

Bearbeitungsbeginn: 01.03.2020

Vorgelegt am: 29.08.2020

Thesis

zur Erlangung des Grades

Master of Arts

im Studiengang Design Interaktiver Medien

an der Fakultät Digitale Medien

Marko Fehrenbach

Matrikelnummer: 260333

Integration einer Physik-Engine in die Lern-Game-Engine FUDGE, sowie Optimierung der Nutzerinteraktion und Darstellung dieser Komponenten

Erstbetreuer: Prof. Jirka Dell'Oro Friedl

Zweitbetreuer: Prof. Dr. Dirk Eisenbiegler

Abstract | Deutsch

Game Engines dienen dazu Spiele und andere interaktive Inhalte darzustellen und mit Funktionen in Echtzeit zu versehen. Auf diesem Markt gibt es eine Vielzahl von etablierten Anbietern, deren Fokus liegt jedoch auf der Produktion von kommerziellen Inhalten.

Der Game Editor FUDGE (Furtwangen University Didactic Game Editor), der durch Prof. Jirka Dell'Oro-Friedl an der Hochschule Furtwangen ins Leben gerufen wurde, wird rund um die Anforderungen der Lehre der Spieleentwicklung entwickelt .

Auf Webtechnologien basierend und für alle Plattformen verfügbar, zugeschnitten auf Transparenz und Austausch mit wichtigen Grundfunktionen zur Erstellung von Spielen und interaktiven Inhalten.

Aufgrund der frühen Phase und Größe des Projektes, fehlen FUDGE noch einige Komponenten, wie z.B. eine Physik Engine um typische Spielmuster, wie Kollision von Objekten, Auslöser oder Auswahl von Objekten per Mauseingabe, einfach umzusetzen. Diese Masterarbeit beschäftigt sich daher mit der Auswahl einer frei verfügbaren Physik-Engine, anhand der für FUDGE wichtigen Anforderungen, und der anschließenden Integration der erarbeiteten wichtigen Funktionalitäten.

Dafür wurden FUDGE, grundlegende Physik Engine Bestandteile und potenzielle Engines, analysiert. Eine Integration für die Oimo Physics Engine wurde entwickelt und programmiert.

Zudem wird sich mit der Nutzerinteraktion zwischen der Physik-Engine und FUDGE Nutzern auseinandergesetzt und Hilfsmaterial entwickelt um zu gewährleisten, dass ein relativ nahtloser Übergang zwischen der Nutzung von FUDGE ohne, sowie mit Physik-Komponenten gewährleistet ist. In diesem Zug wurde die Integration getestet und angepasst.

Abstract | English

Game Engines are build to help create games and interactive applications, with real time interaction. This market is full of established providers, but their focus lies in building tools to create commercial products.

The game editor FUDGE (Furtwangen University Didactic Game Editor), that was created by prof. Jirka Dell'Oro-Friedl at Hochschule Furtwangen University, is purposefully build to support students in their learning process of creating interactive content.

It is based on modern web technologies and is cross platform, specifically tailored to have transparent processes and small project sizes, while providing the basic features needed to create games. This leads to a good communicative learning environment.

Because of its early development status and the size of the project, FUDGE is still lacking features like a physics engine to create common game patterns e.g. collision, triggers and raycast mouse selection. This master's degree paper is therefore selecting an open source web physics engine, with requirements of FUDGE in mind, and the integration of the proposed set of important features.

For this, FUDGE and basic physics engine features were analyzed and potential engines tested. A integration concept for the Oimo physics engine was developed and programmed.

To ensure a good user interaction between the physics engine and FUDGE, the interaction was iteratively optimized, extra features were implemented and learning material was created to provide good onboarding. In the process the integration was tested and adapted.

Inhaltsverzeichnis

Abstract Deutsch	2
Abstract English	3
Inhaltsverzeichnis	4
1. Einleitung	6
1.1 Ausgangslage	6
1.2 Forschungsgegenstand	7
2. FUDGE Furtwangen University Didactic Game Editor	8
2.1 Basis Aufbau	8
2.2 Integration von Modulen	10
3. 3D Physik Engines	11
3.1 Physik Engines in interaktiven Inhalten	11
3.1.1 Körper und Kräfte	13
3.1.2 Physik Welt Kollisionen und Detektionen	16
3.1.4 Verhaltensweisen und Einschränkungen	22
3.2 Anforderungen an eine Physik Engine für Fudge	24
3.3 Auswahl einer Physik Engine	26
3.3.1 Cannon.js	27
3.3.2 Ammo.js	28
3.3.3 Oimo	29
3.3.3.1 Oimo.js	30
3.3.3.2 Oimo Physics	30
3.3.4 Weitere Optionen	31
3.3.5 Tests und finale Auswahl	33
4. Entwicklung der Integration	37
4.1 Physik Engine Integration bei Vergleichsobjekten	37
4.1.1 Unity	38
4.1.2 Babylon.js	40
4.1.3 Unreal Engine	41
4.1.4 PlayCanvas	43
4.2 Daraus abgeleitete Paradigmen für ein Integrationskonzept	45
5. Umsetzung der Integration	47
5.1 Kollision, Kräfte und Materialeigenschaften	50
5.2 Raycast - Objekterkennung	57
5.3 Spielsteuerung durch Physik-Events Kollision und Trigger	59
5.4 Gelenkverbindungen	63

5.5 Besonderheiten und Schwierigkeiten	70
6. Entwicklung und Optimierung der Nutzerinteraktion	73
6.1 Optimierungen für die Nutzung - Visuelles Debugging	73
6.2 Anpassungen für die Nutzung im Editor	76
7. Erstellung Lernmaterial zum Umgang mit der Physik in FUDGE	82
8. Schlussbetrachtung	86
Glossar	88
Abbildungsverzeichnis	90
Quellenverzeichnis	92
Literaturverzeichnis	92
Webquellen	92
Versionsinfos über verwendete Programme und Sprachen	94
Eidesstattliche Erklärung	95
Anhang	96

1. Einleitung

1.1 Ausgangslage

Spieleentwicklung ist längst ein wichtiger Bestandteil der Ausbildung für die Produktion von interaktiven Inhalten, dabei werden Muster und Funktionsweisen vermittelt die über Spiele weit hinaus gehen. Hierfür wurden an der Hochschule Furtwangen lange sog. Game Engines, wie Unity, benutzt, welche bereits ausgereift und einfach zu benutzen sind und z.B. auch in Film/Animation zum Einsatz kommen. Allerdings sind diese auf eine Nutzung im professionellen und kommerziellen Kontext ausgelegt, d.h. es wird weniger Wert auf einfachen Austausch zwischen Schaffenden und vor allem Lernenden gelegt, sowie auf transparente Funktionsweisen.

Zudem hat die native Einbindung von komplexen 3D Inhalten in den Browser, mittels WebGL, die Nutzung von Webtechnologien, für die Lehre, sehr interessant werden lassen. Da die verwendeten Javascript bzw. Typescript Inhalte zur Laufzeit besser einsehbar sind und meist kleinere Projektgrößen mit sich bringen.

Die meisten Anbieter bieten zwar eine Nutzung von WebGL an, aber betrachten dies nicht als Schwerpunkt und fokussieren sich eher auf die Funktionsvielfalt und visuelle Fidelität. Daher hat Prof. Jirka Dell’Oro-Friedl an der HFU den Game Editor, FUDGE (Furtwangen University Didactic Game Editor) ins Leben gerufen, um eine Option für die Lehre zu bieten, deren Ziel es nicht ist eine Alternative für bestehende kommerzielle Game Engines zu sein, die es auch auf Webbasis gibt, sondern einen anderen Betrachtungswinkel auf die Spieleentwicklung, bzw. interaktive Inhalte, zu bieten der so bisher nicht abgedeckt wurde.

Da FUDGE sich noch in einer frühen Phase befindet werden noch nicht alle grundlegenden Funktionen abgedeckt, daher gibt es verschiedene Arbeiten rund um das Projekt. Diese Thesis befasst sich mit dem Aspekt von Physik in Spielen und der dafür nötigen Physik Engine.

Da eine Physik Engine sehr schnell an großem Umfang gewinnt, wäre es wenig sinnvoll eine Eigene zu entwickeln. Daher wird innerhalb dieser Arbeit eine, für die speziellen Anforderungen passende, Physik-Engine ausgewählt und integriert.

Es gab bereits eine Arbeit zur Auswahl einer Physik Engine für FUDGE, diese kam allerdings zu keinem klaren Ergebnis und wird daher für diese Arbeit kaum berücksichtigt. Einzig und allein zu vermerken gilt es, dass einige wenige der zur Auswahl in Betracht gezogenen Physik Engines, in dieser Arbeit ebenso in Betracht gezogen werden.

1.2 Forschungsgegenstand

Physik Engines können verschiedene Funktionsumfänge und einen bestimmten Aufbau mit sich bringen. FUDGE wiederum hat Paradigmen und Anforderungen die bedient werden müssen.

Daher wird zuerst FUDGE selbst untersucht und dargestellt, um anschließend passende, auf webtechnologien basierende, Physik Engines einstufen zu können. Dies beinhaltet die Funktionsweise von FUDGE und die Einbindung anderer Elemente in diese Strukturen.

Es wird zuerst der wichtigste Funktionsumfang für Physik Engines grundlegend hergeleitet, sowie der Nutzen und die Funktionsweise, erklärt, z.B. die Kollision. Davon wird der Funktionsumfang und die Anforderungen speziell für die Lehre abgeleitet.

Darauffolgend wird eine entsprechende Engine ausgewählt und die Integration von Physik Engines innerhalb anderen Game Engines verglichen, um bei der Integration in FUDGE, bewährte und gängige Elemente übernehmen zu können.

Anschließend werden einige Maxime und ein Konzept, für die Integration, entwickelt. Das Hauptziel ist zuerst die Kommunikation der beiden Teile, mit je eigenen Coding Strukturen und anschließend die komplette Integration in FUDGE Strukturen. Teil

dieser Integration ist es einige FUDGE-Komponenten zu entwickeln, welche auch im Editor dargestellt werden können.

Um die Integration greifbar zu machen werden abschließend Lernbeispiele und Inhalte für die Physik in FUDGE erstellt und Prozesse optimiert.

Zum Zeitpunkt der Arbeit befindet sich FUDGE in einem relativ frühen Stadium, besonders der visuelle Editor, daher unterliegen einige Herangehensweisen häufigen Änderungen.

2. FUDGE | Furtwangen University Didactic Game Editor

Wie bereits eingeleitet ist FUDGE eine Alternative zu gängigen Game Engines, angepasst und erschaffen speziell für die Anforderungen der Lehre.¹ Was zu besonderen Ansätzen führt, wie z.B. die Vermittlung von bestimmte Konzepte. Daher muss man diese Engine bzw. Editor genauer betrachten.

2.1 Basis Aufbau

Fudge besteht zum einen aus einer Engine, FudgeCore.js. Diese wird als Kern des FUDGE Konstruktes bezeichnet und liefert daher den Namespace FudgeCore. Der Engine Teil bedeutet eine Bündelung von Funktionalitäten zum Erstellen von interaktiven Inhalten durch Code.

Für das Rendering wird der WebGL 2 Standard verwendet, bei dem 3D Inhalte auf ein HTML5 Canvas Element gezeichnet werden. Dafür wird ein Rendering-Kontext erstellt, welcher Puffer aus Objektinformationen, mittels Vertex- und Fragmentshadern zeichnet, analog zu klassischen Game Engines. Ebenso klassisch ist die Bereitstellung der Informationen, zum Zeichnen, über Objekte, diese werden in FUDGE Nodes (engl. Knoten) genannt.

Diese Nodes können in Eltern/Kind Beziehungen gesetzt werden, wodurch sich ihre Transformationen in hierarchische Strukturen verketten lassen, einem sog. Szenengraph.

¹ [5] FUDGE - <https://github.com/JirkaDellOro/FUDGE/wiki/Motivation>

Jede Node besitzt Komponenten, durch die sie erweitert wird. Dabei enthalten die Knoten kaum eigene Informationen, aber jede Komponente ermöglicht eine neue Funktionalität.

Fudge bietet, zu Beginn der Thesis, die Möglichkeit einer *ComponentTransform* für die mathematische Positionierung des Knotens im Szenengraph.

Eine *ComponentMesh* welche die geometrische Form der Node bestimmt, zusammen mit *ComponentMaterial*, die für die visuelle Darstellung und Shader verantwortlich ist.

Zudem gibt es noch Komponenten für Audio und andere bisher implementierte Funktionalitäten, die für die Physik eine untergeordnete Rolle spielen. Über Veränderungen an entsprechenden Komponenten und Knoten Strukturen lassen sich Spielkonzepte und Simulationen erstellen, dafür bietet FUDGE zusätzlich Event und Zeitmanagement.

Diese Arbeit beschäftigt sich entsprechend mit der Erweiterung der Funktionalitäten, mittels einiger Komponenten für die Physik, die einen physischen Körper und weitere Elemente ergänzt. Im Gegensatz zu bisherigen Komponenten besitzt die Physik allerdings besondere Ansprüche auf die in Kapitel 3 eingegangen wird.

Eine FUDGE Szene besteht immer aus einer HTML Datei, in der das kompilierte FudgeCore-Script eingebunden und ein Canvas bereitgestellt wird. Dazu eine eigene szenenbeschreibende Javascript Datei, die aus einer Typescript Datei kompiliert wird, diese wird üblicherweise main.js genannt.

Zur FUDGE Engine kommt der namensgebende FUDGE Editor, der zum Zeitpunkt der Arbeit noch in seinen Anfängen steckte. Dieser soll, ähnlich wie bei anderen Editoren, Möglichkeiten bieten Szenen über Interface-Elemente statt Code zu bearbeiten, und entsprechende Änderungen direkt zu visualisieren. Der Editor basiert ebenfalls auf Webtechnologien und wurde mittels Electron² umgesetzt. Electron nutzt den Chromium Browser als Basis um Javascript Inhalte darzustellen, daher ergibt sich für FUDGE als Testumgebung der Chrome Browser.

² vgl. <https://www.electronjs.org/> Javascript Anwendungsentwicklung

Ziel des Editors, im Hinblick auf die Lehre und daher kleine Projektgrößen, sowie Lesbarkeit des Code, ist es, dass der Editor JSON Dateien verarbeitet. Und so FUDGE Szenen über das Austauschen einzelner Dateien verfügbar gemacht werden können.

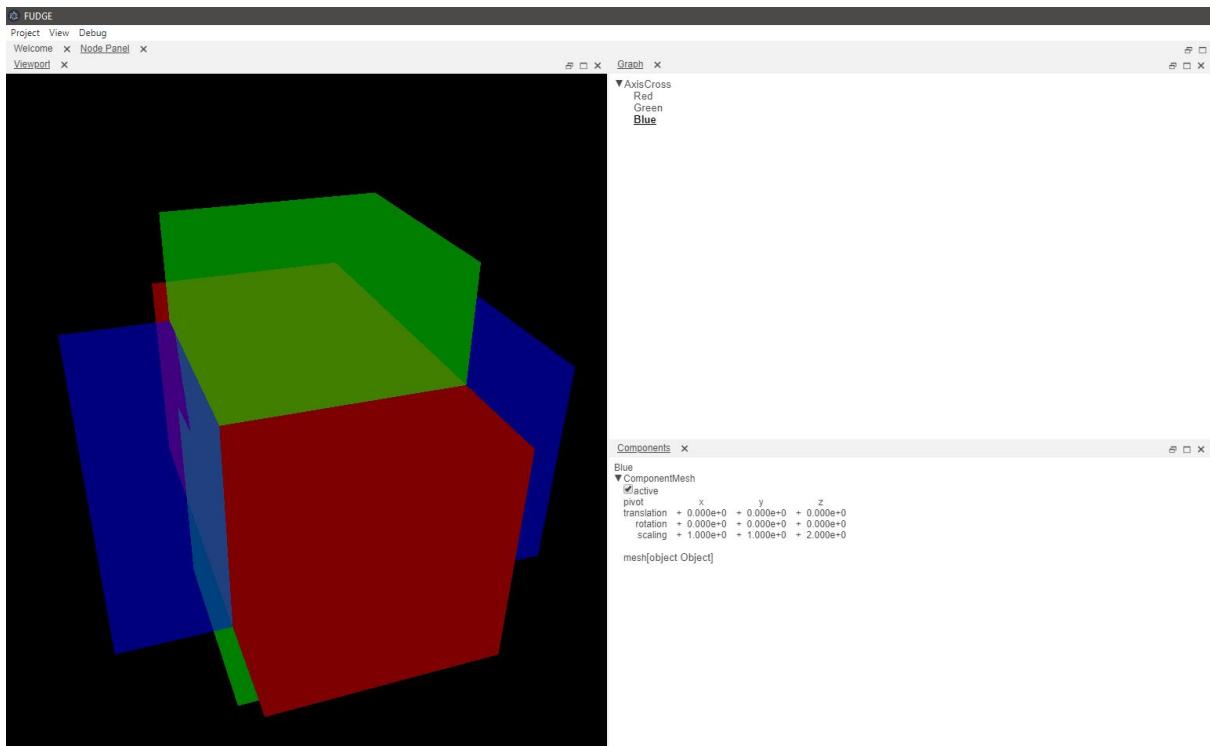


Abb. 1 - Stand des Fudge Editors ohne Physik Integration. Speichern/Laden einfacher Szenen. (10.07.2020)

Bis zur Beendigung der praktischen Arbeit bot der Editor bisher nur die Funktionen JSON Dateien zu speichern/laden und aus den geladenen Informationen eine Szene aus Nodes mit Komponenten zu erstellen, sowie diese ggf. anzuzeigen. Teilweise ließen sich Werte, wie die Position, durch die Transform Komponente direkt beeinflussen.

In der Arbeit sollen die Physik Komponenten weitestgehend im Editor geladen und angezeigt werden können.

2.2 Integration von Modulen

Zudem besteht FUDGE noch aus ergänzenden Modulen, wie *FudgeUI* und *FudgeHelper*. Hierbei handelt es sich um Funktionalitäten die zur Erstellung von User Interfaces verwendet werden und eine Sammlung von Funktionen für spezifische

Anwendungsfälle. Diese sind nicht Teil des Kerns und müssen extra in die HTML Datei eingebunden werden.

Zu Beginn gab es die Überlegung die Physik ebenso zu behandeln, da klar ist, dass die ausgewählte Physik Engine nicht in den Kern integriert wird, d.h. sie muss stets separat in der HTML geladen werden. Und man daher die Komponenten, sowie andere Bestandteile der Integration ebenso modular hinzufügen könnte, analog zum UI. Allerdings stellt die Physik einen integralen Bestandteil dar und die Integration sollte zum Kern gehören, da sie in einige Bereiche, wie Rendering, direkt eingreifen muss und nicht als abgeschlossenes Modul betrachtet werden kann.

Neue Funktionalitäten werden möglichst getrennt in FUDGE integriert, d.h. sie bilden eine Komponente die an eine Node angehängt werden kann, aber nicht muss. Alle alten und neuen FUDGE Szenen mussten funktionieren **ohne** die Physik Engine einzubinden, oder neuen Code mit physikalischer Verbindung zu schreiben. Dies galt daher als oberstes Paradigma für die Integration, *direkt* in den FudgeCore.

3. 3D Physik Engines

3.1 Physik Engines in interaktiven Inhalten

Fokus dieser Arbeit ist die Integration und nicht die Kreation einer Physik Engine, weshalb die Funktionsweise einer Physik Engine mit ihren Bestandteilen, Aufgaben und Problemen dargestellt wird, allerdings in vereinfachter konzeptioneller Form.

Die Basis einer Physik im virtuellen Raum ist die Annahme von Regeln über einen Zeitraum, wobei der Zeitraum in diskrete Abschnitte eingeteilt ist (von variabler Größe). Zudem gibt es normalerweise keine kontinuierliche Physik (selbst sog. kontinuierliche Kollisionsdetektion, wird über Voraussagen und kleinere Zeitabstände realisiert).

Im Gegensatz zur Realität, gibt es keine physischen Phänomene wenn sie nicht

definiert und angewandt werden. In den diskreten Zeiträumen wird die logische Einhaltung dieser definierten Regeln überprüft und Körper entsprechend angepasst. Dieses Konstrukt nennt man Simulation, bzw. Physik Welt. Sie wird von einer Physik Engine generiert und verwaltet, zudem agiert sie eigenständig, unabhängig von der Transformationswelt, dem mathematischen Hierarchie Konstrukt, der Szenengraph.

Es können ebenso Phänomene definiert werden die in der Realität nicht vorkommen können, wie eine Box die nicht von der Gravitation beeinflusst wird, während eine andere Box, mit denselben Eigenschaften, bis auf ihren Gravitationsfaktor, fällt wie ein Stein.

Die Basis dieser Konzepte bilden Körper. Körper bezeichnet ein Objekt mit Masse und Form. In der Realität gibt es quasi nur ein bekanntes Objekt was nicht direkt als Körper identifiziert wird, Licht. Im digitalen Raum allerdings, ist ohne eine implementierte Physik, jede Node nur ein Objekt und reagiert direkt auf Code, ungeachtet einer Form, einer Masse oder wirkenden Elementen.

Man unterscheidet zwischen einem Festkörper engl. *Rigidbody*, häufig abgekürzt als *Body*, da sie bisher den Hauptgegenstand einer Physik Engine darstellten. Sie beschreiben Körper die in ihrer Form unveränderlich sind, wie z.B. ein Ziegelstein (hier beginnt schon die erste Vereinfachung einer Simulation gegenüber der Realität, Festkörper sind an sich unzerbrechbar bzw. unkaputtbar).

Der zweite Typ von Körpern sind sog. Weichkörper, engl. *Softbody*, zu denen auch Liquide gehören. Beispiele hierfür sind z.B. Seile, bzw. Stoff, also Körper die ihre Form bei Krafteinwirkung verändern. Ihre Berechnung ist ungleich aufwendiger und wurde häufig annähernd durch Kombinationen von rigiden Körpern realisiert. Z.B. ein Seil aus einem Verbund vieler kurzer Objekte, die per Gelenk ein ähnliches Verhalten aufweisen. Oder Flüssigkeiten aus einer Vielzahl von einzelnen Kugeln.

Da die Basis einer Physik sich auf rigide Körper beschränkt und FUDGE sich auf die Lehre der Basis fokussiert, wird im Folgenden auf die 4 großen Funktionalitäten von

Rigidbodies eingegangen, welche sich im Vergleich mit anderen Game Engines und Physik Engines als die wichtigsten herausstellten.

3.1.1 Körper und Kräfte

Körper bestehen aus Masse und Form. Dabei ist die Masse vereinfacht ein Gewicht in kg, welches am Schwerpunkt der Form wirkt, daher ist die Form der ausschlaggebende Anteil. Die Form wird in der Physik Kollisionsobjekt, kurz engl. Collider, genannt und kann von der visuellen Ebene abweichen, entweder zur Steigerung der Performanz, oder weil es aus Game Design Gründen so gewollt ist.

Der Collider ist eine geometrische Definition einer Form, die einen spezifischen Raum innerhalb der Simulation einnimmt. Einnehmen bedeutet andere Objekte haben keinen Zutritt zu diesem eingenommenen Raum. Da es sich um Festkörper handelt kann der eingenommene Raum theoretisch nicht verändert werden, d.h. es entsteht Spannung zwischen Objekten wenn sie den Raum einnehmen wollen, dafür muss das andere Objekt reagieren. Es entsteht wechselseitige Kräfte und Kollisionen.

Die Form des Colliders ist dabei in der grundlegenden Physik Engine immer konvex. D.h. mathematisch geht kein Strahl der innerhalb des Körpers begonnen hat zweimal durch die Hülle des Körpers. Vereinfacht gesagt das Objekt hat kein Loch, einen mathematischen Genus von 0. (Sehr vereinfacht, ein Objekt mit einer Biegung, z.B. ein Haken, hat kein *geschlossenes* Loch, aber dennoch ist es nicht konvex). Es gibt jedoch fortgeschrittene Physik Engines mit Möglichkeiten zur Berechnung von konkaven Collidern, häufig Meshcollider genannt.

Es ist möglich mehrere Collider zu einem zu verbinden sog. *Compound Shapes*. Ein Collider ist daher meist eine Annäherung an das visuell dargestellte Objekt und demnach ist die Simulation des Körpers bereits durch die Auswahl des Colliders mit Fehlern belastet.

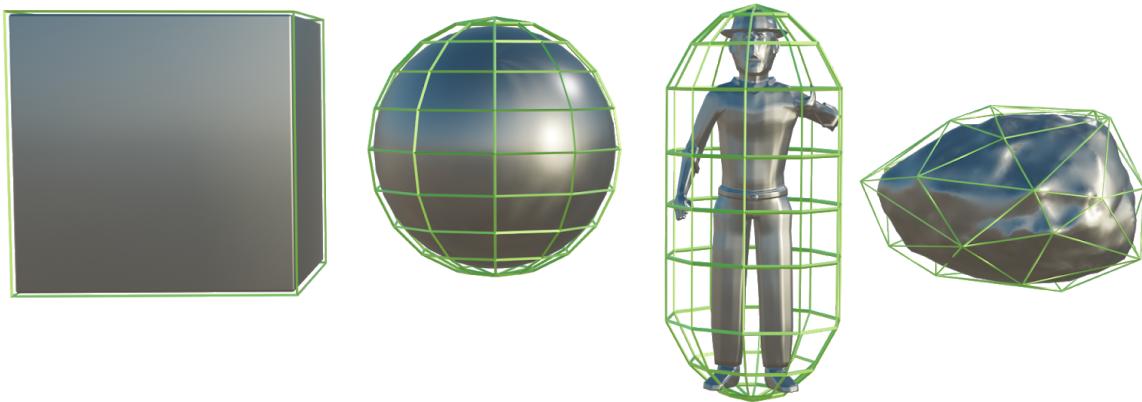


Abb. 2 - Collider (grün) umschließen visuelle Objekte und machen sie dadurch zu physischen Körpern.
Sie sind umso mehr Annäherungen umso komplexer das visuelle Objekt ist.

Der rigide Körper besteht zusätzlich dazu noch aus Eigenschaften die in der Realität zwar auch durch Masse und Form, aber auch durch verwendete Materialien, beeinflusst werden.

Zum einen das Konzept des Widerstands gegenüber anderen Oberflächen, der Luft und zum anderen die Restitution bei einem Aufprall. Dies ist für eine korrekte Darstellung der Physik zu beachten. Während das Aussehen z.B. einer Metallkugel, ein anderer Teil der Game Engine ist, muss in der Physik Engine definiert werden, dass eine Metallkugel kaum Restitution besitzt d.h. sie prallt schlecht ab. Ebenso ist eine Metallkugel glatt, d.h. ihre Friktion bzw. Reibung mit einem Teppich wäre niedriger als die einer Stoffkugel.

Diese Eigenschaften wirken sich auf Energie bzw. die Bewegung der Körper aus, dem Hauptteil einer Physiksimulation.

Kräfte

Bisher wurde ein Körper mit seinen Eigenschaften definiert. Ein Körper bewegt sich allerdings erst, wenn Kräfte auf die Masse einwirken. Eine Kraft ist grundlegend eine Art kontinuierlicher Druck auf den Körper, sie kann als Schieben verstanden werden. Die Formel **Force = Mass * Acceleration** (**Kraft = Masse * Beschleunigung**), liegt diesem Prinzip zugrunde. D.h. jede wirkende Kraft geteilt durch die Masse ergibt eine wirkende Beschleunigung des Körpers im Raum. In der Berechnung ergibt dann die Summe aller Kräfte und Impulse, also kinetischer Energie, eine Geschwindigkeit mit der, der Körper dann agiert.

Diese Kraft kann nun linear wirken, also auf den Schwerpunkt des Körpers, d.h. er wird bewegt ohne Drehung, oder Angular wirken, also ein Drehmoment (engl. *Torque*) erzeugen. Dies ist das vereinfachte Grundprinzip wie Körper bewegt werden und basiert auf den Gesetzen der Bewegung, definiert durch Newton³, sowie Gesetzen von Kepler und Euler.⁴

In Physik Engines kann eine Kraft z.B. durch einen schiebenden Körper realisiert werden, aber auch durch Anwenden von Funktionen um eine Kraft hinzuzufügen, sowie z.B. die Gravitationskraft.

Impulse

Der Impuls kann ähnlich verstanden werden wie die Kraft, es wird ebenso kinetische Energie erzeugt die zu einer Veränderung der Geschwindigkeit führt. Allerdings handelt es sich beim Impuls um eine instantane Wirkung, die bei der Kollision zweier Körper stattfindet, er ist eher als Stoß zu verstehen.

