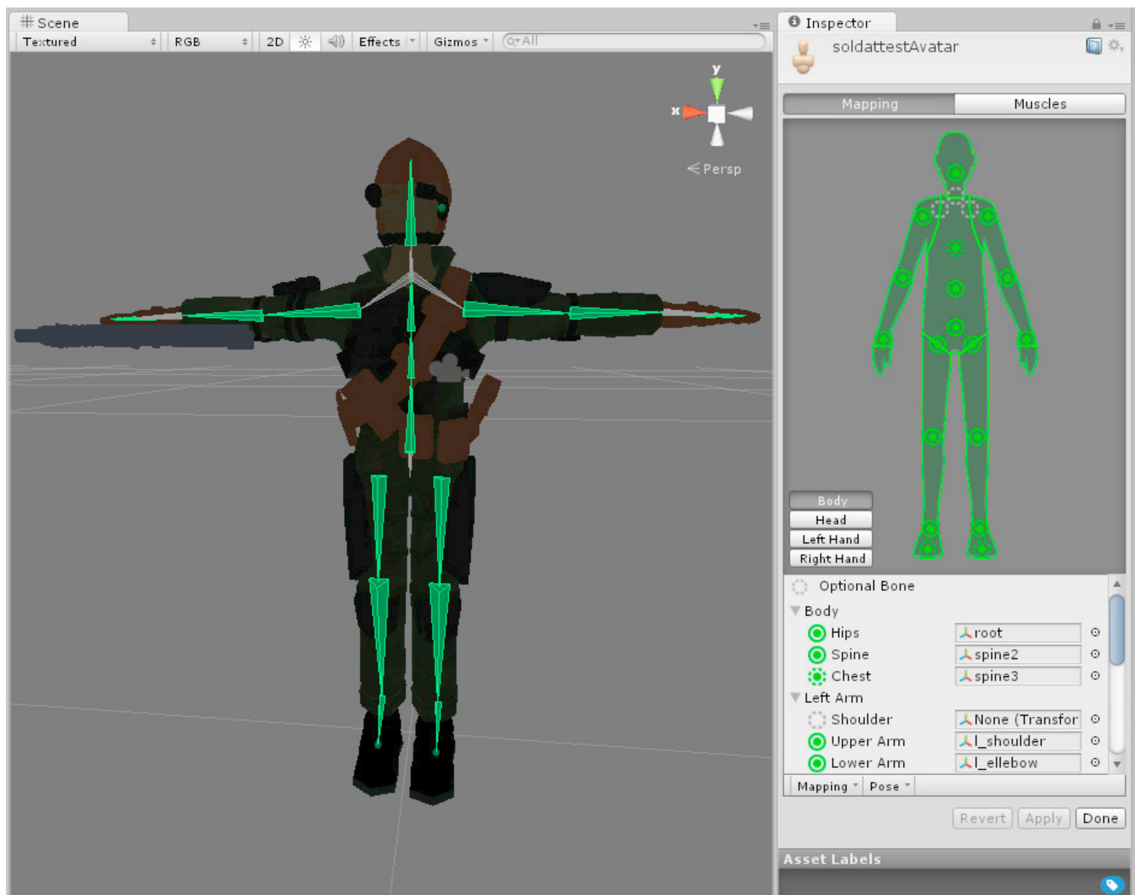

Abb. 49: Skelettdarstellung (links) und ***avatar mapping*** (rechts). Unten rechts können ggf. nicht korrekt erkannte Verbindungen korrigiert oder angepasst werden. Ist eine korrekte Zuordnung erfolgt, arbeitet Unity intern ausschließlich mit dem System der hier rechten Abbildung.

150 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 398-401

In einem externen Programm erstellte Animationen können in einzelnen Clips oder einem langem Clip, sowie mit und ohne zugehörige Modelle in die Engine importiert werden. Dies muss lediglich beim Importvorgang jeweils angegeben werden. Befinden sich die Animationen alle in einem einzelnen langen Clip, bietet Unity die Möglichkeit diesen Clip zu editieren und mit simplen Start- und Endpunkt Angaben daraus mehrere Clips zu gewinnen, ähnlich den Techniken üblicher Programme für den Videoschnitt.[151]

Um ein Objekt im Spiel nun tatsächlich zu animieren reicht es nicht aus lediglich eine entsprechende Animation zu hinterlegen. Jedes ***game object,*** das animiert werden soll, benötigt dazu eine ***animator***-Komponente und einen ***animator controller***. Der c***ontroller*** definiert und steuert bei mehreren hinterlegten Animationen zu welchem Zeitpunkt welche Animation abgespielt werden soll und wie zwischen den verschiedenen Animationen gewechselt wird. Die ***animator***-Komponente stellt dabei die Verbindung zwischen dem Objekt und dem ***controller*** her.

Der ***animation controller*** fungiert im Grunde wie die ***state machines*** der Unreal Engine. Er definiert, kontrolliert und steuert die Zustände eines Objektes und die zugehörigen Animationen. Jeder ***controller*** besitzt dabei standardmäßig zunächst drei Zustände: ***entry, exit*** und ***any state***. Die ersten beiden Zustände sind selbsterklärend, sie symbolisieren den Ein- und Ausstieg in den ***controller*** und sind beispielsweise bei der Verwendung von ***sub-state machines*** erforderlich. Das bedeutet, wenn gewisse Animationen sozusagen dezentral gespeichert werden, damit auch andere Objekte damit verknüpft werden können. Denkbar wären hier Laufanimationen eines humanoiden Charakters. Der ***any state*** repräsentiert jeden beliebigen Zustand des ***controllers*** und dient dem Wechseln in Zustände, die unabhängig davon sind, in welchem Zustand sich das Objekt gerade befindet. Ein gutes Beispiel für einen Zustand, der mit dem ***any state*** verbunden ist, ist die Sterbeanimation eines Charakters. Egal ob dieser gerade tatenlos herumsteht, sich im Lauf befindet oder sich gerade im Sprung befindet, sterben kann er in jeder dieser Situationen. Entsprechend muss die Sterbeanimation auch jederzeit und sofort abgespielt werden können. Um einen neuen Zustand anzulegen, muss einfach ein Animations-Clip via Drag & Drop in den ***controller*** gezogen werden. Der erste angelegte Zustand wird automatisch als ***default state*** definiert, welcher immer zu Beginn des Spiels ausgeführt wird.[152]

