Techniken der 2D-Skelettanimation

Studiengang Informatik

Bachelorarbeit

vorgelegt von

Benedikt Jensen
geb. in Langenfeld (Rheinland)

durchgeführt an:
Technische Hochschule Mittelhessen (THM), Gießen

Referent der Arbeit:	Prof. Dr. Andreas Gogol-Döring
Koreferent der Arbeit:	Herr Lukas Gail

Gießen, 2022

Danksagung

Ich bedanke mich bei Professor Dr. Gogol-Döring, welcher sich bereit erklärte, die Rolle des Betreuers für meine Bachelorarbeit zu übernehmen. Er unterstützte mich mit zielführendem Feedback und ermutigte mich, als ich mir unsicher war. Ich spreche auch meinen Dank an Lukas Gail, meinen Koreferenten, aus. Er nahm sich die Zeit, meine Arbeit sorgfältig Korrektur zu lesen und gab mir Ratschläge, um meine Arbeit verständlicher und informativer zu machen. Sein Feedback trug entscheidend zur Vervollständigung der Arbeit bei. Des Weiteren möchte ich meiner Familie für ihr Korrekturlesen danken.

Selbstständigkeitserklärung

Ich erkläre, dass ich die eingereichte Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Gießen, November 2022

Benedikt Jensen

Inhaltsverzeichnis

Zusammenfassung

Diese Arbeit untersucht Techniken zur 2D-Skelettanimation, welche vorwiegend in interaktiven Anwendungen wie z.B. Videospielen genutzt wird. Als Ziel der Arbeit soll ein Editor zur Erstellung von 2D-Skelettanimationen erstellt werden. In Vorbereitung darauf wird zunächst definiert, was eine erstrebenswerte Animation ist und wie ein guter Animations-Workflow aussieht.

Anschließend werden die Datenstrukturen und Techniken erklärt, welche als Bestandteile des Editors implementiert werden. Ein fundamentales Konzept der Skelettanimation ist das Aufteilen des zu animierenden Körpers in Skin, welches der sichtbare Teil ist, und Skelett, welches die Datenstruktur ist, die der Transformation der Skin zugrunde liegt. Auf diesem Konzept aufbauend können Keyframes (engl. für Schlüsselbild) erstellt werden. Durch Interpolation wird aus diesen Keyframes eine flüssige Animation generiert.

Erweiternde Techniken ermöglichen den flüssigen Übergang zwischen Animationen, das Hinzufügen physikalisch simulierter Komponenten als Teil einer Animation, *Inverse Kinematics* zur Beschleunigung des Animationsprozesses und zur flexiblen Anpassung einer Animation während der Laufzeit, das Generieren und Verformen von Meshes und Weiteres.

Abschließend wird ein Ausblick auf Techniken gegeben, welche nicht näher im Umfang dieser Arbeit untersucht werden, jedoch die Qualität von Animationen und den Workflow des Animierens weiter verbessern können.

Kapitel 1

Einleitung

1.1 Ausgangslage

Animation ist die Technik, durch Darstellen einer Sequenz von Einzelbildern die Illusion von flüssiger Bewegung zu schaffen. Ursprünglich wurden Animationen Bild für Bild erstellt. Jedes Bild wurde von Hand gezeichnet. Dieses Vorgehen beansprucht allerdings sehr viel Zeit und Änderungen können nach Fertigstellung der Animation nur mit großem Aufwand erreicht werden. Ihre Flexibilität ist sehr stark begrenzt. Mit Entwicklung der Technik haben sich auch die Methoden zur Animationserstellung verändert. Heutzutage werden Animationen größtenteils mithilfe von Computern erstellt. Eine bewährte Technik der Computeranimation ist die Skelettanimation. (vgl. [Cor17])

Im Folgenden wird zwischen 3D- und 2D-Animation unterschieden. Als 3D-Animation versteht sich eine Animation, welche mit einem 3D-Modell als Basis erstellt wird. Bei 2D-Animationen ist dies nicht der Fall. Obwohl Skelettanimation sowohl in 3D als auch in 2D möglich ist, wird sie vorwiegend für 3D-Animationen verwendet, z.B. für das Erstellen von 3D-Animationsfilmen. Das hat den im Folgenden erklärten Hintergrund. (vgl. [PW94]: 832)

Für das Animieren einer Person wird nur ein einziges 3D-Modell benötigt, welches durch Transformationen in jede gewünschte Position gebracht werden kann. Die Natur eines 3D-Modells ermöglicht es, die Person aus jedem beliebigen Betrachtungswinkel abzubilden.

Im 2D-Bereich ist das nicht möglich. Die 2D-Skelettanimation wird mittels einer oder mehrerer Bilddateien verwirklicht. Ein Bild kann einen Körper allerdings, im Gegensatz zu einem 3D-Modell, nur von einer einzigen festgelegten Perspektive her abbilden. Für jede weitere Perspektive ist ein weiteres Bild erforderlich. In fast allen Bewegungsabläufen kommt es allerdings zur Drehung des ganzen oder eines Teils des Körpers. Das ist der Grund dafür, dass 2D-Skelettanimationen oft etwas unnatürlich wirken. Es existieren allerdings fortgeschrittene Techniken, welche diesem Problem entgegenwirken. Beispielsweise kann die Skelettanimation mit der Einzelbildanimation (siehe Abschnitt 6.2.1) kombiniert werden. Auch durch

gezielte Mesh-Deformierung kann die Illusion eines Perspektivenwechsels geschaffen werden. (vgl. [PW94]: 833)

Zudem ist nicht bei allen Animationen Realismus der schwerwiegendste Faktor. Besonders in der Entwicklung von 2D-Videospielen ist es oft wichtiger, dass eine Animation effizient erstellt und flexibel angewandt werden kann. Variation und die Möglichkeit zur Personalisierung spielen eine große Rolle. Die Skelettanimation ist in diesem Bereich weit verbreitet, da sie all diese Vorteile mit sich bringt. Es ist möglich, die gleiche Animation für beliebig viele Charaktere zu verwenden, auch wenn deren Erscheinungen nicht identisch sind (vgl. [PW94]: 833). Des Weiteren ermöglicht die Skelettanimation mithilfe prozeduraler Animationstechniken eine größere Interaktion des animierten Charakters mit der Spielwelt.

1.2 Zielsetzung

Ziel der Arbeit ist es, einen Editor für 2D-Skelettanimationen zu implementieren. Der Editor soll nutzerfreundlich sein und es ermöglichen, mit geringem Aufwand lebendig wirkende Animationen zu erstellen. Als grober Maßstab werden die 12 Animationsprinzipien von Johnston und Thomas [TJ81] herangezogen.

Der gewünschte Workflow lässt sich in 4 Schritte aufteilen:

1) Erstellen bzw. Zusammenstellen der Grafiken, welche die Skin bilden
2) Erstellen des Skeletts
3) Erstellen der Skin, bzw. Anbinden der Grafiken an das Skelett
4) Erstellen der Keyframes, welche den Bewegungsablauf bilden

Nach Möglichkeit sollen Arbeitsschritte automatisiert werden. Animationen sollen gespeichert und geladen werden können. Die Anwendung soll letztendlich per WebAssembly auf einer Website installiert werden und frei zugänglich sein.

