HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Abschlussarbeit

in

Allgemeine Informatik B. Sc.

Skelett-System für FUDGE

Referent:	Prof. Jirka Dell'Oro-Friedl
Korreferent:	Prof. Dr. habil. Olaf Neiße
Vorgelegt am:	28.02.2022
Vorgelegt von:	Matthias Roming, 260614 Lienberg 60 78714, Schramberg matthias.roming@hs-furtwangen.de

Abstract

The Furtwangen Didactic Game Editor (FUDGE) is a game editor from Furtwangen University that students can use to intuitively learn how to develop games. Skeleton systems are used in games as a tool to make complex animations easier to define and render. In this work, the functioning and structure of existing skeleton systems are examined, it is clarified why the integration of such a system from another game engine is not practical and – derived from the research - a separate skeleton system for FUDGE is developed.

Der Furtwangen Didactic Game Editor (FUDGE) ist ein Game Editor der Hochschule Furtwangen, mit dem Studentis auf möglichst intuitiver Weise das Entwickeln von Spielen erlernen können. Skelett-Systeme werden in Spielen als Werkzeug verwendet, um komplexe Animationen leichter definieren und darstellen zu können. In dieser Arbeit werden die Funktionsweise und Struktur vorhandener Skelett-Systeme untersucht, klargestellt warum die Integration eines solchen Systems von einer anderen Game Engine nicht sinnvoll ist und – von der Untersuchung abgeleitet – ein eigenes Skelett-System für FUDGE entwickelt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Liste der Codeblöcke

Abkürzungsverzeichnis

FUDGE Furtwangen Didactic Game Editor

f FudgeCore

WebGL Web Graphics Library

GPU Grafikprozessor, engl. Graphics Processing Unit

glTF GL Transmission Format

JSON JavaScript Object Notation

SSD Skeleton Subspace Deformation

URI Uniform Resource Identifier

FBX Filmbox

1 Einleitung

1.1 Ausgangssituation

Wann immer in einem Spiel oder einem Film ein menschlicher Charakter zu sehen ist, der sich auch wie ein Mensch zu bewegen scheint, dann stecken dahinter in der Regel Skelett-Animationen, wie zum Beispiel in Animationsfilmen wie "Toy Story". [1] Es wurden verschiedene Ansätze für solche Skelett-Animationen verfolgt. [2] Ein Entwickleri (geschlechterneutrales Neutrum nach [3]), das mit Engines wie Unreal und Unity vetraut ist oder Tools wie Mixamo und Blender verwendet, hat dabei womöglich mit Skeleton Subspace Deformation (SSD) zu tun gehabt. Da SSD ein sehr verbreitetes Prinzip für Skelett-Animationen ist [4], soll in dieser Arbeit ein Skelett-System nach diesem Prinzip in FUDGE eingebunden werden. Dabei muss aber auch die Zielsetzung von FUDGE beachtet werden. Ziel ist es zum Einen nicht hochperformant oder besonders umfangreich, sondern leicht verständlich zu sein, sodass Studentis mit FUDGE möglichst leicht lernen können, wie Computerspiele entwickelt werden. Des Weiteren soll FUDGE Plattformunabhängig sein und basiert daher auf Webtechnologien, wie die Web Graphics Library (WebGL) der Kronos Gruppe. Dadurch ist es möglich mit FUDGE auf einem beliebigen Gerät zu entwickelt und Projekte besonders leicht als Web-Anwendung zu veröffentlichen. Mit WebGL können außerdem 3D-Grafiken im Browser hardwarebeschleunigt dargestellt werden, sodass der Grafikprozessor, engl. Graphics Processing Unit (GPU), für das Rendern der Grafiken verwendet wird. [5] Die GPU durchläuft dabei ein eigenes Programm, einen sogenannten Shader. In dieser Arbeit wird unter anderem ein Shader entwickelt, um Skelettanimationen umzusetzen. Denn mithilfe eines Shaders können Transformationen von Punkten eines Körpers bzw. Meshes parallel berechnet werden.

1.2 Zielsetzung

Bisher wurden in FUDGE keine Skelettanimationen unterstützt, weshalb sich diese Arbeit damit beschäftigt ein entsprechendes Skelettsystem zu integrieren. Dabei spezialisiert sich diese Arbeit auf die Unterstützung von SSD. SSD ist ein Skelettsystem, bei dem Gewichtungen gesetzt werden können, um so Übergänge von einer

Knochendeformierung in die nächste Deformierung schaffen zu können. Vorerst sollen verschiedene Skelettsysteme analysiert werden, um so zu ermitteln, ob ein bereits existierendes Skelettsystem integriert werden kann oder, ob davon abgeleitet ein eigenes Skelettsystem für FUDGE entwickelt werden sollte. Da FUDGE auf Webtechnologien basiert, sollen in dieser Arbeit nur Skelettsysteme aus ebenfalls webbasierten Engines betrachtet werden, wie Three.js. Außerdem soll das Skelettsystem auch das Kriterium erfüllen, leicht verständlich zu sein, um mit der Zielsetzung von FUDGE übereinzustimmen. Dementsprechend soll ggf. eine weniger performante Lösung gewählt werden, um so die Implementierung verständlicher zu halten.

2 Grundlagen

2.1 Allgemeines zu Skelettsystemen

Ein Skelett-System ist eine Struktur, die es erlaubt Skelettanimationen definieren und darstellen zu können. Unter einer Skelettanimation wird das Verformen eines Körpers, abhängig von einem skalarem Eingangswert, in der Regel der Zeit, verstanden. Die Verformung des Körpers wird durch verschiedene Stellungen einer Knochen- bzw. Gelenkstruktur beschrieben, an dessen Knochen Teile des Körpers gebunden sind. Ein Körper wird in der Computergrafik in der Regel durch ein sogenanntes Mesh definiert. Ein Mesh beschreibt eine Punktwolke, wobei die Punkte zu Dreiecken verbunden werden, um so eine Oberfläche zu bilden. Um ein Mesh ohne ein Skelett zu animieren, kann für jeden Punkt eine Transformation definiert werden, die diesen entsprechend verschieben soll. Mit Skelettanimationen hingegen kann das Animieren eines Meshes stark vereinfacht werden. Anstatt für jeden Punkt, muss lediglich für jeden Knochen eine Transformation definiert werden und in der Regel werden deutlich weniger Knochen für eine Skelettanimation benötigt, als ein Mesh Punkte hat. Die Punkte des Meshes werden dann über diese Transformationsmatrizen verschoben. Doch nicht nur die geringere Anzahl an zu definierenden Matrizen vereinfacht das Animieren, sondern auch die Tatsache, dass nur die lokale Transformation eines jeden Knochens festgelegt werden muss. Mit einer lokalen Transformation ist die Transformation relativ zum Koordinatensystem des übergeordneten Knochens (Elternknochen) gemeint.

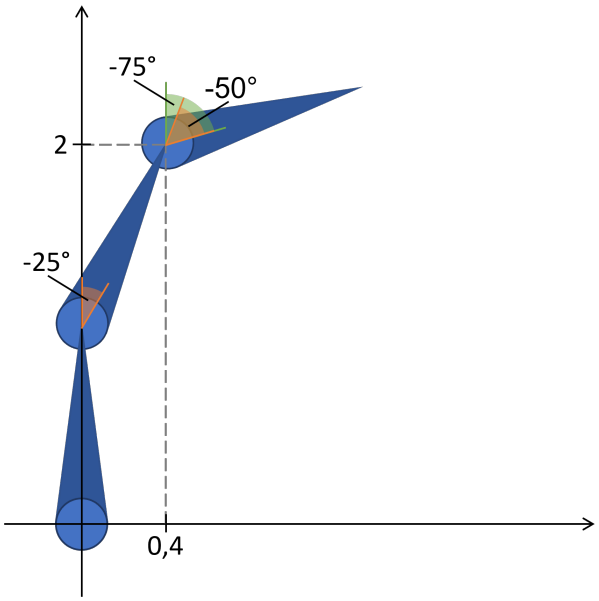

Abbildung 1: Verkettung der Knochentransformationen (Eigene Abbildung)

In der Abbildung 1 ist zu sehen, dass die Welttransformation des obersten Knochens eine Rotation von $-75°$ und eine Verschiebung von $(0,4; 2)$ beschreibt. Die lokale Transformation beschreibt hingegen lediglich eine Rotation von $-50°$. Es wurde also keine Verschiebung definiert, sondern diese ergibt sich dadurch, dass der Elternknochen um $-25°$ rotiert ist. Die Knochen in einem Skelett sind um solche Verkettungen von Transformationen zu ermöglichen hierarchisch gegliedert (siehe Abbildung 2).

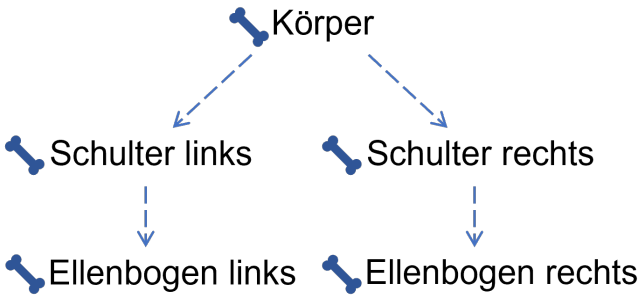

Abbildung 2: Hierarchische Struktur der Knochen (eigene Abbildung)

Speziell bei Skeleton-Subspace Deformation wird für jeden Punkt eines Meshes definiert, wie stark er von welchen Knochen eines Skeletts beeinflusst werden soll. Damit kann ein gewichteter Mittelwert der Knochentransformationen berechnet werden, durch den schließlich ein Punkt eines Meshes transformiert wird.

2.2 Koordinatentransformationen

Ein grundlegendes Verständnis für Koordinatentransformationen ist eine Voraussetzung, um die Funktionsweise eines Skelettsystems verstehen zu können. Solche Koordinatentransformationen sind bereits in FUDGE verankert, nämlich in den Transformationskomponenten für die Knoten, mit denen Szenen aufgebaut werden. Tatsächlich ist die Knochenstruktur eines Skeletts nichts anderes als ein Graph von Knoten, wie es auch die Szenen in FUDGE sind. Für jeden Knoten bzw. Knochen ist somit eine Transformationsmatrix definiert, mit der Punkte von einem Mesh transformiert werden können. So wird ein Mesh in FUDGE an die Stelle eines Knotens gebracht, entsprechend ausgerichtet und skaliert. Durch die hierarchische Struktur können einfache lokale Transformationen definiert werden, die zusammen eine komplexere Transformation bilden. Dabei beschreibt eine lokale Transformation die Transformation des Knotens relativ zum Koordinatensystem des Elternknotens. Die tatsächliche Transformation eines Knotens wird durch das Produkt dessen lokaler und aller übergeordneter Matrizen beschrieben, beginnend mit der lokalen Transformationsmatrix des Wurzelknotens einer Szene bzw. eines Skeletts. Diese Transformation wird in der Computergrafik auch Weltmatrix genannt.