---

151 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 401-402

152 Vgl. Ebd. S. 404-406

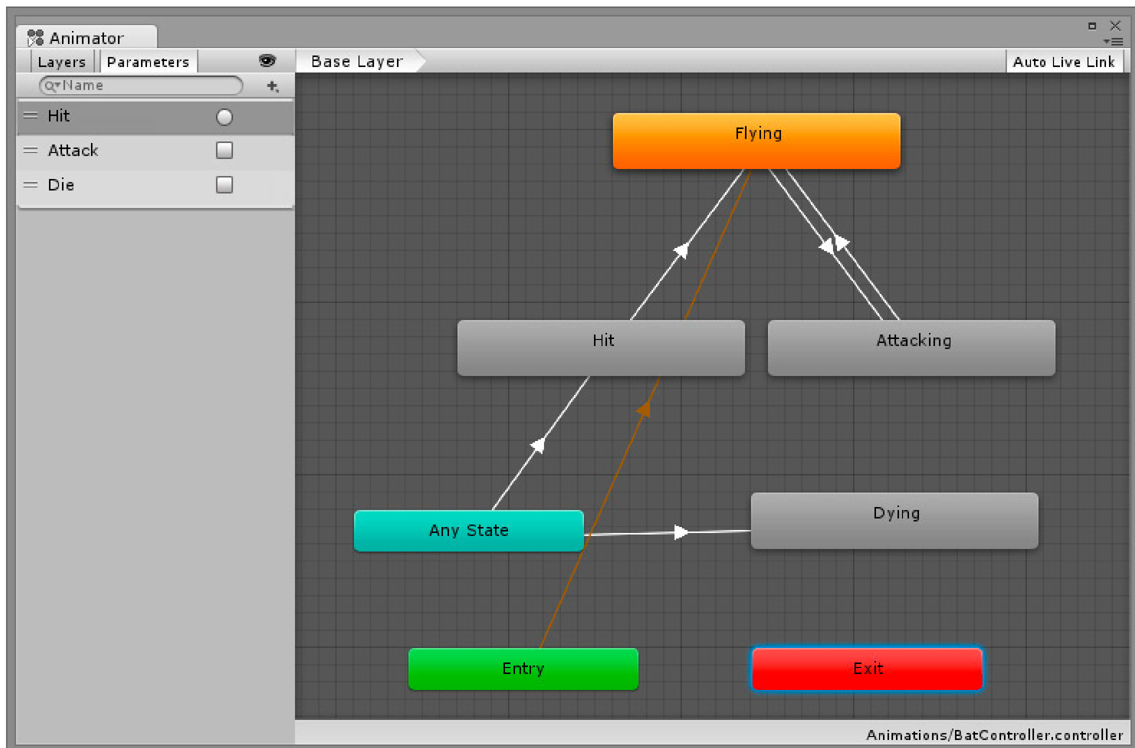

Abb. 50: Beispiel eines ***animation controllers*** in der Unity Engine. Der orangefarbene Zustand kennzeichnet den ***default state***.

Der Wechsel zwischen den Zuständen ist erneut an Bedingungen gebunden, die der Entwickler frei definieren kann. Dazu können beliebige Variablen angelegt und abgefragt werden. Des Weiteren können Übergänge zwischen zwei Animationen durch die sogenannte ***exit time*** verfeinert werden. Wird dieser Parameter aktiviert, besteht die Möglichkeit einen festen Austrittszeitpunkt zu bestimmen. Das bedeutet es kann festgelegt werden, dass Animation A noch vollständig zu Ende ausgeführt wird, obwohl alle Bedingungen für einen Wechsel in den nächsten Zustand und damit in Animation B, bereits erfüllt sind.[153] Eine Besonderheit bei den Zuständen stellen die ***blend trees*** dar. Dabei handelt es sich um Zustände, in denen mehr als eine Animation hinterlegt ist. Mittels eines ***float***-Parameters (***float*** steht in der Informatik in der Regel für eine Gleitkommazahl[154]) wird innerhalb des Zustandes festgestellt, welche Animation abgespielt werden soll. Der Parameter fungiert sozusagen als eine Weiche oder ein Pendel. Schlägt er mehr in die eine Richtung aus, wird Animation A abgespielt, bei der anderen Richtung die Animation B. Befindet sich der Wert des Parameters in der Mitte, wird eine Art Mischform beider Animationen abgespielt, eine Überblendung die nach dem Parametergewichtet ist. Dies ermöglicht einen flüssigen Wechsel zwischen zwei Animationen und wird oft bei ähnlich gelagerten Clips verwendet,

153 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 406-408

154 S. Gleitkommazahl- https://de.wikipedia.org/wiki/Gleitkommazahl (letzter Aufruf: 16.07.2016)

zum Beispiel bei Animationen für das Gehen und das Laufen. Der entscheidende Parameter wäre in diesem Fall die Geschwindigkeit des Charakters.[155]