1.3 Aufbau der Arbeit

Zunächst werden die Grundlagen erläutert. Dazu gehört eine knappe Erklärung der 12 Animationsprinzipien und des Technologie-Stacks. Des Weiteren werden mathematische Grundlagen zu Quaternionen erklärt, welche als Repräsentation von Rotationen verwendet werden.

Im Anschluss werden grundlegende Prinzipien der Skelettanimation und notwendige Datenstrukturen erklärt. Dazu gehören die Natur eines Animationsskeletts und deren Skin,

Forward Kinematics, Keyframes und Interpolation.

Darauf aufbauend werden Techniken untersucht, welche den Animationsprozess einfacher oder effizienter gestalten und Techniken, welche die Möglichkeiten der Animationssoftware erweitern. Dazu gehören Mesh-Deformierung, Inverse Kinematics, Physiksimulation und Animation-Layering und -Blending.

Abschließend wird ein Ausblick gegeben. Es wird evaluiert, inwiefern gesetzte Ziele erreicht wurden und es werden Techniken aufgezeigt, die aufgrund begrenzter Zeit nicht implementiert wurden, welche jedoch die Anwendung in Nutzbarkeit und Funktionalität weiter verbessern könnten.

Kapitel 2

Grundlagen

2.1 Prinzipien der Animation

Frank Thomas und Ollie Johnston veröffentlichten 1981 das Buch *The Illusion of Life: Disney Animation* [TJ81]. In diesem erklären sie zwölf Prinzipien, deren Einhaltung bei der Erstellung von Animationen zur Illusion lebendiger Figuren und Welten maßgebend beiträgt.

1. Squash and Stretch *Squash and Stretch*, auf Deutsch Stauchung und Streckung, bringt die Wirkung von Kräften auf eine Figur zum Ausdruck. Diese führen dazu, dass die Figur zusammengedrückt oder in die Länge gezogen wird. Das Volumen der Form muss dabei erhalten bleiben, um die gewünschte Wirkung zu erhalten.

2. Anticipation Anticipation, auf Deutsch Vorwegnahme, dient dazu, die Aufmerksamkeit des Betrachters auf eine bestimmte Stelle zu lenken, kurz bevor eine Aktion eintrifft. Dieses Prinzip ist z.B. beim Ausholen einer Person vor dem Wurf eines Balls sichtbar. Durch Befolgen dieses Prinzips werden Aktionen lesbarer und wirken natürlicher.

3. Staging Staging, auf Deutsch Inszenierung, ist die klare Präsentation einer Idee. Der Fokus sollte auf dem liegen, was wichtig ist. Ein Weg, das zu erreichen, ist die zentrale Positionierung wichtiger Elemente in einer Szene. Des Weiteren kann die Bewegung aller weniger wichtigen Objekte in einer Szene minimiert werden, um nicht von dem Hauptgeschehen abzulenken.

4. Straight Ahead & Pose-to-Pose Die Einzelbilder einer Animation können in Sequenz erstellt werden (Straight Ahead). Alternativ können vorerst Schlüsselpositionen erstellt werden, zwischen welchen die übrigen Einzelbilder eingefüllt werden (Pose-to-Pose). *Straight Ahead* eignet sich besser zur Erstellung flüssiger, realistischer Bewegungen. *Pose-to-Pose* bietet mehr Kontrolle, insbesondere bezüglich *Timing*.

5. Follow Through & Overlapping Action *Follow Through*, auf Deutsch zu Ende bringen, beschreibt das Verhalten von Teilen einer Figur, sich nach Vollendung der Hauptaktion weiterzubewegen. Das Haar einer Person bewegt sich für einen Moment weiter, nachdem eine vorher laufende Person zum Stillstand gekommen ist. *Overlapping Action*, auf Deutsch überlappende Handlung, beschreibt die Natur von Teilen einer Figur, sich versetzt voneinander zu bewegen. Läuft eine Person, so bewegt sich der Kopf meist versetzt von Armen und Beinen.

6. Slow In & Slow Out Das Prinzip *Slow In & Slow Out*, auf Deutsch Beschleunigung und Verlangsamung, besagt, dass Bewegungen meist langsam beginnen, dann schneller werden und dann wieder langsamer werden, bevor sie zum Halt kommen. Das liegt am Gesetz der Trägheit, welchem zufolge ein Körper dazu neigt, seine Geschwindigkeit beizubehalten. Eine Änderung der Geschwindigkeit bedarf einer äußeren Krafteinwirkung.

7. Arcs Dem Prinzip Arcs (engl. für Bogen) nach folgen die meisten Bewegungen einem Bogen. Beim waagerechten Wurf hat das geworfene Objekt eine konstante horizontale Geschwindigkeit und beschleunigt mit konstanter Geschwindigkeit, aufgrund der Erdbeschleunigung, nach unten. Das resultiert in einer bogenförmigen Flugbahn. Ein Mensch bewegt seine Gliedmaßen in einem Bogen um die dazugehörigen Gelenke.

8. Secondary Action Das Ergänzen einer Animation um eine *Secondary Action* (engl. für Nebenhandlung), kann der Figur mehr Charakter geben und die Haupthandlung betonen. Dabei sollte die Nebenhandlung nicht von der Haupthandlung ablenken. Es ist auch möglich, mit der Nebenhandlung zusätzliche Informationen zu vermitteln. Tut eine Person z.B. etwas Verbotenes, signalisiert ein Umherblicken, dass sie nicht entdeckt werden will.

9. Timing Die Wirkung eines Bewegungsablaufs kann sich abhängig vom Timing verändern. Der gezielte Einsatz von *Timing* kann Intention und Gefühlslage einer Figur und auch die Beschaffenheit eines Objekts zum Ausdruck bringen.

10. Exaggeration *Exaggeration*, auf Deutsch Übertreibung, macht Posen und Handlungen animierter Figuren deutlicher. Sie macht die Animation dynamischer, gibt ihr Ausdrucksstärke und hilft dabei, dass dem Betrachter wichtige Details nicht entgehen.

11. Solid Drawing *Solid Drawing*, auf Deutsch solides Zeichnen, betont die Wichtigkeit, dass ein Körper über den Zeitraum der Animation hinweg seine charakteristische Struktur konsistent beibehält. Ein starrer Körper behält seine Struktur unabhängig davon, aus welcher Perspektive er betrachtet wird. Wird *Solid Drawing* nicht befolgt, so kann die räumliche Struktur nicht vermittelt werden, oder der Körper scheint sich zu verformen.

12. Appeal *Appeal*, auf Deutsch Charme, besagt, dass das Aussehen einer Figur gefällig und dem Charakter der Figur angemessen sein soll. Die inneren Qualitäten einer Figur sollen effektiv dargestellt werden.

2.2 Technologie-Stack

Der Animationseditor wird mit der Programmiersprache Rust und der Game-Engine Bevy implementiert. In Rust ist es unkompliziert, Code für verschiedene Plattformen, darunter auch Web, zu kompilieren. Bevy stellt für dieses Projekt hilfreiche Funktionalitäten bereit, welche in Abschnitt 2.2.2 erläutert werden.