In der Realität werden bei einer Kollision beide Körper zusammengedrückt und sie prallen ab. Dabei geht Energie verloren für Wärme und Audio, aber die restliche, übertragene Energie wird für eine Geschwindigkeitsänderung benutzt. Um das Ganze zu vereinfachen wird die Restitution als konstanter Faktor verwendet um den Erhalt von Energie zu berechnen. Ein Gummiball erhält sehr viel Energie, zwei Billardkugeln mit gleicher Geschwindigkeit die aufeinander prallen bleiben stehen.⁵

Wie Kräfte können Impulse in Physik Engines auch ohne zweiten interagierenden Körper über Funktionen an einem Körper angewandt werden.

³ [2] vgl. Game Physics Engine Development S.208-117 stark vereinfacht

⁴ [1] vgl. Game Physics David Eberly S.88 genaue Berechnungsformeln für eine eigene Implementation

⁵ [3] vgl. Game Engine Architecture Kap. 12.4.7.2 - Impulse, Collision Response

3.1.2 Physik Welt Kollisionen und Detektionen

Diese Regeln, die sich durch die Körper ergeben, müssen nun in einer Physics Engine innerhalb einer Welt angewandt werden. Wie bereits beschrieben, kann dies als Raum verstanden werden der eingenommen wird.

Um eingenommene Räume und die wirkenden Energien adäquat darzustellen, benötigt es Konzepte um festzustellen, ob eine Energie angewandt werden muss. Durch die definierten Collider, der Körper, kann mittels Algorithmen festgestellt werden ob sich ein Collider innerhalb eines anderen Colliders befindet, wie tief dieses Eindringen ist und wie die beiden Körper darauf reagieren müssen. Dieser Vorgang nennt sich Kollisionsdetektion. Diese geschieht in diskreten Zeitabständen und iterativ aufeinander. D.h. die letzte Berechnung liegt der nächsten zugrunde.⁶

Diese Berechnungen basieren auf Vereinfachungen und Annahmen über die Form und Schwerpunkte, der Collider.

Jeder gewählte Collider kann repräsentiert werden durch eine Box, die sogenannte Bounding Volume bzw. Axis-Aligned Bounding Boxes (AABB), welche den Collider umschließt, immer in der Grundachse zur Welt und rechteckig.

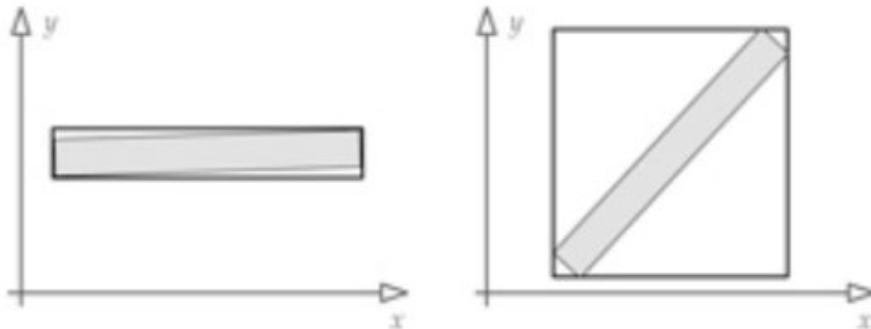


Abb. 3 - AABB's die Körper als Vereinfachung repräsentieren. (Game Engine Architecture Kap. 12.3)

Mehrere Bounding Volumes können zu einer Gruppe, sog. Cluster zusammengefasst werden und jeweils anhand ihrer Nähe zueinander in größere Cluster. Daraus ergibt sich eine Hierarchie an Berechnungen sog. Bounding Volume Hierarchy. Dadurch

⁶ [3] vgl. Game Engine Architecture Kap. 12.4.4

kann eine Kollision möglichst früh ausgeschlossen werden.

Eine einfache Detektion ob z.B. ein Punkt sich innerhalb einer Kugel befindet, ist es den Abstand des Punktes P vom Ursprung U, der Kugel, zu berechnen. Ist dieser größer als der Radius der Kugel befindet er sich außerhalb. Dies ist allerdings nur bei Kugeln zutreffend, da die Hülle immer denselben Abstand zur Mitte besitzt.⁷

Dies macht den Aufwand der Detektion und die Menge an unterstützten Collider-Formen von Physik Engines häufig variabel, denn verschiedene Formen brauchen unterschiedliche Detektionen.

Die Kollision wird in Phasen überprüft um einzuschränken wer beteiligt sein könnte und so Rechenleistung zu sparen.

Es fängt an mit der *Broad Phase*, in der alle Objekte simplifiziert betrachtet und in Cluster sortiert werden (dies kann auf mehrere Arten geschehen hier wird der AABB vs. AABB Test vorgestellt).

Darauf folgt die *Narrow Phase*, in der über genaue Algorithmen die Kollision einzelner Körper bestimmt und Resultate an die Physik Engine geliefert werden.

Es gibt nun zwei wichtige Ansätze die benutzt werden um die meisten Collider-Formen zu detektieren.

AABB trifft auf AABB

Dadurch, dass die Bounding Boxen parallel zu Achsen liegen, können ihre Intersektionen auf jeder Achse überprüft werden, um festzustellen ob die Bounding Boxen kollidieren. Hierfür wird festgestellt ob die Minimal- und Maximalwerte einer Achse sich überschneiden. Ist dies für alle Achsen der Fall wird erst die genauere aufwendige Kollisionserkennung ausgeführt.

⁷ [3] vgl. Game Engine Architecture Kap. 12.3.5.1 Collision Detection

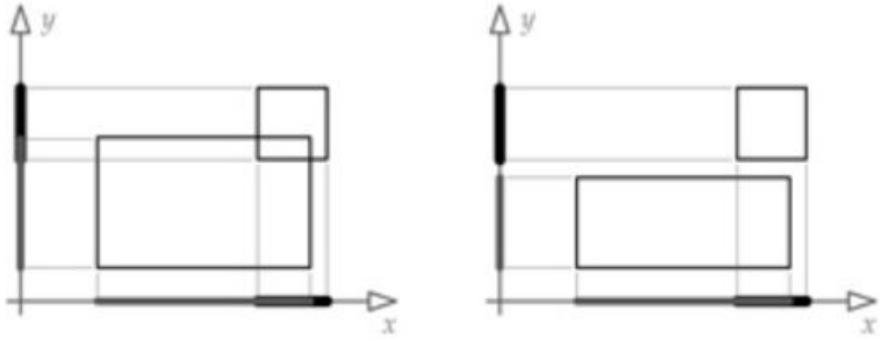


Abb. 4 - AABB vs. AABB Kollisionstest.
Links Kollision, beide Achsen überschneiden sich, rechts dagegen nur eine. (Game Engine Architecture Kap. 12.3)

Convexe Kollision mittels GJK Algorithmus

Der Gilbert-Johnson-S.Keerthi (GJK) -Algorithmus, ist rechenintensiv und wird daher auf möglichst wenige Körper verwendet, deshalb findet diese Detektion in der *Narrow Phase* statt. Er basiert auf dem Vorgang, jeden Punkt innerhalb von Collider A von Collider B paarweise zu subtrahieren, was eine sog. Minkowski Differenz ergibt. Im Falle, dass zwei Körper kollidieren, ungeachtet ihrer Form, muss die Differenz einen Punkt enthalten der den Ursprung der Welt beinhaltet.⁸

Die Minkowski Differenz ist eine AABB für Würfel, aber wird mit der Anzahl der Punkte der Körper zu einer komplexeren Geometrie.

Deshalb wird im GJK Algorithmus ein einfacher Tetrahedron (4 Seitiger Dreiecks Körper) Körper definiert, der in die Minkowski Differenz passt und immer noch den Koordinatenursprung beinhaltet.

Dafür startet der Algorithmus iterativ um ein sog. Simplex zu erweitern. Ein Punkt ist ein Simplex der Stufe 1 und jeder weitere Punkt erhöht die Stufe, bis ein Tetrahedron erreicht ist, die 4. Stufe. Dabei werden Punkte der Hüllen darauf untersucht ob es einen Punkt gibt der den Ursprung umschließt. Gibt es einen wird das Simplex erweitert, wenn nicht gibt es keine Überlappung der Körper.⁹

⁸ [3] vgl. Game Engine Architecture Kap. 12.3.5.5

⁹ [20] Building a Collision Detection for 2D Blog

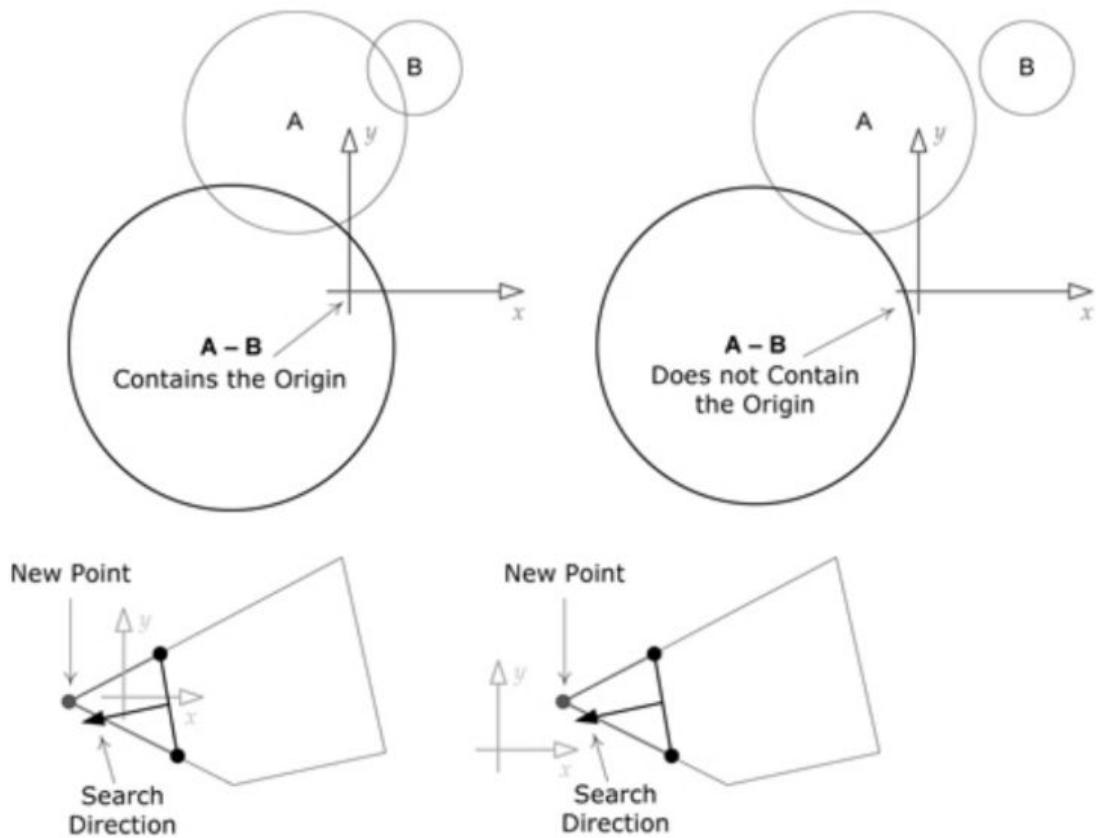


Abb. 5 - GJK Algorithmus Konzepte. Finden der Minkowski Differenz.
Anschließend Support Vertex Suche bis ein Tetrahedron entsteht. (Game Engine Architecture Kap. 12.3.5)

Aus diesen Vorgängen lassen sich der Punkt der Überlappung, sowie Tiefe, Normale uvm. herleiten und entsprechend Körper voneinander abstoßen. Wobei hier nach der Kollisionsdetektion viele weitere Schritte getätigigt werden müssen, z.B. Bestimmung des Kollisionstypus (Vertex-Kollision, Kanten-Kollision), und viele weitere Schritte, die hier sinnvollerweise außer Acht gelassen werden.¹⁰

Der GJK Algorithmus funktioniert nur für konvexe Collider, dafür für jede Form, weswegen er eine beliebte Implementation ist, allerdings die eingangs erwähnte häufige Beschränkung auf diese bedeutet.

Iterativ werden diese Ergebnisse von der Physik Engine verwendet um Änderungen an Körpern vorzunehmen, bis keine penetrierenden Kollisionen innerhalb eines Frames stattfinden, dieser Status wird dann gerendert.

Zur Vereinfachung aller Berechnungen werden Körper in einen Schlaf Status versetzt,

¹⁰ [2] vgl. Game Physics Development - S.294-302 Generating Contacts and Contact Data

d.h. sie erhalten keine Energie, sofern diese zum vorherigen Frame keine Änderungen erfahren die über einer Schwelle liegen und müssen daher nicht neu berechnet werden. Dieser Status kann die Akkuratheit der Berechnungen beeinflussen, durch ungenaue Berechnungsfehler, weil die zuletzt berechneten Daten keine Mikro Änderungen enthalten, daher ist das Sleep/Awake Management ein wichtiger Teil der Arbeit mit Körpern.¹¹

Physikalische Funktionalitäten durch Detektion

Durch Detektion von Collidern ergeben sich viele Funktionalitäten zusätzlich zur Kollision und Bewegung durch kinetische Energie. Die drei Basisfunktionalitäten die daher auch für FUDGE genutzt werden sollen, sind der Raycast, ein Kollisions Event und ein Auslöser Event. Events werden hierbei durch die Integration der Physik Engine ausgelöst, aber die Physik liefert die Werkzeuge um sie auszulösen und mit Informationen zu befüllen.

Der **Raycast** beschreibt einfach dargestellt einen mathematischen Strahl, der keine Breite besitzt, somit eine Linie im Raum darstellt, die exakt von einem Punkt zu einem anderen geht. Dies wird speziell für die Auswahl von Objekten eingesetzt, weshalb diese Funktionalität in fast jedem Spiel verwendet wird, auch wenn für Spieler scheinbar keine Physik verwendet wird.

Ein Strahl ist mathematisch definiert als Start und Endpunkt, wobei jeder Punkt entlang der Linie mit Collidern auf eine Kollision überprüft wird. Zurück geliefert wird die getroffene Stelle des Strahls, die Distanz zum Ursprung des getroffenen Körpers und die Normale. Dieser Vorgang ist auch als Raytracing bekannt, also Verfolgung von Strahlen, wodurch z.B. auch Lichtreflektionen berechnet werden können.

Das **Kollisions Event**, engl. *Collision Event*, ist eine Fortführung der Kollisionsdetektion. Wird eine Kollision erkannt, kann diese von der Game Engine genutzt werden, um ein Event auszulösen, was zur Steuerung von Spiellogik, oder weiteren Physik Vorgängen genutzt werden kann. Es beinhaltet Informationen über Impulse und Ort der Kollision, sowie beteiligte Körper. Diese Funktionalität wird

¹¹ [2] vgl. Game Physics Development - S.422ff

ebenso als Grundfunktion angesehen, so können z.B. Schaden bei Berührung von Spielern mit Schadensquellen, oder bei Aufprall Soundeffekte mit realistischer Lautstärke, umgesetzt werden.

Zuletzt das **Auslöser Event**, engl. *Trigger Event*, ist ein vereinfachtes Überlappungs Event. Sobald zwei Körper überlappen wird es ausgelöst. Da rigide Körper in der realen Physik nicht überlappen können, weil es immer zu einer Kollision kommt, wird dafür entsprechend immer ein Körper benutzt der zwar einen Collider besitzt, aber keine kinetische Energie von anderen Körpern annimmt, d.h. er kollidiert nicht. Dies ist ebenso eine Grundfunktionalität für die meisten Spiel Logiken, bzw. Spawnsysteme. So kann z.B. wenn ein Bereich verlassen wird ein anderer geladen werden um Performanz zu sparen, oder bei Betreten eines Bereichs ein Spielziel vollführt werden, beispielsweise in einem Rennspiel fährt man durch ein Ziel aber kollidiert nicht mit dem Ziel. Trigger sind üblicherweise daher unsichtbar, da es sich eher um ein gedachtes physisches Konstrukt handelt.

All diese Elemente führen zu einer komplexen Physik Engine welche eins der wichtigsten Elemente einer Game Engine darstellt.

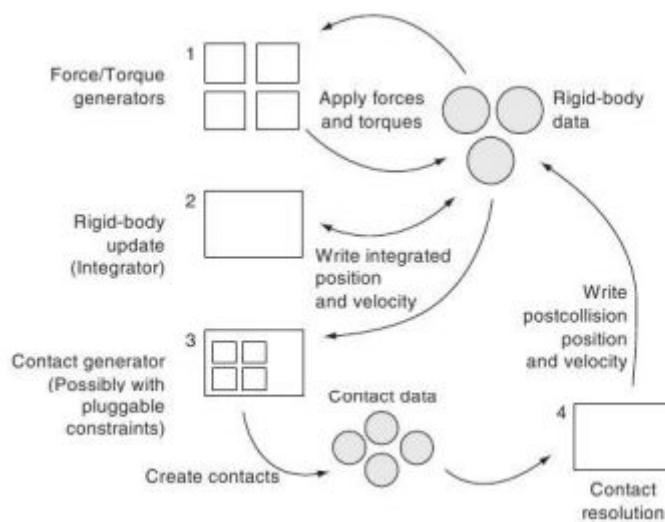


Abb. 6 - Datenfluss Darstellung innerhalb einer Physik Engine
die Game Engine benötigt davon nur die Ergebnisse innerhalb des Rigidbody (Game Physics Development S.365)

3.1.4 Verhaltensweisen und Einschränkungen

Körper können unterschiedlich auf die Physik Berechnungen reagieren, abhängig von ihrem Typ und ihren auferlegten Beschränkungen.

Physik Verhaltens-Typen

Ein Körper der auf Energie reagieren kann wird **dynamisch** genannt. D.h. die Physik übernimmt die volle Kontrolle über das Objekt, alle Transformationen aus der Hierarchie und außerhalb der Physik Engine werden ignoriert. Er reagiert nur auf kinetische Energie.

Der zweite Typ ist der **statische** Körper, er gibt kinetische Energie ab, aber erfährt keine. Es handelt sich hierbei um eine Form des Körpers die nicht direkt einem realen Phänomen entspricht. Verwendet wird er für z.B. Mauern/Böden in Spielen. Er soll kollidieren, aber niemals seinen eingenommenen Raum verändern. Dadurch können Berechnungen gespart werden, da er sich schließlich nicht bewegen wird, auch wenn die realistischen Berechnungen dies ergeben würden. Damit wird in etwa das Verhalten von verankerten Körpern realistisch dargestellt.

Der dritte Typ ist nicht realistisch, aber ein nötiger Teil um die per eigenem Code beschriebene Transformations Welt und die Physik Welt zu verbinden. Der **kinematische** Körper, beschreibt einen Körper der zwar kinetische Energie an dynamische Körper übergeben kann, aber keine erfährt. Diese Eigenschaft hat er mit dem statischen gemeinsam, allerdings kann der kinematische Körper bewegt werden, aber nur über Transformationen, die nicht von der Physik ausgeführt werden. D.h. er kann durch z.B. Animationen direkt um 1 Meter verschoben werden, ohne Beachtung von Masse oder Momentum.¹²

Verhaltens Beschränkungen - Gelenke

Ein Gelenk, engl. *joint / constraint*, ist die Verkettung zweier Körper miteinander. Dadurch werden die Bewegungsmöglichkeiten dieser Körper beeinflusst.

¹² [3] Game Engine Architecture - Physics/Game Driven and fixed Rigidbodies Kap.12.5.1.x

Dies ist eine Grundfunktion, da die Physik Engine für dynamische Körper die Szenengraph-Hierarchie außer Kraft setzt. Also muss eine Verbindung über eine Art mechanische Bauteile geschaffen und ergänzt werden.

Körper besitzen Freiheitsgrade, 6 an der Zahl, wenn sie im Raum schweben. Sie können sich in jede der drei Achsen drehen und in alle drei Richtungen bewegen.¹³

Gelenke schränken dies nun ein, beide verbundenen Körper teilen sich alle Grade miteinander die nicht frei sind und für freie wird ein Verhalten definiert.

Es gibt Freiheitsgrad-Limitierungen sowohl für Translationen, als auch für Rotationen, und diese können meistens gefedert stattfinden. Gelenke verändern das Verhaltensmuster von Körpern und werden durch spezielle Solver berechnet.

Durch sie werden Bedingungen definiert, die eingehalten werden müssen zwischen zwei Körpern, ist dies nicht der Fall werden in jedem Schritt der Simulation Impulse angewandt um die Bedingungen einzuhalten.¹⁴ Daher kann es hier je nach Qualität des Solvers zu Unterschieden zwischen dem gewollten Verhalten und dem tatsächlichen kommen. Man unterscheidet dabei zwischen einem schnellen iterativen Solver, der einzelne Gelenke prüft und einem direkten (globalen) Solver, der alle Gelenke einbezieht, was ihn langsamer aber verlässlicher macht. Gelenke werden über sequentielle Impulse gesteuert, was sie sprunghaft macht und Restitutionen, wie auch Friktionen ignorieren lässt.¹⁵ Ketten von Gelenken und speziell verschiedenen Gelenktypen sind daher häufig instabil und sollten vermieden, bzw. nur mit globalem Solver verwendet werden.¹⁶

Gelenke sind in Physik Engines häufig vordefiniert, für häufige Einsatzgebiete wie z.B. das Schiebegelenk, sog. prismatisches Gelenk. Es besitzt eine Achse, in der sich bewegt werden kann, es hat daher einen Freiheitsgrad.

Er verbindet zwei Körper, der eine steckt im anderen. Und begrenzt die Bewegung des einen Körpers auf die eigene Achse. Dies kann mit einer Feder in Verbindung

¹³ [3] Game Engine Architecture - Kap. 12.4. Separability of Linear and Angular Dynamics / Constraints Kap. 12.4.8

¹⁴ [21] Constraint Rigidbody Simulation Article by Software Engineer Nilson Souto

¹⁵ [22] Modelling And Solving Constraints - Erin Catto Blizzard Entertainment

¹⁶ [3] Game Engine Architecture - Kap. 12.4.8.6 Constraint Chains

stehen um mehr Freiheit zu ermöglichen,¹⁷ oder an beiden Enden begrenzt sein um Limitierungen zu ermöglichen.

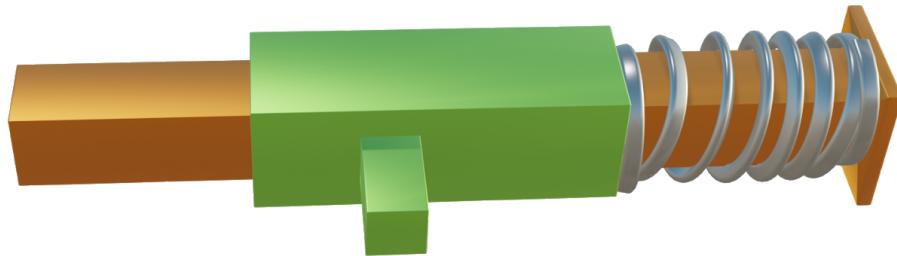


Abb. 7 - Ein einfacher Schieberegler, bei dem der grüne Körper nur der Achse des orangenen Körpers folgen kann.
Eine Feder erlaubt sich gedämpft zu bewegen.

3.2 Anforderungen an eine Physik Engine für Fudge

FUDGE, soll alle beschriebenen Funktionalitäten unterstützen, wird aber nicht von ausgebildeten Entwicklern bzw. in kommerzieller Umgebung genutzt, sondern in der Lehre. Dadurch ergeben sich veränderte Bedingungen, die sich auf die Integration jeglicher neuer Funktionalität auswirken.

Ganz einfach betrachtet sind diese, gute Austauschbarkeit und Transparenz, sowie wichtige Basisfunktionen und Möglichkeiten für die Erkundung von Funktionalitäten. Was entsprechend nicht abgedeckt ist, sind hochkomplexe große Projekte, mit sehr fortgeschrittenen Funktionalitäten und Beachtung einer hohen Performanz.¹⁸

Bei FUDGE geht es weniger um die Ausgefeiltheit von Funktionalitäten im Vergleich zu Konkurrenzprodukten, sondern es geht um die Akteure die sie nutzen. Da der Fokus auf der Lehre liegt, ist auch die Bindung zu FUDGE kein Ziel, sondern eher die Vermittlung von FUDGE zu komplexeren Anwendungen hin.

Die zu betrachtenden Akteure, für diese Arbeit, sind also Lehrkräfte und besonders Lernende. Die besonderen Aspekte die sich dadurch ergeben sind ausschlaggebend

¹⁷ [3] Game Engine Architecture - Kap. 12.4.8.2 Stiff Springs

¹⁸ [5] <https://github.com/JirkaDellOro/FUDGE/wiki/Motivation#what-is-fudge-not>

für den Funktionsumfang und die Integration einer Physik Engine. Diese Punkte wurden erarbeitet durch mehrfache Gespräche mit Prof. Dell'Oro-Friedl und Analyse von bisherigen FUDGE Integrationen/Projekten:¹⁹

1. Ein guter Austausch muss stattfinden können d.h. kleine Dateigrößen, sowie wiederkehrende und konsistente Konzepte zwischen allen Funktionalitäten von FUDGE, einschließlich der Physik.
2. Möglichst wenig Komfortfunktionen integrieren, nur was über Kombination von Basiswissen hinaus geht, Lernfähigkeit steht im Zentrum.
3. Projekte müssen ohne Physik funktionieren können, die Nutzung ist nicht zwingend.
4. Transparenz in der Funktionsweise, nach Möglichkeit, keine Vorgänge die passieren ohne dass die Lernenden involviert sind.
5. Erweiterbarkeit - Da nicht alle Funktionen integriert werden sollen, soll es die Möglichkeit geben für Akteure über die Integration hinaus direkt mit der Physik Engine zu kommunizieren.
6. Konzept Vermittlung steht über der Qualität und Performanz. In normalen Anwendungsfällen sollten die 30 Bilder pro Sekunde nicht unterschritten werden. Realismus sollte soweit gegeben sein, dass reale Anwendungsfälle vermittelt werden können.

Hier schließen sich dann entsprechend auch die Anforderungen an den Funktionsumfang, speziell für die gewählte Physik Engine, an:

1. Festkörperphysik:

- Physikalische Materialien - Reibung und Abprall-Kraft
- Möglichkeit zur Einflussnahme durch Kraft, Impuls, Geschwindigkeit
- Basis-Collider Formen - Würfel, Kugel, Zylinder, Kapsel
- Filterung von Kollisionen

¹⁹ Einzusehen in beiliegenden Thesis-Gesprächsnnotizen und FUDGE Wikieinträgen zu Akteuren und Motivation <https://github.com/JirkaDellOro/FUDGE/wiki/Actors-and-UseCases>

- Abdeckung von statischen, dynamischen und kinematischen Körpertypen (da eine Animations Komponente vorhanden ist und diese kinematische Körper bewegen können soll)
- Einstellbare Zeitabstände der Physikberechnung, Anpassung an Engine Zeitskalierung

2. Raycast:

- Filterbarer Raycast
- Rücklieferung eines custom Objektes, so dass Nodes in FUDGE Form übertragen werden können, anstatt die Rigidbodies der Physik Engine

3. Physikalische Events:

- Kollisions Event: Ort des Aufpralls, Aufprall Stärke, Normalenvektor, Involvierter Festkörper
- Auslöser Event - Involvierter Festkörper, Ort des Auslösens

4. Gelenke:

- Einfache Verbindungsgelenke um Festkörper zu verbinden
- Definierung von Freiheitsgraden
- Möglichkeiten zur Erstellung von simplen physikalischen Zusammenhängen wie Türscharniere, Feder Masse Systemen.

Alle Funktionalitäten darüber hinaus sind Bonus und beeinflussten nicht unbedingt die Wahl der Physik Engine. Es gilt festzustellen, dass lightweights gegenüber komplexen Physik Engines Vorzug erhalten, da so fast alle Funktionalitäten innerhalb dieser Arbeit integriert werden können. Außerdem sind Vorgänge innerhalb dieser Engines häufig klarer und einfacher strukturiert, was der Lehre zugute kommt.

3.3 Auswahl einer Physik Engine

Vorneweg muss gesagt werden, alle Engines machen gewisse Abstriche zu vergleichbaren Engine Angeboten für Desktop bzw. native Anwendungsfälle, dies ist

auf den neuen, wandelbaren Status von webbasierten Techniken, sowie Fokus der kommerziellen Produktionen, zurückzuführen. Alle Engines in der engeren Auswahl sind in die Web Game Engine Babylon.js integriert²⁰ und die Entwicklung wird durch diese maßgeblich vorangetrieben.