Neben der gerade erwähnten weichen Überblendung von zwei ähnlichen Animationen können allerdings auch zwei verschiedene Animationen gleichzeitig abgespielt werden. Dazu können im ***animator controller*** Animationsschichten, sprich ***layer*** definiert werden. Jeder Schicht bzw. Ebene wird dann eine Animation zugewiesen und die gewählte Mischform legt fest, wie sie kombiniert werden sollen. Zur Auswahl stehen hier die Optionen dass eine Ebene die andere komplett überschreibt, beide gemischt werden oder eine Gewichtung nach einer der beiden Ebenen erfolgen soll. Zusätzlich kann jedem Modell eine ***avatar mask*** hinterlegt werden. Mit dieser Maske ist es möglich gewisse Körperteile des Charakters auszuschließen. Beispielsweise soll ein Charakter in der Lage sein sowohl aus dem Stand als auch aus der Bewegung heraus etwas zu werfen. Das heißt es würde in diesem Fall Sinn ergeben, für die Wurfanimation eine Maske zu hinterlegen, die nur den oberen Teil des Körpers beinhaltet. Befindet sich der Charakter dann im Zustand „Laufen“ und der Spieler löst die Werfen-Funktion aus, wird die entsprechende Animation nur für den oberen Teil des Körpers ausgeführt während der untere Teil sich weiter im „Laufen“ Zustand befindet.[156]

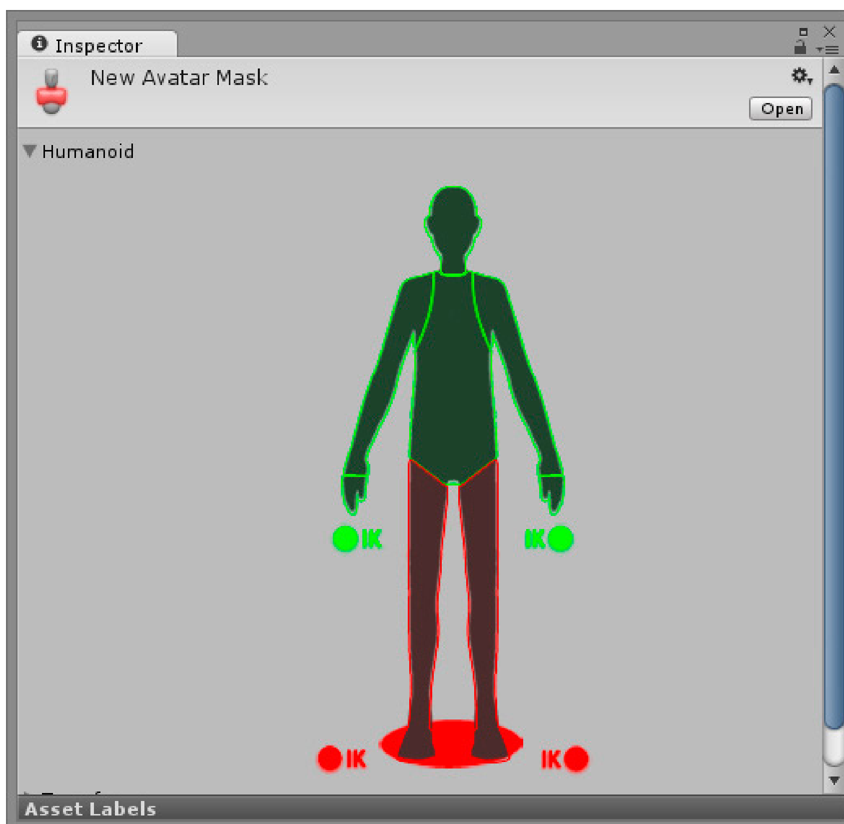

Abb. 51: Darstellung einer ***avatar mask,*** bei der die untere Körperhälfte des ***avatars*** nicht berücksichtig werden soll.

155 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 408-409

156 Vgl. Ebd. S. 409-411

In Unity ist es ebenfalls möglich zeitlich exakt bestimmte Events zu definieren. Jeder Animation kann zu jedem Zeitpunkt ein ***animation event*** hinzugefügt werden, welches dann eine frei wählbare Funktion oder Methode eines Skriptes aufruft und einen Parameter übergibt. Dies wird in der Zeitleiste des jeweiligen Animations-Clips getan und ist relativ einfach zu handhaben. Zu beachten ist nur, dass das aufzurufende Skript auch dem jeweiligen ***game object*** hinzugefügt ist. Sonst kann keine Kommunikation zwischen den Elementen stattfinden.[157]

5.2.7 KI – Künstliche Intelligenz

Die generelle Herausforderung, die das Programmieren einer künstlichen Intelligenz darstellt, wurde im entsprechenden Kapitel der Unreal Engine bereits kurz erläutert. Es ist eine sehr komplexe und anspruchsvolle Aufgabe, besonders aus dem Grund, weil eine KI für jedes Spiel angepasst bzw. mit anderen Fähigkeiten versehen werden muss. In diesem Kapitel soll daher noch einmal auf eine der Problematiken eingegangen werden, mit denen sich die meisten künstlichen Intelligenzen zumindest rudimentär befassen müssen: die Wegfindung.

Die erforderlichen Berechnungen zur Wegfindung erledigt in Unity die ***nav mesh agent***-Komponente, die einem ***game object,*** soll es von der KI gesteuert werden, zugewiesen werden muss. Der ***nav mesh agent*** greift dabei auf Informationen des ***navigation mesh*** zu und berechnet den optimalen Weg von der gegenwärtigen Position zur gewünschten. Wie üblich können für den ***nav mesh agent*** (nachfolgend einfach nur Agent genannt) diverse Einstellungen vorgenommen werden. Zum Beispiel der Radius, der den Abstand zu anderen Objekten definiert, die maximale Bewegungsgeschwindigkeit, ab welchem Abstand der Agent das Objekt stoppen/bremsen soll oder mit welcher Genauigkeit er anderen Objekten oder Agenten ausweicht. Auch hier gilt, dass je genauer das Verhalten des Agenten sein soll, desto rechenintensiver wird der Prozess.[158]