2.2.1 Programmiersprache Rust

Nach Jung ist Rust "eine junge systemnahe Programmiersprache, die es sich zum Ziel gesetzt hat, die Lücke zu schließen zwischen Sprachen mit hohem Abstraktionsniveau, die vor Speicher- und Nebenläufigkeitsfehlern schützen, und Sprachen mit niedrigem Abstraktionsniveau, welche dem Programmierer detaillierte Kontrolle über die Repräsentation von Daten und die Verwaltung des Speichers ermöglichen." ([Jun20]: Zusammenfassung)

Ownership *Jedes Programm muss die Allokation und Deallokation von Speicher regeln. Einige Sprachen verwenden einen Garbage Collector, um nicht mehr gebrauchten Speicher freizugeben, in anderen Sprachen muss der Programmierer Speicher explizit allokieren und freigeben. Rust verwendet einen dritten Ansatz: Ownership.* (vgl. [KN19]: 75-76) Das Ownership-Prinzip bringt einige Regeln mit sich, welche es in Rust zu befolgen gilt:

- Jedes Objekt hat eine Variable, genannt *Eigentümer*.
- Es kann nur einen *Eigentümer* auf einmal geben.
- Wenn der Geltungsbereich des *Eigentümers* endet, wird der eingenommene Speicher wieder freigegeben.
- Es können beliebig viele Referenzen auf ein Objekt ausgeliehen werden, solange keine von ihnen eine mutable Referenz ist. Ist eine gültige mutable Referenz vorhanden, kann es keine weiteren Referenzen auf das Objekt geben.

Die Einführung dieser Regeln macht es möglich, auf einen Garbage Collector zu verzichten und somit Speicher und Rechenaufwand zu sparen.

2.2.2 Game Engine Bevy

Bevy ist eine junge Game-Engine, programmiert in Rust, welche das Entity-Component-System Paradigma (ECS) verwendet (vgl. [bev]: Getting Started → ECS). ECS ist ein Lösungsansatz zu Problemen, welche Objektorientierte Programmierung und Mehrfachvererbung mit sich bringen. Es wird vorwiegend in Game-Engines verwendet. Die Erfindung von ECS wird Adam Martin zugeschrieben, aber schon zuvor experimentierten andere Teams mit ähnlichen Paradigmen. (vgl. [HCSZ21]: 6-7)

ECS teilt eine Software in Entitäten, Komponenten und Systeme auf. Entitäten sind einzigartig und ihnen wird jeweils eine Gruppe von Komponenten zugewiesen. Systeme führen Programmlogik auf diesen Komponenten aus. Dieses Vorgehen ermöglicht eine starke Modularisierung und erhöht die Wiederverwendbarkeit von Quellcode. Zudem ermöglicht es die Parallelisierung von Systemen, solange Datenzugriffe sich dadurch nicht überschneiden. Somit kann mithilfe von ECS eine erhöhte Performanz erreicht werden. (vgl. [HCSZ21]: 6-9)

Bevy stellt grundlegende Funktionalitäten zur Entwicklung von Videospielen bereit. Dazu gehören unter anderem ein Game-Loop und die Verarbeitung von Benutzereingaben. Es ist eine Grafik-API vorhanden, welche plattformunabhängig ist. Sie erlaubt das Lesen, Transformieren und Rendern von Grafiken, sowie den Einsatz von textuierten Meshes.

2.3 Quaternionen

Der Animationseditor verwendet Quaternionen zur Speicherung und Berechnung von Rotationen, da diese auch Bevy-intern verwendet werden. Quaternionen werden häufig in der Grafikprogrammierung genutzt. Sie ermöglichen es, eine Rotation um eine beliebige Achse zu beschreiben. Unter anderem haben sie gegenüber den *Eulerschen Winkeln* den Vorteil, dass sie das Auftreten eines *Gimbal Locks* ausschließen. (vgl. [Muk02]: 1)

2.3.1 Mathematische Eigenschaften

Erklärungen bezüglich Quaternionen beruhen auf dem Text *Mathematische Grundlagen der Quaternionen* [Kun19]. Quaternionen sind vierdimensionale Vektoren mit der Basis 1, i, j, k. Eine Quaternion hat die Form

$$Q = 1 \cdot q_0 + iq_1 + jq_2 + kq_3. \tag{2.1}$$

Die Koeffizienten q_0, q_1, q_2 und q_3 sind reelle Zahlen. Für die Basiselemente i, j und k gilt:

$$\begin{array}{cccc} ij = k & jk = i & ki = j & ii = jj = kk = -1 \\ ji = -k & kj = -i & ik = -j & ijk = -1 \end{array} \tag{2.2}$$

Die Menge der Quaternionen wird als $\mathbb{H}$ bezeichnet. Der Realteil oder Skalarteil der Quaternion ist q_0, und der Imaginärteil oder Vektorteil der Quaternion ist $q = (q_1, q_2, q_3)$. Eine Quaternion, deren Imaginärteil 0 ist, wird reale Quaternion genannt. Hingegen wird eine Quaternion, deren Realteil 0 ist, reine Quaternion genannt. Das neutrale Element der Quaternionen-Multiplikation ist die reale Quaternion $Q = (1, 0, 0, 0)$. Die Konjugierte $\overline{Q}$ einer Quaternion wird gebildet, indem der Imaginärteil negiert wird.

$$\overline{Q} = q_0 - iq_1 - jq_2 - kq_3 \tag{2.3}$$

Das Skalarprodukt $\langle P, Q \rangle$ zweier Quaternionen P und Q ist definiert durch

$$\langle P, Q \rangle = p_0q_0 + p_1q_1 + p_2q_2 + p_3q_3. \tag{2.4}$$

Das Produkt zweier Quaternionen ist eine weitere Quaternion. Quaternionen-Multiplikation ist assoziativ

$$P(QR) = (PQ)R = PQR \tag{2.5}$$

aber nicht kommutativ.

$$\exists P, Q \in \mathbb{H} : PQ \neq QP \tag{2.6}$$

Zur Berechnung des Produkts werden die Rechenregeln aus Formel 2.2 und das Distributivgesetz angewandt:

$$\begin{aligned} QP &= (q_0 + iq_1 + jq_2 + kq_3)(p_0 + ip_1 + jp_2 + kp_3) \\ &= q_0p_0 + iq_0p_1 + jq_0p_2 + kq_0p_3 \\ &\quad + iq_1p_0 + i^2q_1p_1 + ijq_1p_2 + ikq_1p_3 \\ &\quad + jq_2p_0 + jiq_2p_1 + j^2q_2p_2 + jkq_2p_3 \\ &\quad + kq_3p_0 + kiq_3p_1 + kjq_3p_2 + k^2q_3p_3 \\ &= q_0p_0 - q_1p_1 - q_2p_2 - q_3p_3 \\ &\quad + (q_0p_1 + q_1p_0 + q_2p_3 - q_3p_2)i \\ &\quad + (q_0p_2 - q_1p_3 + q_2p_0 + q_3p_1)k \\ &\quad + (q_0p_3 + q_1p_2 - q_2p_1 + q_3p_0)j \end{aligned} \tag{2.7}$$

Die Norm $|Q|$ einer Quaternion Q ist definiert durch

$$|Q| = q_0^2 + q_1^2 + q_2^2 + q_3^2. \tag{2.8}$$

Eine Quaternion mit Norm $N(Q) = 1$ wird Einheitsquaternion genannt.