Alle in der engeren Auswahl stehenden Engines wurden, in einer für die Arbeit definierten Kommunikationsphase, getestet. D.h. es wird eine einfache Verbindung zwischen einer Szene in FUDGE und einer Szene in der gewählten Physik Engine hergestellt, so dass diese nur die Erstellung der Körper und Position/Rotation miteinander austauschen. Damit wurden eine Machbarkeit der Integration geprüft und Basisprobleme aufgedeckt. Die Kommunikation wird genauer getestet und ist Teil des Auswahlprozesses, siehe Kap. 3.3.5. In der Kommunikationsphase wurden zunächst nur Körper und Kräftesysteme untersucht, da davon ausgegangen wurde, dass sofern diese ordnungsgemäß funktionieren die Qualität von Raycast, Event Detektion und Gelenken ebenso für die Umgebung von FUDGE geeignet sind.

3.3.1 Cannon.js

Cannon.js ist eine lightweight Physics Engine, die von Grund auf in Javascript geschrieben wurde. Geboten werden alle erwarteten Features.

Zudem sind extra Funktionalitäten, wie Cloth, vorgefertigte Vehikel Constraints und Heightfield Collider für Terrain, vorhanden.

Dokumentiert ist die API der Engine mit wenigen Erklärungen, aber offensichtlich allen verfügbaren Funktionen. Es gibt gute Hinweise darauf wie die Physik Engine in eine Render Engine integriert werden kann, sowie 20 Demos in denen Cannon.js in Three.js eingebunden wurde.

Cannon.js ist im Januar 2012 entstanden und das letzte Update geschah im Mai 2016. Zum Zeitpunkt der Auswahl stehen ca. 150 Issues offen.²¹ Dies ist

²⁰ [14] Babylon.js Physics Engine Integration

²¹ [9] Cannon.js - Github, einige Issues sind komplett unbeantwortet und speziell der Haupt-Entwickler meldet sich fast nicht.

entsprechend der größte Kritikpunkt dieser Engine, das hohe Alter und wenig Updates. Zudem deuten viele offene Issues darauf hin, dass es noch einige Probleme zu lösen gibt, die potenziell die Integration in FUDGE gefährden könnten, aber vermutlich nicht vom Entwickler angegangen werden.

Es gibt außerdem einen Typescript Port von Cannon.js und zwar Dàpào.²² Dies ist ein sehr interessantes Projekt, da FUDGE in Typescript geschrieben wird und somit die Nähe für eine Integration exzellent wäre. Allerdings ist das letzte Update von Dàpào bereits 2 Jahre alt und besitzt dieselben Schwächen wie Cannon.js. Zudem wurden auch noch nicht alle Features übertragen, speziell im Bereich Collider-Formen.

Zusammengefasst ist Cannon.js eine gute Wahl, im Bezug auf einfache Anwendung und mögliche Features. Mit einigen Zusatzfeatures, die häufig verwendet werden, z.B. vereinfachte Vehikel Einrichtung. Allerdings sind das Alter und offene Probleme ein nicht unwesentlicher Nachteil.

3.3.2 Ammo.js

Ammo.js ist ein Port der vielfach verwendeten Open Source Bullet C++ Physik Engine. Am bekanntesten dürfte der Einsatz der Bullet Engine in der 3D Modelling Software Blender sein. Der Port wird möglich gemacht durch Emscripten²³ was vereinfacht dargestellt, unter anderem C++ in Javascript bzw. asm.js, eine strikte Subvariante von Javascript umwandelt, ebenso ist direktes Webassembly möglich. Bullet Physics ist keine lightweight Engine, sondern eine komplette Physik Software Lösung, die unter anderem auch Soft Bodies beinhaltet.

Dieser Umstand macht den Port in Form von Ammo.js deutlich komplexer gegenüber anderen, allerdings auch stabiler und erweiterbarer.

²² [11] Dàpào - <https://github.com/TheRohans/dapao>

²³ vgl. <https://github.com/emscripten-core/emscripten>

Ammo.js ist sowohl in Babylon.js als auch in PlayCanvas eingebaut und somit auch weit verbreitet im Einsatz. Das letzte Update vor der Auswahl geschah im März 2020 und der Port wurde im Mai 2011 gestartet. Es stehen ca. 100 Issues offen.

Schwachpunkt dieser Engine ist die Dokumentation und Bedienbarkeit, außerhalb einer Integration, sowie die Integrationsschwierigkeit. Ammo.js besitzt keine eigene Dokumentation, sondern verweist auf die entsprechende Bullet Physics API,²⁴ auf Basis von C++. Zudem sind nicht alle Funktionalitäten (komplett) portiert, dies erfordert einen hohen Testaufwand. Ammo.js muss aufgrund seiner Struktur, vor der Verwendung von Funktionalitäten, als Javascript-Promise²⁵ geladen werden. Dies würde einen starken Eingriff in Strukturen von FUDGE bedeuten. Die Möglichkeit des Webassembly erlaubt einen Performancegewinn zur Laufzeit, aber bedeutet mehr initiale Ladezeit.

Zusammengefasst ist Ammo.js die potenteste Wahl im Funktionsumfang, viel verbreitet und oft aktualisiert. Die Schwächen liegen in dem Integrationsaufwand und den zu überladenen internen Strukturen, sowie der Interaktion der Akteure mit der Engine, falls sie auf das Limit der in der Arbeit ermöglichten Integration treffen und sich mit der Engine befassen müssten.

3.3.3 Oimo

Oimo kommt in zwei Varianten daher. Die originale in Haxe geschriebene Physik Engine wurde ursprünglich in Actionscript geschrieben und genutzt, ist eine langjährige, übersichtliche, lightweight Physik Engine. In einem frühen Stadium hat sich ein Javascript Port davon abgespalten. Seit die Engine in Haxe umgeschrieben wurde, ist das Original ebenso als Javascript Build verfügbar.²⁶ Daher gibt es hier zwei Optionen zu betrachten.

²⁴ vgl. <https://pybullet.org/Bullet/BulletFull/index.html>

²⁵ d.h. sämtlicher Code muss umklammert sein von Ammo.onLoad()

²⁶ Haxe als High Level Cross Platform Programmiersprache ermöglicht entsprechende Builds und Ausführung im Browser.

3.3.3.1 Oimo.js

Der Port ist im Oktober 2013 entstanden und wurde Januar 2019 zuletzt mit einem Update versehen. Allerdings verweist der Entwickler, wenn auf Updates angesprochen, inzwischen direkt auf die nun in Javascript verfügbare originale Variante, in ca. 30 offenen Issues zum Zeitpunkt der Auswahl.²⁷

Diese Engine wurde vor der originalen Oimo Engine betrachtet, aufgrund der vorangegangenen Bachelorarbeit. Trotz Verwandtschaft arbeiten beide etwas unterschiedlich und haben teils unterschiedliche Funktionsumfänge, sie eint, eine einfache, gute Struktur und ihr lightweight Status.

Oimo.js bietet was Collider-Formen betrifft neben den Grundformen, Box, Cylinder, Sphere, auch ein Tetrahedron. Zudem ist kein eigener Raycast vorhanden. Oimo.js ist in Three.js und Babylon.js eingebunden und bekommt in der jeweiligen Integration Funktionalitäten durch deren Integration, wie z.B. Raycast oder Terrain Collider. Während die eigenständige Version dies nicht besitzt.²⁸

Dadurch ist der Funktionsumfang trotz guter Struktur eigentlich zu gering.

3.3.3.2 Oimo Physics

Oimo Physics entstand ursprünglich im August 2012 und wurde zuletzt Januar 2018 aktualisiert. Es gibt jedoch einen Branch, der im März ein Update erfahren hat, auf dem auch Typescript Definitionen für alle Funktionen vorhanden sind. Im Gegensatz zu Oimo.js gibt es mehr vorgefertigte Gelenke, was speziell im FUDGE Nutzungskontext der Lehre Sinn ergibt, um Konzepte langsam zu etablieren.

Alle erwarteten Funktionen sind enthalten, zusätzlich dazu ein konvexer Collider, der über Eingabe von einem Array von Punkten im 3D Raum, versucht eine geschlossene Form zu erstellen.

Das besondere an dieser Engine ist die gute Dokumentation, mit klaren Strukturen und einer Schnittstelle, in der Debugging Informationen von der Physik Engine an eine Render Klasse gesendet werden können, wodurch nur noch der Prozess des

²⁷ vgl. <https://github.com/lo-th/Oimo.js/issues/87>

²⁸ https://github.com/lo-th/Oimo.js/blob/gh-pages/examples/test_terrain.html, es gibt innerhalb von der Source <https://github.com/lo-th/Oimo.js/tree/gh-pages/src>, einige Elemente nicht, aber innerhalb von Beispielen mit Three.js sind sie vorhanden, daher müssen sie wohl Teil der Integration sein.

Zeichnens implementiert werden muss um Funktionalitäten wie Gelenke, Collider, Raycasts visuell zu debuggen.²⁹

Schwächen nach der ersten Kommunikationsphase zeigen sich darin, dass Oimo Physics performanter ist als Oimo.js, aber leichte Schwächen in der realistischen Darstellung besitzt.

Zusammengefasst, ist Oimo Physics die bessere und häufiger geupdate Variante, die trotz lightweight Struktur alle definierten Features für FUDGE bereitstellt und zusätzlich noch mit einer Schnittstelle für visuelles Debugging glänzt, was besonders im FUDGE Kontext interessant ist. Allerdings wirkten die Berechnungen, in den mitgelieferten Demos, etwas weniger genau als bei den anderen.

3.3.4 Weitere Optionen

Es gab weitere Optionen, die aber bereits vor der Phase der Kommunikation nicht weiter betrachtet wurden, da sie den momentanen Ansprüchen von FUDGE nicht entsprechen, oder zu spät gefunden wurden, sie werden dennoch kurz vorgestellt, da sie zu einem späteren Zeitpunkt für FUDGE interessant werden könnten.

*PhysX for Javascript:*³⁰

PhysX wurde erst 2018 als Open Source Lizenz zugänglich gemacht, deshalb wurden die ersten Javascript Ports entwickelt. PhysX ist in der Industrie die am weitverbreitetste Physik Engine, neben Havok.³¹ Es sind daher alle Funktionen vorhanden, in sehr guter Ausprägung.

Wie auch Bullet handelt es sich hier um einen Port via Emscripten, von C++ auf Javascript und der Port ist daher weder vollständig, noch besonders performant und getestet.

Diese Engine stellt an sich die perfekte Wahl dar, aufgrund ihrer Qualität und

²⁹ [7] Oimo Physics - <https://github.com/saharan/OimoPhysics>

³⁰ [13] Physx-JS | Port der Nvidia Physx Physik Engine, <https://github.com/ashconnell/physx-js>

³¹ vgl. <https://blogs.nvidia.com/blog/2018/12/03/physx-high-fidelity-open-source/>

verbreiteten Anwendung, was den Einstieg und Übergang in andere Engines für FUDGE Akteure erleichtert. Dennoch fiel dieser Port raus, weil er relativ neu (Januar 2020) und daher schwierig einzuschätzen ist. Es fallen dieselben Nachteile ins Gewicht die auch Ammo.js besitzt, nur gravierender.

Zum Zeitpunkt der Auswahl war dies also keine Alternative, allerdings handelt es sich um ein sehr aktives Projekt und könnte bald den Status von Ammo.js erreichen, mit dem Vorteil, dass Nvidia PhysX an sich verbreiteter und moderner ist als Bullet.

Energy.js:

Eine lightweight Physik Engine geschrieben für die Nutzung mittels Typescript. Allerdings hat diese Engine eine sehr enge Verbundenheit mit Babylon.js, soweit, dass sie ohne diese Engine nicht nutzbar zu sein scheint.³² Auch innerhalb Babylon.js scheint es eine selten genutzte Physik Engine zu sein, da die interne Konkurrenz scheinbar besser ist. Was sich daran zeigt, dass das Projekt auch seit 2 Jahren kein Update mehr bekommen hat, und wenig Interesse besteht.³³

Es wurde trotzdem betrachtet, weil es eine Option der vorangegangenen Arbeit zur Auswahl einer Physik Engine war und der Ansatz einer Typescript lightweight Physik Engine interessant ist. Entsprechend wurden nur Schlüsse daraus gezogen, welche Features als Lightweight betrachtet werden.

***Goblin.js:*³⁴**

Ist eine Javascript Physik Engine die von Grund auf in Javascript geschrieben wurde, mit den meisten nötigen Funktionen die für FUDGE festgelegt wurden und offenbar, innerhalb einer Demo, sogar Mesh Collidern. Allerdings liegt das letzte Update 5 Jahre zurück und hat damit ähnliche Nachteile wie Cannon.js, dazu kommt eine geringe Verbreitung.

Der große Vorteil hierbei wäre gewesen, sehr stabile Berechnungen, das Versprechen Mobil und auf Desktop ähnlich performant zu sein, eine gute Dokumentation und

³² vgl. <https://github.com/samuelgirardin/Energy.js/blob/master/energy.ts>

³³ Wenig Interaktion mit dem Entwickler.

<https://www.html5gamedevs.com/topic/36691-energyjs-playground/>

³⁴ [12] Goblin Physics | <https://github.com/chandlerprall/GoblinPhysics>

Integrationsmöglichkeiten. Allerdings sind Raycasts kein Bestandteil der Engine und hätten integriert werden müssen. Zusätzlich zum Alter der Engine, wären dies dann etwas zu viele negative Punkte gewesen, da diese Engine auch erst spät im Auswahlprozess entdeckt wurde.

3.3.5 Tests und finale Auswahl

Wie beschrieben, wurden die 3, (eigentlich 4, da Oimo Physics und Oimo.js einzeln getestet wurden), in einer ersten Phase mit FUDGE verbunden für einfache Tests.

Diese Machbarkeits Tests ergaben, dass sich die Engines bei den für FUDGE definierten Funktionalitäten, ähnlich gut verhalten und eher in ihren erweiterten Faktoren, wie Integrierbarkeit, Nutzung und Performanz, Unterschiede aufweisen.

Daher wurden in ihrer Komplexität steigende Tests unternommen.

1. Einzelne Körper unterschiedlichen Typs (Statisch, Dynamisch, Kinematisch) kollidieren miteinander, Würfel oder Kugeln und fallen auf einen Boden. Es wird festgestellt ob sich alle Körper so verhalten wie erwartet und alle Typen vorhanden sind.

Ergebnis: Alle 4 Engines erfüllen die Verhaltensweisen verschiedener Typen und Grundformen. Heraussticht, dass unterschiedliche Verhaltensweisen auftreten, da Parameter wie Luftwiderstand (Linear-/Angular- Damping/Drag) und Restitution, sowie Reibung, auf unterschiedliche Standardwerte gesetzt sind, im Vergleich miteinander und z.B. Unity.³⁵

Es gibt keine grundsätzlichen Probleme, einzig das Fehlen von Quaternionen innerhalb der FUDGE Engine bereitet Probleme, da zu diesem Zeitpunkt die eigene Umrechnung von Quaternionen zu Eulerwinkel, teilweise fehlerhaft war, gab es rein visuelle Probleme, in Form von entgegengesetzt und falsch gedrehten Körpern.

³⁵ Unity Vergleichswerte Drag/Linear Damping = 0, Angular Drag = 0.05. Interessanterweise sind Restitution und Friction in 2D und 3D unterschiedlich und standard auf "Mittelwert" gesetzt, d.h. die Friction und Restitution werden gemittelt zwischen zwei Körpern. Unity bietet die Möglichkeit den Wert der Verrechnung zu ändern, die Javascript Physik Engines nicht.

<https://answers.unity.com/questions/656590/none-physics-material-properties.html>

2. Eine Mauer aus 100 Würfeln (10 Reihen á 10 Würfeln), fällt aus 5 Metern Höhe, es wird beobachtet ob das Verhalten realistisch wirkt und die Bildrate in einem mäßigen Rahmen bleibt. Der Realismusgrad des Verhaltens wird gemessen, an einem Test innerhalb von Unity (PhysX Engine).

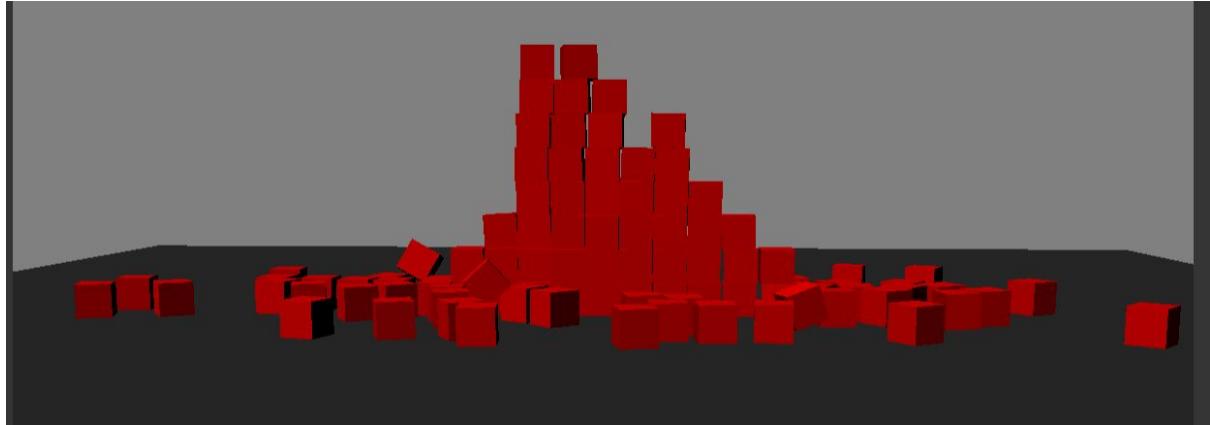


Abb. 8 - Oimo Physics - Fallende Mauer aus einzelnen Objekten Test (100 Rigidbodies). Realistisches Verhalten, einzelne Steine bleiben stehen, einige verteilen sich.

Ergebnis: Auch hier war das Ergebnis, dass die Engines in ihrem Verhalten sehr nahe beieinander liegen. Der Unterschied zeigte sich zum einen darin, dass wenn die Mauer direkt bei Programmstart fiel, starke Einbrüche bei der Bildrate von Ammo.js zu verzeichnen waren, weshalb in einem zweiten Durchgang, eine Zeitverzögerung eingebaut wurde, so dass erst einmal alles geladen wurde, vor Simulationsbeginn. Es wurde zudem die Wiederholbarkeit des Versuchs beobachtet und festgestellt, dass alle nicht jedes Mal gleich zusammenstürzen. Cannon.js hat minimal stärkere Ausreißer und Ammo.js ist erwartungsgemäß am stabilsten, bei leicht geringerer Performanz. Die Mauer stürzt teilweise komplett ein, gelegentlich bleiben einige Würfel gestapelt. (Bis zu diesem Zeitpunkt wurde auch Oimo.js noch mitgetestet, neben Oimo Physics)

Alle Tests fanden mit gleichen Bedingungen statt, bis auf interne Standardwerte für den Algorithmus und die Präzision der Berechnungen.

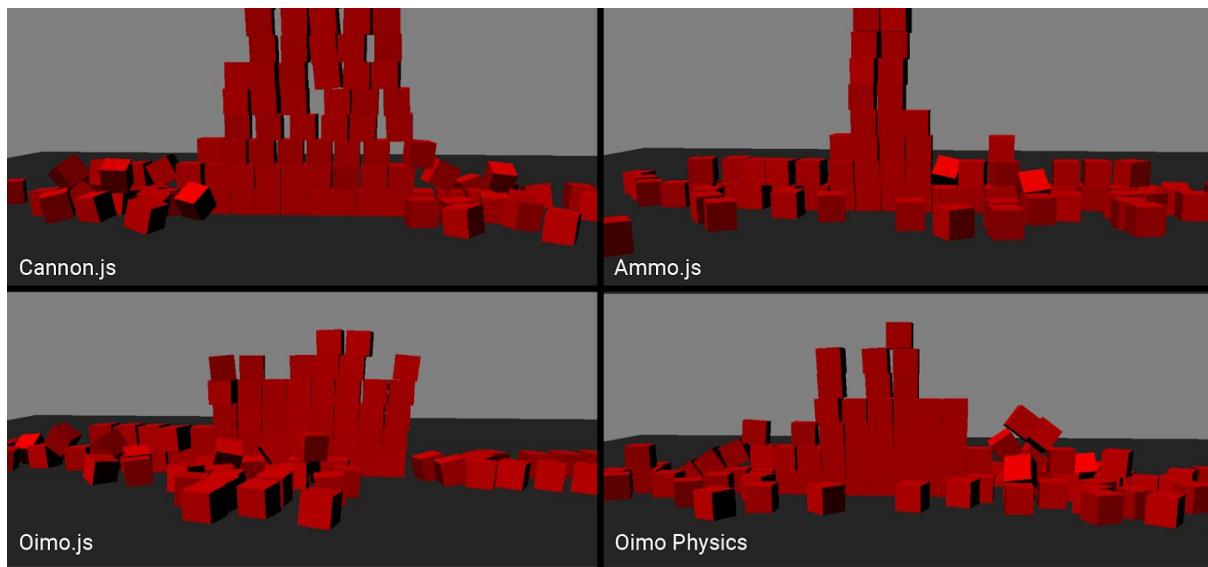


Abb. 9 - Mauer Ergebnisse aller getesteter Physik Engines.
Performanz schwankt zwischen 40-55 Bilder pro Sekunde, während dem Fall.
Auf einem Gaminglaptop als Testgerät. Konstante 60, ca. 3 Sekunden nach Bodenkontakt.

3. In einem Zeitabstand fällt ein zufälliger Körper, Würfel oder Kugel, auf den Boden und verbleibt dort, es wird konstant (100 ms Abstand) ein neuer Körper generiert, der Test wird abgebrochen sobald weniger als 12 Bilder pro Sekunde erreicht werden.

Ziel des Tests ist es, die maximale Anzahl an Körpern und Fehlverhalten bei höherer Anzahl zu testen, außerdem Limits für Anwender-Bereiche zu finden (Physik soll in FUDGE auch auf weniger leistungsstarken Geräten verwendet werden).

Dieser Test basiert auf einem 2014 durchgeföhrten Test für webbasierte Physik Engines. Bei dem hauptsächlich Cannon.js und Ammo.js auf ihre Performanz und Funktionalitäten getestet wurden. Dieser Test kam zu dem Ergebnis, dass Cannon.js performanter ist als Ammo.js, allerdings ist Ammo.js akkurate und mehr Funktionen sind verfügbar.³⁶ In diesem eigenen Test, mit vergleichbarem Versuchsaufbau, wird nun Oimo Physics, welches im Original Test nicht beachtet wurde, ebenso verglichen.

³⁶ [4] Comparison of Physics Frameworks for WebGL-Based Game Engine - Resa Yogya / Raymond Kosala - February 2014 The European Physical Journal Conferences 68

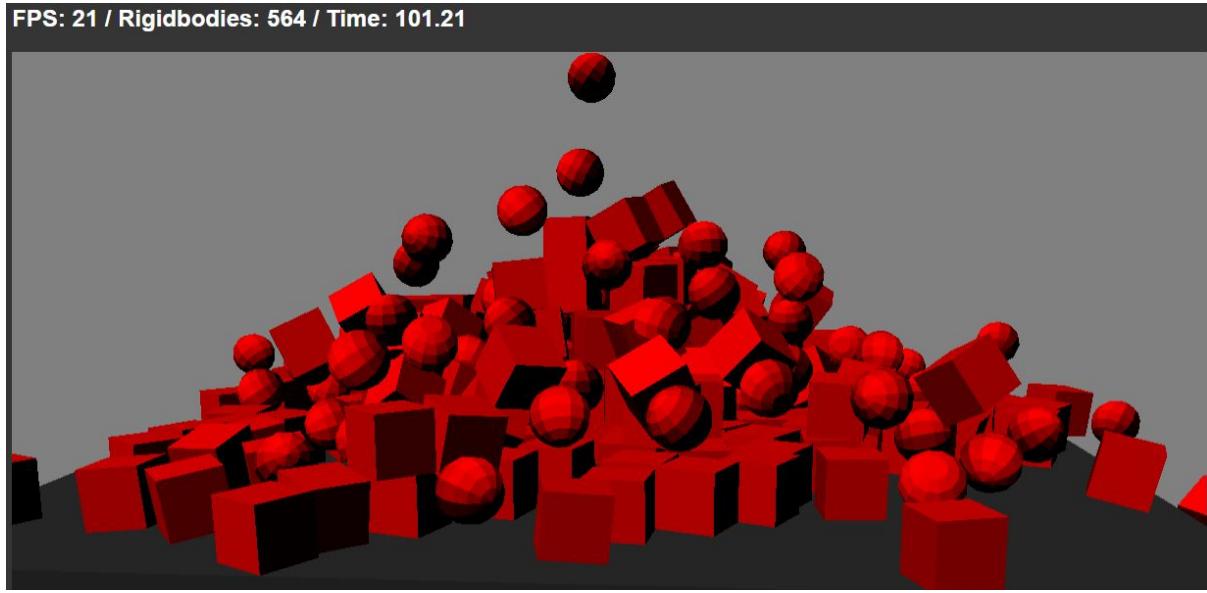


Abb. 10 - Versuchsaufbau - Limit Tests, zufälliger Körper einer Standardform wird zur Szene hinzugefügt, alle vorherigen bleiben bestehen. Bis 12 Bilder pro Sekunde unterschritten werden.

Ergebnis:

Testgerät	Ammo.js	Cannon.js	Oimo Physics
"Low-End" (i5 - 3210m, 2.5 ghz, 6 GB Ram, Nvidia 630m)	486 Körper, 78 Sekunden im Test Kein Fehlverhalten	390 Körper, 63 Sekunden im Test Kein Fehlverhalten	400 Körper, 60 Sekunden im Test Explosionsartige Induktion von Energie
"High-End" (i7 - 7700k, 4.2 ghz, 16 GB Ram, Nvidia 1060 Gtx)	856 Körper, 138 Sekunden im Test Kein Fehlverhalten	840 Körper, 174 Sekunden im Test Kein Fehlverhalten	924 Körper, 142 Sekunden im Test Induktion von Energie

Aus der Tabelle lässt sich Folgendes schließen: Die Körper und die Zeit kombiniert, zeigen wie schnell die Performance einbricht. Da die Anzahl neuer Körper langsam einbricht, sobald die Bildrate sinkt, weil der Browser an sich langsamer wird und die 100 ms zwischen der Generierung nicht mehr einhalten kann. Es zeigt sich also, dass Oimo Physics länger konstant hohe Performanz besitzt, dann aber sprunghaft abfällt. Während Ammo.js und Cannon.js beide kontinuierlich abfallen, Cannon.js jedoch früher nachlässt und zum Schluss sehr langsam neue Körper generiert.

Die Besonderheit hierbei ist, dass Oimo Physics ab ca. 200 Körpern (interessanterweise je nach Testgerät, nur +- 20 Körper), anfängt Kräfte in die Simulation zu injizieren die nicht korrekt sind, alle Körper werden wieder vom

Zentrum aus explosionsartig mit Kräften versehen. Dieses Verhalten steigert sich mit der Anzahl der Körper. Das Phänomen ließ sich verringern, fast ausschließen, indem anstatt die Berechnungszeit des momentanen Bildes (Frametime von FUDGE), eine feste Schrittweite (z.B. 60 Bilder pro Sekunde \sim 17 ms), als Schrittweite für die Physiksimulation angewandt wurde.

Interessanterweise ist Ammo.js performanter auf leistungsschwachen Geräten, aber wird von Oimo Physics weit überholt auf modernen Geräten.

Auswahl:

Schlussendlich entschied man sich gemeinsam für Oimo Physics. Oimo Physics besitzt zwar Fehlverhalten bei großen Anzahlen von Körpern, allerdings wurde gemeinsam mit Prof. Dell’Oro-Friedl entschieden, dass innerhalb von FUDGE selten so viele Körper zum Einsatz kommen, als dass die Induktion von Energie ein Problem darstellt, da die anderen Vorteile von Oimo Physics überwiegen. Cannon.js scheidet aus aufgrund des Alters, bei vergleichbarem Funktionsumfang mit Oimo.

Vorteile sind speziell die einfache Handhabung, eine mitgelieferte Typescript Definitionsdatei, gute Integrationsmöglichkeiten, durch ordentliche Dokumentation und lightweight Status. Mit Bonus einer integrierten visuellen Debugging Schnittstelle. Aufgrund diverser Tests und Auswahlkriterien setzt sich für diese Arbeit Oimo Physics durch. Im Hinblick auf in Zukunft gewollten Funktionsumfang und Genauigkeit wäre Ammo.js die bessere Wahl, dafür aber mit weniger Transparenz und internem Debugging.