---

157 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 427

158 Vgl. Ebd. S. 430-431

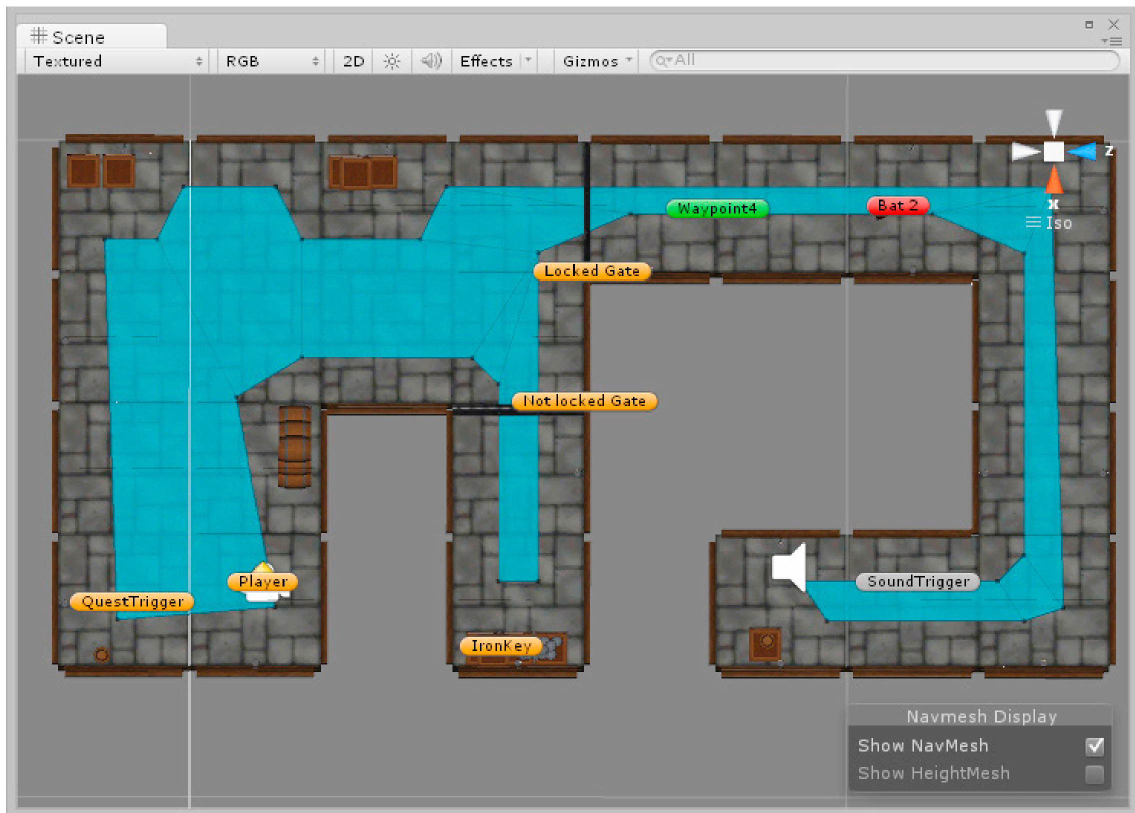

Abb. 52: Darstellung eines ***navigation mesh*** aus der Vogelperspektive. Die Bereiche, die der Agent betreten darf sind bläulich eingefärbt.

Wie bereits erwähnt benötigt der Agent zu Berechnung des Weges Informationen die das ***navigation mesh***, oder kurz ***nav mesh***, bereitstellt. Dort ist die tatsächliche Begehbarkeit der Szene gespeichert. Es definiert also welche Flächen begehbar sind, wo Hindernisse existieren und welche Wege gegebenenfalls zu bevorzugen sind. Diese Daten sind die Grundlage für den Agenten um den idealen Weg zu berechnen. Die Erzeugung oder Festlegung eines n***av mesh*** unterteilt sich in die Bereiche ***object, bake*** und ***areas***. Im Reiter ***object*** werden alle statischen Objekte einer Szene hinterlegt, die für die Berechnung des Weges berücksichtigt werden sollen bzw. müssen. Das bedeutet, es ist mit den in der Engine integrierten Mitteln zum Beispiel nicht möglich, dass die KI einen Aufzug oder eine sich bewegende Plattform verwendet. Dazu wären Lösungen von Fremdanbietern oder spezifische Änderungen auf Ebene des Quellcodes nötig. Unter ***bake*** können Einstellungen getätigt werden, die die Eigenschaften eines Bereiches definieren, damit dieser vom Agenten begehbar ist. Dazu zählt beispielsweise erneut der Radius. Dieser meint hier den Abstand des ***nav mesh*** zu einer Wand und sollte folglich mindestens dem Radius des Agenten entsprechen, da dieser sonst mit der Wand kollidiert. Ist der Radius größer als der des Agenten, hält er einen größeren Abstand zur Wand ein. Des Weiteren kann die maximale Steigung definiert werden, die der Agent erklimmen kann oder darf. Im ***areas*** Reiter können sogenannte ***navigation areas*** bestimmt werden. Dabei handelt es sich um einzelne spezielle Bereiche auf dem ***nav mesh***. Jedem Areal wird dabei ein Kostenfaktor zugewiesen, der bei der Berechnung des idealen Weges be-

rücksichtigt wird. Damit kann ein gewisses Verhalten vom Agenten provoziert werden. Als Beispiel soll folgende Entscheidungsfrage dienen: Ist es besser den kurzen Weg durch den Fluss zu nehmen oder den längeren über die weiter entfernte Brücke? Der Fluss weist dabei einen höheren Kostenfaktor auf als der Weg über die Brücke, dieser ist allerdings länger. Der Agent muss also anhand von Kosten und Entfernung abwägen, welcher Weg der sinnvollere ist. Zusätzlich können komplette Areale als ***not walkable*** definiert werden. Diese werden dann bei der Wegfindung gänzlich ignoriert.[159]