2.3.2 Drehung von Vektoren

Eine Einheitsquaternion kann eine Rotation repräsentieren. In diesem Fall wird sie auch Rotationsquaternion genannt. Anders als bei Winkelrechnungen in Grad oder Radianten, ist es das Produkt von zwei Rotationsquaternionen, welches die Kombination der beiden Rotationen bildet. Zur Berechnung des gedrehten Vektors w eines 3D-Vektors v mittels einer Rotationsquaternion Q, wird die Quaternion mit dem 3D-Vektor und der Konjugierten der Quaternion multipliziert:

$$w = Qv\overline{Q} \tag{2.9}$$

Um die Quaternion mit dem Vektor zu multiplizieren, wird zunächst eine Quaternion aus dem Vektor gebildet. Dazu wird der Vektor einfach um den Realteil 0 ergänzt. Somit ergibt sich die folgende Formel.

$$w = (q_0 + iq_1 + jq_2 + kq_3)(0 + iv_1 + jv_2 + kv_3)(q_0 - iq_1 - jq_2 - kq_3) \tag{2.10}$$

Wie [Kun19] zeigt, lässt sich diese Gleichung wie folgt vereinfacht schreiben:

$$w = (q_0^2 - |q|^2)v + 2\langle q, v\rangle q + 2q_0(q \times v) \tag{2.11}$$

Der folgende Pseudocode dreht einen 3D-Vektor v mit der Rotationsquaternion q. Die Benennung der Quaternionkomponenten folgt der Bevy-Norm. Hier bilden x, y und z den Vektorteil, während w der Realteil ist.

```
// Drehe den Vektor v mit der Quaternion q
fn quat_rot_vec3(v: Vec3, q: Quat) -> Vec3 {
    // Vektorteil
    let q_vec = Vec3::new(q.x, q.y, q.z);

    // Rechnung nach Formel 2.11
    res = 2.0 * dot_product(q_vec, v) * q_vec
          + (q.w*q.w - dot_product(q_vec, q_vec)) * v
          + 2.0 * q.w * cross_product(q_vec, v);

    return res;
}

// Skalarprodukt zweier Vektoren
fn dot_product(u: Vec3, v: Vec3) -> f32 {
    return u.x*v.x + u.y*v.y + u.z*v.z;
}

// Kreuzprodukt zweier Vektoren
fn cross_product(u: Vec3, v: Vec3) -> Vec3 {
    let x = u.y*v.z - u.z*v.y;
    let y = u.z*v.x - u.x*v.z;
    let z = u.x*v.y - u.y*v.x;
    return Vec3::new(x, y, z);
}
```

Listing 2.1: Drehung eines Vektors mit einer Quaternion

Stellt der Einheitsvektor $u = iu_1 + ju_2 + ku_3$ die Rotationsachse dar, so lässt sich die Rotationsquaternion Q mit $|Q| = 1$, welche eine Rotation um diese Achse und um den Winkel θ repräsentiert, wie folgt bestimmen:

$$Q = cos\left(\frac{\theta}{2}\right) + u \cdot sin\left(\frac{\theta}{2}\right) \tag{2.12}$$

Da unsere Anwendung ein 2D-Animationseditor ist, rotieren wir immer um die z-Achse, welche vom Einheitsvektor $(0, 0, 1)$ dargestellt wird. Deshalb können wir $u = k$ festlegen. Es folgt ein Pseudocode der Funktion $quat\_from\_angle$, welche eine Rotationsquaternion zurückgibt. Diese Quaternion repräsentiert eine Drehung gegen den Uhrzeigersinn um den Winkel $angle$:

```
fn quat_from_angle(angle: f32) {
    // Realteil = cos(theta / 2)
    let w = (angle / 2.0).cos();

    // Vektorteil = sin((0,0,1) / 2)
    let z = (angle / 2.0).sin();

```

```
8       // Die Komponenten x und y haben in 2D den Wert 0
9       return Quat::from_xyzw(0.0, 0.0, z, w);
10  }
```

Listing 2.2: Drehung eines Vektors mit einer Quaternion

2.3.3 Quaternionen-Interpolation

Zwischen zwei Rotationsquaternionen kann interpoliert werden, z.B. um eine flüssige Bewegung von einer Ausrichtung zu einer anderen darzustellen. Der einfachste Weg das zu tun, ist, linear zwischen den beiden Quaternionen zu interpolieren. Die interpolierte Quaternion lässt sich bei $Q, P \in \mathbb{H}$ und $x = [0, 1], x \in \mathbb{R}$ mit der Formel

$$quat\_lerp(Q, P, x) = Q \cdot (1 - x) + P \cdot x \tag{2.13}$$

bestimmen. Das Ergebnis ist eine Quaternion, deren Norm meist nicht 1 ist, und muss daher normalisiert werden, bevor es als Rotationsquaternion weiter verwendet werden kann. Diese Art zwischen zwei Quaternionen zu interpolieren ist recheneffizient. Aber sie hat den Nachteil, dass sie nicht linear zwischen den Winkelpositionen, welche von den Quaternionen repräsentiert werden, interpoliert. Stattdessen beginnt eine zwischen Q und P durch Interpolation berechnete Rotation langsam, beschleunigt bis zur Mitte und verlangsamt sich dann wieder.

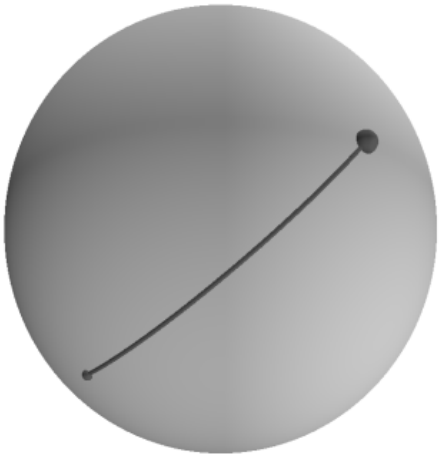

Abbildung 2.1: Kugel im 4D-Raum und interpolierte Werte [DKL98]

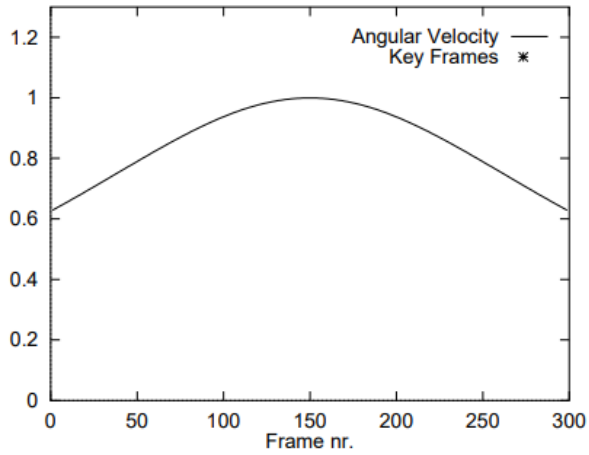

Abbildung 2.2: Winkelgeschwindigkeit der interpolierten Rotation [DKL98]