2.2.1 Matrizenoperationen

Matrizenaddition

Eine Matrix A wird mit einer Matrix B addiert, indem jeder Eintrag a_{ij} mit dem Eintrag b_{ij} addiert wird:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{pmatrix} \tag{2.1}$$

Skalarmultiplikation

Eine Matrix A wird mit einem Skalar λ multipliziert, indem jeder Eintrag mit dem Skalar multipliziert wird:

$$\lambda \cdot \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} \lambda \cdot a_{11} & \lambda \cdot a_{12} \\ \lambda \cdot a_{21} & \lambda \cdot a_{22} \end{pmatrix} \tag{2.2}$$

Matrizenmultiplikation

Das Produkt zweier Matrizen A und B ergibt eine Matrix C von der ein Eintrag c_{ij} dem Skalarprodukt des Zeilenvektors $\vec{a_j}$ und Spaltenvektors $\vec{b_i}$ entspricht:

$$\begin{pmatrix} \vec{a_1} \\ \vec{a_2} \end{pmatrix} \cdot \begin{pmatrix} \vec{b_1} & \vec{b_2} \end{pmatrix} = \begin{pmatrix} \vec{a_1} \cdot \vec{b_1} & \vec{a_1} \cdot \vec{b_2} \\ \vec{a_2} \cdot \vec{b_1} & \vec{a_2} \cdot \vec{b_2} \end{pmatrix} \tag{2.3}$$

Dabei ist z. B. $\vec{a_1} \cdot \vec{b_1}$ ein Skalarprodukt. Die Gleichung (2.4) zeigt den Aufbau der sogenannten Einheitsmatrix.

$$E = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 \end{pmatrix} \tag{2.4}$$

Für eine Matrix A und der Einheitsmatrix E gilt $A \cdot E = A$ und $E \cdot A = A$.

2.2.2 Transformationen

Mit Matrizen können verschiedene Transformationen beschrieben werden, wie Drehung, Skalierung und Scherung. Um einen Punkt zu transformieren wird er (als Ortsvektor betrachtet) mit einer Transformationsmatrix multipliziert, während für eine Verschiebung eine Scherungsmatrix im höherdimensionalen Raum definiert wird (siehe Gleichung (2.11)). Entspricht die Transformationsmatrix der Einheitsmatrix gilt $T \cdot \vec{p} = E \cdot \vec{p} = \vec{p}$, wobei T die Transformationsmatrix und $\vec{p}$ der zu transformierende Ortsvektor ist. Mehrere Transformationen können auf einen Vektor angewandt werden, indem die entsprechenden Matrizen miteinander multipliziert werden. Dabei wird ein Vektor abhängig von der Reihenfolge der Multiplikationen von rechts nach links gelesen transformiert. Anders betrachtet wird der Raum in dem sich die Vektoren befinden von links nach rechts gelesen transformiert.

Drehung

Ein Vektor $\vec{p}$ kann wie folgt um den Winkel α im zweidimensionalen Raum rotiert werden:

$$\vec{p'} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \vec{p} \tag{2.5}$$

Für Drehungen im dreidimensionalen Raum muss eine Achse gewählt werden, um die rotiert werden soll. So können Drehmatrizen für die Achsen x, y und z definiert werden:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \tag{2.6}$$

$$R_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \tag{2.7}$$

$$R_z(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.8}$$

Die drei Drehmatrizen können miteinander multipliziert werden, um eine Drehmatrix zu erhalten, mit der ein Ortsvektor im dreidimensionalen Raum in alle Richtungen rotiert werden kann, wie sie auch in FUDGE über die Methode *Matrix4x4.ROTATION* definiert wird:

$$R_{xyz}(\alpha,\beta,\gamma) = \begin{pmatrix} \cos\gamma\cdot\cos\beta & \sin\gamma\cdot\cos\beta & -\sin\beta \\ \cos\gamma\cdot\sin\beta\cdot\sin\alpha - \sin\gamma\cdot\cos\alpha & \sin\gamma\cdot\sin\beta\cdot\sin\alpha + \cos\gamma\cdot\cos\alpha & \cos\beta\cdot\sin\alpha \\ \cos\gamma\cdot\sin\beta\cdot\cos\alpha + \sin\gamma\cdot\sin\alpha & \sin\gamma\cdot\sin\beta\cdot\alpha - \cos\gamma\cdot\sin\alpha & \cos\beta\cdot\cos\alpha \end{pmatrix} \tag{2.9}$$

Skalierung

Ein Ortsvektor $\vec{p}$ kann wie folgt um einen Vektor $\vec{\lambda}$ im dreidimensionalen Raum skaliert werden:

$$\vec{p'} = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix} \cdot \vec{p} \tag{2.10}$$

Verschiebung

Um einen Ortsvektor zu verschieben, kann er mit einem Verschiebungsvektor addiert werden, also $\vec{p'} = \vec{p} + \vec{b}$. Um den gleichen Effekt durch das Anwenden einer Matrix zu erhalten, müssen der Ortsvektor als auch die Matrix um eine Dimension erweitert werden. Ein Ortsvektor $\vec{p}$ kann dann wie folgt um einen Vektor $\vec{b} = (b_1, b_2, b_3)$ im dreidimensionalen Raum verschoben werden:

$$\vec{p'} = \begin{pmatrix} 1 & 0 & 0 & b_1 \\ 0 & 1 & 0 & b_2 \\ 0 & 0 & 1 & b_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} \tag{2.11}$$

Wie in der Gleichung zu sehen ist, wird die Matrix in der vierten Spalte um den Verschiebungsvektor erweitert. Außerdem folgt dem Verschiebungsvektor und dem Ortsvektor eine Eins an vierter Stelle. Daher haben auch in FUDGE Transformationsmatrizen eine Dimension mehr als die Vektoren, die damit transformiert werden sollen.

2.3 Aufbau eines Skeletts

Ein Skelett ist eine Sammlung von Knochen. In der Computergrafik bilden die Knochen eines Skeletts kein physisches System, das durch die Kontraktion von Strängen bewegt wird; sondern jeder Knochen beschreibt eine Transformation, die Einfluss auf ein Mesh nehmen kann. Die Knochen eines Skeletts sind hierarchisch gegliedert. So kann das Skelett als Graph und die Knochen als Knoten des Graphens betrachtet werden. Wie in der Abbildung 3 zu sehen ist, gibt es einen Wurzelknochen, den *Körper*, und mehrere Kindknochen wie z. B. *rechter Arm* und *linkes Bein*, auf die der Wurzelknochen verweist. Die Kindknochen können auf weitere Kindknochen verweisen. Wenn einer

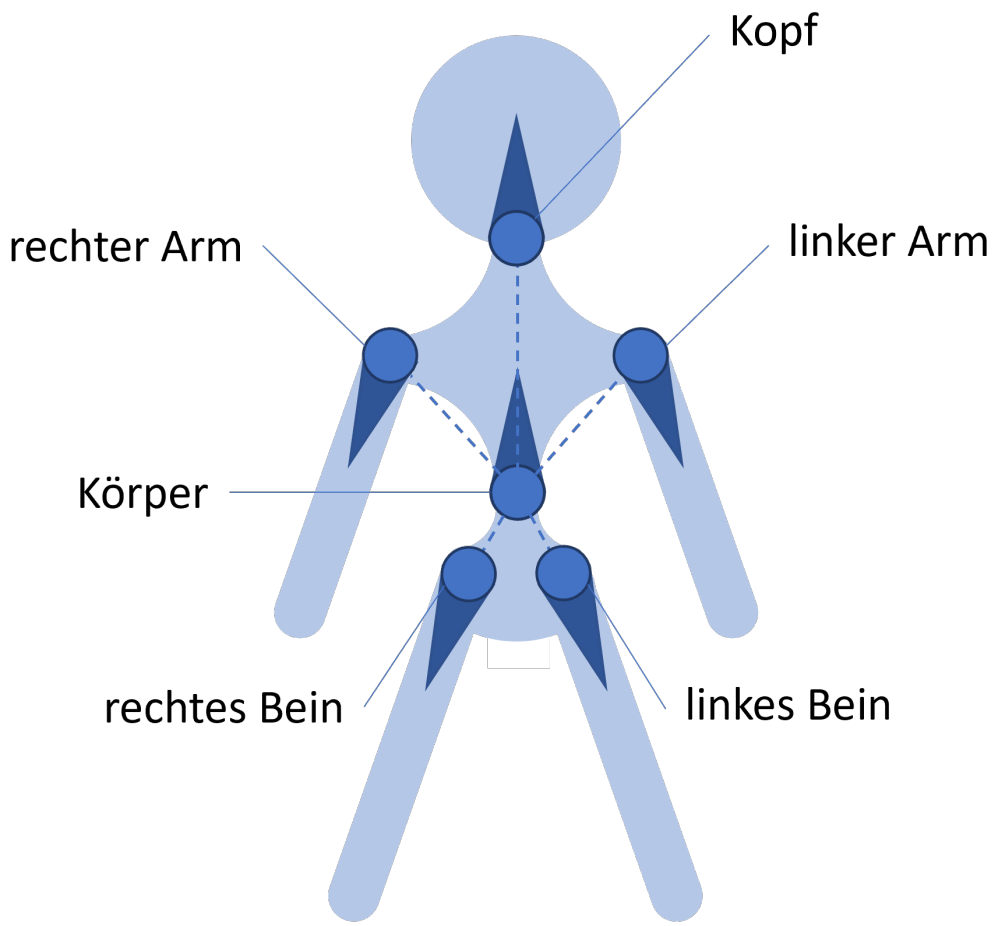

Abbildung 3: Aufbau eines einfachen Skeletts (in Anlehnung an [6])

dieser Knochen bewegt wird, soll der Teil des Meshes sich mit bewegen, der an den Knochen gebunden ist, als auch die untergeordneten Knochen (Kindknochen). Z. B. soll der linke Arm der Figur auch nach oben rotieren, wenn der Knochen *linker Arm* entgegen des Uhrzeigersinns rotiert wird. Natürlich sind *linker Arm* und *rechtes Bein* unübliche Bezeichnungen für die Knochen, jedoch werden in der Computergrafik häufig Knochen nach dem benannt, was sie bewegen. Synonym für Knochen wird auch die Bezeichnung Gelenk verwendet, da die Knochen auch wie Gelenke agieren.

Außer auf eine Sammlung von Knochen verweist ein Skelett auf die inversen Matrizen der Grundstellung der Knochen. Z. B. ist der *linke Arm* von unserem einfachen Skelett um 4 entlang der x-Achse und 5 Einheiten entlang der y-Achse verschoben und um 290° rotiert (siehe Abbildung 4). Wenn ein Punkt von einem Mesh bzw. ein Vektor mit der Matrix multipliziert wird, die die Verschiebung, Rotation und Streckung eines Knochens zu einem bestimmten Zeitpunkt einer Animation beschreibt, dann steckt darin auch die Transformation, die nötig ist um den Knochen in seine Grundstellung zu bringen. Gesucht ist jedoch nur die Verschiebung, Rotation und Streckung, die über der Transformation in die Grundstellung hinaus nötig ist, um den Knochen in

seine aktuelle Stellung zu bringen. Schließlich befindet sich der linke Arm schon an der Stelle des zugehörigen Knochens und würde sonst nur noch weiter weg vom Ursprung des Skeletts geschoben werden. Anders betrachtet lässt sich auch sagen, dass die Vektoren zuerst in den Raum des Knochens gebracht werden müssen.