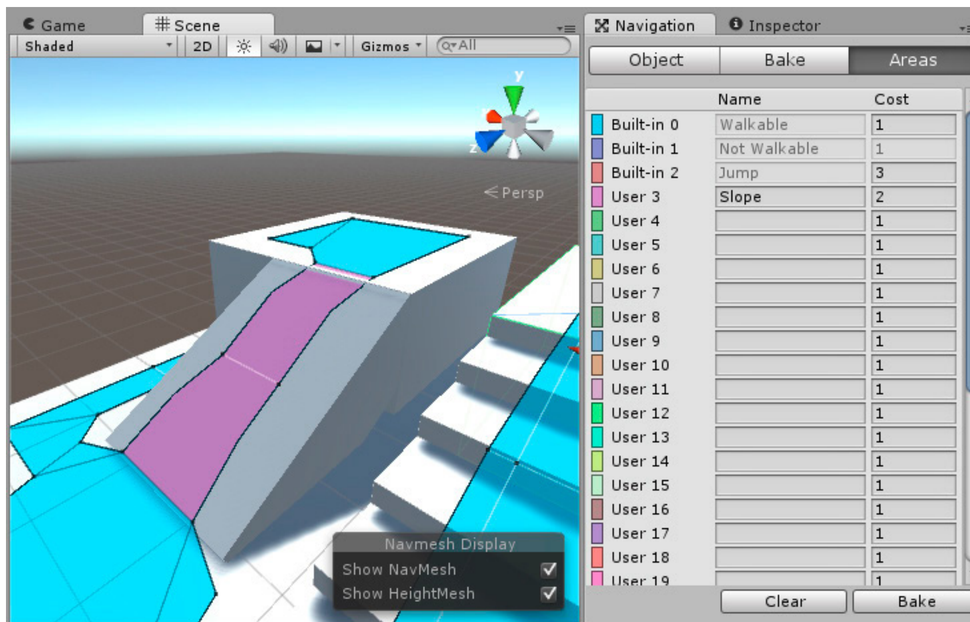

Abb. 53: Verschiedene Areale innerhalb eines n***avigation mesh*** mit ihren zugehörigen Kostenfaktoren. Der violette Bereich bezeichnet eine Steigung, die für den Agenten „teurer" zu benutzen ist als die flachen bläulichen Bereiche.

Mit dem ***off-mesh link*** ist es möglich, baulich getrennte Bereiche eines ***nav mesh*** miteinander zu verbinden. Dies erlaubt dem Agenten entweder durch hinunterfallen oder herüberspringen, je nach Einstellung, zu einem anderen Teil des ***meshs*** zu gelangen. Einzige Bedingung dabei ist, dass der andere Teil für den Agenten auch auf anderem Wege zugänglich ist. Das heißt ein Agent kann nicht von einem begehbaren in einen nicht begehbaren Bereich springen. Dieser Form der Bewegung wird ebenfalls ein Kostenfaktor hinterlegt und ***off-mesh links*** können entweder automatisch oder manuell erzeugt werden.[160]

159 Vgl. Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag, S. 432-436

160 Vgl. Ebd. S. 437-439

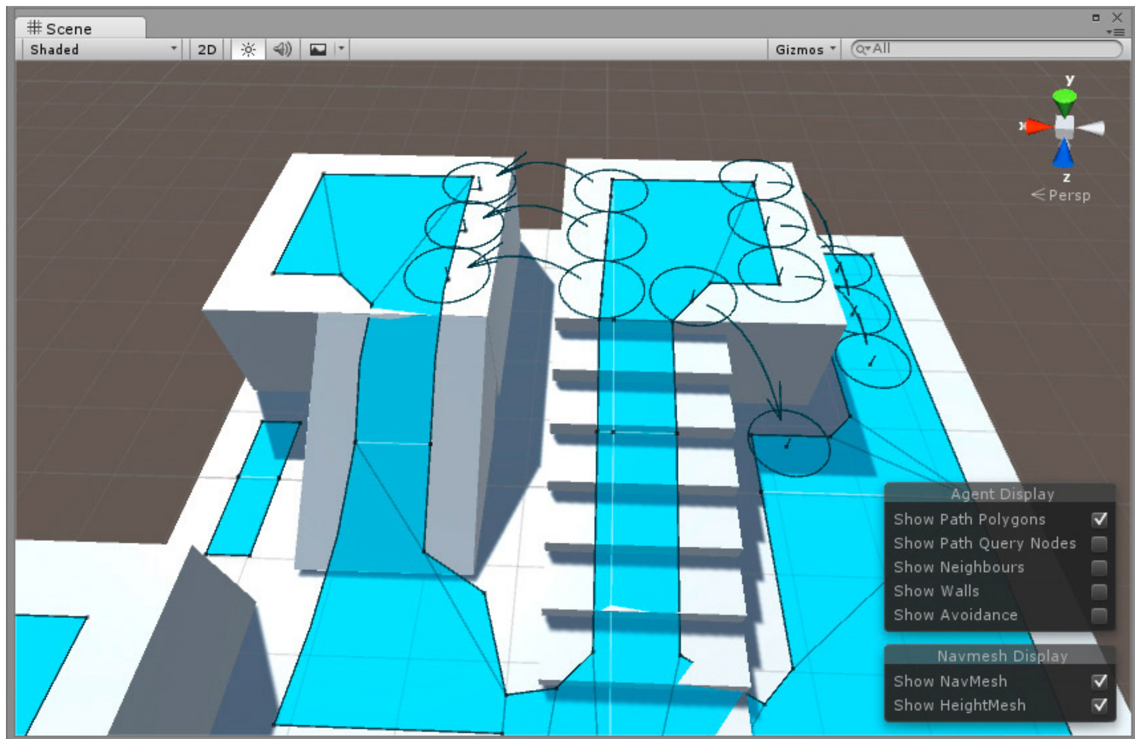

Abb. 54: ***off-mesh links,*** dargestellt durch die dunklen Kreise, ermöglichen es einem Agenten von der Plattform links auf die Plattform rechts zu springen oder von dort aus hinunter in das tiefergelegene Areal.