Abbildung 2.2 zeigt einen Plot der Winkelgeschwindigkeit einer über 300 Frames hinweg interpolierten Rotation. Die Menge der Einheitsquaternionen bildet eine Kugel im vierdimensionalen Raum. Die zwischen Q und P interpolierten Quaternionen bilden eine gerade Linie, welche durch das innere dieser Kugel verläuft. Abbildung 2.1 veranschaulicht diesen Zusammenhang. (vgl. [DKL98])

Um mit gleichbleibender Winkelgeschwindigkeit zu interpolieren, kann die in [Sho85] von Shoemake beschriebene Formel der *Sphärischen linearen Interpolation* (engl. spherical linear interpolation) verwendet werden. Für $Q, P, R \in \mathbb{H}$ und $\langle Q, P \rangle = cos(\theta)$ gilt:

$$quat\_slerp(Q, P, x) = \frac{sin((1-x) \cdot \theta)}{sin(\theta)} Q + \frac{sin(x \cdot \theta)}{sin(\theta)} P \tag{2.14}$$

Der Name weist darauf hin, dass die interpolierten Quaternionen eine Linie auf der Oberfläche der vierdimensionalen Kugel bilden. Es ist anzumerken, dass $quat\_slerp$ nicht immer auf dem kürzesten Weg interpoliert. Die zwei Ausrichtungen, welche von Q und P repräsentiert werden, umschließen meistens einen *stumpfen Winkel*[1] und einen *überstumpfen Winkel*[2], welche zusammen einen *vollen Winkel bilden*[3]. Ist $\langle Q, P \rangle < 0$, dann interpoliert $quat\_slerp$ entlang dem überstumpfen Winkel. Um immer auf dem kürzesten Weg zu interpolieren, kann bei $\langle Q, P \rangle < 0$ die Quaternion Q negiert und mit $|\langle Q, P \rangle| = cos(\theta)$ gerechnet werden. Es folgt der Pseudocode zu $quat\_slerp$.

```
fn quat_slerp(mut p: Quat, q: Quat, x: f32) {
    // Das Skalarprodukt von p und q berechnen
    let dotproduct = p.dot(q);

    // Ist das Skalarprodukt kleiner als 0, negiere p
    if dotproduct < 0 {
        p = -p;
    }

    // Theta berechnen
    let theta = dotproduct.abs().acos();
    if theta.sin()==0.0 {
        return a;
    }

    // Berechnung nach Formel 2.14
    let coefficient_a = ((1.0 - x) * theta).sin() / theta.sin();
    let coefficient_b = (x * theta).sin() / theta.sin();
    return a * coefficient_a + b * coefficient_b
}
```

Listing 2.3: Sphärische linearen Interpolation

[1] ein Winkel größer 0° und kleiner 180°
[2] ein Winkel größer 180° und kleiner 360°
[3] ein Winkel von 360°

Kapitel 3

Funktionsweise

In diesem Kapitel werden die Grundelemente der Skelettanimation erklärt. Es wird ein Skelett benötigt, welches die Grundstruktur einer Figur bestimmt, und eine Skin, welche deren Erscheinungsbild bestimmt. Für diese werden sogenannte Keyframes festgelegt, welche Schlüsselposen beinhalten, zwischen welchen interpoliert wird, um eine flüssige Animation zu erstellen.

3.1 Skelett

Das Skelett ist das Modell eines gegliederten Körpers. Es wird als Baum verstanden, dessen Knoten Knochen sind. Jeder Knochen hat einen oder keinen Elternteil und beliebig viele Kinder, welche ebenfalls Knochen sind. Diese Datenstruktur wird verwendet, um den Anschein zu erwecken, dass Kind-Knochen an deren Eltern-Knochen befestigt sind.

Jeder Knochen setzt sich aus lokalen Werten zu Rotation, Skalierung und Verschiebung zusammen. Die Verschiebung gibt die Position der Stelle an, wo ein Glied mit dem Elternteil verbunden ist, und wird als Vektor mit den drei Koordinaten x, y und z gespeichert. Auch die Skalierung wird als 3D-Vektor gespeichert, welcher die Streckung bzw. Stauchung in x-, y- und z-Richtung (relativ zur Orientierung des Knochens) enthält. Es werden mehrere Koordinaten gespeichert, damit auch eine uneinheitliche Skalierung möglich ist. Im 2D-Bereich ist die z-Koordinate redundant, sie wird jedoch zugunsten einer einfacheren Implementierung beibehalten. Weil es sich bei unserer Anwendung um einen 2D-Editor handelt, genügt es für die Rotation einen einzigen Winkel zu speichern. Aber da Bevy, so wie viele beliebte Game-Engines, Rotationen mithilfe von Quaternionen abhandelt, verwenden auch wir Quaternionen. Mit *Forward Kinematics* wird von den lokalen auf die globalen Transformationswerte eines Knochens geschlossen.

3.2 Forward Kinematics

Als Forward Kinematics bezeichnet man das Problem, bei bekannten Transformationen der kinematischen Kette, die Position des Endeffektors zu bestimmen. Beim FK Problem sind die Gelenkwinkel sowie die Längen der Glieder bekannt. (vgl. [ALCS18]: 36)

Es wird zwischen globalen und lokalen Transformationswerten unterschieden. *Global* bedeutet relativ zu einem absoluten Nullpunkt und einer Standardrotation und -skalierung. *Lokal* bedeutet relativ zu den Transformationswerten des Elternobjekts. Bei einem Knochen ohne Elternteil sind globale und lokale Position und Orientierung identisch.

Um von den lokalen Transformationswerten eines Knochens mit Elternteil auf die globalen Transformationswerte zu schließen, müssen diese mit den globalen Transformationswerten des Eltern-Knochens kombiniert werden. Bei der Skalierung und bei der Rotation ist der kombinierte Wert das Produkt aus der globalen Skalierung bzw. Rotation des Eltern-Knochens und der lokalen Skalierung bzw. Rotation des Kind-Knochens. Bei der Bestimmung der globalen Position muss der lokale Positionsvektor zunächst mit dem globalen Skalierungsvektor des Eltern-Knochens multipliziert und dann um dessen globale Rotation gedreht werden. Die Summe des so berechneten Vektors und der globalen Position des Eltern-Knochens ist die globale Position des Kind-Knochens. Rekursiv können so die globalen Transformationswerte aller Knochen bestimmt werden.