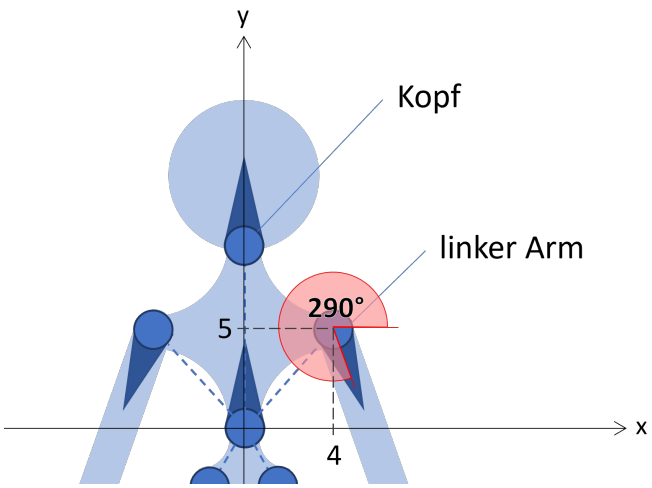

Abbildung 4: Transformation des Gelenks linker Arm (eigene Abbildung)

In der Abbildung 5 befindet sich Knochen 2 im ersten Bild genau in der Mitte von den Vektoren des Meshes, wobei alle Vektoren an diesen Knochen gebunden sind. Der Ursprung des Koordinatensystems für diese Vektoren befindet sich jedoch auf Höhe von Knochen 1. Wird die Rotation des Knochens ohne weiteres auf die Vektoren des Zylinders angewandt, wird unser Zylinder wie im zweiten Bild rotiert. Was jedoch erwartet wird, ist eine Rotation um Knochen 2. Dazu müssen vorerst alle Vektoren des Zylinders nach unten verschoben werden. Genau genommen müssen sie um die gleiche Strecke nach unten verschoben werden, wie Knochen 2 nach oben verschoben ist. Somit bildet der Ursprung des Koordinatensystems, den Mittelpunkt der Vektoren, so wie zuvor Knochen 2 den Mittelpunkt gebildet hat. Auch für Rotationen und Streckungen muss jeweils die umgekehrte Transformation auf die Vektoren angewendet werden, damit sie relativ zum Ursprung genau so stehen wie zuvor relativ zum Knochen. Im dritten Bild werden also alle Vektoren mit B_2^{0-1} multiplizert, der inversen Matrix der Grundstellung des Knochens 2. Danach werden diese Vektoren, wie im vierten Bild zu sehen, mit der aktuellen Transformationsmatrix des Knochens 2 B_2^{δ} multipliziert.

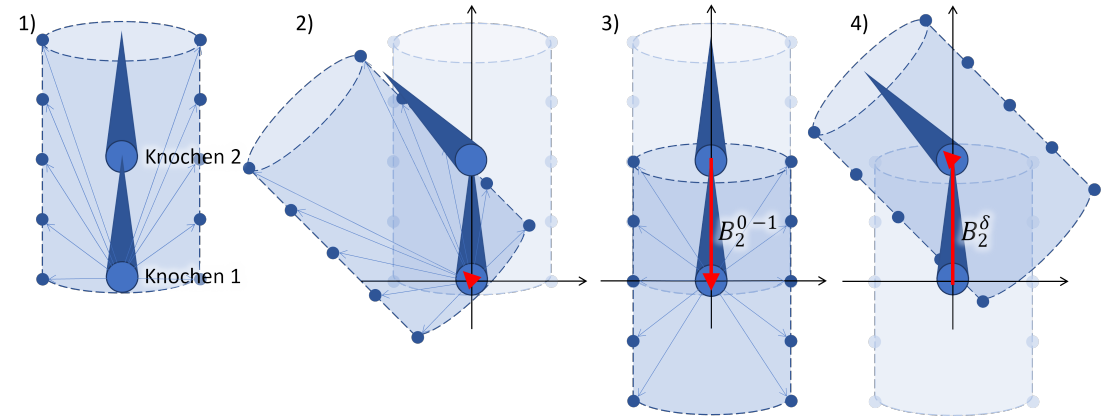

Abbildung 5: Transformation eines Meshes durch einen Knochen (in Anlehnung an [6])

2.4 Deformierung eines Meshes

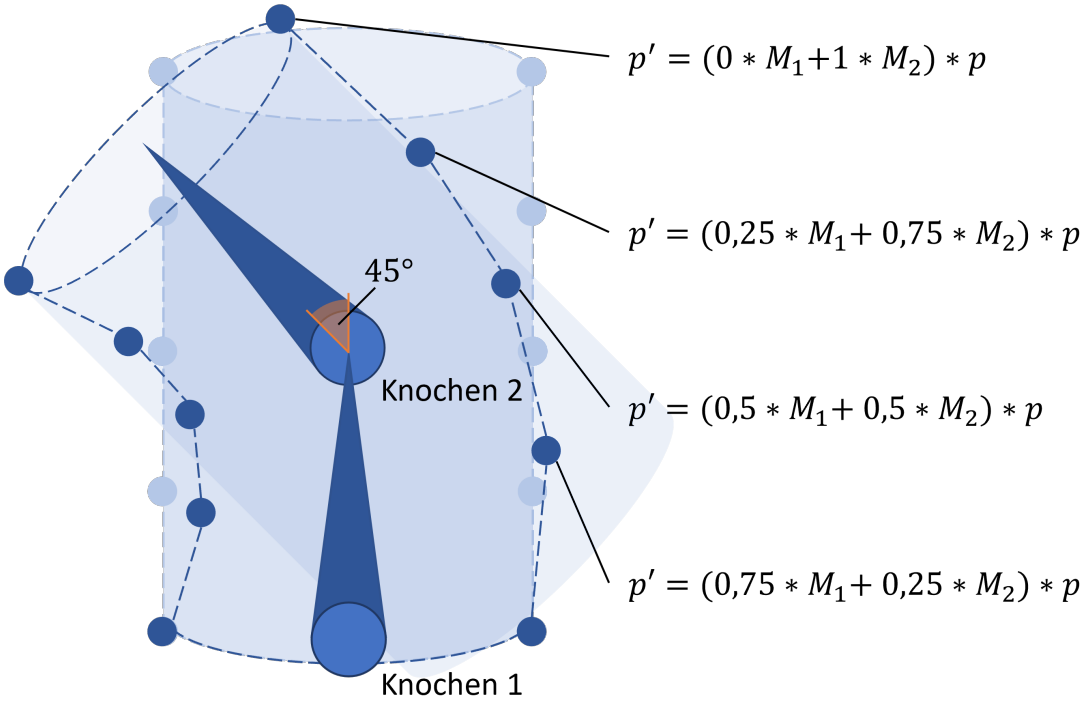

Abbildung 6: Übergang zwischen den Transformationen zweier Knochen (in Anlehnung an [6])

Für jeden Vektor in einem Mesh wird definiert von welchen Knochen er beeinflusst wird und wie stark. In der Abbildung 6 wird dargestellt wie der Einfluss der Knochen auf die einzelnen Vektoren definiert ist und wie dadurch ein Übergang zwischen den Transformationen von zwei Knochen geschaffen werden kann. Die Punkte des Meshes werden dabei als Vektoren betrachtet. Die Berechnung der Transformation eines

Vektors p_i kann wie folgt formuliert werden [2], [4], [6]:

$$p'_i = (\sum_{j=1}^{m} w_{i,j} * T_j) * p_i \tag{2.12}$$

Es wird also ein gewichteter Mittelwert der einzelnen Knochentransformationen T_j berechnet. Dabei muss die Summe der Gewichtungen $w_{i,j}$ eins ergeben, damit keine unerwünschte Skalierung entsteht. Tatsächlich wird in Engines wie Three.js üblicherweise nicht für einen Vektor p_i der Einfluss aller Knochen angegeben, sondern nur für maximal vier Knochen. Dazu werden für jeden Punkt vier Indizes, die auf die Transformationsmatrizen verweisen, und vier Gewichtungen definiert. Dem Shader wird also eine indizierte Liste von Knochen und pro Punkt im Mesh Knochenindizes und -gewichtungen übergeben. Dadurch kann im Shader der Einfluss der Knochen für jeden Vektor individuell berechnet werden. Die Berechnung der Transformationsmatrix T_j, welche die Transformation eines Vektors durch einen Knochen beschreibt, kann wie folgt formuliert werden [2], [6]:

$$T_j = N^{-1} * B_j^{\delta} * B_j^{0-1} \tag{2.13}$$

B_j^{0-1} beschreibt hier die inverse Matrix der Grundstellung eines Knochens. B_j^{δ} ist die zum Rendern berechnete Weltmatrix des Knochens, also die aktuelle Stellung des Knochens in der Welt. Genauso ist N die Weltmatrix des Knotens, an dem das Skelett gebunden ist. Nur wird hier die inverse Matrix N^{-1} benötigt, um die Transformation dieses Knotens aus den Weltmatrizen der Knochen rauszukürzen. Das Mesh hängt in der Regel an dem selben Knoten, wie das Skelett und wird dadurch bereits von dem Knoten transformiert und würde ohne der Multiplikation mit der Inversen N^{-1} nochmal vom Skelett mit derselben Transformation transformiert werden. Theoretisch könnte die Knochenhierarchie einfach nicht an einen anderen Knoten gehängt werden. Somit wäre die Multiplikation mit N^{-1} nicht notwendig, jedoch werden dann nicht mehr die Weltmatrizen der Knochen zum Rendern berechnet, da sie nicht mehr Teil der zu rendernden Szene sind. Des Weiteren sind die Knochen, wenn sie Teil der Szene sind, auch funktionale Knoten, die durch Meshes visualisiert werden können. Außerdem können ihnen dann auch andere Knoten angehängt werden, um z.B. einen Gegenstand mit der Hand mitbewegen zu lassen.

3 Analyse von Skelettsystemen

3.1 three.js

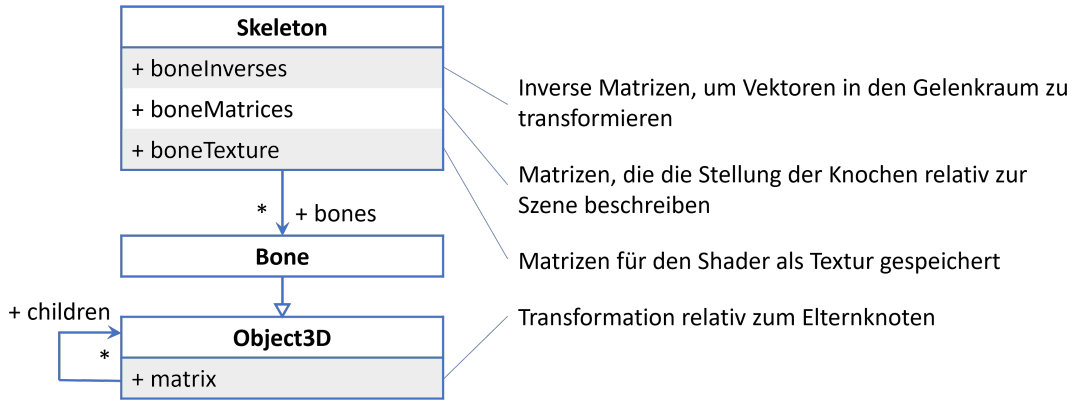

Abbildung 7: Die Skeleton-Klasse von three.js [7]–[9]

Ein Skelett in three.js besitzt eine flache Liste von Knochen. Dadurch kann vom Skelett aus jeder Knochen in der Knochenhierarchie über einen Index gefunden werden. Dass die Knochen in three.js hierarchisch gegliedert sind, sieht man an der Eigenschaft *children* von der Klasse Object3D (siehe Abbildung 7). Object3D entspricht hier der Klasse Node von FUDGE. Unter *boneInverses* sind die inversen Matrizen der Grundstellung der Knochen hinterlegt. Aber auch die berechneten Knochenmatrizen werden in der Skelettklasse von three.js zwischengespeichert.