4. Entwicklung der Integration

4.1 Physik Engine Integration bei Vergleichsobjekten

Da die mit FUDGE lernenden Akteure letztendlich wahrscheinlich mit anderen Engines arbeiten werden, ist es von Interesse die Integration, mit gewissen Anpassungen an die Anforderungen von Lernenden, möglichst nahe an verbreitete Engines anzulehnen. Daher werden Integrationen in Javascript Game Engines betrachtet, wie Babylon.js und PlayCanvas, aber auch die Integration in die größten

Vertreter, im Sektor für frei zugängliche Engines, Unity und Unreal Engine. Dabei liegt der Fokus auf der Integration der nötigen Features, die auch in FUDGE integriert werden sollen, aber es wird auch auf die Art der Integration, im Bezug auf die Akteure die, die Engines nutzen, eingegangen. Es wird ebenso festgestellt welche Physik Engines benutzt werden und weshalb.

4.1.1 Unity

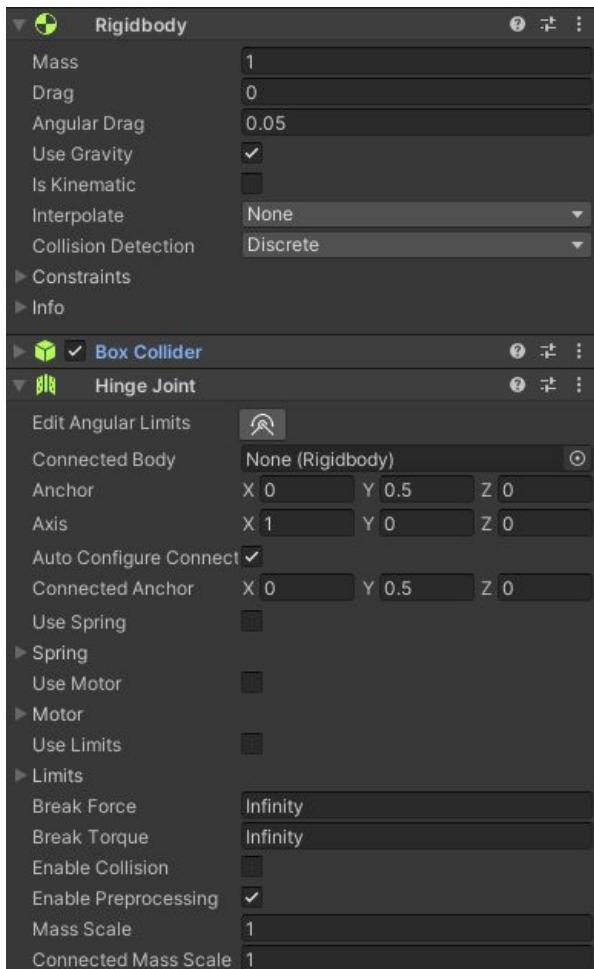


Abb. 11 - Physik Elemente in Unity, deren bestimmte Eigenschaften und Darstellung

Da dieser Arbeit einige Erfahrung in der Spieleentwicklung, mittels Unity, vorausgeht, werden einige Entscheidungen getroffen die auf diesen erlernten Strukturen basieren. Dies deckt sich damit, dass auch FUDGE mit einigen Strukturen die an Unity erinnern, entwickelt wurde. Daher ist für die Physik Integration Unity der initiale Status Quo.

Unity nutzt für die Physik Berechnungen derzeit PhysX von Nvidia, arbeitet aber auch an einer Integration der Havok Physik Engine. In Unity wird die Physik mittels Komponenten an Objekte in der Hierarchie geheftet, FUDGE agiert analog. Dabei wird grundlegend vom Nutzer nur verlangt einen Collider, also eine Form des Objektes, anzuhängen,

damit er der Physik Welt bekannt ist. Dies erlaubt nur Veränderungen der Form des Körpers. Um Objekte dann offiziell als Körper zu definieren benötigt es eine zweite

Komponente, den Rigidbody. Hier finden sich Optionen über das physikalische Verhalten für physisches Material, Masse und Luftwiderstände.³⁷

Hat ein Objekt nur einen Collider aber keinen Rigidbody, kann es zwar mit Objekten/Körpern kollidieren, löst aber keine Kollisions-Events aus. Man kann daher davon ausgehen, dass in Unity trotzdem im Hintergrund eine Art Rigidbody existiert, aber nicht als sichtbare Komponente, die beeinflusst werden kann.

Es gibt die Möglichkeiten der drei Physik Arten, statisch, dynamisch und kinematisch. Sie verhalten sich entsprechend den Erwartungen aus Kapitel 3.

Es lassen sich grundlegend alle Parameter einstellen die das physische Verhalten beeinflussen und es kann jede Form von Collider-Formen gewählt werden. Ungünstig für die Lehre kann jedoch die Trennung von Form und Physik-Objekt sein. Da Einstellungen, die für den gesamten Körper relevant sind, jeweils beim einen oder anderen zu finden sind.

Für Raycasts wird der Anfang, die Richtung und die Länge sowie optional eine LayerMask (für Filterung) angegeben. Und erhält als Rückgabewert ein ganzes Informationsobjekt.

Events sind vorprogrammiert, so dass Nutzer nur eine Funktion wie *OnCollisionEnter(other : collision)* schreiben müssen die dann automatisch vom Physik System aufgerufen wird, ohne dass der Nutzer sich darum kümmern muss, jeden Körper darauf zu prüfen ob er mit einem anderen kollidiert. Es gibt Kollisions-Events und Trigger-Events, zweitere werden durch Körper ausgelöst, die als Trigger markiert sind und dann nicht mehr mit anderen Körpern kollidieren.

Gelenke sind eine Komponente, die an einem Körper hängen und ein anderer wird als Parameter (verbundener Körper) gesetzt. Hierbei gibt es einen "ConfigurableJoint", welcher ein freies Gelenk darstellt welches so verändert werden kann, dass es gängigen Gelenktypen entspricht. Allerdings gibt es mit z.B. "SpringJoint" und "HingeJoint" Gelenktypen die bereits spezialisiert sind, für häufige Anwendungsfälle

³⁷ [15] vgl. Unity Engine Version 2019.3

und einen einfachen Einstieg. Gelenke bestehen aus Motoren und Limit-Einstellungen, sowie Achsen-Einstellungen.

Aus diesem Aufbau kann man den Schluss ziehen, dass es für Nutzer praktisch sein kann, Collider und Rigidbody zusammenzuziehen, da so Einstellungen zentral sind und in der Physik-Welt beide Komponenten nur gemeinsam auftreten. Ebenso sollten Raycasts so integriert sein, dass sie mehr als nur das getroffene Objekt zurückliefern. FUDGE sollte deshalb auch Informationen wie Distanz, Impuls und Normalen liefern.

Davon beeinflusst sollten auch Kollisions/Auslöser-Events vorprogrammiert sein, da diese auszulösen ein trivialer Schritt ist, der von Nutzern zwar machbar ist, ihnen aber nicht viel Lerninhalt bietet und nicht so viele Informationen über das Event liefern kann wie ein automatisch ausgelöstes Event, das von der Physik Integration befüllt wird, dem sie nur eine Listener-Funktion hinzufügen müssen.

Für Gelenke scheint Unity einen guten Mittelweg zu gehen, der gut für Lernende geeignet ist, in dem es bereits vorgefertigte Gelenke mit einem Zweck gibt, also einen erklärenden Charakter mit sich bringt, bzw. häufige Anwendungsfälle vereinfacht, aber auch ein generisches Gelenk um alle Möglichkeiten in einer Komponente zu haben.

4.1.2 Babylon.js

Babylon.js ist eine weit verbreitete WebGL Engine, die wohl bisher Fortgeschrittenste (Neben Three.js). Entwickelt von zwei Microsoft Mitarbeitern, integriert sie vier Physik Engines (Cannon.js/Ammo.js/Oimo.js/Energy.js), mehr oder weniger vollständig und identisch abrufbar, mit eigenen Einflüssen.

Hier zeigt sich wieder der unterschiedliche Fokus im Vergleich zu FUDGE, Nutzer müssen sich erst einmal für eine entscheiden und anschließend diese richtig ansprechen. Hierbei setzt Babylon.js auf verschiedene entwickelte Plugins für jede Engine, die entsprechend als Wrapper fungieren, so dass die Engines möglichst identisch angesprochen werden können, aber durch sog. *nativeParameter* Engine

spezifische Werte übergeben werden. Die unterschiedlichen Funktionsumfänge müssen jedoch beachtet werden.³⁸

Untypisch scheint hier die Bezeichnung von Collidern als Imposter, was in der Physik Programmierung durchaus mehrfach auffindbar ist, aber nicht in die Strukturen der Vergleichsobjekte passt. Bei dem Imposter handelt sich um eine Kombination aus Rigidbody und Collider. Neben diesem Unterschied scheint die Integration aber nahe anderer Vergleichsobjekte zu sein.

Einem Objekt wird ein Collider/Rigidbody hinzugefügt und dieser wird beeinflusst. Für Gelenke wird einem Objekt ein Joint hinzugefügt, der ein anderes mit ihm verbindet. Es gibt also ein Träger Objekt für das Gelenk, das Gelenk ist somit auch eine eigenständige Komponente.

Events gibt es keine vorgefertigten, wie bei Unity, der Nutzer muss selbst überprüfen ob ein Objekt mit einem anderen eine Kollision auslöst und entsprechend eine Funktion aufrufen.

Babylon.js bietet keinen Editor, nur einen sogenannten Playground, der den entwickelten Code direkt anzeigt, weshalb hier keine Schlüsse für die Optimierung der Nutzerinteraktion für den FUDGE Editor erschlossen werden können.³⁹

Es ergibt sich daher für die Integration einer Physik Engine in FUDGE, dass es zur Vereinfachung valide ist eine Kombination aus Rigidbody und Collider zu verwenden, da diese in anderen Vergleichsobjekten nur visuell getrennt sind. Ansonsten sollten die einzelnen Features ähnlich integriert werden wie in Unity.

4.1.3 Unreal Engine

Der zweite Primus im Bereich frei zugängliche Game Engines, Unreal Engine, nutzt auch die PhysX Engine von Nvidia, entsprechend sind auch alle Features und mehr

³⁸ [14] vgl. Babylon.js - Physik Integration

³⁹ [14] vgl. <https://playground.babylonjs.com/> Code kann erstellt und abgespielt werden, aber keine Editor ähnliche Interaktion

vorhanden (wird zukünftig teilweise bzw. ganz abgelöst durch Chaos Destruction Physics).

Die Integration sieht hierbei eine Darstellung in Form von Physik-Tab und Collider vor. Jeder Actor (pendant zum Unity GameObject bzw. Objekt in einem Szenen-Baum allg.), besitzt automatisch eine Physik, die Einstellungen über Material und Eigenschaften in der Physik-Welt ermöglicht. Und ein Collider Objekt mit einer festgelegten Form. Hier, wie auch bei Unity, sind alle Formen und Meshes möglich.

Raycast und Events funktionieren, ähnlich zu Unity, über Code, aber auch als Blueprint (Visual Code). D.h. sie sind vorgefertigt und liefern viele Informationen über den getroffenen Körper und den Treffer ansich. Der einzige kleine Unterschied, Raycasts lassen sich, neben Start und Richtung, auch über Start-/Endpunkt definieren und wiederum mit einer anderen Funktion sind auch gefilterte Raycasts möglich. Hier sind häufig viele Funktionen vorhanden, mit leicht unterschiedlichen Namen und Ansprechweise. Eine durchgehende Eigenheit ist, dass Unreal Engine Elemente anders benannt, als Vergleichsobjekte. Was dafür spricht, dass die Benennung nicht zwangsläufig einheitlich sein muss, aber innerhalb FUDGE konsistent.

Der große Unterschied liegt hierbei bei den Gelenken. Es gibt im Vergleich zu Unity nur ein Gelenk, den *PhysicsConstraintActor* welches in etwa dem *ConfigurableJoint* ähnelt, indem er so konfiguriert werden kann, dass jedes Gelenk abgebildet werden kann.⁴⁰

Zusammenfassend kann man sagen, dass die Strukturen aus Unity sich verstärken. Einige Dinge werden auch hier abgenommen, z.B. die Erstellung einer Physik-Welt, oder das Handling von Kinematischen Körpern versus statischen Physik-Objekten. Dies sind allerdings Vereinfachungen, die das Verständnis von Lernenden beeinträchtigen können.

⁴⁰ [16] vgl. Unreal Engine, Version 4.25

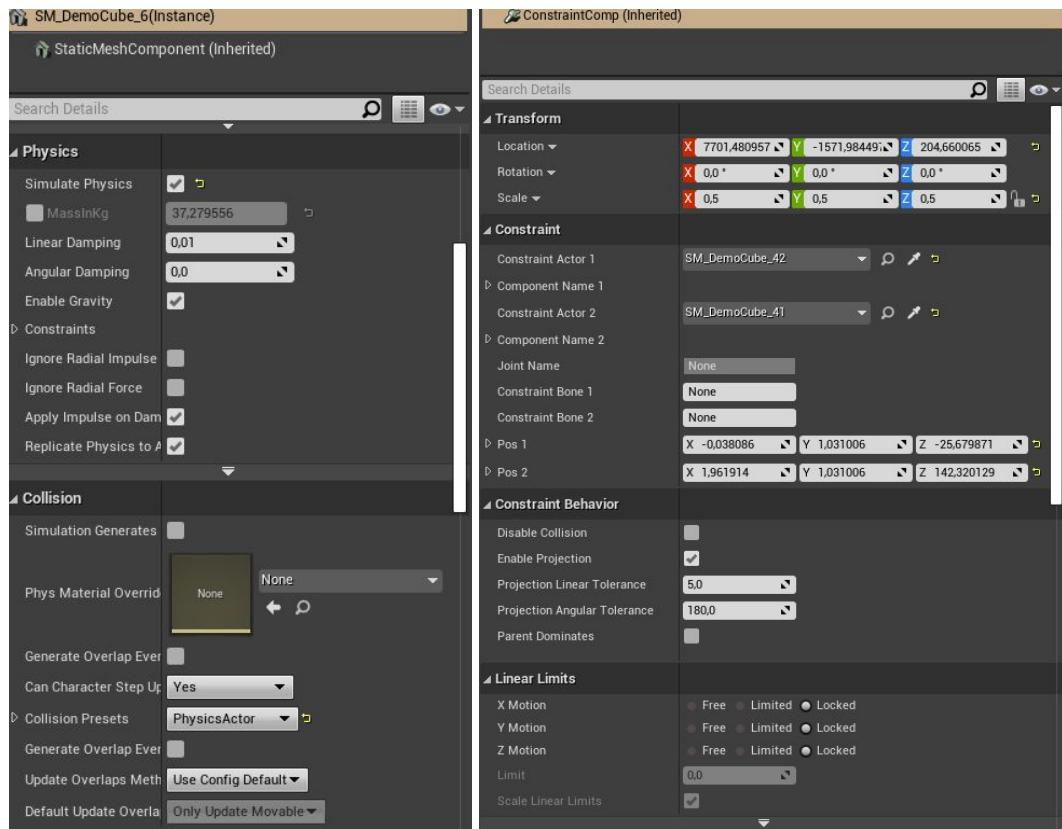


Abb. 12 - Darstellung und Eigenschaften von Körpern und Gelenken im Unreal Editor

4.1.4 PlayCanvas

PlayCanvas ist der wohl größte Konkurrent bzw. das ähnlichste Projekt zu FUDGE. Es handelt sich dabei um eine WebGL Game Engine mit eigenem Editor. Ein Projekt welches unter anderem von Mozilla unterstützt wird und sich an professionelle Spieleentwicklung mittels WebGL wagt, aber mit einem großen und wichtigen Unterschied zu FUDGE, die Zielgruppe sind Editor, Nutzer die sich mit Game Engines und Spieleentwicklung bereits auseinandergesetzt haben, trotz gutem Onboarding, mittels Schritt für Schritt Tutorials in einem modern designten Editor.

Im Bereich Physik nutzt PlayCanvas die bereits vorgestellte Ammo.js Physik Engine. Die vermutlich genutzt wird aufgrund des Funktionsumfangs und der hohen Verbreitung, was für ein so großes Projekt, Sicherheit für Fehlerbehebung und Zukunftsorientierung bietet. Integriert ist diese ähnlich der Unity Integration. Es wird

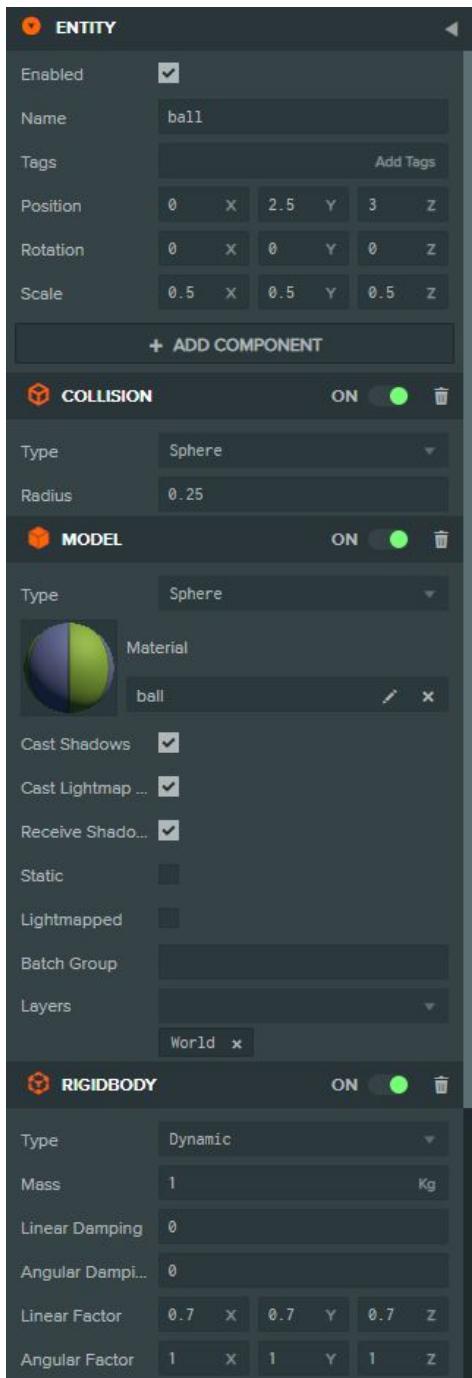


Abb. 13 - Rigidbodies in PlayCanvas

mit Komponenten gearbeitet, für Collider und Rigidbody getrennt, sowie Skripten in denen Raycasts und Events über bekannte Muster aufgerufen werden können. Es wird in Javascript entwickelt, was etwas Übersicht nimmt, da Strukturen wie Klassen nur über Prototype bekannt sind. So wird z.B. ein Kollisions-Event mittels `Teleport.prototype.onTriggerEnter = function (otherEntity)` aufgerufen, was den Nutzern eben nicht übersichtlich Spieleanwendung vermittelt, sondern darauf setzt, dass Javascript Besonderheiten bekannt sind.

Was PlayCanvas dem Nutzer allerdings abnimmt, ist das Laden und Einbinden der Physik Engine, was die grundlegende Nutzung vereinfacht. Sie ist allerdings auch so integriert, dass PlayCanvas ohne genutzt werden kann und somit kleinere Builds ermöglicht.⁴¹

Fest in den Editor und die Engine scheinen zu diesem Zeitpunkt nur das Kräftesystem und Events der Physik integriert zu sein, um z.B. ein Gelenk zu erstellen gibt es keine Komponente und es muss auf direkte Aufrufe der Ammo.js API per Code zurückgegriffen werden.⁴²

Für eine Integration in FUDGE ergibt sich dadurch die Schlussfolgerung, dass eine übersichtliche Integration in Form von einfachen Befehlen und Strukturen, sowohl ein

⁴¹ [17, PlayCanvas Github], Ammo.js wird über einen Loader integriert, der im Editor extra geladen wird, falls ein Physik Feature verwendet wurde (Stand 17.05.20)

⁴² [17, PlayCanvas User Manual]

Alleinstellungsmerkmal, als auch ein Muss ist. Ebenso sollte die Einbindung der Physik Engine separat möglich sein. Dies entspricht auch den bisherigen Leitlinien von FUDGE.

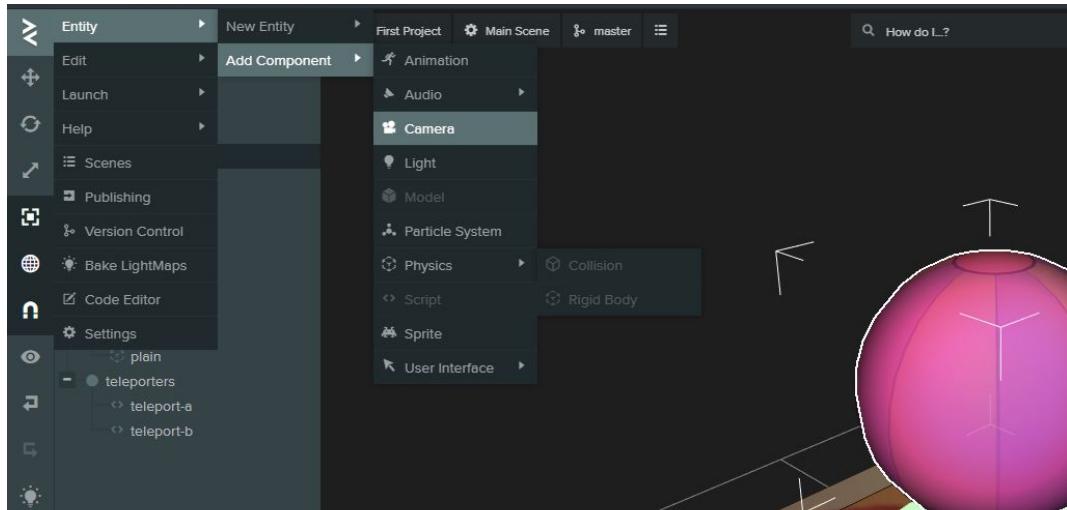


Abb. 14 - Mögliche Physik Objekte im PlayCanvas Editor

4.2 Daraus abgeleitete Paradigmen für ein Integrationskonzept

Aus den gesammelten Informationen ergeben sich große Standards für Physik Integrationen:

- Die Physik besteht aus einer Komponente am Objekt, die objektspezifische Einstellungen für das Physik-Verhalten erlaubt. Die Form des Körpers hängt nicht vom visuellen Mesh ab, daher sollten in FUDGE mehr Formen als die bisher möglichen abgedeckt werden, bestenfalls konvexe Hüllen.
- Event Auslösung, Raycast und Informationsbeschaffung sollte durch die Integration geschehen und nicht durch den Lernenden.
- Die Lernenden haben wenig Interaktion mit der Physik-Welt generell, sondern nur mit Komponenten. D.h. sie erstellen keine Physik-Welt und werden selten dazu aufgefordert mit irgendeiner Klasse zu interagieren, die keine Komponente ist, ausgenommen dem Raycast.

- Gelenk Komponenten sollten so integriert werden, dass sie aussagekräftig und einfach eingesetzt werden, universal konfigurierbare Gelenke sind eher für die Nutzung im professionellen Kontext geeignet.
- Die herausgearbeiteten Funktionsumfänge für die Integration in FUDGE: Festkörper / Kräftesystem, Raycast, Collision/Trigger-Events und Gelenke, können mit dem Angebot vergleichbarer WebGL Engines mithalten und sind ein guter Standardumfang.
- Benennungen sind nicht zwangsweise einheitlich, aber sollten nah an der gewählten Oimo Physics sein und innerhalb der Integration bzw. FUDGE konsistent.

Daraus ergibt sich ein Konzept einer Physik Hauptklasse, die eine Physik-Welt erstellt/veraltet, sowie Raycasts für diese ausführt. Diese soll Einstellungen für die gesamte Physik-Simulation beinhalten.

Eine starre Körper Klasse, die sich bei der Hauptklasse an-/abmeldet und die Eigenschaften von Collider und Körper vereint. Ebenso sollte das Registrieren von Events auf dieser Komponente selbst passieren.

Gelenke müssen an ein Elternobjekt als Komponente angeheftet werden, aber nur aktiv in der Physik Welt werden, wenn zwei Rigidbodies miteinander verbunden sind, sonst sollen sie als Informations Lager dienen, für eventuelle Aktivierung. Gelenke sollten eine gewisse Benennung und Interaktionsmöglichkeiten besitzen, die den Lernenden, mit vorgefertigten Elementen, entgegenkommt, da diese Spieleentwicklung und GameEngine-Strukturen erlernen sollen, ohne spezifisches Wissen über Gelenke zu besitzen. Daher werden mehrere Gelenk Komponenten entwickelt.

Ebenso muss sichergestellt werden, dass die drei Typen der Physik, statisch, dynamisch und kinematisch vorhanden sind und ohne Eingriff korrekt in ihrer

Funktionsweise in der Physik Engine und der Transformations-Hierarchie von FUDGE agieren.

Zuletzt muss, bei jedem weiteren Integrationsschritt sichergestellt werden, dass FUDGE immer verwendet werden kann, ohne OimoPhysics zu laden.

5. Umsetzung der Integration

Die Integration bedeutete Oimo Physics Strukturen zum einen so zu überbrücken, dass sie mit den Strukturen von FUDGE arbeiten. D.h. ein sinnvoller Datenfluss kann garantiert werden, der wichtige Daten austauscht, aber Verarbeitungsweisen getrennt hält.

Zudem wurden Wrapper geschrieben, die Datentypen und nicht FUDGE typische Verarbeitungsweisen verpacken, in für Akteure Nutzbare Funktionen. Es handelt sich um eine Art des Versteckens, so dass sich die integrierte Engine anfühlt als sei sie direkt Teil von FUDGE und für FUDGE entwickelt worden.

Die dritte Struktur die etabliert wurde, ist die Entwicklung von Funktionalitäten die zwar durch Werkzeuge der Physik Engine umgesetzt werden können, aber nicht Teil dieser sind. Dazu gehörten z.B. das Auslösen von physikalischen Events.

Die Basis bildet die **Physics.ts** Klasse, sie übernimmt die Schnittstelle zwischen FUDGE und Oimo.world, also der Physik Welt in der die gesamte Simulation stattfindet.

Die Aufgaben sind folgende:

- Initialisierung und Re-initialisierung der Physik Welt (*initializePhysics, createWorld* - Funktionen, resultieren in einer aktiven Instanz von Physics im Singleton Pattern, es gibt eine Physik FUDGE Integration, die eine Oimo.World verwaltet)
- Ausführen der Simulation, mit wählbarem Zeitabstand, gekoppelt mit der FUDGE Zeitskalierung (*simulate* - Funktion)
- An-/Abmelden von neuen Körpern und Gelenken bei der Oimo.Welt (*add/removeRigidbody* - Funktion, die durch bestehende FUDGE Events bei Komponenten Kreation/Entfernung ausgelöst werden)

- Führen von Listen an Körpern, Triggern, Gelenken (*entsprechende Arrays der Datentypen*)
- Verteilen einer ID für jeden Körper, Suchen eines bestimmten Körpers über seine ID
- Verarbeitung von Raycasts für die gesamte Welt, gefiltert für einzelne Gruppen, Berechnung des Strahlendes, Füllen von Informationen (*raycast, getRayEndpoint, getRayDistance - Funktionen*)
- Einstellungen der Simulation ermöglichen, Standardwerte, Gravitation, Genauigkeit uvm. (hauptsächlich durch eine Instanz der Helfer Klasse *PhysikSettings*, die Oimo.settings beeinflusst)
- Verwalten der Physik Debug Draw Klasse

Für Akteure ist diese Klasse meist nur für die Einstellungen und den Start und der Simulationsfunktion der Physik relevant. Da Verwaltung weitestgehend intern geschieht, da sie für den Akteur nicht gewinnbringend ist.

Allerdings gibt es eine Besonderheit auf die hingewiesen werden muss. Analog zu den analysierten Engines soll sich der Collider an die Transform Komponente anpassen, sofern nicht vom Akteur anders bestimmt, außerdem berechnet die Physik immer mit Welt Transformationen, daher muss die Welttransformation (*Node mtxWorld*) berechnet sein. Um dies zu gewährleisten wurde die bereits bestehende FUDGE Funktion des RenderManagers *.setupTransformAndLights(_sceneTree);* öffentlich gemacht. Allerdings benötigt diese für das Setup, die Szenen Hierarchie. Der Akteur muss vor dem ersten berechneten Frame der Szene, die Hierarchie der Physik bekannt machen, indem er *Physics.start(_szenenRoot);* aufruft. Dies ist ein analoges Verhalten zum bereits in FUDGE etablierten ViewPort, der ebenso die Wurzeltransformation der Szene kennen muss. Für ihn wurde die Welt Transformation allerdings erst im ersten Frame berechnet.