	Position	Rotation	Skalierung
A	(x: 0.2, y: 0.0)	45°	(x: 1.0, y: 1.0)
B	(x: 0.0, y: 1.0)	45°	(x: 0.5, y: 0.5)
C	(x: 0.0, y: 1.0)	-45°	(x: 2.0, y: 2.0)

Abbildung 3.1: Lokal

	Position	Rotation	Skalierung
A	(x: 0.2, y: 0.0)	45°	(x: 1.0, y: 1.0)
B	(x: 0.9, y: 0.7)	45°	(x: 0.5, y: 0.5)
C	(x: 1.4, y: 0.7)	-45°	(x: 2.0, y: 2.0)

Abbildung 3.2: Global

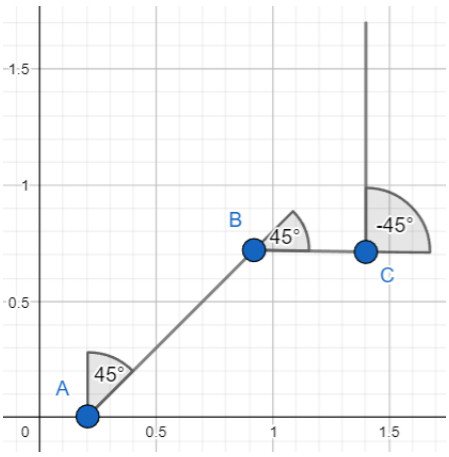

Abbildung 3.3: Forward Kinematics

Abbildung 3.3 zeigt ein Beispiel drei hierarchisch angeordneter Knochen. C hat den Eltern-Knochen B, B hat den Eltern-Knochen A und A hat keinen Eltern-Knochen. Abbildung 3.1 zeigt die lokalen Transformationswerte der Knochen. Abbildung 3.2 zeigt die globalen Transformationswerte. Alle Koordinaten sind auf die erste Nachkommastelle gerundet.

Listing 3.1 beinhaltet einen Pseudocode zur Lokalisierung des Endeffektors einer kinematischen Kette. Der darin verwendete Datentyp *Transform* beinhaltet die lokalen Transformationswerte von Skalierung, Rotation und Verschiebung eines Glieds, dessen Felder die Namen *scale*, *rotation* und *translation* haben. Es wird davon ausgegangen, dass jedes Glied die uniforme Länge 1 hat welche mit der y-Skalierung verrechnet wird.

```
// Lage des Kindes hinsichtlich des Koordinatensystems des Elternteils
fn get_combined_tf(parent: &Transform, child: &Transform) -> Transform {
    Transform {
        translation: parent.translation
            + parent.rotation.mul_vec3(child.translation * parent.scale),
        rotation: parent.rotation * child.rotation,
        scale: parent.scale * child.scale,
    }
}

// Finde die globalen Transformationswerte eines Glieds.
// [chain] ist die kinematische Kette, geordnet von Kindern zu Eltern.
// Das letzte Element von [chain] hat keinen Elternteil.
fn get_gl_transform(index: usize, chain: &Vec<Transform>) -> Transform {
    let mut gl_transform = chain[index];
    for i in (index + 1)..chain.len() {
        gl_transform = get_combined_tf(&chain[i], &gl_transform);
    }
    gl_transform
}


// Finde die Spitze eines Glieds anhand der globalen Transformationswerte
fn get_link_tip(transform: &Transform) -> Vec3 {
    transform.translation
    + Vec3::new(0.0, transform.scale.y, 0.0).rotate(transform.rotation)
}

// Finde die Position des Endeffektors
fn get_endeffector_pos() -> Vec3 {
    let gl_transform = get_gl_transform(0, chain);
    get_link_tip(&gl_transform)
}
```

Listing 3.1: Forward Kinematics

3.3 Skin

Die Skin (engl. für Haut) legt die Erscheinung der zu animierenden Figur fest. Die einfachste Art ein Skelett mit einer Skin auszustatten, ist jedem Knochen eine Bilddatei zuzuweisen. Diese Bilddatei wird entsprechend der globalen Transformationswerte des Knochens skaliert, gedreht und verschoben. Somit entsteht die Illusion, dass das Bild am Knochen befestigt ist.

Unser Animationseditor verwendet eine etwas komplexere Technik, welche mehr Möglichkeiten bietet. Es werden 2D-Meshes zu einem Set von Texturen generiert. Jeder Mesh-Vertex steht in Beziehung zu einer beliebigen Anzahl an Knochen. Entsprechend der Bewegung des Skeletts werden die Vertices verschoben, und somit das Mesh verzerrt. Mehr Details dazu folgen in Abschnitt 4.4.

3.4 Keyframes

Bei der Skelettanimation wird nicht jedes Einzelbild von Hand erstellt. Stattdessen werden Keyframes (engl.: Schlüsselbilder) erstellt, aufgrund welcher die Gesamtheit aller Einzelbilder programmatisch generiert wird. Für jeden Keyframe werden die entsprechenden Transformationswerte der Skelettknochen, eine Interpolationsfunktion und ein Zeitpunkt gespeichert.

3.5 Interpolation

Um aus Keyframes eine flüssige Animation zu machen, wird die Technik der Interpolation genutzt. Die Transformation der Knochen über den Zeitraum zwischen zwei Keyframes hinweg wird als Funktion verstanden. Im Folgenden wird dies am Beispiel der Interpolation der Position eines Knochens veranschaulicht. Folgende Angaben sind bekannt:

t_0	der Zeitpunkt des 1. Keyframes
t_1	der Zeitpunkt des 2. Keyframes
t	Zeitpunkt der Betrachtung, $t_0 \leq t \leq t_1$
a	Translation des Knochens zu t_0
b	Translation des Knochens zu t_1

Gesucht ist die Translation des Knochens zum Zeitpunkt t.
Die Interpolationsfunktion kann je nach gewünschtem Ergebnis definiert werden. Das simpelste Beispiel ist die lineare Interpolation (lerp):

$$x = \frac{t - t_0}{t_1 - t_0} \tag{3.1}$$

$$lerp(a, b, x) = a \cdot (1 - x) + b \cdot x \tag{3.2}$$

Zur Interpolation von Skalierung genügt es, diese an Stelle der Translation einzusetzen. Zur Interpolation der Rotation wird die sphärische lineare Interpolation angewandt (siehe Formel 2.14).

Kapitel 4

Weiterführende Techniken

Bisher wurden die grundlegenden Elemente, welche die Skelettanimation möglich machen erläutert. In diesem Kapitel werden Techniken untersucht, welche helfen Animationen höherer Qualität zu erstellen und den Animationsprozess zu vereinfachen und zu beschleunigen.

4.1 Variieren der Interpolationsfunktion

Die oben aufgezeigte Interpolationsfunktion ist in den meisten Anwendungsfällen nicht geeignet. Es wird eine Figur betrachtet, welche ihren Arm hebt. Zwei Keyframes werden definiert: die Figur mit gesenktem Arm und mit gehobenem Arm. Wird diese Animation in einer Schleife abgespielt, hebt und senkt die Figur ihren Arm wiederholt. Jedoch kommt es dabei zu abrupten Richtungswechseln, welche unnatürlich wirken. Das liegt daran, dass sich physikalische Objekte, wie das Animationsprinzip *Slow In & Slow Out* besagt, nach dem Gesetz der Trägheit bewegen. Um eine Veränderung der Bewegungsrichtung zu erreichen, muss das Objekt beschleunigt werden.