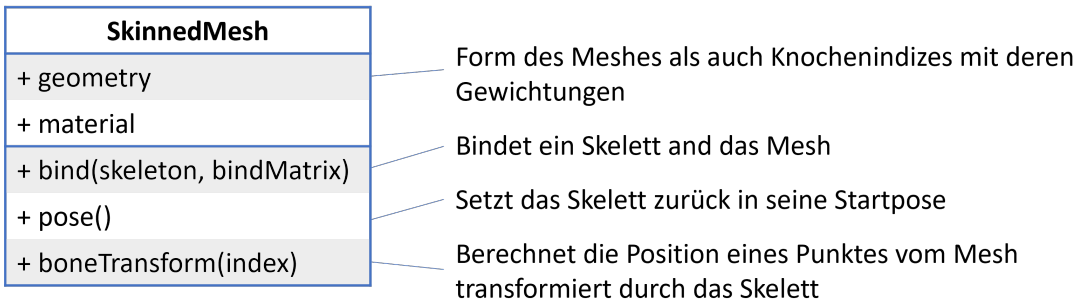

Abbildung 8: Die Klasse SkinnedMesh von three.js [10], [11]

In three.js gibt es eine eigene Klasse für Meshes, die durch ein Skelett beeinflusst werden sollen. Diese ist hier SkinnedMesh genannt, von der Mesh-Klasse abgeleitet

und besitzt einige spezielle Methoden im Bezug auf Skelette. Von diesen Methoden ist die *bind*-Methode besonders wichtig, da ohne deren Aufruf nicht klar ist, welche Knochen mit den Knochenindizes gemeint sind, die in den Punkten des Meshes hinterlegt sind. Die Punktwolke, steckt in der Eigenschaft "geometry". Dort können wie im Codeblock 1 dargestellt Knochenindizes und -gewichtungen gesetzt werden.

```
geometry.setAttribute(
  'skinIndex',
  new THREE.Uint16BufferAttribute(skinIndices, 4)
);
geometry.setAttribute(
  'skinWeight',
  new THREE.Float32BufferAttribute(skinWeights, 4)
);
```

Codeblock 1: Setzen von Knocheneinflüssen in three.js [10]

Eine integration des Skelett-Systems von three.js ist sehr aufwendig, da die Berechnung der Knochenmatrizen und natürlich auch das Zeichnen des skelettanimierbaren Meshes an die Render-Methode von three.js gebunden ist, wobei FUDGE seine eigene Render-Methode hat. Das heißt die Berechnung der Knochenmatrizen muss in FUDGE neu implementiert werden, als auch ein Shader zum zeichnen des animierten Meshes. Somit bleibt nur die Datenstruktur, welche integriert werden kann, jedoch nicht die Funktionalität.

3.2 Away3D

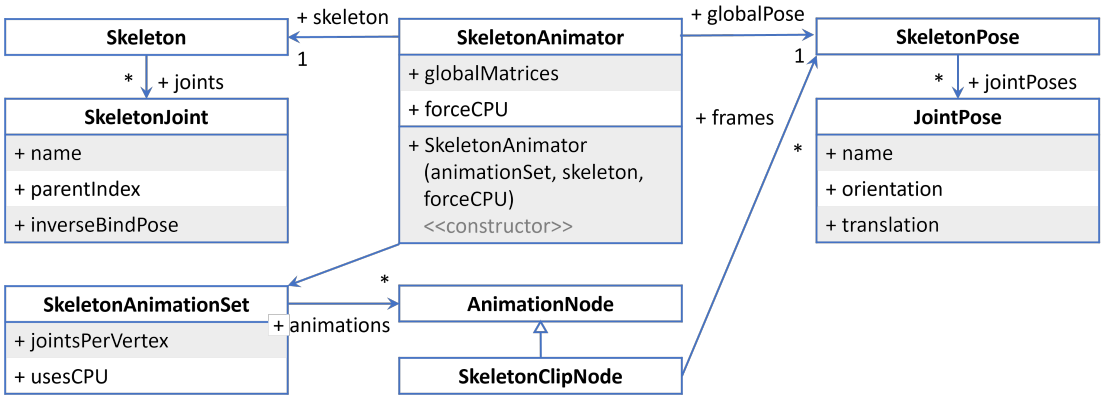

Abbildung 9: Das Skelett-System von Away3D [12]–[18]

In Away3D wird für die Knochen eine neue Struktur definiert, die hier *SkeletonJoint* heißt. Diese bildet sich aus nur drei Eigenschaften, darunter die Inverse der Grundstellung des Knochens und der Index des Elternknochens. Mit dieser Struktur wird nur

das Minimum an Daten gespeichert, das nötig ist, um die Knochen-Hierarchie eines Skeletts beschreiben zu können. Nur beim Bezeichner wird hier nicht gespart, der hier als *name* hinterlegt ist und eine Zeichenkette anstatt einen einfachen Index darstellt. Um die lokale Transformation eines Knochens zu beschreiben, wird in Away3D eine Instanz von *JointPose* erzeugt, in der aber nur eine Rotation und Verschiebung festgelegt werden kann. Skalierungen werden also von den Knochen in Away3D nicht unterstützt. Die Transformation einer *JointPose*-Instanz kann dann über den Namen dem entsprechenden Knochen zugewiesen werden. Die verschiedenen Posen der Knochen sind in einer Instanz von *SkeletonPose* gespeichert, die wiederum einem Schlüsselbild in Form einer Instanz von *SkeletonClipNode* gespeichert sind. Das heißt die Transformation eines Knochens kann nur über eine Animation verändert werden. Die Aktuelle Pose eines Skeletts ist als *globalPose* im *SkeletonAnimator* hinterlegt, wobei die berechneten Knochenmatrizen als *globalMatrices* gespeichert sind. Das Mesh besitzt in Away3D keine Assoziation zu einem Skelett, sondern zum *SkeletonAnimator*. Diese Assoziation wird über die Eigenschaft *animator* von Mesh gesetzt.

Mit dem Skelett-System von Away3D gibt es das Problem, dass die Knochenmatrizen über den *SkeletonAnimator* berechnet werden und somit nicht das Animationssystem von FUDGE nicht ohne weiteres für dieses Skelett-System genutzt werden kann. Dazu müsste eine Implementierung entwickelt werden, die über das Animationsystem von FUDGE im Hintergrund den *SkeletonAnimator* ansteuert. Des weiteren bleibt noch die Entwicklung eines skelettanimierbaren Meshes übrig, da das Mesh von Away3D aus dem selben Gründen wie bei Three.js nicht integriert werden kann.

3.3 Fazit der Analyse

Ein Skelettsystem von einer anderen Engine, wie Three.js oder Away3D zu integrieren, ist in Anbetracht der Probleme, die zu den Skelett-Systemen dieser Engines aufgeführt wurden weniger sinnvoll. Der Aufwand ein bestehendes Skelett-System zu integrieren ist damit höher als die Vorteile, die daraus gezogen werden.

Ein eigenes Skelett-System für FUDGE zu entwickeln erweist sich als sinnvoller, denn in FUDGE ist z. B. bereits eine Klasse *Mesh* enthalten, welche nur um die zwei Eigenschaften Knochenindizes und -gewichtungen erweitert werden muss um von einem Skelett beeinflusst werden zu können. Außerdem gibt es auch ein Knotensystem, mit die Knochenstruktur eines Skelettes beschrieben werden kann. Hier bedarf es einer neuen Klasse mit größerem Aufwand als der für das Mesh, welcher aber voraussichtlich auch in einem überschaubaren Rahmen bleibt. Wenn mit den Knoten von FUDGE gearbeitet wird, gibt es auch den Vorteil, dass diese bereits über das Animationssystem

animiert werden können. Natürlich muss auch ein neuer Shader entwickelt werden, jedoch ist dies genauso für die Integration eines bestehenden Skelett-Systems nötig. Des weiteren ist ein eigenes Skelett-System nicht von einer anderen Engine abhängig. Es muss also in der Dokumentation nicht auf die einer anderen Engine verwiesen werden und Anpassungen sind leichter möglich.

4 Konzeption und Implementierung

4.1 Anwendungsfälle

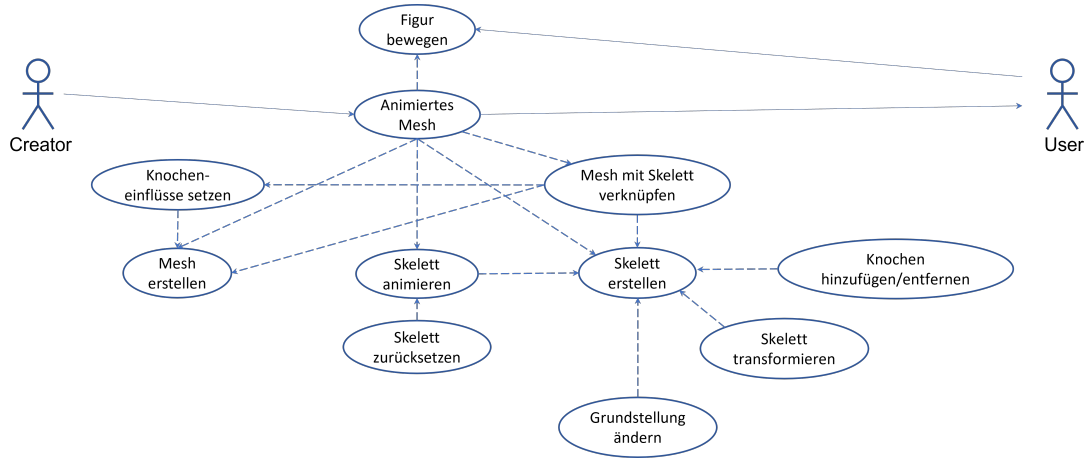

Abbildung 10: Use Case-Diagramm

Wenn ein Creator ein Spiel entwickeln möchte, in dem der User eine Figur bewegen kann, soll er für die Visualisierung ein animiertes Mesh erstellen können. Dazu braucht es, wie in der Abbildung 10 zu sehen ist ein Mesh, ein Skelett und Animationen für das Mesh. Die Möglichkeit Animationen zu erstellen, welche Knoten und deren Komponenten animieren, ist in FUDGE schon gegeben, jedoch sollen Knochen, welche auch Knoten sind, von ihrem Skelett aus animiert werden können. Damit sich durch die Animationen nicht nur die Knochen des Skeletts bewegen, sondern auch Punkte vom Mesh mitbewegt werden, muss ein Mesh mit einem Skelett verknüpft werden können. Dazu gehört dann auch die Knocheneinflüsse, also die Knochenindizes und -gewichtungen, für jeden Punkt im Mesh zu setzen. Ein animiertes Skelett soll in seine Grundstellung zurückgesezt werden können und damit ein Skelett in seiner Größe und Ausrichtung an ein Mesh angepasst werden kann, soll es als gesamtes transformiert werden können. Um die Knochenstruktur eines Skeletts aufzubauen, müssen Knochen hinzugefügt werden können. Weitere Funktionen können vorgesehen werden in Betracht auf die Erstellung von Skeletten in einem Editor. Dazu gehört das Entfernen von Knochen und das Ändern derer Grundstellung.