So viel zur Problematik der Wegfindung einer künstlichen Intelligenz in der Unity Engine. Die hier erläuterten Vorgänge beschreiben dabei nur die grundlegenden Funktionen und Möglichkeiten, um die Problematik zu lösen. Dem Entwickler steht es frei eigene Lösungen zu implementieren oder auf Angebote von Fremdanbietern zurückzugreifen. Des Weiteren ist die KI natürlich prinzipiell ebenfalls in der Lage mehr Steuereinheiten zu übernehmen als nur die Bewegung eines Charakters. Die dafür erforderliche Tiefe stände allerdings erneut in Konflikt mit dem hier gewählten Rahmen. Daher sollen die hier vermittelten grundlegenden Kenntnisse als ausreichend betrachtet werden, um einen Einblick in dieses Themengebiet zu erhalten.

6. Fazit

In der einleitenden Frage ging es darum, ob es heutzutage für Jedermann möglich ist, ein eigenes Spiel zu entwickeln. Nach den hier gewonnenen Einblicken und Erkenntnissen kann die Antwort auf diese Frage nur ein mehrdeutiges „Jein“ sein.

Der Prozess der Spieleentwicklung ist überaus komplex, besonders wenn man die Industriestandards betrachtet. Nicht umsonst sind an der Entwicklung eines großen Spiels mitunterer mehrere Teams involviert die aus vielen Spezialisten und Vertretern ihres Handwerks bestehen. Darunter befinden sich die Ingenieure und Programmierer, die Künstler und Designer für die verschiedenen Bereiche eines Spiels, die Produzenten und auch andere Abteilungen wie beispielsweise das Marketing.[161] Alle kommen zusammen, um über Monate und Jahre hinweg ein einziges Spiel zu erschaffen. Oft handelt es sich dabei um einen Zeitraum, der an große Filmproduktionen heranreicht oder sie sogar übertrifft. Und wie beim Film kommen auch bei einem Computer- oder Videospiel viele einzelne Bereiche zusammen, die in sich und in Verbindung miteinander stimmig sein müssen, um schlussendlich den Kunden zu überzeugen.

Natürlich hat ein Privatanwender oder ein eventueller Indie-Entwickler in der Regel nicht den gleichen Anspruch wie ein großes Entwicklerstudio. Doch auch die Definition eines vermeintlich simplen Spiels verändert sich mit jeder Weiterentwicklung der Branche. Das bedeutet, es wird auch immer schwieriger etwas Einfaches zu kreieren, weil das Einfache durch steigende Ansprüche immer komplexer wird. Daher sollte der Prozess der Spieleentwicklung, unabhängig des angestrebten Ergebnisses, in keinem Fall unterschätzt werden.

Was die technischen Lösungen angeht, so es ist berechtigt zu behaupten, dass der Weg zum eigenen Spiel geebnet ist. Große und namhafte Engines stehen jedem zur Verfügung und können sogar kommerziell genutzt werden. Die Communities rund um diese Engines sind jetzt schon sehr groß und wachsen stetig weiter. Es gibt unzählige frei verfügbare Formen an Hilfsmitteln. Seien es Video Tutorials, Foren oder Online-Dokumentationen. Auf Plattformen wie dem Unreal Marketplace[162] und dem Asset Store von Unity[163] können einzelne Spielelemente erworben oder teilweise ebenfalls kostenlos heruntergeladen

161 Vgl. Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press, S. 5-8

162 S. MARKETPLACE- https://www.unrealengine.com/marketplace (letzter Aufruf: 18.07.2016)

163 S. Asset Store- https://www.assetstore.unity3d.com/en/ (letzter Aufruf: 18.07.2016)

werden, um den Entwicklungsprozess für Unerfahrene zu vereinfachen. Der Traum vom selbst entwickelten Computerspiel ist zweifelsohne so greifbar wie nie zuvor.[164]

Allerdings ist es eben doch nicht so einfach zu erreichen wie es auf einen ersten schnellen Blick erscheinen mag. Die hier gewährten Einblicke in die Unreal und Unity Engine haben gezeigt, dass die Fundamente zwar für Jedermann gegeben sind, die tatsächliche Umsetzung allerdings doch einen nicht zu unterschätzenden Schwierigkeitsgrad aufweist. Egal ob die genutzte Engine auf manuelle Programmierung setzt, wie die Unity Engine, oder eine visuelle Skriptsprache bietet wie die ***blueprints*** in der Unreal Engine oder das ***flow graph*** System der CryENGINE, welches ähnlichen Prinzipien folgt.[165] Ein Computerspiel erfordert komplexe Verkettungen von Logikelementen, damit alles so funktioniert wie es beabsichtigt ist und ein reibungsloser Spielablauf entsteht. Dafür sind Kenntnisse aus der Programmierung und Softwareentwicklung unabdingbar. Wer noch keine Berührungspunkte mit diesen Bereichen hatte, wird nicht drum herumkommen, sich die grundlegenden Prinzipien und Konzepte vor der eigentlichen Spieleentwicklung anzueignen. Das Genre des Spiels, die Form oder der eigene Anspruch sind dafür unerheblich. Letzten Endes entwickelt man immer ein Stück Software, welches seinen eigenen üblichen Regeln und Gesetzmäßigkeiten unterliegt.

Hinzu kommen die eventuell benötigten Elemente aus anderen Teilbereichen der Informationstechnik. Hat man zum Beispiel für sein Spiel einen speziellen Charakter vor Augen, muss dieser zunächst einmal erschaffen werden. Dafür sind Kenntnisse aus der Computergrafik nötig. Der Charakter soll sich in der Regel in irgendeiner Art und Weise bewegen können, also werden auch Animationen benötigt. Sind diese handwerklich nicht zumindest akzeptabel angefertigt, überzeugen sie nicht und führen zu Unmut beim Spieler. So entsteht eine ganze Reihe an teilweise essentiellen Elementen, die mit der eigentlichen Spieleprogrammierung oder –Logik noch gar nichts zu tun haben.

Aus diesem Grund kann man noch einen Schritt weitergehen und festhalten, dass nicht nur Kenntnisse in der Programmierung notwendig sind, sondern dass eine generelle Affinität zu vielen Bereichen der Informations- und Computertechnik gegeben sein muss, um erfolgreich ein eigenes Spiel zu entwickeln. Selbst dann noch, wenn benötigte Elemente aus anderen Quellen hinzugekauft werden. Eine ganz grundsätzliche Ahnung der Handhabung muss einfach vorhanden sein um ein Projekt abzuschließen.