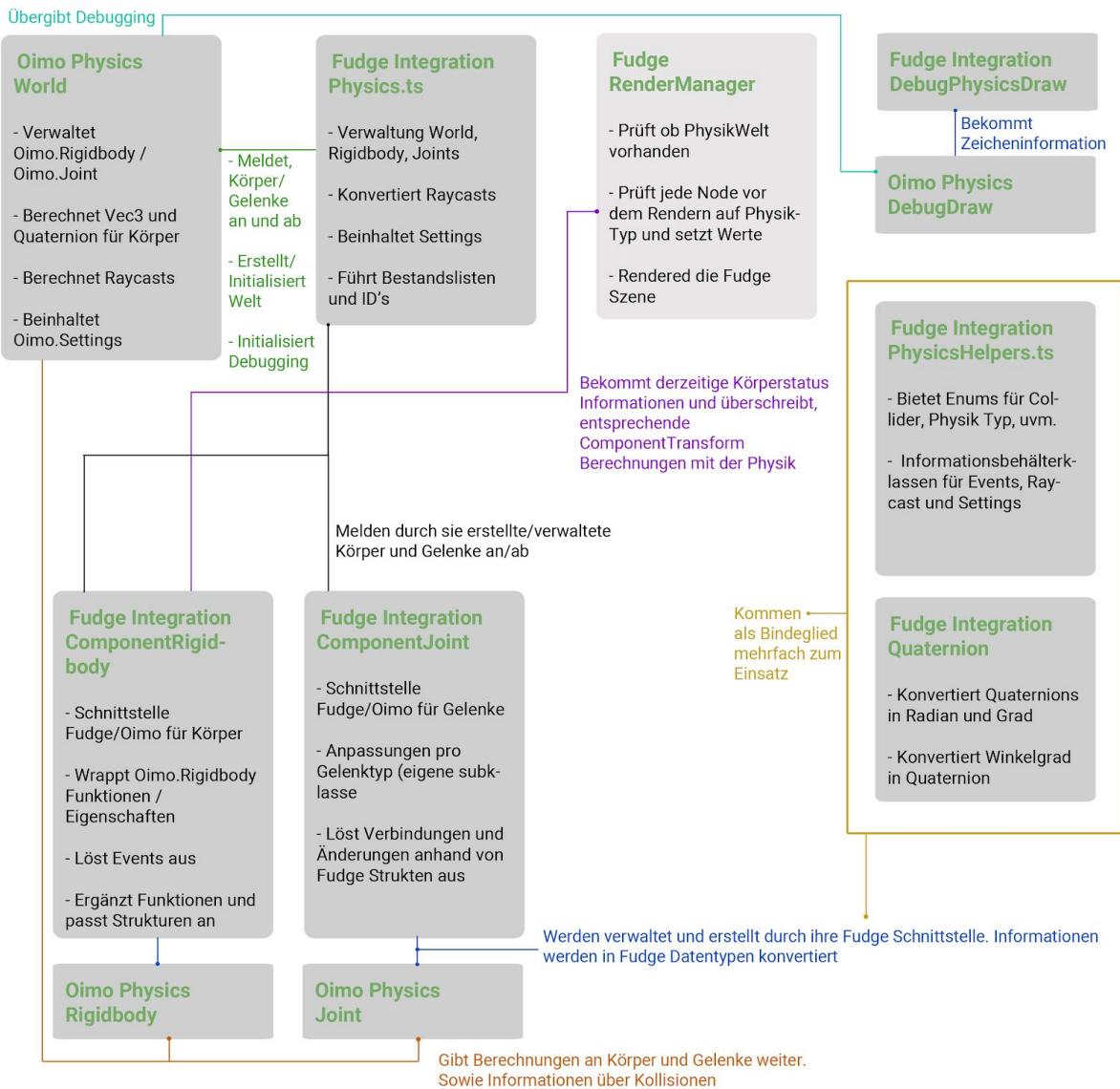


Abb. 15 - Vereinfachter Informationsfluss und Verwaltung der Integration

Zur einfachen Referenz sind alle eigenen Physik relevanten Klassen innerhalb der Source von FUDGE aufzufinden im Physik Ordner.⁴³

⁴³ <https://github.com/JirkaDellOro/FUDGE/tree/PhysicsDevelopment/Core/Source/Physics>

5.1 Kollision, Kräfte und Materialeigenschaften

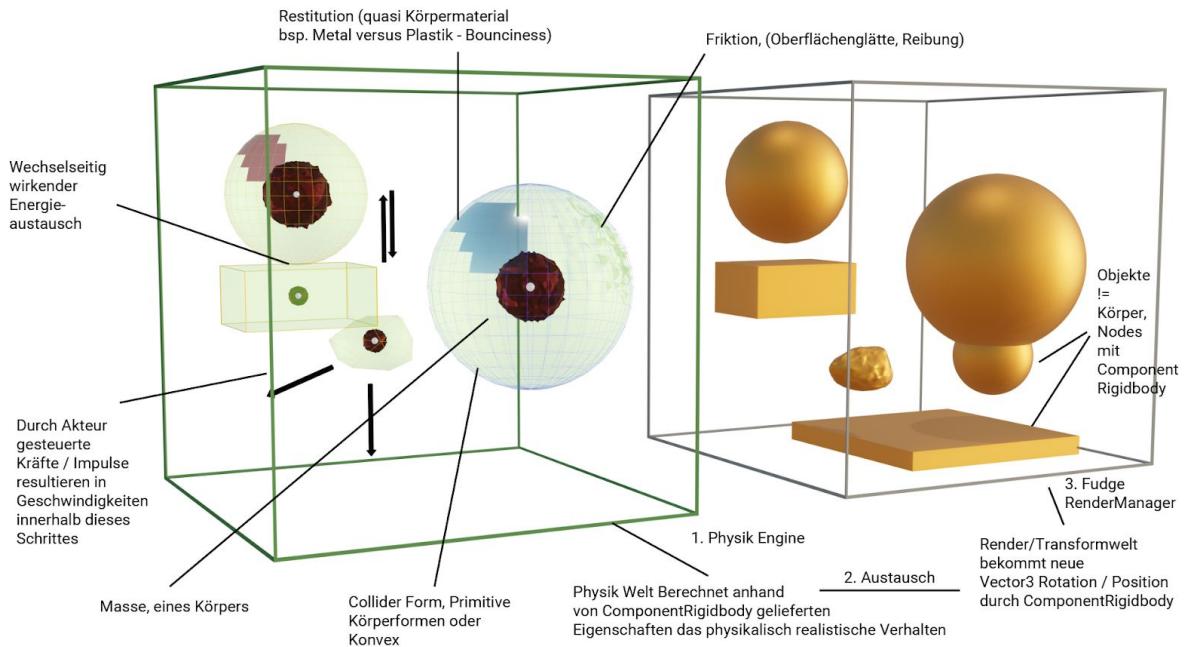


Abb. 16 - Überblick über Körpereigenschaften und den Unterschied zwischen Physik und Rendering/Transformationswelt

Die Basis aller Überlegungen in der Physik sind die starren Körper, Rigidbodies kurz Bodies. Daher müssen nach einer Physik Welt, auch die Körper entwickelt werden, die diese bevölkern. Wie alle anderen Funktionalitäten in FUDGE, wird hierfür eine Komponente entwickelt, dem Namensschema folgend *ComponentRigidbody*.

Die Aufgabe der Komponente:

- Verwalten von Körpereigenschaften für Akteure sichtbar und im gewohnten Format
- Selbst in der Physik Welt an-/abmelden
- Informationen von der Physik Welt an die Render Welt weiterleiten
- Funktionen für Einflussnahme auf Bewegungseigenschaften (Kräfte, Impulse, Geschwindigkeiten)
- Schnittstellen bieten um weitere Funktionalitäten zu ergänzen, Identifikation von ComponentRigidbody, anstelle von Oimo.Rigidbody

ComponentRigidbody:

Für die Entwicklung der ComponentRigidbody wurde eine Vereinfachung

vorgenommen, die sich in der Analyse von anderen Integrationen, als Möglichkeit ergeben hat. Die Kopplung von Collider an die Körper Komponente.

Bestandteile dieser Komponente sind also die Verwaltung einer geometrischen Kollisionsform (Collider) und Einstellungen für die Simulation des Körpers dieser Form.

Collider passen sich typischerweise, sofern nicht anders definiert, an die Transformation des Objektes an dem er Körper hängt an. Die sollte auch in FUDGE funktionieren, wobei FUDGE die Besonderheit besitzt jede Komponente mit einem eigenen Ursprung, engl. *pivot* zu versehen. So kann eine Veränderung des Mesh-Pivot anstatt der Transformation zur Folge haben, dass der Collider dies nicht übernimmt. Entsprechend wurde auch der Körper mit einem pivot versehen, so kann der Collider manuell angepasst werden.

Damit diese automatische Anpassung vorgenommen werden kann, wurde die Funktion *updateFromWorld()* entwickelt, die vor dem ersten Frame der Simulation durch die Physics Klasse aufgerufen wird. In dieser wird der Collider mit den veränderten Einstellungen neu erstellt. Außerdem wird der Körper einmalig auf durch die Transformation gegebenen Werte gesetzt, bis ab dem ersten Simulationsschritt die Physik übernimmt.

Der Collider kann, aus Oimo Physics Sicht, primitive Grundformen und einen konvexen Collider abbilden. Der Konvexe wird aus übergebenen Punkten in Form eines Float32 Arrays erstellt, wobei die Reihenfolge der Punkte eine Rolle spielt. FUDGE besaß zum Integrationszeitpunkt bereits eine Form, die Pyramide, die keine primitive Grundform darstellt.

Alle möglichen Formen wurden in einem Enum definiert, aus dem der Nutzer sich den entsprechenden bei der Konstruktion des Körpers auswählt.

Die Pyramide nutzt hierbei die für konvexe Collider entwickelte Funktion, die aus einem Float32Array, die Punkte ausliest und sie dem konvexen Collider Generator von Oimo Physics übergibt.

Das System von Enums für Einstellungen wurde für alle Parameter der Konstruktion eines Körpers übernommen, bis auf die Masse. Um Eigenschaften wie *PhysikTyp* (Statisch, Dynamisch, Kinematisch) und *PhysikGruppe* (Wer kollidiert mit wem) lesbar zu machen.

Die Komponente besteht zu weiten Teilen aus Wrappern für Oimo Funktionen und Eigenschaften die Datentypen verschleiern bzw. Einflussnahme durch Akteure nur über Getter/Setter zu erlauben, damit eventuelle Prüfungen durchgeführt werden können.

Eigenschaften sind hierbei die bereits beschriebenen Masse, Form, Friction, Restitution und Dämpfungen in der Bewegung, Rotation. Aber auch besondere, in der realen Physik nicht vorkommende Eigenschaften, wie Einfluss der Gravitation oder Sperrung von Rotationen, die allerdings für Muster in der Spieleentwicklung häufig unerlässlich sind.

In Form von:

```
public get restitution(): number {
    return this.bodyRestitution;
}

/*Falls die Restitution gesetzt wird, während kein Collider vorhanden ist
Wird er für zukünftige Konstruktionen eines Colliders intern gespeichert.
Falls vorhanden wird er dem Oimo Rigidbody übergeben. */
public set restitution(_restitution: number) {
    this.bodyRestitution = _restitution;
    if (this.rigidbody.getShapeList() != null)
        this.rigidbody.getShapeList().setRestitution(this.bodyRestitution);
}

//Verwendete Eigenschaft für Restitution bei Konstruktion/Änderung
private bodyRestitution: number;
```

Ähnlich wird hier auch für Funktionen vorgegangen, wobei häufig der nicht kompatible Datentyp Vec3 und Quaternion von Oimo verschleiert wird, damit konstant mit FUDGE Datentypen gearbeitet wird.

```
public applyForceAtPoint(_force: Vector3, _worldPoint: Vector3): void {  
    this.rigidbody.applyForce(new OIMO.Vec3(_force.x, _force.y, _force.z), new  
    OIMO.Vec3(_worldPoint.x, _worldPoint.y, _worldPoint.z));  
}
```

Dabei werden alle gängigen Interaktionen abgedeckt, Kräfte / Impulse (am Schwerpunkt oder an Weltpunkt), Geschwindigkeiten, direktes Bewegen von Position/Rotation.

Intern wird für die Verpackung immer der eigentliche Körper, Rigidbody verwendet und Akteure interagieren immer mit der *ComponentRigidbody*. Damit fortgeschrittene Akteure auch kleinere Funktionalitäten nutzen können die nicht integriert wurden, kann auch von außen auf den Körper mit *getOimoRigidbody()* darauf zugegriffen werden. Hier wird dem Akteur bewusst gemacht, dass er die Integration verlässt.

Oimo Physics hat an eine Integration gedacht, insofern, dass jeder Oimo.rigidbody eine Eigenschaft namens **userData** besitzt. Dieser Umstand kommt in mehreren Fällen zum Einsatz und es sollte bei einer Integration immer beachtet werden, ob eine solche Struktur vorhanden ist. Diese Eigenschaft kann, bei Interaktionen zwischen Körpern, frei genutzt werden um eigene Datentypen zu übergeben. Sie wird daher genutzt um bei Interaktionen immer die *ComponentRigidbody*, also die komplette Komponente zu übergeben. Wodurch Identifikationen vereinfacht werden, da so innerhalb der Physik Engine ohne neue Konstrukte, das FUDGE System angewandt werden kann.

Verbindung zwischen Physik Welt und FUDGE Welt:

Die Oimo Physik Welt sendet ihre Berechnungen automatisch an alle angemeldeten *Oimo.Rigidbody*, diese sind mittels der *ComponentRigidbody* nun in FUDGE integriert.

Für die Anzeige und Verwendung müssen die errechneten Werte noch an das Rendering übertragen werden.

Hierbei gilt die beschriebene Interaktionsstruktur für Physik Körper. Dynamisch überschreibt sämtliche Transformationen außerhalb der Physik Engine, mit den berechneten Welt Transformationen. Statisch bewegt sich überhaupt nicht und Kinematisch schreibt die errechneten Welt Transformationen in die *ComponentRigidbody*.

Das Rendering wird durchgeführt im FUDGE RenderManager.ts,⁴⁴ dort wird in der Funktion *drawGraph* rekursiv jeder Knoten berechnet und gezeichnet. Außerdem wird das Physik Rendering eingefügt und überschreibt entsprechende Werte. Hierbei muss vorher natürlich geprüft werden, ob eine Physik Welt vorhanden ist und der Knoten überhaupt eine *ComponentRigidbody* besitzt. Entsprechend des etablierten *PHYSICS_TYPE Enums* werden dann Werte von der Physik Engine übergeben oder andersherum.

Diese Übertragung sowie die Berechnung der Physik findet in jedem Frame statt, dabei ist die Standard Schrittweite, festgesetzt auf 60 Bilder pro Sekunde, also ~17ms sind in der Physik vergangen. Dies hat zur Folge, dass auch bei geringer Bildrate immer dieselbe Schrittweite genutzt wird. Deshalb wird eine visuelle Verlangsamung der Physik dargestellt, da sie in weniger Wiederholungen mit demselben Zeitabstand berechnet wird. Dies führt aber zu sicheren Ergebnissen, wie in Tests in Kapitel 3.3.5 beschrieben wurde. Akteure können die Schrittweite innerhalb der *Physics.world.simulate(_schrittweite)* Funktion beeinflussen.

Quaternions und Rotationen

Fudge nutzt für Rotationen Matrizen vom Typ Matrix4x4, welche zum Speichern aller Transformationen genutzt werden. Oimo, sowie sämtliche getestete und analysierte Physik Engines, berechnen die Rotationen von Körpern allerdings in Quaternionen. Daher muss bei der Konstruktion von Körpern und dem Austausch von Daten eine Umwandlung stattfinden. Bisher konnte FUDGE diesen Datentyp nicht weshalb die

⁴⁴ FUDGE RenderManager, berechnet für alle Nodes die Weltkoordinaten und übergibt Matrizen an das Rendering
<https://github.com/JirkaDellOro/FUDGE/blob/PhysicsDevelopment/Core/Source/Render/RenderManager.ts>

Quaternions Klasse im Zuge dieser Arbeit etabliert wurde. Hierbei handelt es sich um eine sehr kleine Klasse, die sich nur mit der Speicherung von Quaternionen und der Konvertierung in andere Datentypen beschäftigt. Funktionalitäten, wie die Multiplikation von Quaternionen, oder andere Operationen sind nicht vorgesehen.

Quaternionen werden deshalb von Physik Engines genutzt um bei ihren Berechnungen den Gimbal Lock zu verhindern, welcher wohl durch die Berechnung mittels Euler Winkeln bzw. Matrizen auftreten kann, sowie anderen Vorteilen.⁴⁵ Da Quaternionen allerdings ein Datentyp mit einem mathematisch reellen Anteil und einem imaginären Anteil sind, sind sie für die meisten Akteure nahezu nicht verwendbar.

Daher wurde diese Klasse, anhand verschiedener Referenzen,⁴⁶ zur Umwandlung und Nutzung von Quaternionen entwickelt und damit den Austausch von Daten zwischen Physik Engines und FUDGE möglich gemacht. Möglich ist dabei die Umwandlung zwischen Quaternionen und Vector3, Winkel in Grad und Radian.

Grundlegende Nutzung und Aufbau Physik Integration:

Es braucht eine Physik Welt die in Physics.ts verwaltet wird. Diese muss für die Anpassung von Collidern bzw. des Rigidbody selbst vor dem ersten Simulationsschritt einmal einen Start erfahren, mittels der Funktion, *Physics.start(Szenenwurzelobjekt)*. Es muss, in diskreten Zeitabschnitten, die Physik berechnet werden über *Physics.world.simulate(optionalZeitabschnitt)*.

Für die Berechnung werden dann entsprechend die beschriebene *ComponentRigidbody* hinzugefügt.

Diese wird konstruiert.

```
//Eine neue FUDGE Komponente erstellen mit dem entwickelten Typ
let cmpRigidbody: f.ComponentRigidbody = new f.ComponentRigidbody(_mass,
_physicsType, _colType, _group);

constructor(_mass: number = 1, _type: PHYSICS_TYPE = PHYSICS_TYPE.DYNAMIC,
.colliderType: COLLIDER_TYPE = COLLIDER_TYPE.CUBE, _group: PHYSICS_GROUP =
```

⁴⁵ [19] vgl. GameDev Diskussion do we really need Quaternions - Auf die genauen Vorteile und Funktionsweisen wird aufgrund des Fokus dieser Arbeit nicht eingegangen.

⁴⁶ [19] vgl. Euclidean Space / Mathematics - Quaternion to Angle

```

Physics.settings.defaultCollisionGroup,      _transform: Matrix4x4 = null,
_convexMesh: Float32Array = null) {
//Konstruktionscode, Vereinfacht, Übergabe von Werten. An-/Abmeldung durch von
FUDGE etablierte Komponenten Events. Zusammenbauen eines Körpers aus
einem Körper und einer Form.
this.createRigidbody(_mass,      _type,      this.colliderType,      _transform,
this.collisionGroup);
this.addEventListener(EVENT.COMPONENT_ADD, this.addRigidbodyToWorld);
this.addEventListener(EVENT.COMPONENT_REMOVE, this.removeRigidbodyFromWorld);
}

//Erstellung eines komplett neuen Körpers
private createRigidbody(ParameterList): void{
//Umwandeln von FUDGE Vector3/Matrix4x4 zu Oimo Vec3
//Oimo.rigidbody erstellen
//Oimo.rigidbody Eigenschaften initialisieren
createCollider(_scale, COLLIDER_TYPE.BOX) //bsp. Collider Type
this.collider = new OIMO.Shape(this.colliderInfo);
this.rigidbody.addShape(this.collider);
//Oimo.rigidbody.getShapeList(). alle Eigenschaften die sich speziell
auf den Collider beziehen initialisieren
}

//Erstellung einer Shape, die als neuer Collider hinzugefügt werden kann
private createCollider(_scale: OIMO.Vec3, _colliderType: COLLIDER_TYPE): void
//Switch für Collider Form und je nach Form wird nur ein Teil
der Skalierung verwendet, bsp. Sphere definiert sich nur über die x Komponente
den Radius.
//Falls Konvex -> this.createConvexGeometryCollider(this.convexMesh : 
Float32Array, _scale);
//Collider Konfiguration in privater colliderInfo Variable speichern
{

```

Die konstruierte *ComponentRigidbody* entsprechend an eine Node anhängen.

```
node.addComponent(cmpRigidbody);
```

Anschließend greifen dann die anderen etablierten System und Funktionalitäten. Für die Erstellung eines Colliders bzw. Rigidbody ist noch zu beachten, dass die

Reihenfolge von Transformationen die ein Akteur auf eine Node anwendet entscheidend ist, um Phänomene wie Shearing zu verhindern, d.h. eine rechtwinkliger Körper muss rechtwinklig bleiben, sonst kann es vorkommen, dass er sich zu einem Trapez verformt. Daher sollten Rotationen, die nach einer Skalierung vorgenommen werden, in der Abhandlung der Transformationen links stehen. Dies macht FUDGE typischerweise nicht von allein, kann aber in der Rotation als Parameter angegeben werden.

```
//Best practices Reihenfolge von Transformationen für eine Node, wenn Physik  
nach der Initialisierung der Node verwendet werden soll  
Node.mtxLocal.translate(new f.Vector3(value, value, value));  
Node.mtxLocal.scale(new f.Vector3(value, value, value));  
Node.mtxLocal.rotateY(value, fromLeft);
```

5.2 Raycast - Objekterkennung

Ein mathematischer Strahl, besteht aus einem Anfang und einem Ende, eine Linie ohne Dicke. So ist er in den meisten Physik Engines entsprechend auch definiert und ermöglicht die Detektion von Körpern.⁴⁷

In der Spieleentwicklung ist allerdings selten bekannt wo eine Linie aufhören soll, sondern zumeist nur eine Richtung und eine Distanz. Wie bei einem Schuss. Dieser Umstand führt dazu, dass der Oimo Raycast besonders verpackt werden muss. Hierfür wurde die Funktion des Raycast in die Hauptklasse bzw. Welt integriert in Form einer Funktion die allerdings einen Ursprung, eine Richtung und eine Länge entgegennimmt.

Hierfür muss das Ende des Strahles durch die Integration berechnet werden. Typescript erlaubt leider nur die Verwendung von mathematischen Operatoren zwischen ihm bekannten Datentypen wie Zahlen, aber keinen Vektoren. Der Endpunkt wird entsprechend berechnet:

UrsprungsVektor3 + (RichtungsVektor3 * Länge)

⁴⁷ Oimo Dokumentation - World.raycast auf alle Objekte in der Welt
<https://saharan.github.io/OimoPhysics/oimo/dynamics/World.html#rayCast>

Alle FUDGE Vektoren müssen immer in *Oimo.Vec3* umgewandelt werden und entsprechend wird dann der *Oimo.world.raycast* ausgeführt. Oimo liefert ein *RayCastClosest* zurück, was für Akteure nicht verwendbar wäre. Entsprechend wurde eine Datenklasse *RayHitInfo* entwickelt die FUDGE übliche Daten und Zusatzdaten zurückliefert, über die getroffene Körper Komponente, getroffener Punkt, Normale, Startpunkt, Ende des Strahles, Distanz und natürlich ob überhaupt ein Körper getroffen wurde.

In der Physik Integration wird durchgehend mit Rückgabewerten, die *ComponentRigidbody* enthalten, gearbeitet und nicht mit der getroffenen Node. Dies gibt Akteuren deutlich zu verstehen, dass alle Physik Funktionalitäten sich auf Körper beziehen und nur diese mit dem implementierten Raycast detektierbar sind, außerdem ist der eigentliche Knoten dennoch per *ComponentRigidbody.getContainer()* schnell zu erreichen.

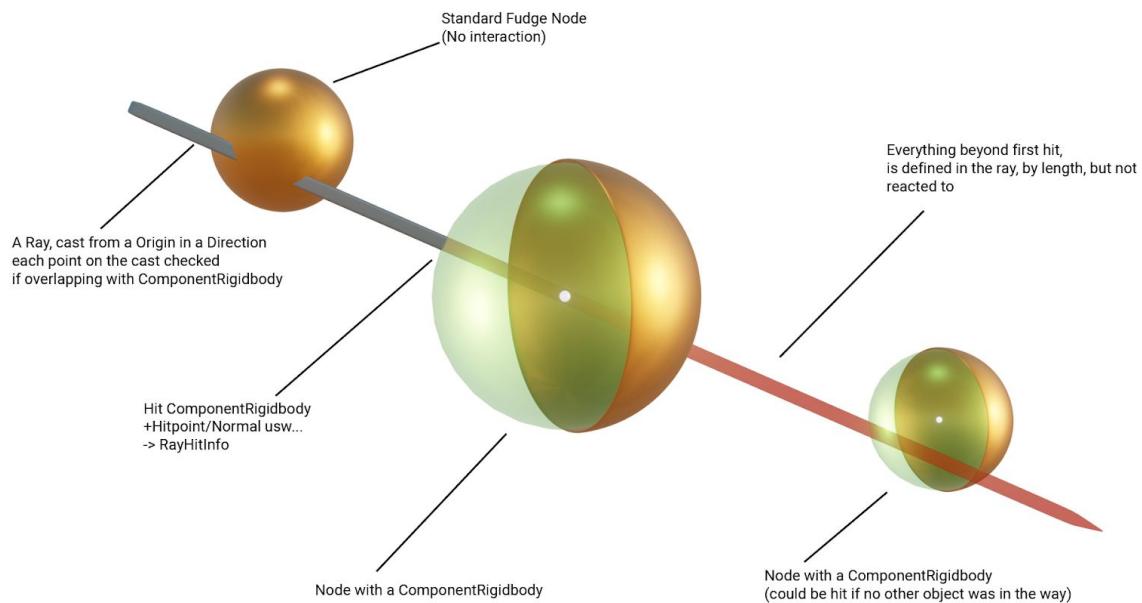


Abb. 17 - Raycast Funktionsweise, Überblick. (grünes Orb impliziert ComponentRigidbody)

In Spielen ist die Filterung von Raycasts ein üblicher Anwendungsfall. So sollen z.B. bei Shootern einige Körper keine Deckung bieten, indem der Strahl eine dünne Holzwand nicht erkennt. Daher wurde der Raycast so integriert, wie ihn Oimo auch bereits vorsieht, einmal als *world.raycast*, um alle Objekte zu treffen und *rigidbody.raycast*, um nur einen einzelnen Körper zu treffen. Allerdings fehlt hier die

Filterung für einzelne Gruppen wie z.B. nur Spieler-Körper, dies wurde selbst implementiert durch:

1. Alle Körper in der Welt von *Physics.ts* erfragen, Übergabe *bodies* Array von *getBodyList()* - Funktion
2. Jeden Körper auf seine zugehörige Gruppe testen, wenn er in der Zielgruppe ist einen *ComponentRigidbody.raycast(origin, direction,length)* ausführen und die zurückgegebene *RayHitInfo* einem Array von *RayHitInfo* hinzufügen
3. Testen welche *RayHitInfo* ist die Nächste und diese als getroffene *RayHitInfo* zurückgeben.

Auf diese Weise wird der integrierte Raycast genauso programmiert, wie in den üblichen Game Engines über die Welt und mit einer möglichen Filterung versehen, sofern gewollt.

```
let hitInfo : f.RayHitInfo = f.Physics.Raycast(origin, direction, length);  
oder  
let hitInfo : f.RayHitInfo = f.Physics.Raycast(origin, direction, length,  
f.PHYSICS_GROUP.ENUMVALUE);  
oder  
let hitInfo : f.RayHitInfo = ComponentRigidbody.Raycast(origin, direction,  
length);  
  
if(hitInfo.hit){ //Detektion erfolgreich }
```

5.3 Spielsteuerung durch Physik-Events | Kollision und Trigger

Die physikalischen Events sind von der Physik Engine zu berechnen, durch Kollisions- bzw. Überlappungsdetektion, allerdings kann sie diese nicht auslösen, da sie natürlich nicht die Struktur in die sie eingebunden wird kennt.