4.2 Klassenstruktur

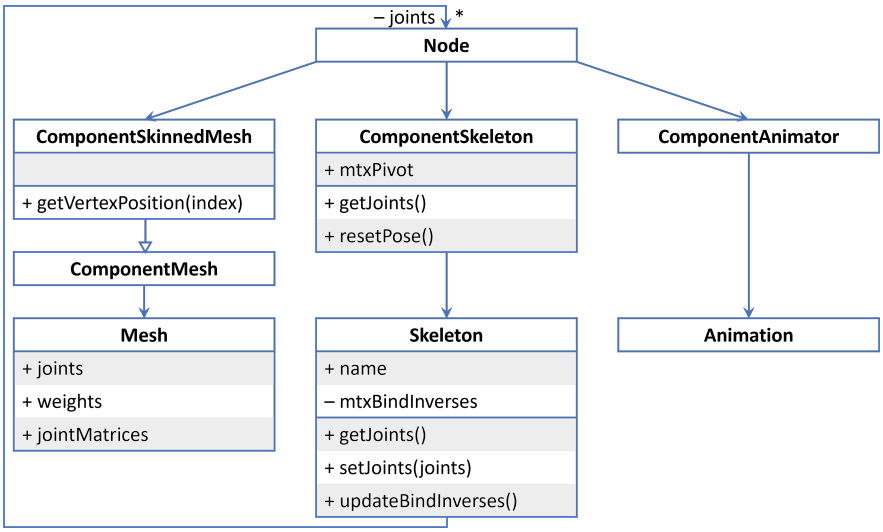

Abbildung 11: Erster Entwurf für die Klassenstruktur des Skelett-Systems als Klassendiagramm

4.2.1 Skelettklassen

Abgleitet von der Analyse bestehender Skelett-Systeme, waren für das Skelett zuerst zwei Klassen geplant. Zum Einen eine Klasse *Skeleton*, um die Knochen eines Skeletts und die Inversen der Grundstellung der Knochen zu speichern. Zum Anderen die Komponente *ComponentSkeleton*, um ein Skelett an einen Knoten zu binden. Für Knochen sollte dabei die Klasse *Node* von FudgeCore (f) verwendet werden, dem Kernnamensraum von FUDGE. Die Skelett-Komponente sollte auf eine Pivot-Matrix verweisen, um ein Skelett an ein Mesh anpassen zu können.

Die Knochen müssen Teil der von FUDGE gerenderten Szene sein, damit deren Weltmatrizen von der f*.Render.renderPrepare*-Methode berechnet werden und außerdem andere Knoten angehängt werden können, um zum Beispiel eine Waffe mit einer Hand mitbewegen zu lassen. Dazu sollte die Skelett-Komponente die Knochen des Skeletts an seinen Knoten binden, wenn sie an einen angeheftet wird. Dieser Ansatz hatte jedoch das Problem, dass wenn die Knochen von einer Skelett-Komponente bewegt werden, die Veränderung auch für alle Skelett-Komponenten mit dem gleichen Skelett galt. Dabei sollten die Skelett-Komponenten unabhängig voneinander animiert werden können. Der Lösungsansatz dafür war von der Skelett-Komponente einen Klon des Skeletts erstellen zu lassen.

Bei der Suche nach einer Klon-Methode, kam schließlich das Thema Serialisierung auf. FUDGE bietet eine eigene Struktur für die Serialisierung von Objekten. Damit

lassen sich zum Beispiel Knoten von FUDGE mit allen Komponenten und Kindknoten in ein JavaScript Object Notation (JSON)-String umwandeln, der wieder zu dem ursprünglichen Knoten deserialisiert werden kann. Die Idee ist, dass damit auch Klone von Knoten erzeugt werden, indem sie serialisiert und als neuer Knoten wieder deserialisiert werden. Jetzt sind aber die Knoten von FUDGE nicht selbst als Ressource gedacht, die in einem Projekt gespeichert sind, wohingegen ein Skelett unabhängig von einer Szene in einem Projekt als Ressource gespeichert werden können soll. Wenn eine Knotenstruktur gespeichert werden soll, wird in FUDGE aus dem Knoten ein Graf erzeugt und soll dieser Graf wiederum in eine Szene gebracht werden, so wird davon eine Instanz mit dem Typ *ƒ.GraphInstance* erstellt. Darum ist es sinnvoll, die Klasse *Skeleton* von *ƒ.Graph* abzuleiten. Die Skelett-Komponente soll jetzt nicht mehr ein Skelett selbst, sondern nur die Instanz eines Skeletts an seinen Knoten binden. Darum muss auch eine Klasse *SkeletonInstance* entwickelt werden, die von *ƒ.GraphInstance* abgeleitet ist (siehe Abbildung 13).

Da ein Graf bereits die Methode *addChild* besitzt, um Kindknoten anzuhängen, ist die Idee, diese Methode zu verwenden, um auch Knochen an das Skelett zu hängen. Mit diesem Ansatz kennt ein Skelett aber nicht nur seine Knochen, sondern alle Kindknoten, die an es gebunden sind. Zur Unterscheidung wurde also eine von *ƒ.Node* abgeleitete Klasse *Bone* definiert. Um in der *Skeleton*-Klasse weiterhin eine flache Liste der Knochen zu haben, kann nun über die Ereignisse *ƒ.EVENT.CHILD_APPEND* und *ƒ.EVENT.CHILD_Remove* eine entsprechende Liste mit der Liste der Kindknoten synchronisiert werden.

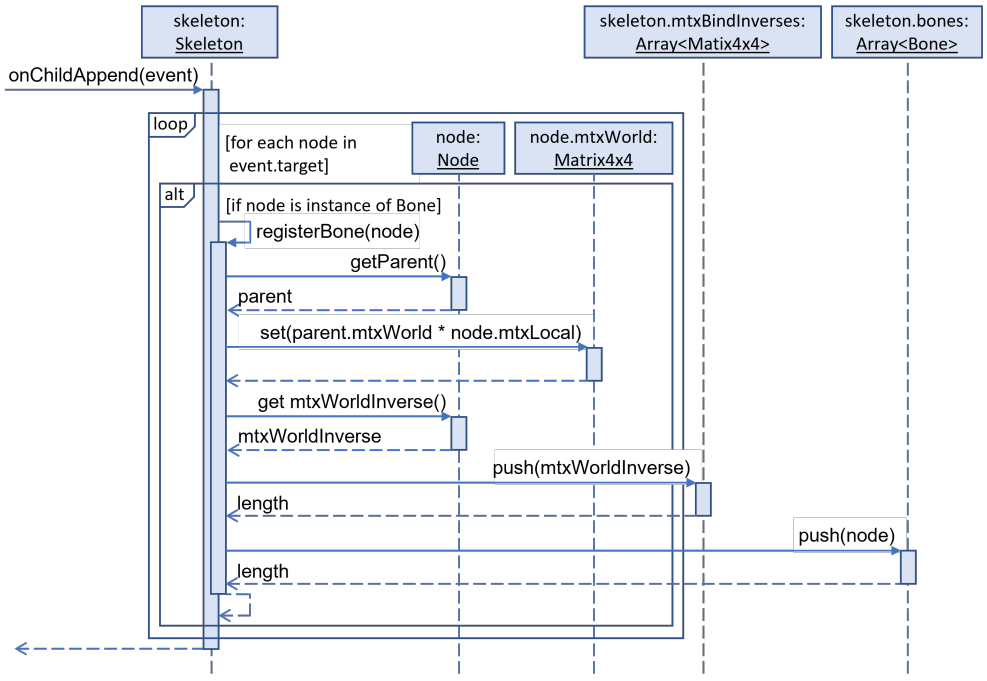

Abbildung 12: onChildAppend-Methode der Skelettklasse

Abbildung 12 zeigt den Ablauf der *onChildAppend*-Methode in einem Sequenzdiagramm. Darin wird ein Knoten, der angehängt wurde, als Knochen registriert, falls er eine Instanz der Klasse *Bone* ist und dessen Weltmatrix berechnet, damit schließlich auch die Inverse der Grundstellung des Knochens berechnet werden kann.

Die Verwendung einer Klasse *Bone* hatte jedoch das Problem, dass der *GLTFLoader* unterscheiden musste, ob es sich bei einem Knoten, der in der glTF-Datei definiert ist, um einen Knochen handelt, da es in glTF diese Unterscheidung nicht gibt. Stattdessen kann ein Skelett auch anhand einer Liste von registrierten Namen der Knochen unterscheiden, ob es sich bei einem Kindknoten um einen Knochen handelt. Da die Knochen eines Skeletts für eine Animation über dessen Namen referiert werden, sind auch die inverse Matrizen der Grundstellung der Knochen in einer über den Namen indizierten Liste gespeichert und somit kann über alle Knochennamen iteriert werden. Damit wird aber eine Methode *AddBone* in der *Skeleton*-Klasse benötigt, weil jetzt beim Ereignis *ChildAppend* nicht mehr unterschieden werden kann, ob es sich bei dem Knoten um einen Knochen handelt oder nicht.

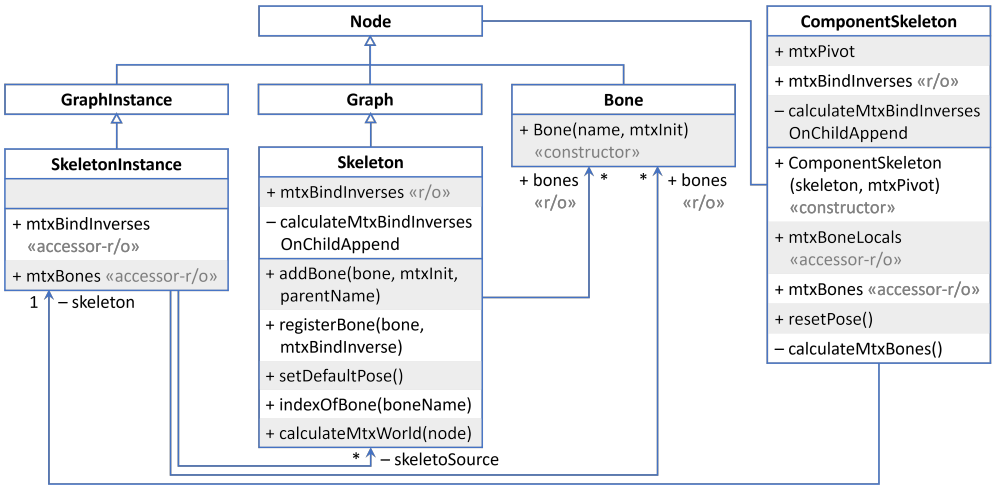

Abbildung 13: überarbeitete Struktur der Skelettklassen

4.2.2 Mesh-Klassen

Für die Strukturen des Skelett-Systems zur Umsetzung des sklettanimierbaren Meshes, wurde zuerst nur eine neue Klasse geplant, nämlich die Komponente *ComponentSkinnedMesh*. Außerdem sollte f*.Mesh* nicht abgeleitet, sondern die Klasse selbst um drei Eigenschaften erweitert werden. Was dafür spricht ist, dass Primitive Körper, die von dieser Klasse abgeleitet sind, somit auch von einem Skelett beeinflusst werden können. Dieses Konzept wäre jedoch ein starker Eingriff in die Strukturen von FUDGE, weshalb eine abgeleitete Klasse besser wäre, um gegebene Strukturen nicht zu verändern. Außerdem müsste ein Creator immer noch von den Primitiven Körpern ableiten,

um die Knochenindizes und -gewichtungen setzen zu können. Wenn eine abgeleitete Klasse *MeshSkin* entwickelt wird, dann kann ein Creator, der einen Primitiven Körper erzeugen und animieren will, von dieser Klasse ableiten und über Delegation auf die *calculateVertices*-Methode und Berechnungsmethoden der Primitiven Körper verweisen. In die abgeleitete Klasse kann ein Creator dann auch entsprechende Methoden einbauen, um die zugehörigen Knochenindizes und -gewichtungen für die Vertices zu setzen. Der Name *MeshSkin* anstatt *SkinnedMesh* wurde gewählt, da zuerst die übergeordnete Bezeichnung *Mesh* kommen soll und dann die Erweiterung, die darauf hinweist, dass es sich um ein Skelett-beeinflussbares Mesh handelt. *Skin* wurde anstatt *Skinned* gewählt, da es sich um eine verformbare "Haut" handelt und nicht um etwas "gehäutetes", auch wenn der Begriff Skinning als Synonym für SSD verwendet wird.