---

164 Vgl. Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag, S. XIII

165 Vgl. Tracy, Sean; Reindell, Paul (2012): CryENGINE 3 Game Development. Birmingham, UK: Packt Publishing Ltd., S. 98-118

Es ist dabei natürlich nicht ausgeschlossen, dass die Spielebranche tatsächlich einmal an den Punkt kommt, an dem die romantische Vorstellung, man könne sich sein eigenes großes und aufregendes Computerspiel einfach „zusammenklicken“ wahr wird. Betrachtet man den Entwicklungsprozess der Game-Engines, haben bereits enorme Veränderungen in die Richtung dieser Vorstellung stattgefunden. Doch zum gegenwärtigen Zeitpunkt muss gesagt werden, dass es noch nicht für jeden, der gerne Computer- und Videospiele konsumiert, möglich ist diese auch selbst zu erschaffen.

I. Quellenverzeichnis

Literaturverzeichnis

Gregory, Jason (2014, 2. Aufl.): Game Engine Architecture. Second Edition. Boca Raton, Florida: CRC Press

Reuther, Philipp: Kingdom Come: Deliverance – Cryengine am Limit, in PC Games 283 (03/2016)

Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag

Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag

Thorn, Alan (2010): Game Engine Design and Implementation: Foundations of Game Development. Jones & Bartlett Publ Inc

Tracy, Sean; Reindell, Paul (2012): CryENGINE 3 Game Development. Birmingham, UK: Packt Publishing Ltd.

Weiß, Stefan: Kingdom Come Deliverance. Auf zur Schnitzeljagd nach Böhmen: Wir haben die Beta ausführlich gespielt!, in: PC Games 283 (03/2016)

Internetverzeichnis

Assetstore: https://www.assetstore.unity3d.com/en/ (letzter Aufruf: 18.07.2016)

Autodesk: http://www.autodesk.com/products/navigation/overview (letzter Aufruf: 08.07.2016)

Bitkom: https://www.bitkom.org/Presse/Presseinformation/Gaming-hat-sich-in-allen-Altersgruppen-etabliert.html (letzter Aufruf: 18.07.206)

Cryengine: https://www.cryengine.com/get-cryengine (letzter Aufruf: 05.07.2016)

Epic Games: https://epicgames.com/ (letzter Aufruf: 05.07.2016)

Fachadmin: http://www.fachadmin.de/index.php/Client-Server_Prinzip (letzter Aufruf: 07.07.2016

Frostbite: http://www.frostbite.com/about/this-is-frostbite/ (letzter Aufruf: 05.07.2016)

Gamedev: http://archive.gamedev.net/archive/reference/programming/features/scenegraph/index.html (letzter Aufruf: 07.07.2016)

Geforce: http://www.geforce.com/hardware/technology/physx (letzter Aufruf: 07.07.2016)

GNU: http://www.gnu.de/documents/gpl.de.html (letzter Aufruf: 05.07.2016)

Göth, Christoph (21.02.2001): Binary Space Patitioning Trees. - https://www.uni-koblenz.de/~cg/veranst/ws0001/sem/Goeth.pdf (letzter Aufruf: 12.07.2016)

Havok: http://www.havok.com/physics/ (letzter Aufruf: 07.07.2016)

id Software: http://www.idsoftware.com/de-de (letzter Aufruf: 04.07.2016)

IT Wissen: http://www.itwissen.info/definition/lexikon/Voxel-volume-pixel-Volumenpixel.html (letzter Aufruf: 04.07.2016)

IT Wissen: http://www.itwissen.info/definition/lexikon/Middleware-middleware.html (letzter Aufruf: 06.07.2016)

IT Wissen: http://www.itwissen.info/definition/lexikon/Rendering-rendering.html (letzter Aufruf: 07.07.2016)

Kern, Sabine; Neumayer, Ingo (07.07.2016): Computer und Roboter. Künstliche Intelligenz - http://www.planet-wissen.de/technik/computer_und_roboter/kuenstliche_intelligenz/ (letzter Aufruf: 08.07.2016)

Kühl, Eike (13.10.2012): PC-Spiele erreichen das nächste Level - http://www.zeit.de/digital/games/2012-10/pc-konsole-videospiele-entwicklung (letzter Aufruf: 18.07.2016)

Khronos: https://www.khronos.org/opengl/ (letzter Aufruf: 06.07.2016)

Microsoft: https://msdn.microsoft.com/de-de/library/kx37x362.aspx (letzter Aufruf: 14.07.2016)

MODDB: http://www.moddb.com/games/unreal-tournament-2004/mods (letzter Aufruf: 05.07.2016

Naughty Dog: http://www.naughtydog.com/ (letzter Aufruf: 06.07.2016)

Ogre3D: http://www.ogre3d.org/ (letzter Aufruf: 06.07.2016)

Scratch: https://scratch.mit.edu/ (letzter Aufruf: 05.07.2016)

Stampfl, Nora (2014): ZWERGENAUFSTAND. INDIE-GAMING-SZENE IN DEUTSCHLAND. - https://www.goethe.de/de/kul/mol/20445752.html (letzter Aufruf: 18.07.2016)

Sarathi Paul, Partha; Goon, Surajit; Bhattacharya, Abhishek (2012): HISTORY AND COMPARATIVE STUDY OF MODERN GAME ENGINES - http://bipublication.com/files/IJCMS-V3I2-2012-07.pdf S. 246 (letzter Aufruf: 04.07.2016)

Tutorialspoint: http://www.tutorialspoint.com/operating_system/os_process_scheduling.htm (letzter Aufruf: 06.07.2016)

Unrealengine: https://www.unrealengine.com/marketplace (letzter Aufruf: 18.07.2016)

Unity3D: https://unity3d.com/public-relations (letzter Aufruf: 13.07.2016)