Zu einem Event gehört ein Objekt welches es auslöst und eines welches darauf hört. Im klassischen Javascript wird hierfür ein *EventListener* an das hörende Objekt angeheftet und ein Event *dispatched* (*entsendet*) an ein Ziel, welches durch die Hierarchie dieses Ziels wandern und gehört werden kann. Da es sich um Events

handelt die von dem Physik Teil von FUDGE registriert werden sollen, werden sie von den *ComponentRigidbody* ausgelöst und auch registriert. Der Akteur muss nur noch einen Listener einrichten, der die gewünschte Funktionalität benutzt.

Ein Kollisions Event kann zwei Zustände besitzen, entweder ein neuer Körper trifft den Körper, oder er beendet den Kontakt. Also entweder *Enter* oder *Exit*, häufig gibt es noch *Stay*, der Kontakt bleibt bestehen, aber dies ist eine Funktionalität, die Akteure selbst aus der Kombination von Enter/Exit erstellen müssen. Innerhalb von FUDGE sollen keine trivialen Arbeiten abgenommen werden.

Oimo bietet eine Liste von Kontakten⁴⁸ an, innerhalb eines sog. *Collision Manifolds*, welches Informationen über, Teilnehmer, Impulse uvm enthält. Andere Engines bieten im Normalfall ähnliche Funktionalitäten, da Detektion ein Teil jeder Engine ist, weshalb die Integration von Events ähnlich funktionieren sollte.

Innerhalb eines Frames bzw. eines Schritts der Simulation muss nun folgendes für alle Kontakte innerhalb der Funktion *checkCollisionEvents()* geprüft werden:

Collision_Enter Event

- Existiert ein Kontakt und ist einer davon, nicht der prüfende Körper selbst. Hierbei arbeitet Oimo leider so, dass in eine Kollision immer zwei Körper verwickelt sind, aber selbst überprüft werden muss, welcher Körper der kollidierende ist. Dieser kann nicht vorher ausgeschlossen werden.
- Der Körper wird identifiziert über die bereits beschriebenen *UserData*, so dass hier eine FUDGE Komponente bekannt ist und kein *Oimo.Rigidbody*
- Ist der Körper bereits bekannt, wenn nicht, den Körper hinzufügen zur Liste der Körper mit denen ein Kontakt besteht. -> Event einmalig auslösen
- Informationen hineinfüllen, z.B. durch Ermittlung des Zentrums des Kontakts (*collisionCenterPoint* - Funktion), Addition der Impulse aus allen

⁴⁸ vgl. Oimo Physics Dokumentation

<https://saharan.github.io/OimoPhysics/oimo/dynamics/rigidbody/RigidBody.html#getContactLinkList>

Kollisionspunkten. Übergabe des involvierten FUDGE Körpers über die *Oimo.Rigidbody.userData*

Collision_Exit Event

- Der Körper prüft ob innerhalb seiner Kontaktliste ein Körper noch vorkommt, wenn er sich bisher in der Liste der kollidieren Körpern befand.
- Ist er nicht mehr in Kontakt, wird er aus der Kollisionsliste entfernt und ein Event wird ausgelöst.

Trigger Events sind für Physik Engines ein meist nicht bedachtes Konstrukt. Weil es in der realen Physik keine Bereiche gibt, die prüfen ob sich etwas in ihnen befindet und dann etwas auslösen: Es findet kein Energieaustausch statt und daher keinen Kontakt, bzw. Kollision. Daher müssen für die Kreation von Triggern einige andere Funktionalitäten ausgenutzt werden.

Im Grunde funktionieren die Strukturen hier ähnlich der Kollision-Events. Es wurde ebenso ein Funktionsteil entwickelt, der Körper prüft und einer Liste hinzufügt, so dass Enter Events nur einmal ausgelöst werden und beim Verlassen des Auslösers, noch einmal Exit Events ausgelöst werden.

Die Unterschiede liegen hierbei allerdings in der tiefergehenden Funktionsweise, da Trigger eben keine reagierenden Körper sind. Es kann nicht auf die Kontaktliste von *Oimo.Rigidbody* zurückgegriffen werden, sondern die Überlappung anhand der Detektion der Physik Engine überprüft. (*collidesWith* - Funktion). Hier wird eine simplifizierte Annahme getätigt, dass Trigger wenig ungewöhnliche Formen besitzen, da sie zumeist als unsichtbare Boxen in Spielen verwendet werden. Oimo bietet nur die Möglichkeit eine Überlappung zwischen den Bounding Boxes festzustellen.⁴⁹ D.h. hier kann die Collider-Form und die Registrierung von Trigger Events, in untypischen Anwendungsfällen, leider leicht abweichen.

⁴⁹ vgl. Oimo Dokumentation AAbb Intersection
<https://saharan.github.io/OimoPhysics/oimo/collision/geometry/Aabb.html#getIntersection>

Die Überlappung von Körpern soll entsprechend nur mit Triggern ausgelöst werden. Da es so in der Physik keine Körper gibt die diese Funktionalität ermöglichen, wird dies darüber gelöst, dass Körper die man als Trigger benutzen möchte der entsprechenden Physik Gruppe, in der Konstruktion, oder über die Funktion `set collisionGroup(_value: PHYSICS_GROUP)` zugeteilt werden. Dadurch wird der Physics Klasse mitgeteilt, dass dieser Körper in die Liste der Trigger aufgenommen und entsprechend geprüft werden soll.

Zuletzt wird nicht nur geprüft, ob der Körper in einen Trigger bewegt wurde, sondern alle Körper die der Trigger Gruppe angehören, prüfen ob irgendein Körper sich in ihnen befindet.

Wird ein Trigger Event ausgelöst, werden ähnliche Informationen wie bei der Kollision verschickt, allerdings keine Impulse, da es schließlich keinen Energieaustausch gibt.

Javascript Custom Events zu FUDGE Events

Physik Events sollten ursprünglich integriert werden als Javascript Custom Events, eine spezielle Form von Events, die Daten zusätzlich zu normalen Javascript Events enthalten. Allerdings mit der Folge, dass anstatt eines normalen EventListeners und EventHandlers, entsprechend Custom Varianten verwendet werden müssen. Außerdem wäre der zusätzliche Inhalt, wie involvierter Körper, in einer nicht aussagekräftigen Variable in Form von `event.detail` gespeichert.⁵⁰

Daher wurde in einem zweiten Schritt das verschickte Event umgeschrieben auf eine erweiterte `EventTarget` Klasse, die sog. `Mutable` Klasse (Basisklasse aller Komponenten in Fudge). Durch diese Struktur kann eine Komponente Ziel einer vererbten Form von Event sein und so kann eine Event Klasse erstellt werden, die besondere eigene Eigenschaften enthält, anstatt dem normal überlieferten Event. Hierfür muss eine Event Klasse erstellt werden, in diesem Fall `EventPhysics`, welche anstatt in einer `event.detail` Eigenschaft, viele Informationen zu übergeben, einzeln Eigenschaften wie `event.componentRigidbody`, oder `.normalImpulse` speichern und für Akteure übersichtlich implementieren kann.

⁵⁰ https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Creating_and_triggering_events

Um dies zu nutzen muss FUDGE die neue Event Klasse als zulässigen Typ beigebracht werden, innerhalb von *Event.ts*.⁵¹

Dadurch entsteht die übliche Javascript Event Struktur für physikalische Events, und nahe Implementation zu anderen Game Engines, eines selbstauslösenden Events auf welches nur gehört werden muss.

In Unity z.B. muss ein Script auf einem Objekt mit Rigidbody/Collider liegen, welches eine Funktion *OnCollisionEnter(_col: collision)* enthält, in der dann eine Funktionalität programmiert werden kann z.b. *_col.rigidbody*, um den involvierten Körper zu bekommen..

In FUDGE wird dies nun gelöst über:

```
//Listener an den Körper heften, Typ aus ENUM auswählen und Funktion die das  
Event handhaben soll auswählen  
rigidbodyKomponente.addEventListener(f.EVENT_PHYSICS.COLLISION_ENTER,  
hndCollisionEventEnter);  
//Funktion bekommt ein angepasstes Physik Event als Parameter übergeben  
function hndCollisionEventEnter(_event: f.EventPhysics): void {  
_event.cmpRigidbody //involvierter Körper  
}
```

So wurde anhand von vorhandenen Strukturen von FUDGE und Werkzeugen von Oimo Physics eine Brücke gebaut, um die Integration eines der wichtigsten Muster im Game Design zu ermöglichen.

5.4 Gelenkverbindungen

Gelenke (geläufiger engl. *Joints*) stellen eine ganz besondere Art von Komponenten dar, wie sie bisher in FUDGE nicht vorkamen. Sie sind zum einen wie alle Physik Komponenten in ständiger Kommunikation mit einem externen Element, aber sie sind auch nur funktional, wenn zwei andere Komponenten referenziert werden. Um ein Gelenk zu erstellen müssen zwei *ComponentRigidbody* miteinander verbunden werden und diese Verbindung muss der *Oimo.world* bekannt sein, damit im Simulationsschritt dieses Gelenk auf die Körper angewandt wird. Im Endeffekt ist ein

⁵¹ <https://github.com/JirkaDellOro/FUDGE/blob/PhysicsDevelopment/Core/Source/Event/Event.ts>

Gelenk vereinfacht nur eine Komponente um ein Verhalten für zwei *ComponentRigidbody* zu kreieren, aus technischer Sicht müsste sie keine Komponente an einer Node sein.

Dies wurde allerdings so umgesetzt weil dies in Game Engines üblich ist, dass die Informationen über die Gelenkverbindung als Komponente an den ersten Körper dieser Verbindung angehängt wird.

Eine Gelenkkomponente kann typischerweise bestehen ohne aktiv zu sein, z.B. wenn derzeit nur ein Körper des Gelenks gesetzt ist. Um diese Funktionalität umzusetzen wurde eine Verarbeitungsstruktur entwickelt die eine Art *dirty* (dt. dreckig) Status beinhaltet. Eine bekannte Struktur, bei der etwas Bearbeitetes z.B. ein Dokument anzeigt, dass es Änderungen gibt die noch nicht gespeichert bzw. angewendet wurden und erst noch bereinigt werden müssen. Analog verhält sich die *ComponentJoint*, es kann die Komponente bestehen und an einer Node angehängt werden, aber die tatsächliche Konstruktion der Oimo Gelenke findet nur statt, wenn der Physik Integration gemeldet wird, dass sich ein bereites, aber noch unverbundenes Gelenk im System befindet. Dieser Umstand wird dann vor dem nächsten Simulationsschritt bereinigt.

Dieses System bietet einen zweiten Vorteil, es dient der Verringerung von Fehlern, da dies eine sehr komplexe und referentielle Komponente ist, müssen zum Zeitpunkt der Konstruktion beide *Oimo.Rigidbody* vollständig in der Physik Welt bekannt sein. Aufgrund verschiedener FUDGE und vor allem Javascript interner Strukturen, kann es aber dazu kommen, dass die *Oimo.Rigidbody* sich noch im Bau befinden, der Akteur allerdings schon ein Gelenk kreiert und zwei *ComponentRigidbody* als Parameter übergibt. Ohne dieses System würde daher, eine Konstruktion eines *Oimo.Joints*, ohne etablierte *Oimo.Rigidbodies* stattfinden. Dieses Verhalten konnte zu Beginn der Integration von Gelenken beobachtet werden und es war ein entsprechend schwieriges Unterfangen dies herauszufinden.

Die weitere Basis-Integration ist zum größten Teil das Wrappen von Oimo Funktionen in FUDGE Strukturen. Hierzu wurden wieder Getter/Setter Funktionen für Gelenk-Eigenschaften erstellt, die Datentypen von Oimo verstecken und ggf. weitere

Funktionalitäten ausführen. Es gibt daher jeweils eine *public get/set* für z.B. die Feder Frequenz und eine *private, interne jointSpringFrequency*, die dann innerhalb der Gelenkkonstruktion verwendet wird. Eigenschaften wie Achsen und Verankerung haben eine Neu-Konstruktion des Gelenks zur Folge, dies liegt an der Oimo internen Struktur, da eine neue Konfiguration von lokalen Achsen und Verankerungspunkten stattfindet.

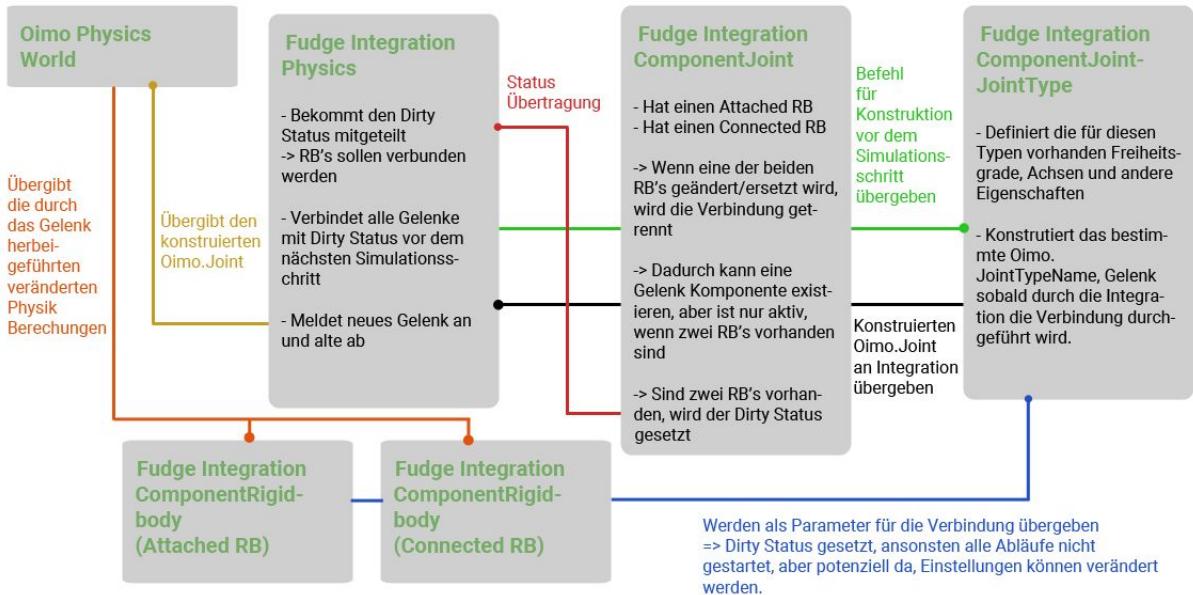


Abb. 18 - Überblick der Gelenk Integration

Ein Gelenk besteht in Oimo, bzw. im Allgemeinen, aus einer Verankerung, an der beide Körper angebracht sind, einer/mehreren Achsen auf der Freiheitsgrade vorhanden sind, sowie Motor- und Federeigenschaften. Es gibt verschiedene Gelenktypen mit unterschiedlichen Eigenschaften.⁵²

Es wurde entschieden alle Gelenke einzeln zu integrieren, statt eines konfigurierbaren Gelenks, da dies dem Aufbau von Oimo Physics entspricht, welche eine Basisklasse für Eigenschaften für alle Joints hat und einige Subklassen mit speziellen Konfigurationsstrukturen für die unterschiedliche Typen. Die Typen entsprechen dabei häufig verwendeten Gelenken, was dem Ziel von FUDGE zugute kommt verständlich in die Physik Nutzung in der Spieleentwicklung einzuführen.

⁵² vgl. Dokumentation Oimo bsp Prismatic Joint
<https://saharan.github.io/OimoPhysics/oimo/dynamics/constraint/joint/PrismaticJoint.html>

Oimo bietet 6 verschiedene Gelenktypen an, wovon 5 typische Standardfälle abdecken und ein zusätzliches *Ragdoll* Gelenk. Dieses benötigt etwas mehr Verständnis, um Strohpuppen Physik zu realisieren, was fortgeschrittener Nutzung entspricht.

Um die Eigenschaften die zur Verfügung stehen kurz zu erläutern:

- Verankerung - *Anchor* - An diesem Punkt sind beide Körper verbunden, er wird üblicherweise lokal angegeben zum ersten Körper (Oimo Physics gibt ihn in Weltposition an und es wurde daher eine Konvertierung eingebaut, da Nutzung mit lokaler Angabe üblich)
- Achse - *Axis* - In diese Richtung besteht der Freiheitsgrad, üblicherweise steht die Achse für jeweils beide Freiheitsgrade Translation / Rotation

Für Federn:

- Feder Frequenz - *Spring Frequency* - Die Steife der Feder, bei 0 ist keine Feder vorhanden. Standardwert daher 0 für Gelenke. Angegeben in Hertz. Dadurch wird die Anzahl der Schwingungen in Sekunden angegeben.⁵³
- Federdämpfung - *Spring Damping* - Die Dämpfung der Feder die hilft die Nulllage zu erreichen.⁵⁴

Für Federn kann festgestellt werden, dass hier zwei sehr vereinfachte Werte für die Darstellung verwendet werden und man für den realistischen Einsatz einige Umrechnungen benötigt. Weil die Frequenz die ist, die Physik Engine intern wohl von der Feder erreicht wird, aber eigentlich von der Masse abhängig sein sollte. Daher

⁵³ vgl. komplette Herleitung, für die Verwendung muss diese Eigenschaft häufig ausgetestet werden.
<https://www.ingenieurkurse.de/physik/schwingungen/ungedaempfte-harmonische-schwingungen/schwingungsgleichung-federpendel.html>

⁵⁴ vgl. komplette Ausführung in der Mechanik
<https://www.leifphysik.de/mechanik/mechanische-schwingungen/grundwissen/federpendel-gedaempft>

kann davon ausgegangen werden, um mit steigender Masse des verbunden Objektes trotzdem die Frequenz zu erreichen, wird intern die Federkonstante erhöht.

Für Motoren:

- Oberes Limit - *Upper Limit* - Die obere Grenze in Welteinheiten (Meter), die sich beide Körper von der Verankerung auf ihrem Freiheitsgrad bewegen können. Standardwert für Rotation 360° und Translation 10 Meter (neue Gelenke sind also nahezu unbeschränkt, der Akteur soll nach gemeinsamer Entscheidung, die Beschränkung absichtlich selbst setzen)
- Unteres Limit - *Lower Limit* - Die untere Grenze (0° Rotation, -10 Meter Standardwert). Ist das untere Limit größer als das Obere, gibt es keine Limits. Limits können auf 0/0 gesetzt werden, dann wird versucht keine Verschiebungen stattfinden zu lassen (engl. *Welding* - Fest verbinden).
- Kraft / Drehkraft - *Force / Torque* - Die Kraft mit der die Verbindung gehalten wird. Und die Kraft mit der die Motorgeschwindigkeit versucht wird zu erreichen.
- Geschwindigkeit - *Speed* - Wie sich die Verbindung eigenständig versucht zu bewegen/lösen, entlang des definierten Freiheitsgrades, in Meter pro Sekunde. Beispielsweise ein Schieberegler wird versucht, entlang der Achse in positiver Richtung, um 1 m/s zu verschieben und umgekehrt.

Oimo Physics versucht immer die entsprechenden Limits für Motoren einzuhalten. Allerdings je nach Akkuratheit der Berechnung können auch komplett limitierte Körper kurzzeitig weiter auseinandergehen. Auch kinetische Körper können verbundene Körper über die Limits hinaus bewegen, da kinetische Körper nicht nachgeben, kann Oimo Physics die theoretisch unendliche Gegenkraft des kinematischen Körpers nicht zurückdrängen.

Zusätzliche Einstellungsmöglichkeiten sind:

- Die Fähigkeit auseinander zu brechen bei zu großen Kräften. Wie in Game Engines üblich wird hierbei nur die Verbindung gelöst und nicht die Komponente entfernt.
- Die Möglichkeit die Körper die an der Verbindung beteiligt sind miteinander kollidieren zu lassen oder interne Kollisionen zu ignorieren (Initialwert).

Um kurz die integrierten Gelenktypen vorzustellen:

Das **prismatische Gelenk**, mit einem Freiheitsgrad in der Translation. Es besteht daher aus einer Feder und einem Motor für Translation.

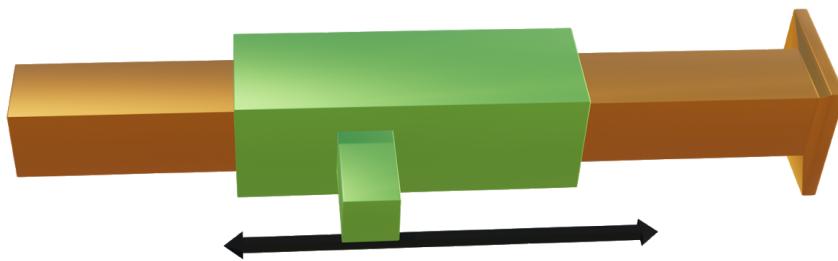


Abb. 19 - Modellierte Darstellung eines Einsatzes für ein prismatisches Gelenk, innerhalb eines Schiebereglers

Das **revolute Gelenk**, mit einem Freiheitsgrad in der Rotation. Einer Feder und einem Rotationsmotor.

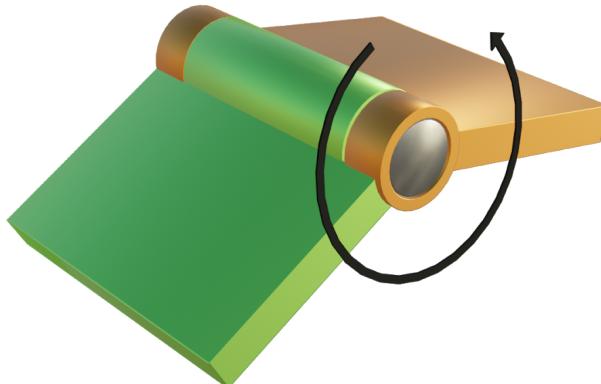


Abb. 20 - Modellierte Darstellung eines Einsatzes für ein revolutes Gelenk, innerhalb eines (Tür-) Scharniers

Das **Cylinder-Gelenk**, als Kombination aus den vorherigen Gelenken. Eine freie Rotation und Translation auf einer Achse. Mit je einem Motor und Feder für Rotation/Translation.

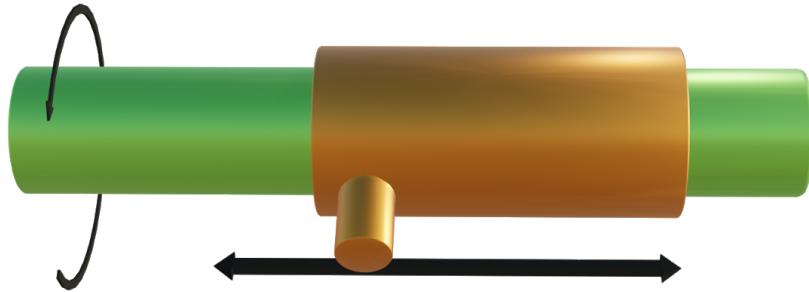


Abb. 21 - Modellierte Darstellung eines Einsatzes für ein cylindrisches Gelenk, innerhalb eines (Tür-) Riegels

Das **Kugelgelenk**, auch bekannt als Ball and Socket Joint. Mit 3 Freiheitsgraden in der Rotation, aber keiner Bewegung. Es kann frei schwingen und drehen.

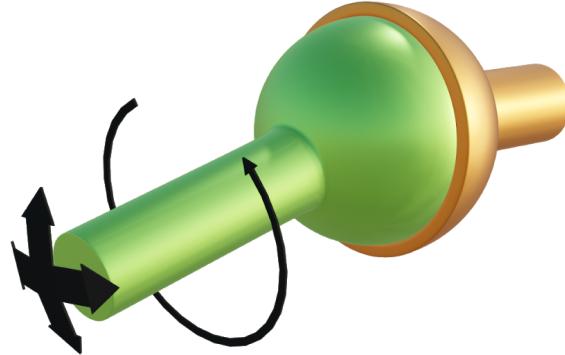


Abb. 22 - Modellierte Darstellung eines Einsatzes für ein Kugelgelenk, innerhalb einer Schulter ähnlichen Verbindung

Das **Universale Gelenk**, mit zwei definierten Achsen der freigegebenen Rotation. Mit dazugehörigen Motoren und Federn. Es gibt zwei Schwung-Achsen, aber die interne Rotation ist verbunden, was typischerweise für Umlenkung von Rotation auf einen anderen Körper genutzt wird.

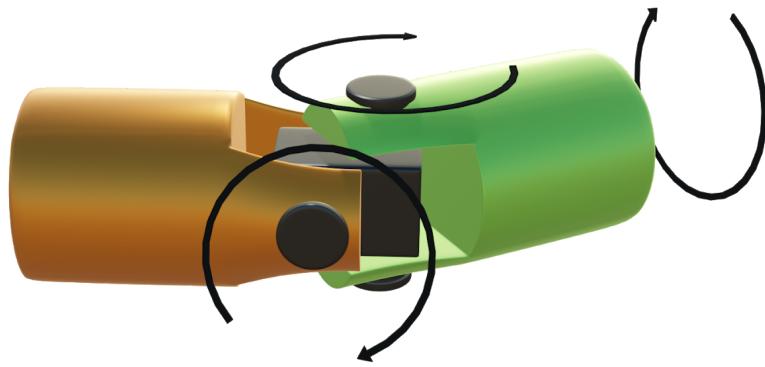


Abb. 23 - Modellierte Darstellung eines Einsatzes für ein Universal Gelenk, innerhalb einer typischen Verbindung

Das **Ragdoll Gelenk**, welches ähnlich dem Kugelgelenk funktioniert, aber spezielle Vorteile für das Erstellen von Strohpuppen Physik beinhaltet, und z.B. für fallende bewusstlose Humanoiden, verwendet werden kann. Mit zwei Achsen und Möglichkeiten zur Winkelbegrenzung.

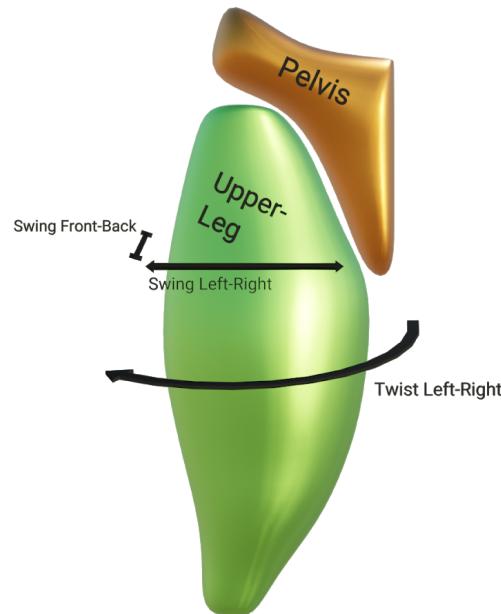


Abb. 24 - Modellierte Darstellung eines Einsatzes für ein Ragdoll Gelenk, innerhalb einer Bein, Rumpf Verbindung

5.5 Besonderheiten und Schwierigkeiten

Bei der Integration gab es einige Schwierigkeiten, manche sind zurückzuführen auf die Entscheidungen die für FUDGE getroffen sind, einige sind leider aber auch der

Wahl von Oimo geschuldet.

Zum einen wäre hier der Vorteil, der **mitgelieferten Typescript Definitionen**, was ansich die Integration, als auch die erweiterte Nutzung durch Akteure, vereinfacht. Allerdings sind diese Definitionen Teil eines Updates auf den neueren Haxe 4.0 Standard. In Folge dessen wird von Javascript auf Javascript Modules als Hauptanwendungsbereich gewechselt. Was zur Folge hat, dass die Definitionen für Modules sind. In diesen Modules sind Subklassen der Oimo Engine aufgeteilt auf *common*, *dynamics*, sowie weiteren. Daher sind alle Funktionen innerhalb dieser Subnamespaces und können nicht in der normalen Javascript Engine verwendet werden, die alle Funktionalitäten in *Oimo* als Namespace zusammenpackt.

Folglich mussten die Definitionen umgeschrieben werden.

Die Nutzung von Javascript Module Varianten von Oimo Physics ist ausgeschlossen, da dies eine Verwendung serverseitig nach sich zieht und somit nicht für guten Austausch sorgt. Im Zuge dieser Entdeckungen wurde auch entdeckt, dass sich die in Arbeit befindende Version von Oimo Physics, auch die Funktionalität des visuellen Debuggings verändert bzw. nahezu streicht.

Ein Detail was in der Entwicklung vieler Javascript Physik Engines nicht betrachtet wird, ist die oft realitätsferne Simulation die Spiele darstellen, obwohl dies wohl der größte Anwendungsbereich ist. In FUDGE als Spiele Editor sollte die Funktionalität gegeben sein, dass die **Collider der Körper sich auch während der Laufzeit bzw. nach Erstellung des Körpers verändern können**, um z.B. einen Spielercharakter zum Riesen werden zu lassen.