Dementsprechend soll auch die zugehörige Komponente *ComponentMeshSkin* heißen. Die Komponente hatte in dem ersten Ansatz nur die Aufgabe, auf eine Skelett-Komponente zu verweisen, von der das Mesh deformiert werden soll. Darüber hinaus verfügte sie über eine Methode, um die Position eines Vertex vom Mesh zur CPU-Laufzeit zu bestimmen. Das Problem an diesem Ansatz ist, dass für ein Skelett-animiertes Mesh immer eine Skelett- und eine Mesh-Skin-Komponente benötigt werden, obwohl diese nie unabhängig voneinander benötigt werden. Dementsprechend können die beiden Komponenten zur Vereinfachung zusammengeführt werden. Also verweist *ComponentMeshSkin* direkt auf ein Objekt der Klasse *SkeletonInstance*.

Ein Problem bleibt jedoch bestehen, welches mit der Render-Methode von FUDGE zusammenhängt. Dort wird nach Mesh-Komponenten gesucht, um deren Meshes rendern zu können. Diese werden über deren Klassennamen in den Knoten gefunden. Der Klassenname von den Mesh-Skin-Komponenten ist jedoch *ComponentMeshSkin* und nicht *ComponentMesh*, weshalb die Render-Methode nicht die Skelett-beeinflussbaren Meshes rendert. Als Lösung gibt es zwei Möglichkeiten. Entweder wird in der Render-Methode speziell nach Mesh-Skin-Komponenten gesucht oder die beiden Komponenten werden zusammengeführt. Dabei erwies sich das Zusammenführen als sinnvoller, da es zum Einen eine größere Überarbeitung der Render-Methode von FUDGE vermeidet. Zum Anderen kann ein Knoten somit nur eine Mesh-Komponente besitzen entsprechend dem Singleton-Muster. Singleton ist ein Muster, bei dem nur eine Instanz einer Klasse existieren darf bzw. im Falle der Komponenten von FUDGE bezieht sich das nur auf einen Knoten. Das heißt ein Knoten darf nur auf eine Instanz einer bestimmten Komponente verweisen, falls diese als Singleton markiert ist. Dementsprechend sollte ein Knoten nur eine Mesh-Komponente besitzen, jedoch gäbe es ohne die Zusammenführung das Problem, dass ein Knoten eine Mesh- und eine Mesh-Skin-Komponente besitzen könnte. Dementsprechend müsste ein Sonderfall berücksichtigt

werden, sodass diese Komponenten sich gegenseitig ausschließen. Für die Zusammenführung gilt, möglichst wenig Eigenschaften, die sich auf Funktionalitäten vom Skelettsystem beziehen, in die Mesh-Komponente einzubauen. Tatsächlich können die meisten Funktionalitäten in die Klasse *SkeletonInstance* übertragen werden (siehe Abbildung 14). Außerdem muss die Klasse *MeshSkin* nichts von den Knochenmatrizen wissen, wie es sonst bisher angedacht war. Da in der Render-Methode über die Mesh-Komponente auf das Mesh zugegriffen wird, können genauso direkt von der Skelett-Instanz die Knochenmatrizen an die *useRenderBuffers*-Methode von *MeshSkin* übergeben werden.

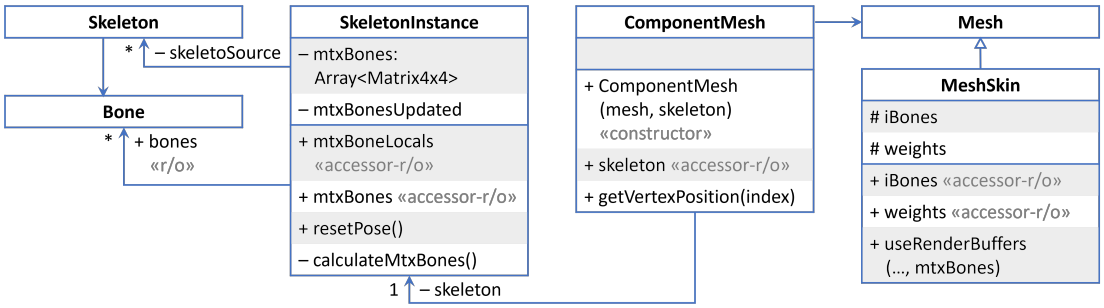

Abbildung 14: Überarbeitung der Klassen MeshSkin, ComponentMesh und SkeletonInstance

Die Abbildung 15 zeigt nochmal das gesamte Klassendiagramm zum Ende der Entwicklungsphase. Hier sind auch Methoden aufgelistet, die geerbt sind und überschrieben werden oder im Falle der Klasse *ComponentMesh* geändert wurden.

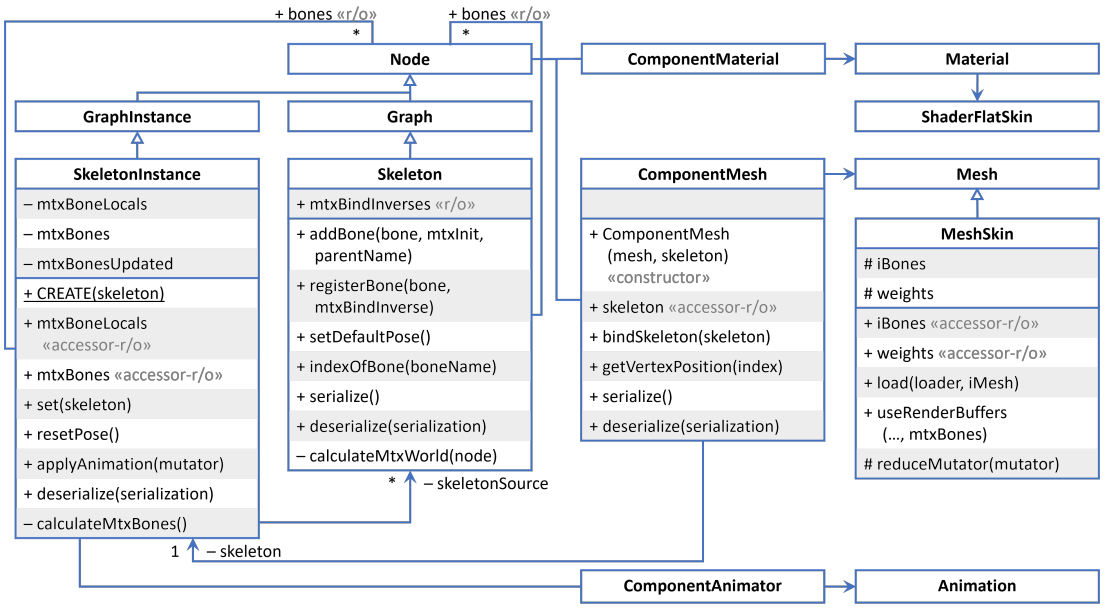

Abbildung 15: Finales Klassendiagramm des Skelett-Systems

4.3 Implementierung des Shaders

In WebGL als auch allgemein in der Computergrafik wird in der Regel ein Vertex- und ein Fragment-Shader implementiert, um ein Mesh zu rendern. Ein Vertex-Shader durchläuft parallel alle Vertices und berechnet deren Positionen im Kamerakoordinatensystem. Im Vertex-Shader können die Punkte nochmal verschoben und Daten für den Fragment-Shader berechnet werden. Im Fragment-Shader werden für jedes Dreieck die einzelnen Pixel durchlaufen, die zu zeichnen sind, und deren Farbe berechnet. Die Interpolationen der vom Vertex-Shader überlieferten Daten (ein Dreieck setzt sich schließlich aus drei Vertices zusammen) bilden die Eingangswerte des Fragment-Shaders, mit denen weiter gerechnet werden kann. Für das Skelettsystem muss nur ein neuer Vertex-Shader entwickelt werden, da sich dessen Einfluss nur auf die Manipulation der Vertex-Positionen beschränkt. Für den Vertex-Shader müssen folgende neue Variablen definiert werden:

```
1 struct Bone {
2   mat4 matrix;
3 };
4
5 const uint MAX_BONES = 10u;
6
7 in uvec4 a_iBone;
8 in vec4 a_weight;
9 uniform Bone u_bones[MAX_BONES];
```

Codeblock 2: Variablen für den Skin-Vertex-Shader

Der Codeblock 3 zeigt, wie die neue Position eines Vertex im Shader berechnet werden kann. Des Weiteren ist auch die Berechnung der neuen Normale zu dem Dreieck des jeweiligen Vertex zu sehen.

```
mat4 skin =
    a_weight.x * u_bones[a_iBone.x].matrix +
    a_weight.y * u_bones[a_iBone.y].matrix +
    a_weight.z * u_bones[a_iBone.z].matrix +
    a_weight.w * u_bones[a_iBone.w].matrix;

gl_Position = u_projection * skin * vec4(a_position, 1.0);

vec3 normal = normalize(
    mat3(u_normal) *
    transpose(inverse(mat3(skinMatrix))) *
    a_normalFace
);
```

Codeblock 3: Berechnung der Position und Flächennormale im Shader [6]

4.4 Importieren von Skelettanimationen

4.4.1 glTF

Spätestens, wenn das Erstellen von Skeletten in ein Modelierprogramm für FUDGE eingebunden wird, sollte es möglich sein, diese zu exportieren und schließlich in einem Spiel zu importieren. Oder man möchte mit einem unabhängigen Tool Modelle mit Skeletten erstellen, die importiert werden können, wie z. B. mit Blender. Auf der Suche nach aktuellen Dateiformaten ist dabei glTF als unabhängiges und teils leserliches Dateiformat aufgefallen. glTF ist ein auf JSON basierendes unabhängiges Dateiformat, um Szenen und Assets mit Meshes, Animationen, Skeletten und weiteres speichern zu können. Um die Daten kompakter zu halten, werden Vektoren, Matrizen, Texturen etc. in einer gesonderten Binärdatei gespeichert. Für die Szenenstruktur wird mit Knoten gearbeitet, wie sie in FUDGE bekannt sind. Im Codeblock 4 ist die Definition eines Skeletts in glTF zu sehen.

```
{
  "inverseBindMatrices": 6,
  "joints": [
    5,
    4,
    3,
    2
  ],
  "name": "Skeleton"
}
```

Codeblock 4: Definition eines Skeletts in glTF

Der Index 6 hinter dem Schlüssel *inverseBindMatrices* verweist auf einen Accessor, der letztendlich auf einen Teil in einer Binärdatei verweist, in dem die Inversen Matrizen der Grundstellungen der Knochen gespeichert sind. Die Indizes in dem *joints*-Array verweisen auf Knoten, die in einem Array *nodes* angelegt sind.