Valve: http://www.valvesoftware.com/ (letzter Aufruf: 05.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Pong (letzter Aufruf: 18.07.2016)

Wikipedia: https://de.wikipedia.org/wiki/Doom (letzter Aufruf: 04.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Level_of_Detail (letzter Aufruf: 04.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Ego-Shooter (letzter Aufruf: 04.08.2016)
Wikipedia: https://de.wikipedia.org/wiki/3dfx (letzter Aufruf: 04.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Scratch_(Programmiersprache) (letzter Aufruf: 05.07.2016)
Wikipedia: https://en.wikipedia.org/wiki/Preemption_(computing)#Time_slice (letzter Aufruf: 06.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Prozess-Scheduler (letzter Aufruf: 06.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/DirectX (letzter Aufruf: 06.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Bildsynthese (letzter Aufruf: 07.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Zwischensequenz (letzter Aufruf: 07.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Starrer_K%C3%B6rper (letzter Aufruf: 07.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Starrk%C3%B6rpersimulation (letzter Aufruf: 07.07.2016)
Wikipedia: https://en.wikipedia.org/wiki/Morph_target_animation (letzter Aufruf: 07.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Compiler (letzter Aufruf: 08.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/K%C3%BCnstliche_Intelligenz (letzter Aufruf: 08.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Side-Scroller (letzter Aufruf: 12.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Super_Mario_Bros. (letzter Aufruf: 12.07.2016)
Wikipedia: https://en.wikipedia.org/wiki/Digital_sculpting (letzter Aufruf: 12.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Third-Person-Shooter (letzter Aufruf: 04.08.2016)
Wikipedia: https://de.wikipedia.org/wiki/Digital_Signage (letzter Aufruf: 13.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/C-Sharp (letzter Aufruf: 14.07.2016)
Wikipedia: https://de.wikipedia.org/wiki/Gleitkommazahl (letzter Aufruf: 16.07.2016)

Abbildungsverzeichnis

Abb. 1 Game-Engine Architektur
2 Hardware-Ebene
3 Treiber-Ebene
4 Betriebssystem
5 Middleware
6 Plattformabhängige Ebene
7 Kern-Systeme
8 Ressourcen Manager
9 Low-Level Renderer
10 Szenengraph
12 Visuelle Effekte
13 Front End
14 Profiling & Debugging
15 Physik
16 Skelett Animationen
19 Human Interface Device
20 Audio-Engine
21 Multiplayer
22 Gameplay Grundlagen
23 Spielspezifische Inhalte
http://www.gameenginebook.com/coursemat.html (letzter Aufruf: 01.08.2016)

Abb. 11 Szenengraph
https://www.packtpub.com/sites/default/files/Article-Images/2480-03-11.png (letzter Aufruf: 01.08.2016)

Abb. 17 Ragdoll-Beispiel 01

18 Ragdoll-Besipiel 02
http://www.gamespy.com/articles/489/489917p1.html (letzter Aufruf: 02.08.2016)

Abb.	24	GUI Unreal Engine	S. 14
	25	Beispiel Blueprint	S. 30
	26	Funktion Toggle Visibility	S. 45
	27	Ansicht Sockel	S. 84
	28	Optionen Sockel	S. 84
	29	Physik Einstellungen	S. 139
	30	Ein- und Ausschalten der Physik	S. 145
	31	Zerstörtes Steinmodell	S. 149
	32	Platzhalter Durchgang	S. 167
	33	Heightmap	S. 177
	34	Landschaftsmaterial	S. 195
	35	Aim Offset	S. 310
	36	Anim Graph	S. 317
	37	State Machine	S. 321
	38	Nav Mesh	S. 345
	39	Wegpunkt Steuerung	S. 349
	40	Sichtfeld Visualisierung	S. 351

Richartz, Jonas (2016): Spiele entwickeln mit UNREAL ENGINE 4. München: Carl Hanser Verlag

Abb.	41	GUI Unity Engine	S. 8
	42	MeshFilter und MeshRenderer	S. 113
	43	Standard Shader Beispiel	S. 119
	44	Multiple Sprite Textur	S. 147
	45	Collider-Anordnung	S. 217
	46	Landschaft Gegenüberstellung	S. 327
	47	Platzierung von Bäumen	S. 328
	48	Platzierung von Details	S. 330
	49	Avatar-Mapping	S. 400
	50	Animation Controller	S. 405
	51	Avatar Mask	S. 411
	52	Nav Mesh	S. 433
	53	Nav Mesh Areas	S. 436
	54	Off Mesh Links	S. 438

Seifert, Carsten (2015, 2. Aufl.): Spiele entwickeln mit Unity 5. München: Carl Hanser Verlag

II. Anhang

Eidesstattliche Erklärung:

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

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

Leopoldshöhe, den 10.08.2016

Danksagung

Ich möchte mich an dieser Stelle bei der NVIDIA ARC GmbH für die Möglichkeit bedanken, meine Abschlussarbeit zu einem ihrer innovativen technischen Erzeugnisse anfertigen zu dürfen.

Besonderer Dank gilt auch Prof. Dr. rer. nat. Guido Falkemeier für die Unterstützung im Vorfeld des Verfassens der schriftlichen Ausarbeitung und die Möglichkeit diese als Abschlussprüfung einreichen zu dürfen. Für seine Tätigkeit als Zweitprüfer und die ebenfalls geleistete Unterstützung danke ich außerdem Prof. Dr. phil. Frank Lechtenberg.

Für die allgemeine Unterstützung und Ermutigung während der Abschlussarbeit und die spätere Überarbeitung der schriftlichen Ausarbeitung, danke ich herzlichst Nathalie Lambert und Gregor Beyer.

Nicht zuletzt geht mein großer Dank an meinen Projektpartner: André Düchting. Für die Chance mich am Projekt zu beteiligen, für die geleistete Arbeit, die unzähligen übernommenen Aufgaben während der Produktion und ein Endprodukt, auf das man stolz sein kann.

Vielen Dank!