Um diese Eigenschaft physikalischer Objekte zu simulieren, bedarf es einer Interpolationsfunktion, dessen Ableitung keine Sprünge macht, weder zwischen zwei Keyframes noch über angrenzende Keyframes hinweg. Im Folgenden wird die Funktion $ease\_in\_out$ aus dem Artikel *Easing Functions for Animations* [Feb18] erklärt, welche diese Eigenschaften hat. Zunächst werden die Funktionen $ease\_in$ und $ease\_out$ definiert:

$$ease\_in(x) = x^2 \tag{4.1}$$

$$\begin{aligned} ease\_out(x) &= 1 - (1 - x)^2 \\ &= -x^2 + 2x \end{aligned} \tag{4.2}$$

Abbildungen 4.1 und 4.2 zeigen Plots der Funktionen. Wird $ease\_in$ als Interpolationsfunktion verwendet, beginnt die Animation langsam, wird dann schneller und endet abrupt. Wird $ease\_out$ verwendet, beginnt die Animation abrupt und verlangsamt sich dann, bis sie zum Stillstand kommt. Um sowohl eine Beschleunigung am Anfang der Animation als auch eine

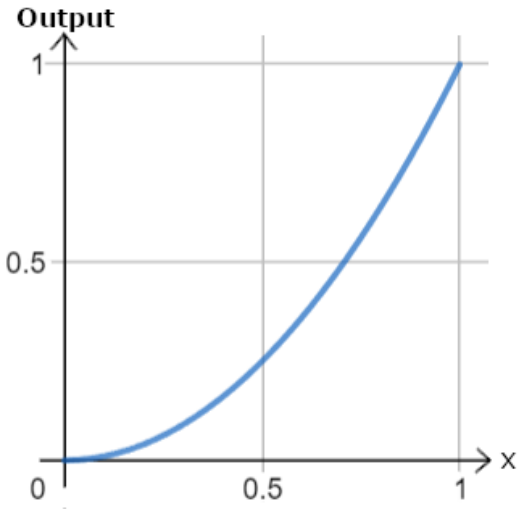

Abbildung 4.1: $ease\_in$

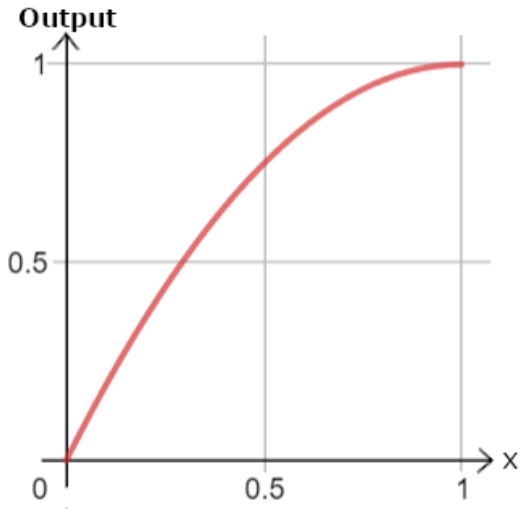

Abbildung 4.2: $ease\_out$

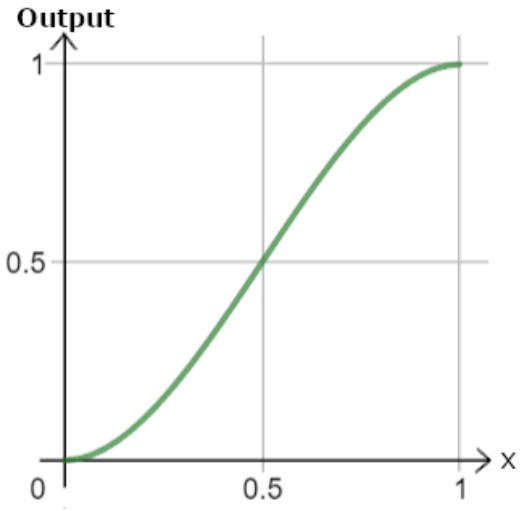

Abbildung 4.3: $ease\_in\_out$

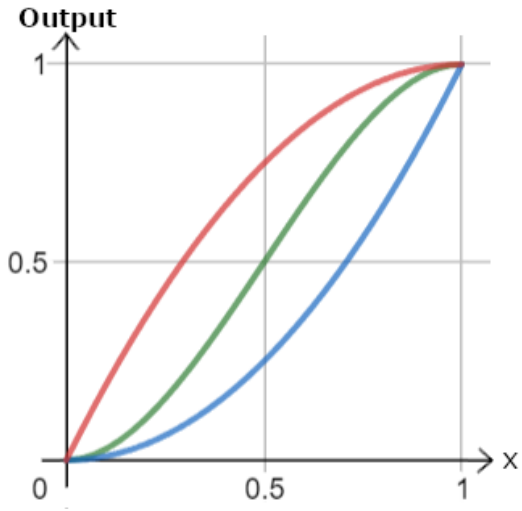

Abbildung 4.4: Kombinierte Ansicht

Verlangsamung am Ende der Animation zu erreichen, werden beide Funktionen mithilfe von $lerp$ kombiniert, um $ease\_in\_out$ zu formen (siehe Abb. 4.3):

$$\begin{aligned} ease\_in\_out(x) &= lerp(ease\_in(x), ease\_out(x), x) \\ &= ease\_in(x) \cdot (1 - x) + ease\_out(x) \cdot x \\ &= x^2 \cdot (1 - x) + (-x^2 + 2x) \cdot x \\ &= -2x^3 + 3x^2 \end{aligned} \tag{4.3}$$

In Abbildung 4.4 ist zu sehen, dass $ease\_in\_out$ links gegen $ease\_in$ und rechts gegen $ease\_out$ konvergiert. Schließlich wird $ease\_in\_out(x)$ an Stelle von x in $lerp$ eingesetzt,

um zwischen zwei Keyframes zu interpolieren:

$$lerp(a, b, ease\_in\_out(x)) \tag{4.4}$$

Der oben beschriebene Algorithmus $ease\_in\_out$ ist nur eine von vielen nützlichen Interpolationsfunktionen. Eine abgeänderte Version der Funktion $ease\_in\_out$ ist $ease\_in\_out\_back$, bei welcher die Bewegung erst in entgegengesetzter Richtung beginnt, dann die Zielpose überschreitet und dann zurückkehrt. Dieses Verhalten folgt den Animationsprinzipien *Anticipation* und *Follow Through*. Abbildung 4.5 zeigt den Plot zu dieser Funktion. Listing 4.1 zeigt den dazugehörigen Pseudocode. (vgl. [Sit] → easeInOutBack)

```
fn ease_in_out_back(x: f32) -> f32 {
    let c1 = 1.70158;
    let c2 = c1 * 1.525;

    if x < 0.5 {
        ((2.0 * x).powi(2) * ((c2 + 1.0) * 2.0 * x - c2)) / 2.0
    } else {
        ((2.0 * x - 2.0).powi(2)
        * ((c2 + 1.0) * (x * 2.0 - 2.0) + c2) + 2.0)
        / 2.0
    }
}
```