4.4.2 glTF-Loader

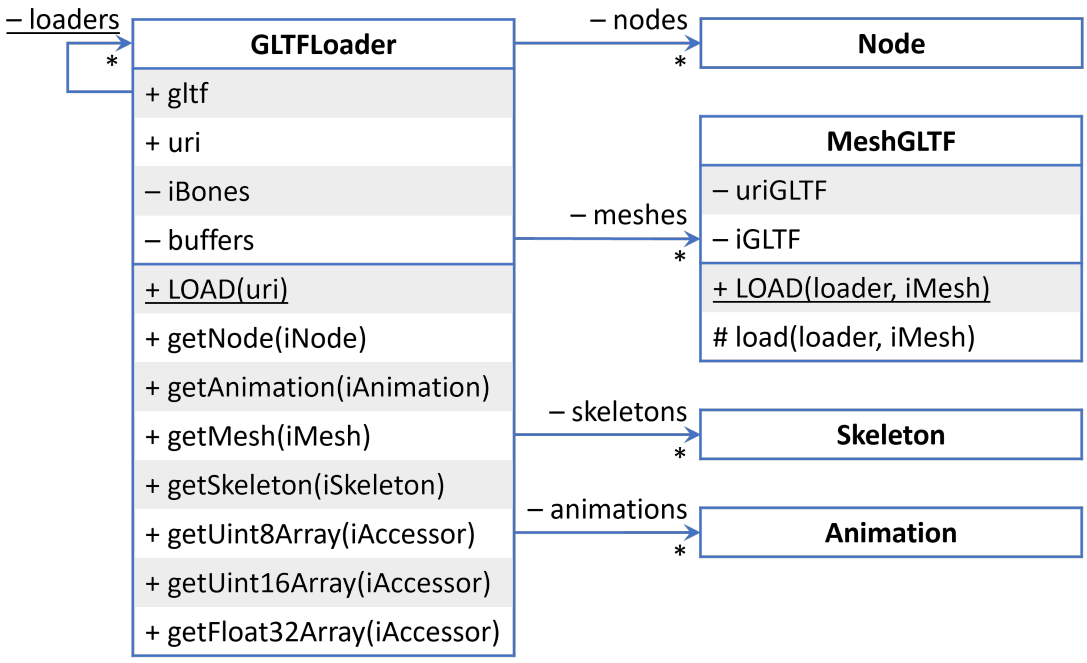

Abbildung 16: Klassendiagramm des glTF-Loader

Der glTF-Loader soll in der Lage sein, verschiedene Assets aus einer glTF-Datei zu laden. Dabei wird erstmal nur die Unterstützung von Knoten, Animationen, Meshes

und Skeletten implementiert. Andere Asset-Typen aus einer glTF-Datei zu laden, ist nicht relevant für das Skelett-System und daher wird vorerst keine Unterstützung für sie implementiert. Einen Knoten laden zu können, ist auch nicht notwendig, jedoch praktisch, um direkt einen Knoten mit Mesh-Komponente und Skelettinstanz laden zu können, ohne diesen selbst zusammensetzen zu müssen. Der glTF-Loader soll nach dem Lazy-Pattern implementiert werden, sodass nicht direkt alle Assets in der glTF-Datei geladen werden, sondern diese nur nach Aufruf in Listen beim Loader hinterlegt werden. So wird nur das nötigste für benötigte Assets geladen und gleichzeitig bereits geladene Assets oder Buffer-Daten wiederverwendet. Außerdem soll jeder Getter im glTF-Loader asynchron definiert sein, sodass wenn ein Asset erst noch geladen werden muss der Thread, in dem der Getter aufgerufen wurde, parallel fortfahren kann. Mit der statischen Methode *LOAD* wird eine Instanz der *GLTFLoader*-Klasse erzeugt. Dabei wird die URI einer glTF-Datei angegeben, sodass diese direkt eingelesen und beim Loader als Eigenschaft gesetzt werden kann. Die glTF-Datei selbst enthält mehrere Listen in denen die Strukturen der verschiedenen Assets beschrieben werden. Die großen Datenmengen stecken aber in den referierten Binärdateien. Auf diese Dateien wird in der glTF-Datei jeweils über ein Uniform Resource Identifier (URI) verwiesen. Um auf Daten in den Binärdateien zuzugreifen, sollen die Getter *getUint8Array*, *getUint16Array* und *getFloat32Array* verwendet werden. Falls der Zugriff auf weitere Array-Typen benötigt wird, müssen entsprechende Getter implementiert werden. Für das Skelett-System sind aber nur diese drei Getter notwendig. Wenn ein Asset geladen werden soll, welches per Accessor auf Daten in einer Binärdatei verweist, so muss der entsprechende Getter aufgerufen werden. Im Falle von Vertices für ein Mesh wird also *getFloat32Array* aufgerufen. Dort wird dann die entsprechende Binärdatei in ein Buffer geladen, dessen Referenz beim Loader hinterlegt wird, sodass auch für andere Assets darauf zugegriffen werden kann. Neben den Gettern, mit dem Assets über deren Namen aufgerufen werden, sollen weitere Getter mit dem Namensmuster *get[AssetType]ByIndex(index)* implementiert werden. Dies ermöglicht Assets auch über deren Index aufrufen zu können. Standardmäßig wird ein Asset über seinen Namen aufgerufen, weil diese Variante für den Creator intuitiver ist.

5 Tests und Anwendungsbeispiele

Das Skelett-System wurde erfolgreich in FUDGE integriert und kann wie in den Anwendungsbeispielen aufgeführt verwendet werden. Ein Skelett kann programmatisch wie folgt definiert werden.

```
const skeleton: f.Skeleton = new f.Skeleton("Skeleton");
skeleton.addBone(
  new f.Node("LowerBone"),
  f.Matrix4x4.TRANSLATION(f.Vector3.Y(0))
);
skeleton.addBone(
  new f.Node("UpperBone"),
  f.Matrix4x4.TRANSLATION(f.Vector3.Y(1)),
  "LowerBone"
);
```

Codeblock 5: Erstellen eines Skeletts

Dabei steht der letzte Parameter beim Hinzufügen von *"UpperBone"* für den Namen des Elternknochens an den der neue Knochen gebunden werden soll. Soll das Skelett an eine Mesh-Komponente gebunden werden, so muss erst eine Skelett-Instanz erstellt werden. Codeblock 6 zeigt, wie eine Skelett-Instanz erstellt und an eine Mesh-Komponente gebunden werden kann.

```
const skeletonInstance: f.SkeletonInstance =
  await f.SkeletonInstance.CREATE(skeleton);
cmpMesh.bind(skeletonInstance);
```

Codeblock 6: Binden eines Skeletts an eine Mesh-Komponente

Dabei gilt es zu beachten, dass die *CREATE*-Methode der *SkeletonInstance*-Klasse, asynchron implementiert ist und daher auf den Abschluss der Methode mit dem await-Schlüsselwort gewartet werden sollte. Sollte die Skelett-Instanz zur Anpassung an das Mesh skaliert werden, gelingt das wie im Codeblock 7 dargestellt.

```
skeletonInstance.addComponent(new f.ComponentTransform(
  f.Matrix4x4.SCALING(f.Vector3.ONE(2)))
);
```

Codeblock 7: Transformieren einer Skelett-Instanz

Wichtig ist, dass die Instanz eines Skeletts transformiert wird und nicht ein Skelett selbst. Damit die Verformung durch das Skelett gerendert wird, muss der Shader *ShaderFlatSkin* verwendet werden. Ein entsprechendes Material ist wie folgt definierbar.

```
const material: f.Material = new f.Material(
  "GreyFlatSkin",
  f.ShaderFlatSkin,
  new f.CoatColored(f.Color.CSS("white"))
);
```

Codeblock 8: Erstellen eines Materials mit dem Flat-Skin-Shader

Um das Skelett schließlich zu animieren, muss vorerst eine Animationsstruktur definiert werden. Diese kann wie im Codeblock 9 dargestellt aussehen.

```
const animationStructure: f.AnimationStructure = {
  bones: {
    "UpperBone": {
      components: {
        ComponentTransform: [ { "f.ComponentTransform": {
          mtxLocal: {
            rotation: {
              z: sequence
            }
          }
        }}]
      }
    }
  }
};
```

Codeblock 9: Definieren einer Animationsstruktur

Hier wird eine Rotation um die z-Achse für den ersten Knochen des Skeletts beschrieben.

Alternativ kann die Struktur über die Abkürzung *mtxBoneLocals* wie im Codeblock 10 zu sehen beschrieben werden.

```
const animationStructure: f.AnimationStructure = {
  mtxBoneLocals: {
    "UpperBone": {
      rotation: {
        z: sequence
      }
    }
  }
};
```

Codeblock 10: Definieren einer Animationsstruktur über die Abbkürzung *mtxBoneLocals*

sequence stellt in beiden Beispielen eine Animationssequenz dar, welche zuvor definiert wurde. Diese kann wie im Codeblock 11 dargestellt aussehen.

```
const sequence: f.AnimationSequence = new f.AnimationSequence();
sequence.addKey(new f.AnimationKey(0, 0));
sequence.addKey(new f.AnimationKey(1000, 45));
sequence.addKey(new f.AnimationKey(2000, 0));
```

Codeblock 11: Erstellen einer Animationssequenz

Der erste Wert eines Schlüsselbilds entspricht dem Eingangswert, in diesem Fall der Zeit in Millisekunden, und der zweite entspricht dem Ausgangswert, also der Rotation in Grad. Dementsprechend beschreibt diese Animationssequenz eine Rotation um 45°, wobei sie mit 0° beginnt und nach 2 Sekunden wieder mit 0° aufhört. Schließlich wird mit der Animationsstruktur eine Animation erstellt, die wie folgt an eine Skelett-Instanz gebunden werden kann.

```
const animation: f.Animation = new f.Animation(
    "Animation",
    animationStructure
);
const cmpAnimator: f.ComponentAnimator =
    new f.ComponentAnimator(animation);
skeletonInstance.addComponent(cmpAnimator);
```

Codeblock 12: Erstellen und Binden einer Animation an eine Skelett-Instanz

Die Abbildung 17 zeigt einzelne Bilder einer Animation von einem skelettanimierten Mesh, das wie in den Anwendungsbeispielen aufgeführt mit Skelett und Animation definiert wurde. Hier wird der Wurzelknochen nach oben verschoben und vergrößert, während ein untergeordneter Knochen rotiert wird. Entsprechend lässt sich in den Bildern erkennen, dass sich der Zylinder nach oben bewegt, vergrößert und in der Mitte knickt. Für weiter oben liegende Punkte ist hier nämlich eine größere Gewichtung für den Kindknochen gesetzt, während für weiter unten liegende Punkte die Gewichtung für den Wurzelknochen höher gesetzt ist. Im dritten Bild ist außerdem zu sehen, dass der Zylinder in der Mitte schrumpft. Dies ist ein bekannter Effekt von SSD. Dabei handelt es sich um das sogenannte Kollabierende-Ellenbogen-Problem. [2]

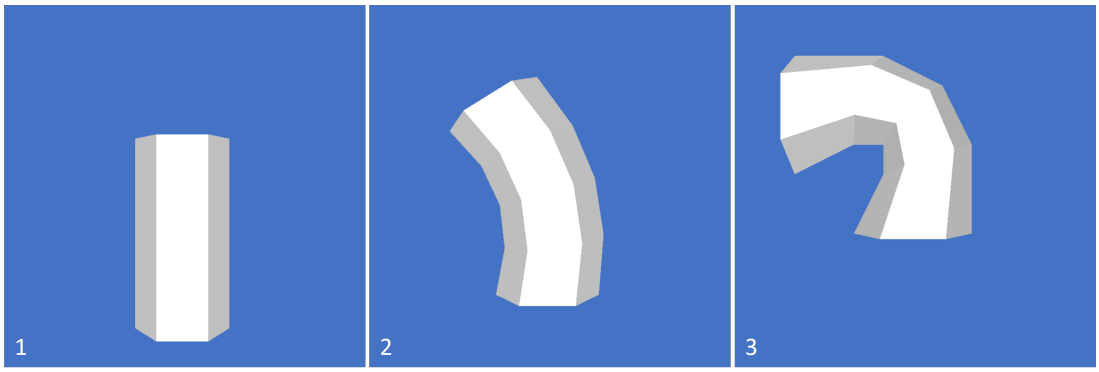

Abbildung 17: Animationsbilder eines skelettanimierten Meshes, welches programmatisch erstellt wurde (eigene Screenshots)

Mit dem *GLTFLoader* kann jetzt ein skelettanimiertes Mesh z. B. wie im Codeblock 13 zu sehen geladen werden.

```
1 const loader: f.GLTFLoader =
2     await f.GLTFLoader.LOAD("./animated_arm.gltf");
3 const arm: f.Node = await loader.getNode("ArmModel");
```

Codeblock 13: Laden eines Knotens aus einer glTF-Datei

Dabei werden ein Knoten zusammen mit einer Mesh-Komponente und einer Skelettinstanz als auch Animierkomponenten mit den Animationen für die Skelettinstanz geladen. Die Animationen werden in diesem Fall automatisch gestartet.