Dafür muss ein Collider skaliert werden. Dies ist aber in Oimo Physics nicht wirklich vorgesehen gewesen, weshalb eine Skalierung nur durch Neu-Kreation des Colliders möglich ist. Das ist grundlegend kein Problem für die Funktionalität, allerdings besaß Oimo Physics einen Bug, wodurch keine Collider zu einem Körper hinzugefügt und entfernt werden konnten. Da beim Entfernen, trotzdem innerhalb Oimos immer auf den bereits entfernten Collider zugegriffen wurde. Dieser Fehler wurde mit Hilfe eines anderen Oimo Nutzers, in längerer Absprache, entdeckt und behoben. Der

Entwickler reagierte nicht auf das Anliegen⁵⁵ und inzwischen muss daher eine bearbeitete Oimo Engine genutzt werden die im FUDGE Physics Ordner zu finden ist. Ohne Behebung dieses Fehlers hätte bei jeder Collider Änderung, statt nur dem Collider wie es üblich ist, der komplette Körper neu erstellt werden müssen, was zur Folge hätte, dass alle wirkenden Energien und Eigenschaften wiederhergestellt werden müssten, was keine Option gewesen wäre.

Drittens stellte sich gegen Ende der Integration heraus, **dass Oimo Physics vermutlich nicht ganz sauber arbeitet, um so performant zu sein.** In der Integration von Einstellungen zur Engine, sowie Tests stellt sich heraus, dass Oimo wohl nur einige wenige tatsächliche Iterationen vornimmt, um die Physik Korrekt zu berechnen, und dann mit Hilfe von Korrektur Algorithmen nachbessert. Dies führt zu Ungenauigkeiten, die auch schon bei der Auswahl, als Induktionen von Energie bemerkt wurden. Es ist nicht ganz nachzuvollziehen ob es bei Oimo Physics immernoch so ist, da dieser Hack für Performanz nur für Oimo.js⁵⁶ nachgewiesen werden konnte. Fest steht allerdings Oimo Physics kann mit Standard Einstellungen deutlich höhere Iterationszahlen, als üblich, ohne Frameeinbrüche berechnen.⁵⁷ Außerdem werden Einstellungen für Solver geboten.

Dies ist kein Ausschlusskriterium, speziell da es erst so spät bemerkt wurde und vlt. kein Einzelfall in Javascript Physik Engines ist, allerdings eine Schwäche die man beachten muss. Für verschiedene Anwendungsfälle sollte also definitiv mit verschiedenen Einstellungen getestet werden. Positiv fällt hierbei die Vielfalt von Einstellungen und verschiedenen Algorithmen auf, allerdings sind die Standardeinstellungen deutlich fehlerbehaftet. In FUDGE sind fast alle Einstellungen nun über *FudgeCore.Physics.settings* zu erreichen und wurden in zahlreichen

⁵⁵ <https://github.com/saharan/OimoPhysics/issues/33> Keine Antwort, aber Hinweis durch mich erbracht für Zukünftige Änderungen. Die Lösung wurde im privaten Gespräch mit einem Nutzer erarbeitet.

⁵⁶ vgl. <https://dzone.com/articles/webgl-physics-based-car-using-babylonjs-and-oimojs> Dieser Funktionalität wurde für Oimo.js herausgefunden, da Oimo Physics die Basis von Oimo.js war, kann davon ausgegangen werden, dass Oimo Physics ähnlich agiert, obwohl es von Grund auf neu geschrieben wurde. Es gibt auch eine Einstellung für den Correction Algorithm.

⁵⁷ Normalerweise ca. 10 Iterationen = Standardwert auch in Oimo Physics, aber es ließen sich auch locker 100-500 Iterationen einstellen ohne Einbrüche, aber mit besseren Ergebnissen.

<https://saharan.github.io/OimoPhysics/oimo/common/Setting.html#defaultContactPositionCorrectionAlgorithm> und ähnliche Einstellungen zu Sovern usw.

Anwendungsfällen getestet. dabei variiert die Qualität stark, vor allem da der Gauss-Seidel, der wohl verbreitetste Standard Algorithmus in Oimo Physics nicht standard ist.

6. Entwicklung und Optimierung der Nutzerinteraktion

6.1 Optimierungen für die Nutzung - Visuelles Debugging

Zur Optimierung der Nutzung der Physik Integration, gehören Entscheidungen wie die Vorbelegung von Werten, z.B. Gelenke sind bei Erstellung nur geringfügig restriktiv und werden von den Akteuren erst eingestellt. Eine Iteration zuvor war dies noch umgekehrt.

Allerdings gibt es auch größere Änderungen. Wie die bei den Events beschriebene Änderung von normalen Javascript Custom Events, für physikalische Events, in die Struktur wie FUDGE sie bei anderen Events verwendet. Wodurch die Nutzung konstant gleich bleibt und mehr Informationen übertragen werden können.

Das visuelle Debugging ist eine spezielle Funktionalität von Oimo Physics und einer der Gründe für die Entscheidung für diese Engine. Eine interne Klasse *DebugDraw*, bekommt Informationen von der *World* Klasse über verschiedene Ereignisse und Parameter der Simulation und verarbeitet diese in mehrfachen Aufrufen von 3 intern leeren Funktionen *point*, *line*, *triangle*. Die Implementierung der Zeichenbefehle muss entsprechend selbst erarbeitet werden.

Diese Klasse nimmt einen wichtigen Teil ab, nämlich das Management wie eine Physik Szene analysiert werden kann. Es muss nicht festgelegt werden wie ein Limit der Drehbarkeit eines Gelenkes aussieht, oder wie ein Würfel-Collider aus Linien gestaltet wird. Diese Funktionalität ist ein großer Bonus für die Anwendungsoptimierung der Integration, da sie mit verschiedenen Modi, Elemente wie Gelenke, Collider, Bounding Boxes zu analysieren vermag. Der Modus *PhysicObjectsOnly* ermöglicht, übersichtlich Probleme und Verbesserungen an der eigenen Szene zu finden und die Integration der Physik Engine generell gut zu testen.

Verschiedene Farben deuten auf einen bestimmten Status wie z.B. schlafend hin. Drahtgestelle zeigen, zusätzlich zum eigentlich definierten visuellen Anteil einer Node, den physischen Körper in Form des Colliders.

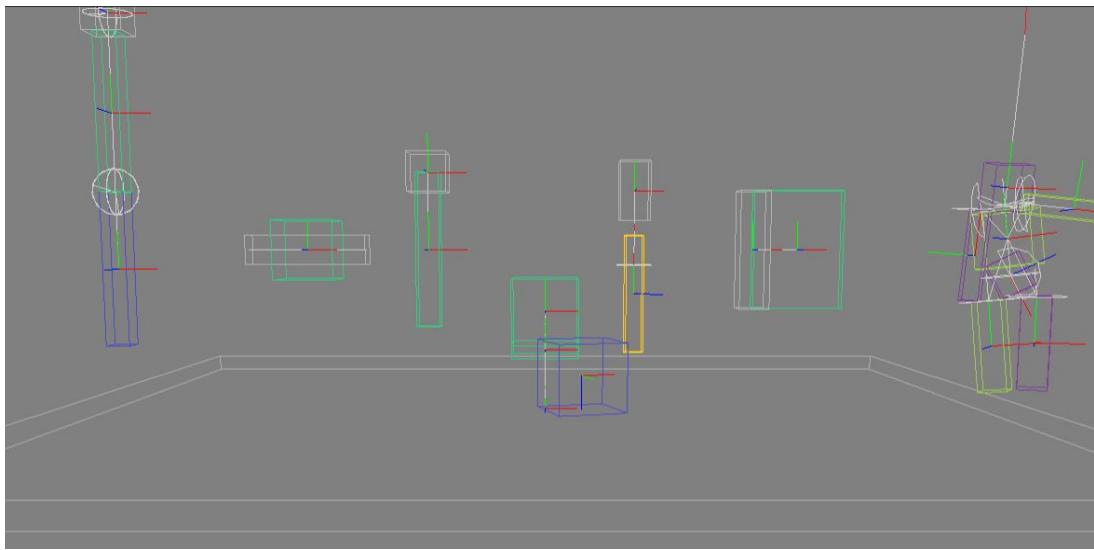


Abb. 25 - Ergebnis eines visuellen Debuggings
Eine Szene mit verschiedenen Gelenken und Körpern, im nur Physik Objekte Drahtgestell Modus

Daher besteht hier die Integration aus der Implementierung der Zeichenbefehle und ihrer Anwendung. Zunächst stößt man hier auf ein Problem mit dem Paradigma unter dem die Integration geschah, dass FUDGE immer ohne das Laden von Oimo Physics nutzbar sein muss. Dies liegt daran, dass die Implementierung der Zeichenbefehle erfordert, die von *Oimo.DebugDraw* gelieferten Funktionen innerhalb einer Vererbung zu überschreiben. Allerdings würde eine von *DebugDraw* erbende Klasse automatisch etablieren, dass FUDGE die Klasse *Oimo.DebugDraw* bekannt sein muss.

Um dies zu umgehen wurde eine Besonderheit in Javascript ausgenutzt. Beliebige Funktionen können zur Laufzeit frei überschrieben werden, weil Javascript nicht typisiert und klassenbasiert ist.

Jede Funktion kann überschrieben werden mittels:

```
OIMO.DebugDraw.prototype.point = function (v: OIMO.Vec3, color: OIMO.Vec3) {
//Funktionsinhalt
}
```

Wodurch stattdessen die neu definierte Funktion verwendet wird.

Daher wurde nur ein System benötigt, welches:

- Die Punkt, Linien und Dreieck Funktionen von *DebugDraw* überschreibt.
- Dem WebGL Renderingcontext das richtige Zeichenprogramm mit kompilierten Vertex/Fragmentschader gibt
- Sog. WebGL Buffer kreiert und verwaltet. Ein Buffer ist eine Art Array das Daten über Vertices enthält, welche an die Grafikkarte geschickt werden können. Sie sind Teil des WebGL Standards.⁵⁸
- Die Daten aus *Oimo.DebugDraw* an den RenderingContext von FUDGE weitergibt

Da FUDGE ein eigens erarbeitetes Rendering System nutzt und die dafür nötigen Elemente in Komponenten wie z.B. *ComponentMaterial* verpackt, kann dieses System nicht ohne erhebliche Änderungen für die Verarbeitung der Debugging Daten genutzt werden. Also wurde entsprechendes Wissen über WebGL Rendering in Erfahrung gebracht.

Oimo Physics enthält in Haxe geschriebene Demos, in denen entsprechende Debugging Funktionalitäten genutzt und über eine einfache Rendering Struktur gezeichnet werden.⁵⁹ Aus den vorhandenen Elementen, die extra die Demos geschrieben worden sind, wurden Strukturen extrahiert, analysiert und übersetzt in Typescript um ähnliche Strukturen in FUDGE zu ermöglichen.

Anfangs wurde überlegt ob es einfacher wäre auf ein zweites HTML Canvas Element nur die Debugging Informationen zu zeichnen, dies hätte aber zur Folge keine Tiefenabgleiche mit der eigentlichen FUDGE Szene zu bekommen, was für mich persönlich nicht ausreichend gewesen wäre.

Die Schwäche der Implementation ist, dass WebGL Buffer zu Beginn erstellt werden, mit einer gewissen Größe, die teilweise nicht ausgefüllt wird, aber leere Array Teile trotzdem vorhanden sind, weshalb außerdem nur eine bestimmte Anzahl an Physik

⁵⁸ Api zu WebGL und entsprechende Erklärungen aus der erarbeitete Informationen entstammen.
<https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/bufferData>
<https://webglfundamentals.org/webgl/lessons/webgl-how-it-works.html>

⁵⁹ [7] Oimo Physics vgl. <https://github.com/saharan/OimoPhysics/tree/master/demos/src/demo> Es sind DemoRenderer, DemoGraphics, WebGLDebugGraphics, Index/Vertex Buffer als Referenz benutzt worden.

Objekten debugged werden können. Dies ist trotzdem performant, da leere Buffer in WebGL gesondert bearbeitet und daher ignoriert werden.

Die erarbeitete Struktur zur Implementation eines visuellen Debuggings geschieht hauptsächlich innerhalb *DebugPhysicsDraw.ts*. Für eine andere Engine könnten die erarbeiteten Strukturen verwendet werden, wobei die Art und Anzahl der *Point*, *Triangle*, *Line* Aufrufe selbst definiert werden müsste.

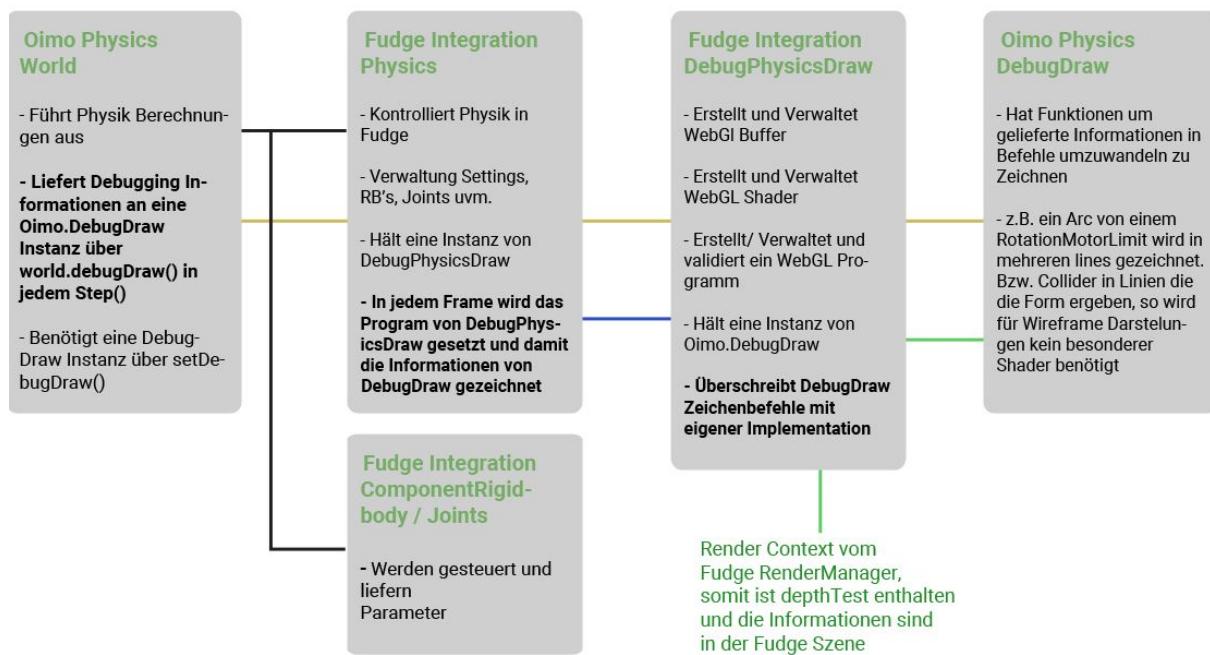


Abb. 26 - Übersicht der nötigen Struktur für ein visuelles Debugging mittels Oimo Physics

6.2 Anpassungen für die Nutzung im Editor

Trotz des Status des Editors, konnten einige Optimierungen umgesetzt werden, die eine Nutzung von Physik innerhalb des Editors möglich machen. Hierfür wurde direkt am Editor Code, der auf Electron basiert und damit Type-/Javascript ist, editiert und an allen Komponenten die innerhalb der Arbeit entstanden sind spezielle Änderungen unternommen.

Während diesen Unternehmungen fielen einige Probleme mit dem momentanen Status des Editors auf, die mit Prof. Dell'Oro-Friedl ausgetauscht wurden und die

Entwicklung des Editors vorangetrieben haben. Kleine Änderungen wie Dateitypen Filter beim Speichern/Laden für die von FUDGE verarbeiteten JSON Dateien, oder das Hinzufügen von Menüeinträgen wurden selbst erledigt.

Ein Punkt der zur Sprache kam und in Zukunft im Hinblick auf Scripting und Physik interessant wird, ist das Fehlen der Editor Funktion, eine Szene ohne Gameloop anzuzeigen und eine entsprechende Gameloop zu starten/stoppen. Dies würde der Physik und Scripten von Akteuren helfen, indem sie zuerst erstellt, eingestellt und dann gestartet werden können. Dies war aber nicht Teil der Arbeit und benötigt tiefergehende Änderungen an der Editor Struktur.

Speichern/Laden von Physik Komponenten:

Fudge speichert und lädt die Eigenschaften von einzelnen Komponenten innerhalb einer JSON Datei. Die Werte werden hierfür serialisiert und deserialisiert. Dies muss für jede Komponente gesondert übernommen werden, aber wird in einer Datei zusammengefasst. Damit die Physik tatsächlich in FUDGE integriert ist, muss sie auch entsprechend im Editor nutzbar sein, was hierdurch ermöglicht wird.

Das Format sieht hierbei vor:

```
{  
  "f.Node": {    //Ein Knoten Objekt - Hier das Root Objekt  
    "name": "AxisCross",      //Identifizierbarer Name  
    "components": {},        //Die einzelnen Komponenten  
    "children": [  
      { "f.Node": {    //Eine tatsächlich genutzte Node  
        "name": "RedFalling",  
        "components": {  
          "ComponentRigidbody": [ //Name der Komponente  
            {  
              "f.ComponentRigidbody": { //Die entsprechende Komponente  
                "pivot": { //Basis der Komponente lokal zum Knoten Ursprung  
                  "translation": { //Komponenten können verschoben werden  
                    "x": 0, "y": 0, "z": 0  
                  },  
                  "rotation": {  
                    "x": 0, "y": 0, "z": 0  
                  }  
                }  
              }  
            }  
          }  
        }  
      }  
    ]  
  }  
}
```

```

        },
        "scaling": { //So kann ein visuell 1m großer Würfel
            "x": 1, //Physikalisch z.B. 2 Meter groß sein
            "y": 1, //D.h. visuell wird er nie etwas berühren
            "z": 1
        },
    },
    //Identifikationsnummer dieses Körpers um zwei klar identifizierbare Körper
    //beim Laden wieder durch ein Gelenk verbinden zu können
    "id": 15,
    "restitution": 1.5, //Restenergie bei Aufprall
    "physicsType": 0, //Art der Interaktion 0 = Dynamisch
    // Es gibt noch mehr Werte, aber es müssen nie alle enthalten sein im JSON
    //Fehlende werden einfach auf den Standardwert gesetzt
    "Component": {
        "active": true
    }
},
//Ergänzt durch weitere Komponenten wie Mesh, Material, Transform
},

```

Dieses Beispiel nimmt die entsprechenden Ergänzungen bereits vorneweg. Es gilt für die Serialisierung/Deserialisierung, dass nur einfache Datentypen gespeichert werden können, d.h. Werte wie Number, Boolean.

Schritt 1 war entsprechend alle Komponenten durchzugehen und Werte so zu hinterlegen, dass sie einfachen Datentypen entsprechen, oder komplexe zu serialisieren/deserialisieren, um aus ihnen einfache Datentypen zu machen. Hierbei fiel auf, dass Gelenke Vektor3 Eigenschaften für z.B. Verankerung verwenden und diese bisher noch nicht, wie Matrix4x4 für Transformationen, deserialisiert wurden. Daher wurde diese Funktionalität ergänzt, über das Mutatoren Prinzip von FUDGE, analog zu Matrix4x4.

```

public deserialize(_serialization: Serialization): Serializable {
    this.mutate(_serialization); //Erhaltene Serialisierung
    return this;
}

```

```
//Interne Werte werden über ein Mutator Assoziatives Array gesetzt, was
Akteuren und dem Editor ermöglicht ohne zu wissen wie Eigenschaften heißen
über assoziierte Strings zu verändern.

public mutate(_mutator: Mutator): void {
    let _x: number = <number>_mutator["x"];
    let _y: number = <number>_mutator["y"];
    let _z: number = <number>_mutator["z"];
    this.x = _x; this.y = _y; this.z = _z;
}
```

Schritt 2 ist die (De-)Serialisierung der Komponente. Das geschieht im Code der einzelnen Komponente. Dieser Vorgang nimmt alle Parameter und verpackt sie, bzw. bekommt alle Parameter und setzt Werte auf die geladenen Werte. Hierbei galt zu beachten, die Standardwerte zu setzen, falls im geladenen JSON kein Wert vorhanden ist und die Werte über publike Getter/Setter zu setzen, sofern diese, außer der Veränderung des Wertes, weitere Funktionalitäten besitzen.

Ein Beispiel: Sobald die Physik-Gruppe in der sich der Körper befindet geändert wird, sollte überprüft werden ob es sich um einen Trigger handelt, ist dies der Fall, muss er in der Physik Verwaltung bekannt gemacht werden. Die Physik-Gruppe, ist daher ein privater Wert, der über Get/Set gesetzt wird und eine Überprüfung der Art des neuen Wertes beinhaltet. Daher muss dies auch bei der Deserialisierung geschehen.

Eine solche (De-)Serialisierungs Funktion sieht entsprechend folgendermaßen beispielhaft, vereinfacht, für *ComponentRigidbody*, aus.

```
public serialize(): Serialization {
    let serialization: Serialization = {
        pivot: this.pivot.serialize(), //Matrix4x4 wird gesondert serialisiert
        in lesbare einzelwerte wie "translation"[ "x" : 5, etc.]
        physicsType: this.rbType,
        mass: this.massData.mass;
    //Struktur NameImJson: this.eigenschaft,
    //        [super.constructor.name]: super.serialize()
    };
    return serialization;
}

public deserialize(_serialization: Serialization): Serializable {
```

```

        this.pivot.deserialize(_serialization.pivot);
        this.physicsType = _serialization.physicsType;
//Struktur Wert = Serialisierung.NameImJson != NichtVorhanden ? WennJa :
WennNein (Standardwert) - Verkürzte If Bedingung, falls Daten nicht vorhanden
        this.mass = _serialization.mass != null ? _serialization.mass : 1;
        super.deserialize(_serialization[super.constructor.name]);
        return this;
    }
}

```

Diese Struktur musste noch erweitert werden um eine ID, als klar wurde auch Gelenke müssen gespeichert und geladen werden, damit zwei Körper wieder identifiziert werden können, auch wenn ihr Name gleich. Deshalb wurde eine ID gebende Funktion in die *Physics.ts* eingebaut, welche eine freie ID sucht und vergibt, bei der Rigidbody Erstellung. Gelenke bekamen eine spezielle Funktion, die bei Serialisierung prüft, ob zwei Körper gefunden wurden und diese dann als Körper festlegt, die im nächsten Frame miteinander verbunden werden sollen.

```

public deserialize(_serialization: Serialization): Serializable {
    super.idAttachedRB = _serialization.attID;
    super.idConnectedRB = _serialization.conID;
    if (_serialization.attID != null && _serialization.conID != null)
        super.setBodiesFromLoadedIDs(); //Rest der Deserialisierung...
}

```

Schritt 3 war die entsprechenden Funktionalitäten zu testen. Hierbei fiel auf, für ein gutes Nutzererlebnis wird benötigt, dass man die visuellen Debugging Werkzeuge auch im Editor zur Verfügung hat, es war sonst schwer abzuschätzen, ob z.B. Achsenrichtung und Verankerung von Gelenken geladen wurden.

Debugging im Editor:

Um die Analyse Funktionalität auch im Editor nutzbar zu machen, wurden neue Menüelemente erstellt, die mit FUDGE kommunizieren. Hierbei muss beachtet werden: Electron Anwendungen bestehen aus Code mittels dem die Anwendung erstellt wird (*Main.ts*) und Code der zur Laufzeit verwendet wird (*Fudge.ts*). Diese können nicht ohne Umwege miteinander kommunizieren, was die Erstellung von

visuellen Toggles erschwert.⁶⁰ Allerdings ist im FUDGE Editor bereits eine einseitige Kommunikation zwischen Anwendung und Laufzeit Code möglich.

Hierfür wurde das Menü erweitert, um entsprechende Knöpfe, die den Physik Debugging Modus der implementiert wurde ein-/auszuschalten, bzw. zwischen den Modi zu wechseln.

In Electron sind die Menüs in einer JSON artigen Verschachtelung aufgebaut. Für die visuelle Erweiterung müssen nur Einträge ergänzt werden. Das Untermenü Debug, was bereits die Browser Konsole öffnet, musste entsprechend erweitert werden. Um es kurz zu halten ein Eintrag besteht aus.

```
{ label: "Physic Debug View", id: String(MENU.PHYSICS_DEBUG), click:  
  menuSelect, accelerator: "CmdOrCtrl+P" },
```

Eigenschaften, wie der Benennung und einer ID wodurch übergeben wird welcher Knopf betätigt wurde, sowie einer Click Funktion, die entsprechende Funktionalität auslöst. Zudem kann ein “Accelerator” bzw. Tastaturkürzel ergänzt werden, welches Möglichkeiten besitzt direkt für Windows und Mac angepasst verwendet zu werden, was aus Optimierungsgründen natürlich geschehen ist. Um ein Menü weiter zu verschachteln kann die Eigenschaft submenu: [], verwendet werden.

Der erstellte Knopf findet seine Funktion über eine spezielle Funktion, die eine Identifikation ausführt und Informationen an den WebContent weiterreicht. Dies ist eine Electron Besonderheit. Der Laufzeit Code nennt sich WebContent, da es eine Ausführung eines Browsers ist und dieser kann Daten empfangen über:

```
_window.webContents.send(_message, _args);
```

Der Electron Rendering Anteil, innerhalb von *Fudge.ts*, sog. *Electron.ipcRenderer* kann nun die gesendeten Events verarbeiten. Diese bereits bestehende Funktionalität wird entsprechend genutzt um die Variablen innerhalb der Physik Integration zu verändern.

```
ipcRenderer.on("togglePhysicsDebugView", (_event: Electron.IpcRendererEvent,  
_args: unknown[]) => {
```

⁶⁰ <https://stackoverflow.com/questions/47756822/change-electrons-menu-items-status-dynamically>
Es wäre eine Rekreation des Menüs nötig und dies ist nicht in FUDGE vorgesehen.

```
f.Physics.settings.debugDraw = !f.Physics.settings.debugDraw;});
```

Ergebnis dieser Erweiterungen, durch das Analysieren und Ergänzen von FUDGE Editor Strukturen, ist eine Physik Debugging Editor Ansicht, die dazu verwendet wurde zu testen, ob Komponenten korrekt geladen werden und in Zukunft Akteuren eine gute Analyse ihrer physischen Szenen ermöglicht.

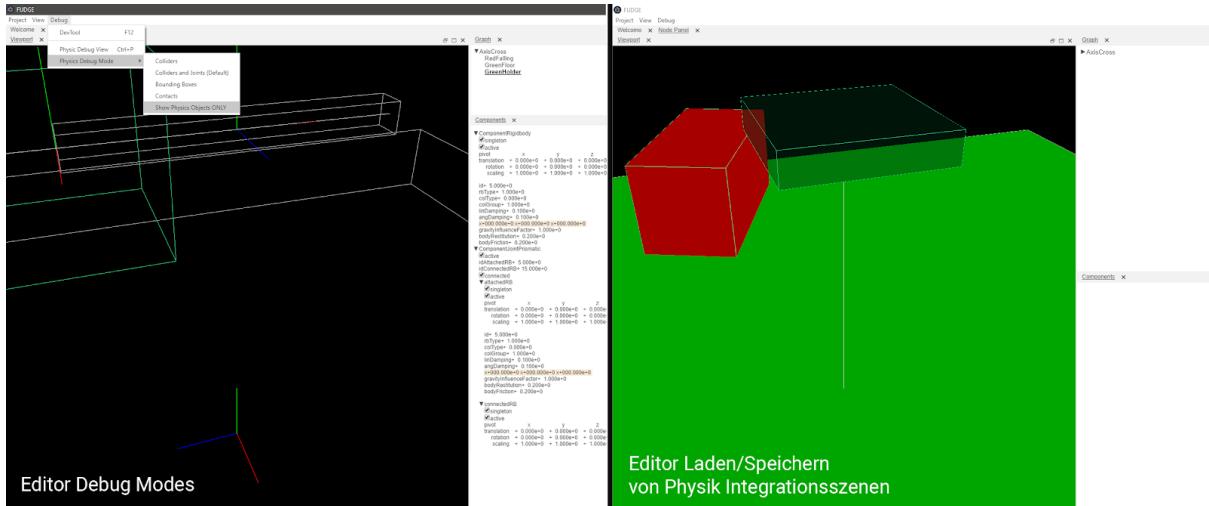


Abb. 27 - Der Editor in seinem finalen Status innerhalb dieser Arbeit. Zu sehen das Physik Debugging und eine geladene Szene, mit einem Schieberegler-Gelenk (links) und einem gefederten Gelenk (rechts)

7. Erstellung Lernmaterial zum Umgang mit der Physik in FUDGE

Von großer Wichtigkeit, im Nutzungskontext von FUDGE, ist das Onboarding der lernenden Akteure, für Funktionsweisen von FUDGE, der Spieleentwicklung und Engine Konzepten. Die Integration einer Physik bringt nur etwas, wenn sie auch mit einer möglichst kleinen Hürde verwendet und vor allem verstanden werden kann, aber genug Tiefe bietet damit einige Anwendungsfälle abgedeckt werden können.