Alternativ kann ein Knoten wie folgt manuell gebaut werden.

```
const arm: f.Node = new f.Node("Arm");

const mesh: f.Mesh = await loader.getMesh("ArmMesh");
const skeleton: f.SkeletonInstance =
    await loader.getSkeleton("ArmSkeleton");
const animation: f.Animation =
    await loader.getAnimation("ArmLift");
skeleton.addComponent(new f.ComponentAnimator(animation));
arm.addComponent(new f.ComponentMesh(mesh, skeleton));

const material: f.Material = new f.Material(
    "ArmMaterial",
    f.ShaderFlatSkin,
    new f.CoatColored()
);
arm.addComponent(new f.ComponentMaterial(material));
```

Codeblock 14: Manuelles zusammensetzen eines Knotens aus einer glTF-Datei

Die Abbildung 18 zeigt einzelne Bilder aus einer gerenderten Animation eines skelettanimierten Meshes, welches wie in den Anwendungsbeispielen geladen wurde.

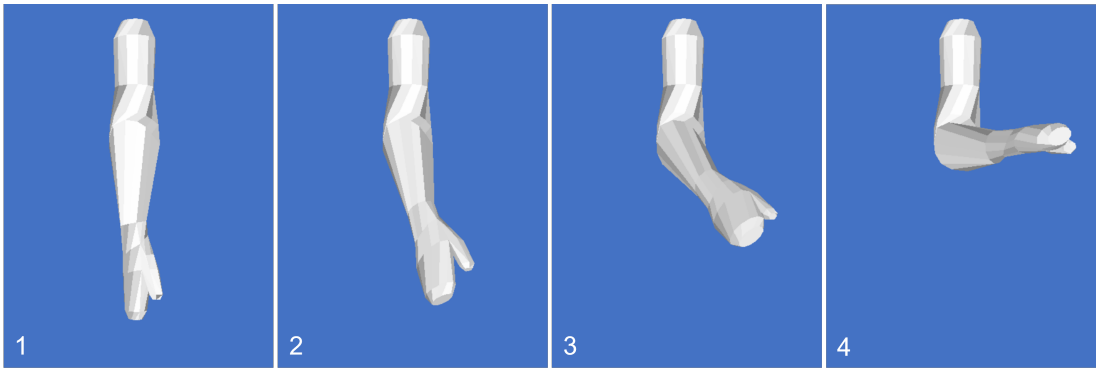

Abbildung 18: Animationsbilder eines skelettanimierten Meshes, welches importiert wurde (eigene Screenshots)

6 Fazit

Ziel dieser Arbeit war es ein Skelett-System in FUDGE einzubinden. Dafür sollten aktuelle Implementationen anderer Engines betrachtet werden und auf Integrationsmöglichkeiten untersucht werden. Dabei wurde festgestellt, dass eine Integration ohne größeren Aufwand nicht möglich und es durch gegebene Strukturen von FUDGE einfacher ist, ein eigenes Skelett-System zu entwickeln.

Die Untersuchung sollte als Grundlage für das eigene Skelett-System dienen. Dementsprechend wurde ein erster Entwurf für eine Klassenstruktur abgeleitet. Diese wurde zusammen mit einem neuen Shader implementiert und über mehrere Iterationen von Anpassungen in die gegebenen Strukturen von FUDGE eingebettet, sodass letztendlich ein skelettanimiertes Mesh in FUDGE erstellt und gerendert werden kann. Abschließend wurden Tests geschrieben, um die Funktion des Skelett-Systems zu prüfen. Letztendlich konnten erwartete Bilder einer Animation gerendert werden. Damit ist das Einbinden eines Skelett-Systems in FUDGE erfolgreich abgeschlossen.

Über das Ziel hinaus, ein Skelett-System in FUDGE einzubinden, wurde ein glTF-Loader entwickelt, um auch das Laden von Skelettanimationen aus glTF-Dateien zu ermöglichen. Dazu wurde das Dateiformat untersucht, wodurch dessen Strukturen Klassen aus FUDGE zugeordnet werden konnten.

Eine einfache Anwendbarkeit des Systems wurde berücksichtigt z. B. dass Knochen über deren Namen bei einem Skelett gefunden werden können, damit sich ein Creator keine Indizes merken muss. Ebenso werden beim glTF-Loader Assets über deren Namen geladen. Außerdem können Objekte der Klassen *Skeleton* und *MeshSkin* serialisiert werden.

7 Ausblick

7.1 Inverse Kinematik

Das Skelett-System, welches in dieser Arbeit entwickelt wurde, unterstützt nur Vorwärtskinematik. Dabei sind für eine hierarchische Kette von Gelenken die Winkelstellungen bekannt, jedoch nicht der resultierende Endpunkt der Kette. Das heißt in einer Animation werden die einzelnen Gelenke animiert. Bei inverser Kinematik hingegen wird ein gewünschter Endpunkt gesetzt und die Winkelstellungen der einzelnen Gelenke werden berechnet. Eine Animation wird durch einen sogenannten Endeffektor kontrolliert. Die inverse Kinematik kann also das Ausrichten zusammenhängender Knochen vereinfachen. Sie ist vor allem wichtig, um z. B. die Hand eines menschlichen Charakters zu einem Gegenstand hinzubewegen, oder um dessen Füße den Boden anzupassen. [19]

7.2 Morph Target Animation

Morph-Target-Animationen werden vor allem dazu verwendet Gesichtsausdrücke zu animieren. Bei Morph-Target-Animationen werden mehrere Formen bzw. Varianten von einem Mesh definiert, zwischen denen interpoliert wird. Das heißt ein Mesh wird mehrmals definiert, wobei die Positionen der einzelnen Punkte variieren. So kann die Form eines Meshes für ein Schlüsselbild genau definiert werden. Dadurch lässt sich ein Gesicht in fest modellierte Gesichtsausdrücke verformen bzw. verschmelzen. Schwierig wird es bei dieser Technik, wenn es darum geht Rotationen zu animieren, da in der Regel linear interpoliert wird. [20]

Trotz allem können Morph Targets dabei helfen, in Skelettanimationen bei bestimmten Schlüsselbildern das Mesh in eine fest definierte Form zu bringen, um so Effekte wie kollabierende Ellenbogen zu kaschieren. Jedoch gibt es auch andere Ansätze zur Lösung des Kollabierenden-Ellenbogen-Problems, welche in dem Werk "SIGGRAPH Course 2014 — Skinning: Real-time Shape Deformation Part I: Direct Skinning Methods and Deformation Primitives" [4] aufgeführt werden und zur Verbesserung des Skelett-Systems betrachtet werden können.

7.3 FBX

Filmbox (FBX) ist ein Dateiformat zum Speichern von 3D-Modellen, unter anderem auch mit Skelettanimationen, welches sehr verbreitet ist und vor Allem von Mixamo als Export-Format verwendet wird. Daher ist es Sinnvoll auch eine Unterstützung von diesem Format in FUDGE zu implementieren. Dieses Format ist jedoch nicht für den Creator leserlich und speichert die Daten des Modells rein binär im Gegensatz zu glTF, weshalb sich diese Arbeit vorerst mit der Unterstützung von glTF beschäftigt hat.

Literaturverzeichnis

[1] J. Lasseter, *Toy Story*. The Walt Disney Company, 1995.

[2] J. P. Lewis, M. Cordner und N. Fong, "Pose space deformation: A unified approach to shape interpolation and skeleton-driven deformation," *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, S. 167, 2000.

[3] P. J. Dell'Oro-Friedl. (2021). "Das geschlechtergerechte Neutrum," [Online]. Verfügbar unter: `https://jirkadelloro.github.io/Neutrum.pdf`. Zugriff am: 01.02.2022.

[4] L. Kavan. (2014). "SIGGRAPH Course 2014 — Skinning: Real-time Shape Deformation, Part I: Direct Skinning Methods and Deformation Primitives," [Online]. Verfügbar unter: `https://www.skinning.org/direct-methods.pdf`. Zugriff am: 01.02.2022.

[5] *WebGL 2.0 Quick Reference*. Khronos Group, Feb. 2017.

[6] *glTF - What the duck? An overview of the basics of the GL Transmission Format For glTF 2.0!* Jan. 2019. [Online]. Verfügbar unter: `https://github.com/KhronosGroup/glTF/blob/main/specification/2.0/figures/gltfOverview-2.0.0b.png`, Zugriff am: 18.11.2021.

[7] *Skeleton - three.js docs*. [Online]. Verfügbar unter: `https://threejs.org/docs/#api/en/objects/Skeleton`, Zugriff am: 07.09.2021.

[8] *Bone - three.js docs*. [Online]. Verfügbar unter: `https://threejs.org/docs/#api/en/objects/Bone`, Zugriff am: 07.09.2021.

[9] *Object3D - three.js docs*. [Online]. Verfügbar unter: `https://threejs.org/docs/#api/en/core/Object3D`, Zugriff am: 07.09.2021.

[10] *SkinnedMesh - three.js docs*. [Online]. Verfügbar unter: `https://threejs.org/docs/#api/en/objects/SkinnedMesh`, Zugriff am: 07.09.2021.

[11] *BufferGeometry - three.js docs*. [Online]. Verfügbar unter: `https://threejs.org/docs/#api/en/core/BufferGeometry`, Zugriff am: 07.09.2021.

[12] *SkeletonAnimator - Away3D Documentation*. [Online]. Verfügbar unter: `http://away3d.com/livedocs/away3d/4.1Alpha/away3d/animators/SkeletonAnimator.html`, Zugriff am: 09.09.2021.

[13] *Skeleton - Away3D Documentation*. [Online]. Verfügbar unter: `http://www.away3d.com/livedocs/away3d/4.0/away3d/animators/data/Skeleton.html`, Zugriff am: 09.09.2021.

[14] *SkeletonJoint - Away3D Documentation*. [Online]. Verfügbar unter: http://w-ww.away3d.com/livedocs/away3d/4.0/away3d/animators/data/SkeletonJoint.html, Zugriff am: 09.09.2021.

[15] *SkeletonPose - Away3D Documentation*. [Online]. Verfügbar unter: http://w-ww.away3d.com/livedocs/away3d/4.0/away3d/animators/data/SkeletonPose.html, Zugriff am: 09.09.2021.

[16] *JointPose - Away3D Documentation*. [Online]. Verfügbar unter: http://w-ww.away3d.com/livedocs/away3d/4.0/away3d/animators/data/JointPose.html, Zugriff am: 09.09.2021.

[17] *SkeletonAnimationSet - Away3D Documentation*. [Online]. Verfügbar unter: http://away3d.com/livedocs/away3d/4.1Alpha/away3d/animators/SkeletonAnimationSet.html, Zugriff am: 09.09.2021.

[18] *SkeletonClipNode - Away3D Documentation*. [Online]. Verfügbar unter: http://w-ww.away3d.com/livedocs/away3d/4.0/away3d/animators/nodes/SkeletonClipNode.html, Zugriff am: 09.09.2021.

[19] J. Hagler. (Feb. 2006). "Kinematik: FK / IK," [Online]. Verfügbar unter: http://www.dma.ufg.ac.at/app/link/Grundlagen%3A3D-Grafik/module/14174?step=all. Zugriff am: 15.02.2022.

[20] C. Liu, *An Analysis of the Current and Future State of 3D Facial Animation Techniques and Systems*. 2006, Seite 12-15.

Eidesstattlicher Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Schramberg, 23.02.2022 Matthias Roming