Hierfür muss ein Onboarding vorhanden sein, was über einfache Funktionsbeschreibungen durch Definition Files hinausgeht. Deswegen werden hierfür erklärende Tutorials, begleitet durch verschiedene Physik Anwendungen, erstellt.

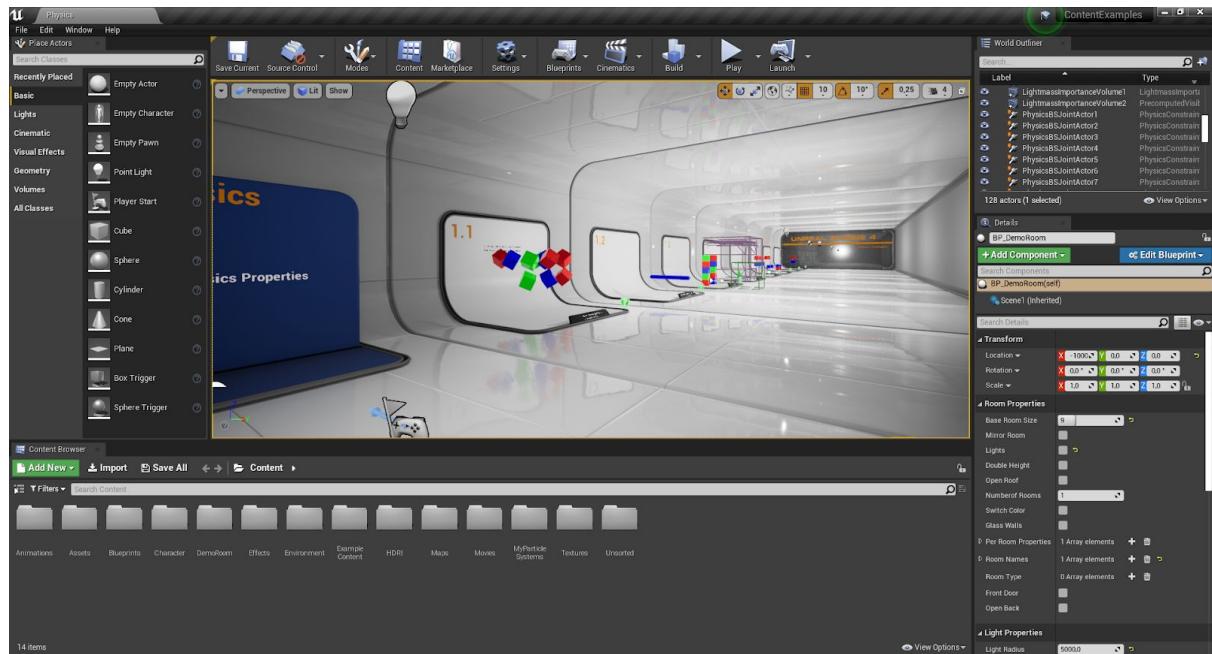
Allerdings gibt es wenige Standards für den Aufbau von Tutorials und diese lassen sich hauptsächlich über Popularität feststellen. Noch weniger gibt es diese

entsprechend für ein Onboarding von Physik Engines. Allgemein ließ sich eine populäre Struktur in Tutorials ableiten, bestehend aus:

Titel, nötiges Vorwissen oder Projektdateien, einer Kurzzusammenfassung und dem eigentlichen Inhalt. Häufig noch ein Hinweis, der zur tiefergehenden Erkundung einlädt, oder darauf den Autor des Tutorials für Fragen anzuschreiben.

Daher wurde auch auf die Onboardings der beiden bereits beschriebenen Hauptvertreter im Bereich frei zugängliche Game Engines, Unity und Unreal, zurückgegriffen.

Es lässt sich von diesen ableiten, dass Unity mit klaren Videotutorials, unterstützt durch Code arbeitet, die einfache Beispiele für ganz bestimmte Funktionen abdecken.⁶¹ Unreal hingegen besitzt weniger klassische Tutorials, als viel mehr erklärende Texte, in Form einer gut ausgearbeiteten Dokumentation, die neben Code, hauptsächlich, im für Einsteiger freundlichen Blueprint Format (visuelles Programmieren), erweiterte Erklärungen zur Funktionsweise liefern.⁶² Zudem bietet Unreal bei der Installation direkt ein Projekt für jeden Teilbereich an, unter anderem die Physik, in der eine Szene mit allen Grundfunktionen enthalten ist und diese direkt beeinflusst werden können.



⁶¹ [15] Unity Engine, Learn Plattform seit 2020 kostenfrei, zuvor ebenso kleine Videos zu spezifischen Funktionen innerhalb der Dokumentation

⁶² [16] Unreal Engine, Tutorials sind in die Dokumentation verwoben

Abb. 28 - Unreal Engine - Ein Raum indem sich Physik Funktionalitäten befinden und deren Werte direkt beeinflusst werden können und Infos neben der Funktionalität eingeblendet werden.

Beide einigen sich auf, die auch durch die ergänzende Theorie herausgearbeiteten, wichtigsten Aspekte der Physik, Körper, Kräfte/Impulse, Raycast, Events und Gelenke. In Unreal werden diese sogar entsprechend als "*Essentials*" bezeichnet. Es wird immer die Basis erklärt und die Entwickler sollen selbst über Definitionen herausfinden wie einzelne Elemente genau verwendet werden.

Das Onboarding für FUDGE sollte hier allerdings mehr bieten, da es sich um einen Nutzungskontext in der Lehre handelt, mit Einsteigern in die Spieleentwicklung selbst, bzw. in den Physik-Aspekt dieser. Deshalb muss das Lernmaterial noch mehr Erklärungen abseits der Funktionsweisen in FUDGE bereitstellen. D.h. Informationen über verschiedene Anwendungsbereiche der Funktionalitäten und den Aufbau von Physik in Game Engines.

Deshalb wurde entschieden, fünf Tutorials in Textform zu verfassen, die mittels, einer Kombination aus beschreibendem Text, der durch Bilder von Phänomenen unterstützt wird, Vorgänge, Gründe und Anwendungsfälle erklärt. Außerdem wird mit gekennzeichneten Code Blöcken die spezifische Anwendung innerhalb von FUDGE aufgezeigt.

Begleitet werden diese Texte von fertigen Szenen, die innerhalb der Texte aufgegriffen werden, die von Akteuren selbst bearbeitet und verstanden werden können, diese sind speziell kommentiert.

Sie sollen ähnlich des Onboardings von Unreal sein und bieten die Möglichkeit zum Ausprobieren, durch die Akteure, ohne vorher etwas gelesen zu haben, durch kleine Veränderungen von Werten im Code. Neben den essentiellen Funktionalitäten der Festkörperphysik, wird auch der Gebrauch von Einstellungen der Simulation und des visuellen Debugging Features erklärt, so dass Akteure nicht nur Anwendungen erstellen können, sondern auch untersuchen und analysieren.

Die Erklärungstexte und Code-Kommentare wurden in englischer Sprache verfasst, da dies im Rahmen von FUDGE üblich ist. Vorgänge wurden dabei simplifiziert, um

Nutzung durch sämtliche Akteure zu ermöglichen.

Damit Akteure direkt in die Physik einsteigen können wurde noch eine Physik-Boilerplate erstellt, eine einfache FUDGE Szene die nichts außer einen bestehenden Körper und einer Funktion zum Erstellen von Körpern bietet. Dies gibt Akteuren die Möglichkeit sich auf die neuen Funktionalitäten der Physik zu konzentrieren, ohne selbst eine FUDGE Szene kreieren zu müssen. Wobei das selbstverständlich Wissen ist, welches vorausgesetzt wird. Alle Tutorials fokussieren sich auf den neuen Physik-Aspekt von FUDGE, deshalb wurden keine FUDGE UI Elemente, oder anderen vorhandenen Strukturen verwendet de nicht unbedingt nötig sind, da sie sonst extra erlernt werden müssten.

Für eine Anwendung dieser Physik Features, innerhalb eines Spiels oder einer Simulation, sind noch zwei Szenen bereitgestellt, in denen einmal der grundlegendste Anwendungsfall eines 3D Platformer Charakters dargestellt wird, also ein Charakter der die Fähigkeit besitzt sich durch eine Welt zu bewegen mit der er kollidiert. Er kann springen, sich drehen und interagieren.

Zudem ein komplettes Spiel in Form eines kleinen Korbwurf Spiels, bestehend aus einem Basketball und extra Zielen, die an Gelenke geheftet sind. Es wurden alle Funktionalitäten die integriert sind verwendet, innerhalb von diesen Beispielen, oder in entsprechenden Experimenten, die im FUDGE *Miscellaneous/Experiments/Marko* Ordner zu finden sind.

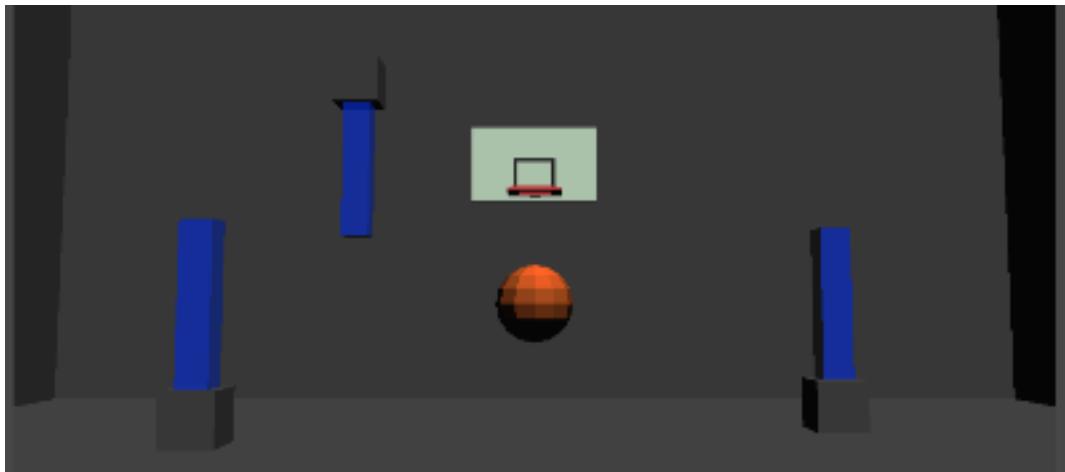


Abb. 29 - Ein einfacher Korbwurf der Maus Richtung und Bewegungsstärke nimmt um einen Ball in den Korb oder auf Gelenk-Ziele zu werfen um Punkte durch Kollisions/Trigger Events zu erzielen

Bei der Erstellung dieser Tutorials wurden alle Features noch einmal durchgetestet, um etwaige Fehler zu finden, nachdem diese bereits in Experimenten getestet wurden. Es handelt sich hier um eine zweite Iteration, die zudem noch den Vorteil bot festzustellen ob die integrierten Strukturen sich problemlos einsetzen und einfach erklären lassen.

8. Schlussbetrachtung

Abschließend lässt sich feststellen, dass die meisten webbasierten Physik Engines nicht so ausgereift sind wie im klassischen Game Engine Bereich und daher mit einigen Nachteilen verbunden sind. Letztendlich lässt sich auch feststellen, dass Oimo Physics Schwächen in der Qualität der Simulation besitzt, welche aber von geringerer Natur sind. Außerdem sind andere Physik Engines ebenso mit Problemen behaftet, oder bieten weniger. Dennoch bietet die gewählte Oimo Physics alles Nötige für die Lehre und durch die Integration kann FUDGE durchaus mit den meisten derzeitigen HTML5 Game Engines mithalten, obwohl dies nicht der Fokus der Arbeit war.

Die Integration war mit einigen Problemen verbunden, aufgrund spezieller Vorgehensweisen und verschiedener Ansätze in den beiden Systemen, aber gelang letztendlich, über mehrere Iterationen, sehr gut und fast alle Funktionen die Oimo Physics bietet können in FUDGE nahtlos verwendet werden. Eine große Rolle spielte

hier das Vorgehen behutsam mehr Funktionalitäten nach und nach zu ermöglichen und konstant einige Anwendungsfälle zu testen.

Es wurden gute Strukturen geschaffen, die in weiten Teilen nicht direkt Oimo spezifisch sind, so dass man mit deutlich geringerem Aufwand eine andere Physik Engine integrieren könnte, sollten in Zukunft Funktionalitäten, wie Meshcollider für konkave Körper, oder Terrain Meshes, sowie Softbody Physik, nötig sein.

Aufgrund der Entscheidung Rigidbody und Collider, als Vereinfachung, in einer Komponente zusammenzufassen, sowie der Funktionsweise von OIMO sind Compound Shapes leider zeitlich, in dieser Iteration, nicht umzusetzen gewesen, daher können nur konvexe Collider, die eine ungefähre Form der Compound Shape darstellen, als workaround genutzt werden.

Der Editor von FUDGE war zum Zeitpunkt der Arbeit noch recht am Anfang und konnte daher nur soweit wie es möglich war bearbeitet werden, um die Physik dort ebenso zu unterstützen. Leider ist es nicht direkt möglich Physik-Szenen im Editor ordentlich zu bearbeiten. Dennoch wurde auf optimierte Nutzerinteraktion geachtet und sich an Beispielen großer Game Engines orientiert, so dass eine gewohnte und komfortable Programmierstruktur gewährleistet ist.

Es ist gelungen genügend Anwendungsfälle als Lernmaterial, oder als Experimente, abzudecken, um die Integration auf ihre wichtigen Bestandteile zu testen und Nutzern einen optimalen Einstieg in die Welt der Physik Spielereien zu ermöglichen.

Glossar

Begriff	Erklärung
Akteur	Agierende Person innerhalb eines besonderen Kontextes. Für diese Arbeit hauptsächlich die Lernenden innerhalb des FUDGE Kontextes.
API	Application Programming Interface - Eine vordefinierte Kommunikations- Schnittstelle zwischen Anwendungen/ Bibliotheken bzw. dem Programmierer und der Software
Body / Rigidbody	Ein Körper bzw. Festkörper, Begriff um einfach zu unterscheiden ob es sich um ein Objekt, oder eben ein physikalisches Objekt handelt, das eine wirkende Form und Masse besitzt und somit ein Körper ist.
Boilerplate	Beschreibt einen Text bzw. Element was auch außerhalb eines spezifischen Kontextes wiederverwendet werden kann. So wie die Fudge-Physics Boilerplate, die außerhalb von den Tutorials verwendet werden kann um FUDGE Spiele mit Physik zu erstellen.
Collider	Geometrische Form die den Schwerpunkt eines Objektes angibt und den Raum der physisch eingenommen wird.
Debugging / Debug	dt. Fehlerbehebung. Beschreibt allerdings darüber hinaus die Auseinandersetzung mit dem Programmcode, in Form von Analyse, die zur Fehlerbehebung dienen kann, aber auch zur Verbesserung oder besseren Darstellung zur Entwicklungszeit. Wird im Standardfall nur vom Entwickler gemacht.
Definition File	Definitions Datei, in Typescript eine Datei die automatisch die Definition und eine kurze Erklärung zur ausgewählten Funktion/Variable/Klasse während dem Schreiben liefert.
Engine	Eine Software - stellt meist ein zentrales, unabhängiges System aus Funktionen und Modulen dar, um bestimmte Aufgaben zu erfüllen. <i>Bsp. Physik-Engine, ein abgeschlossenes System für Physik Berechnungen</i>
Framework	Software Strukturen die als Gerüst dienen um etwas bestimmtes zu bauen. Z.B. Physik Framework, ein Gerüst welches eingebaut und verwendet werden kann um Physik darzustellen. Abgrenzung von Engines unscharf, beide Begriffe oft synonym verwendet. Wobei Frameworks typischerweise kleiner und weniger ausgereift sind.
Lightweight	dt. Leichtgewicht, wenn eine Software Grundfunktionalitäten für die wichtigsten Anwendungsfälle enthält und nicht mit zusätzlichen Features oder Komfortfunktionen aufgebläht ist.
Masse	Ist die theoretische Menge an Materie eines Körpers und somit der Widerstand, der Bewegung beeinflusst. Vereinfacht auch als Gewicht bezeichnet.

Namespace	Codestruktur die es ermöglicht einen abgeschlossenen Bereich zu definieren, in dem bestimmte Namen und Variablen vorkommen. So können bei Mehrfachbenennung zwischen Namespaces Konflikte verhindert werden, weil jeder Name einem Bereich zugeordnet werden kann.
Onboarding	Beschreibt den Vorgang, jemanden auf etwas hinzuweisen bzw. für etwas zu interessieren und es demjenigen zugänglich zu machen.
Rendering	Die visuelle Darstellung Mathematischer Objekte, anhand der definierten Parameter. Geschieht über Vertex/Fragment Shader (Programme die für die Grafikkarte definieren: Wie Punkte im Raum angeordnet sind. / Pixel, die zwischen den Punkten eingeschlossen sind, eingefärbt werden.)
Root	dt. Wurzel, ist die Basis bzw. Start einer Szene, an die neue Knoten bzw. Objekte angeheftet werden. Transformationen dieses Objektes werden nicht verändert, aber es wird als Basis von Rendering und Physik Berechnungen verwendet.
Solver	Problemlösungs-Algorithmus, häufig verschiedene Algorithmen zur Lösung desselben Problems, mit Ansätzen zur Akkuratheit oder Geschwindigkeit als Trade-Off.
WebAssembly	Vereinfacht, eine Möglichkeit mit sog. Binary Files, hoch performanten Code, in Browsern laufen zu lassen, der nativen Einbindungen nahe kommt und sich über Javascript steuern lässt.
Wrapper	Eine Verpackungs-Struktur, die eine Funktionalität einwickelt, um sie für Nutzer entweder lesbarer, komfortabler, oder überhaupt nutzbar zu machen.

Abbildungsverzeichnis

Abb. 1 - Stand des FUDGE Editors ohne Physik Integration. (Quelle: Eigene Darstellung, FUDGE Editor, April 2020).....	S.10
Abb. 2 - Collider geometrische physik Barrieren die visuelle Objekte umschließen (Quelle: Eigene Darstellung).....	S.14
Abb. 3 - AABB's die Körper als Vereinfachung repräsentieren. (Quelle: [3] Game Engine Architecture Kap. 12.3).....	S.16
Abb. 4 - AABB vs. AABB Kollisionstest (Quelle: [3] Game Engine Architecture Kap. 12.3).....	S.18
Abb. 5 - GJK Algorithmus Konzepte. Finden der Minkowski Differenz. Anschließend Support Vertex Suche bis ein Tetrahedron entsteht. (Quelle: [3] Game Engine Architecture Kap. 12.3.5).....	S.19
Abb. 6 - Datenfluss Darstellung innerhalb einer Physik Engine die Game Engine benötigt davon nur die Ergebnisse innerhalb des Rigidbody (Quelle: [2] Game Physics Development S.365).....	S.21
Abb. 7 - Ein einfacher Schieberegler, bei dem der grüne Körper nur der Achse des orangenen Körpers folgen kann. Eine Feder erlaubt sich gedämpft zu bewegen. (Quelle: Eigene Darstellung).....	S.24
Abb. 8 - Oimo Physics - Fallende Mauer aus einzelnen Objekten Test (100 Rigidbodies) (Quelle: Eigene Darstellung).....	S.34
Abb. 9 - Mauer Ergebnisse aller getesteter Physik Engines. (Quelle: Eigene Darstellung).....	S.35
Abb. 10 - Versuchsaufbau - Limit Tests (Quelle: Eigene Darstellung).....	S.36
Abb. 11 - Physik Elemente in Unity, deren bestimmte Eigenschaften und Darstellung (Quelle: Unity Engine Editor Version 2020.1).....	S.38
Abb. 12 - Darstellung und Eigenschaften von Körpern und Gelenken im Unreal Editor (Quelle: Unreal Engine Editor Version 4.25).....	S.43
Abb. 13 - Rigidbodies in PlayCanvas	

(Quelle: Playcanvas Editor Version Mai 2020).....	S.44
Abb. 14 - Mögliche Physik Objekte im PlayCanvas Editor	
(Quelle: Playcanvas Editor Version Mai 2020).....	S.45
Abb. 15 - Vereinfachter Informationsfluss und Verwaltung der Integration	
(Quelle: Eigene Darstellung).....	S.49
Abb. 16 - Überblick über Körpereigenschaften und den Unterschied zwischen Physik und Rendering/Transformationswelt	
(Quelle: Eigene Darstellung).....	S.50
Abb. 17 - Raycast Funktionsweise, Überblick	
(Quelle: Eigene Darstellung).....	S.58
Abb. 18 - Überblick der Gelenk Integration	
(Quelle: Eigene Darstellung).....	S.65
Abb. 19 - Abb. 24 - Modellierte Darstellung von Gelenk Verwendungszwecken	
(Quelle: Eigene Darstellung).....	S.68ff
Abb. 25 - Ergebnis eines visuellen Debuggings, mit allen integrierten Funktionen	
(Quelle: Eigene Darstellung, FUDGE Szene).....	S.74
Abb. 26 - Übersicht der nötigen Struktur für ein visuelles Debugging mittels Oimo Physics	
(Quelle: Eigene Darstellung).....	S.76
Abb. 27 - Der Editor in seinem finalen Status innerhalb dieser Arbeit	
(Quelle: Eigene Darstellung, FUDGE Editor).....	S.82
Abb. 28 - Unreal Engine - Onboarding Physik mit interaktiver Szene	
(Quelle: Unreal Engine Editor Version 4.25).....	S.83
Abb. 29 - Ein einfaches Korbwurf-Spiel als Demonstration	
(Quelle: Eigene Darstellung, FUDGE Szene).....	S.86

Quellenverzeichnis

Literaturverzeichnis

[1] David H. Eberly - 3D Game Engine Architecture - 2013 - CRC Press

ISBN 978-0-12-229064-0

[2] Ian Millington - Game Physics Engine Development, 2nd Edition - 2010 -

CRC Press

ISBN 978-0-12-381976-5

[3] Jason Gregory - Game Engine Architecture, 2nd Edition - 2017 -

A K Peters/CRC Press

ISBN 978-1-4665-6001-7

[4] Comparison of Physics Frameworks for WebGL-Based Game Engine - Resa Yogya

/ Raymond Kosala - February 2014 The European Physical Journal Conferences 68

DOI [10.1051/epjconf/20146800035](https://doi.org/10.1051/epjconf/20146800035)

Webquellen

[5] FUDGE - Prof. Jirka Dell'Oro Friedl

<https://github.com/JirkaDellOro/FUDGE> (Zuletzt aufgerufen 16.01.2020)

[6] Oimo.js Physics Lightweight Engine

<https://github.com/lo-th/Oimo.js/> (Zuletzt aufgerufen am 01.03.2020)

[7] OimoPhysics : Physics Lightweight Engine (Basis Engine für Oimo.js)

<https://github.com/saharan/OimoPhysics> (Zuletzt aufgerufen am 16.06.2020)

[8] Energy.js Physics Engine / Samples

<https://github.com/samuelgirardin/Energy.js> (Zuletzt aufgerufen am 01.03.2020)

<http://www.visualiser.fr/page.php?id=Energy.js>

[9] Cannon.js Physics Engine

<https://schteppe.github.io/cannon.js/> (Zuletzt aufgerufen am 01.03.2020)

[10] Ammo.js Physics Engine

<https://github.com/kripken/ammo.js> (Zuletzt aufgerufen am 01.03.2020)

[11] Dapao Physics Engine

<https://github.com/TheRohans/dapao> (Zuletzt aufgerufen am 28.07.2020)

[12] Goblin Physics | Javascript Physics Engine

<http://www.goblinphysics.com/> (Zuletzt aufgerufen am 27.07.2020)

[13] Physx-JS | Javascript Port der Nvidia Physx Physik Engine

<https://github.com/ashconnell/physx-js> (Zuletzt aufgerufen am 16.06.2020)

[14] Babylon.js Physics Engine Integration

https://doc.babylonjs.com/how_to/using_the_physics_engine

(Zuletzt aufgerufen am 01.03.2020)

[15] Unity Engine

<https://unity.com/> (Zuletzt aufgerufen am 06.03.2020)

<https://learn.unity.com/tutorial/3d-physics#>

Physik Tutorials (Zuletzt aufgerufen am 28.07.2020)

[16] Unreal Engine

<https://www.unrealengine.com/en-US/> (Zuletzt aufgerufen am 10.03.2020)

<https://docs.unrealengine.com/en-US/Engine/Physics/index.html>

Physik Tutorials bzw. erklärende Doku (Zuletzt aufgerufen am 29.07.2020)

[17] PlayCanvas Entwicklung GitHub

<https://github.com/playcanvas/engine> (Zuletzt aufgerufen am 03.05.2020)

<https://developer.playcanvas.com/en/user-manual/physics/calling-ammo/>

(Zuletzt aufgerufen am 17.06.2020)

[18] A Comparison JavaScript Physic Engines | 2012

<https://www.webappers.com/2012/12/11/a-comparison-of-javascript-physics-engines/>

(Zuletzt aufgerufen am 16.05.2020)

[19] Quaternions | Why and how to implement

<https://www.gamedev.net/tutorials/programming/math-and-physics/do-we-really-need-quaternions-r1199/> (Zuletzt am 15.06.2020)

<https://stackoverflow.com/questions/11492299/quaternion-to-euler-angles-algorithm-how-to-convert-to-y-up-and-between-ha> (Zuletzt am 15.06.2020)

<https://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToAngle/index.htm> (Zuletzt am 15.06.2020)

[20] Building a Collision Engine Part 1: 2D

<https://blog.hamaluk.ca/posts/building-a-collision-engine-part-1-2d-gjk-collision-detection/>
(Zuletzt aufgerufen am 01.08.2020)

[21] Video Game Physics Tutorial - Part III: Constrained Rigid Body Simulation

<https://www.toptal.com/game/video-game-physics-part-iii-constrained-rigid-body-simulation>
(Zuletzt aufgerufen am 01.08.2020)

[22] Erin Catto - Modelling And Solving Constraints - Games Developer Conference 2009 - Blizzard Entertainment

http://twvideo01.ubm-us.net/o1/vault/gdc09/slides/04-GDC09_Catto_Erin_Solver.pdf
(Zuletzt aufgerufen 03.08.2020)

Versionsinfos über verwendete Programme und Sprachen

Visual Studio Code (Version 1.43 - 1.47)

FUDGE (Stand Februar-Juli, Grundlegender Funktionsumfang für Spieleentwicklung vorhanden, faktisch unterliegen viele Funktionalitäten stetiger Entwicklung)

FUDGE Editor (Stand März-Juli, erste Verwendung möglich, Speichern und Laden von Daten einzelner Komponenten sowie grundlegende Anzeige und Interaktion möglich)

TypeScript Version 3.8.3 / Javascript (ES6 - ECMAScript 2015)

Google Chrome (Version 81 - 84)

Blender 2.82 - Erstellung von visuellen Darstellungen

Eidesstattliche Erklärung

Hiermit erkläre ich, dass die vorliegende Masterthesis von mir selbstständig verfasst wurde. Außer den angegebenen Quellen und Hilfsmitteln sind keine weiteren verwendet worden. Gedankengut, welches aus fremden Quellen direkt oder indirekt übernommen wurde, ist entsprechend mit gebührender Sorgfalt als solches kenntlich gemacht. Die Arbeit wurde bisher in keiner vergleichbaren Form einem anderen Prüfungsamt vorgelegt. Ich erkläre mich damit einverstanden, dass diese Arbeit mit Hilfe einer Plagiaterkennungs-Software überprüft wird.

24.08.2020 Vöhrenbach

(Datum, Ort)



(Unterschrift)

Anhang

Datenträger mit:

- Versionen von FUDGE während des Bearbeitungszeitraums
- Version von verwendeten Physic Engines während des Bearbeitungszeitraums
- Klassendiagramm der Hauptstruktur
- Bearbeitete FUDGE Dateien und eigenen Dateien
- Tutorials zur Physik Integration
- Beispiele von Physik in FUDGE
- Gesprächsnotizen
- Digitale Fassung der Thesisarbeit