Listing 4.1: Ease In Out Back

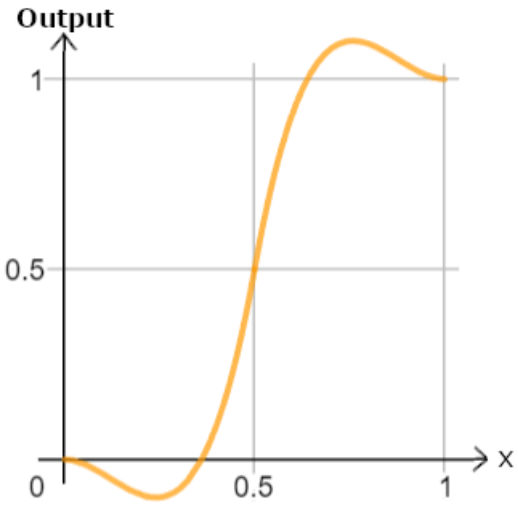

Abbildung 4.5: $ease\_in\_out\_back$

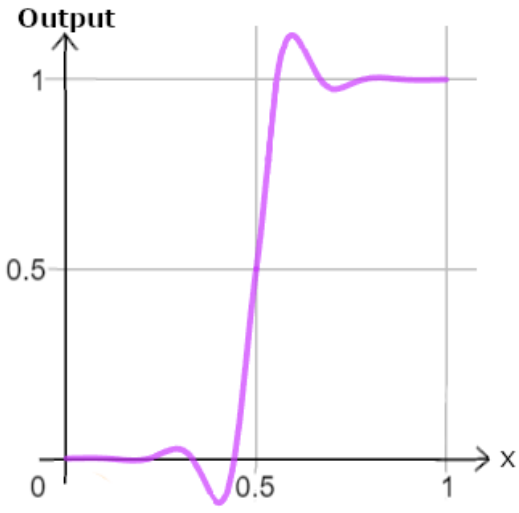

Abbildung 4.6: $ease\_in\_out\_elastic$

Eine weitere Variante zu $ease\_in\_out$ ist $ease\_in\_out\_elastic$ (siehe Listing 4.2). Sie ist der Funktion $ease\_in\_out\_back$ ähnlich, erweckt aber den Anschein, die Bewegung sei gefedert. Abbildung 4.6 zeigt einen Plot der Funktion. (vgl. [Sit] → easeInOutElastic)

```rust
fn ease_in_out_elastic(x: f32) -> f32 {
    let constant: f32 = (2. * std::f32::consts::PI) / 4.5;

    if x == 0. || x == 1. {
        x
    } else {
        if x < 0.5 {
            -(2f32.powf(20. * x - 10.)
            * ((20. * x - 11.125)
            * constant).sin())
            / 2.
        } else {
            (2f32.powf(-20. * x + 10.)
            * ((20. * x - 11.125) * constant).sin())
            / 2. + 1.
        }
    }
}
```

Listing 4.2: Ease In Out Elastic

Es gibt viele weitere Interpolationsfunktionen, welche gezielt eingesetzt werden können, um eine bestimmte Wirkung in der Animation zu erzielen. (siehe [Sit])

4.2 Animation Layering

Besonders in interaktiven Anwendungen kann es hilfreich sein, mehrere Animationen zu kombinieren. Erstellt man eine Animation für einen bestimmten Bereich, dann kann man diesen Bereich unabhängig vom Rest des Körpers animieren. Die Transformationswerte aller betroffenen Gelenke werden je Keyframe abgespeichert. Diese werden interpoliert, um für jeden Frame, wie weiter oben geschildert, mit *Forward Kinematics* die globalen Transformationswerte zu berechnen. Beim *Animation Layering* ersetzt eine Ebene für den betroffenen Bereich alle berechneten Werte darunterliegender Ebenen.

Betrachtet wird folgendes Beispiel: Die zu animierende Figur hat zwei Animationen, *Gehen* und *Winken*. Die Gehen-Animation betrifft alle Gelenke der Figur. Die Winken-Animation hingegen betrifft nur die Gelenke des rechten Arms. Um beide Animationen zu kombinieren, werden die Rotationen der Armgelenke der Gehen-Animation durch die Rotationen der Winken-Animationen ersetzt.

4.3 Additive Animation Blending

Neben dem *Animation Layering* gibt es auch die Technik des *Additive Animation Blending*. Zwei oder mehr Animationen werden zu einer neuen Animation zusammengefügt. Die neue Animation besteht zu vom Animator festgelegten Gewichtungen aus bereits vorhandenen Animationen. Es gibt verschiedene Anwendungsfälle für diese Technik, doch ein häufiger

Anwendungsfall ist der flüssige Übergang zwischen zwei Animationen. Betrachtet wird folgendes Beispiel: Zwei Animationen einer Person wurden erstellt. Die erste ist eine Gehen-Animation und die zweite ist eine Rennen-Animation. Beim Wechsel von der ersten zur zweiten Animation würde es dabei ohne *Additive Animation Blending* zur abrupten Änderung der Pose kommen. Mit *Additive Animation Blending* ist es möglich, diesen Übergang über einen bestimmten Zeitraum hinweg ablaufen zu lassen. (vgl. [FHKS12] und [Joh09]: 13)

Um zwei oder mehr Animationen zu mischen, wird mit der in Abschnitt 3.5 beschriebenen Technik zunächst die Frame-Pose für jede Animation berechnet. Um nun zwischen den Animationen zu interpolieren, kann die *lerp*-Funktion (Formel 3.2) angewandt werden. Es wird jeweils zwischen einer neutralen Pose, bei welcher alle Transformationswerte den Standardwerten entsprechen, und einer der soeben berechneten Frame-Posen interpoliert. Anstatt x über den Zeitpunkt zu ermitteln, wird für x die Gewichtung der jeweiligen Pose eingetragen. Es ist anzumerken, dass die Summe der Gewichtungen aller Teilanimationen immer 1 sein sollte. Als letzter Schritt werden die interpolierten Posen miteinander kombiniert. Für Position und Skalierung wird die Summe, für die Rotation das Produkt berechnet.

Abbildung 4.7:
100% traurig,
(Verändert von: iheartcraftythings.com)

Abbildung 4.8:
50% traurig, 50% glücklich,
(Verändert von: iheartcraftythings.com)

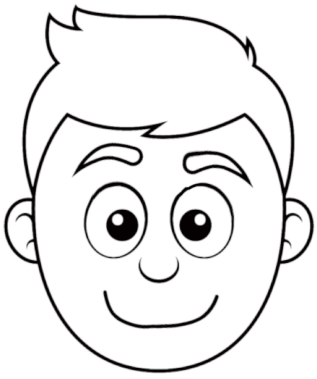

Abbildung 4.9:
100% glücklich,
(Verändert von: iheartcraftythings.com)

Die Abbildungen 4.7, 4.8 und 4.9 zeigen ein Beispiel von *Additive Animation Blending*. Für das gleiche Gesicht werden zwei Animationen erstellt, eine traurige und eine glückliche. Zwischen den beiden Animationen entsteht ein flüssiger Übergang. Für das traurige Gesicht können z.B. zitternde Lippen, für das glückliche Gesicht ein Auf und Ab der Brauen animiert werden. Die Gewichtung der Teilanimationen kann durch eine beliebige numerische Variable bestimmt werden. Das kann der Zeitpunkt sein, oder beispielsweise die Position oder Orientierung eines Objekts.

4.4 Meshes

Ein Mesh (engl. für Netz) ist eine Ansammlung von Punkten und Dreiecken, welche sich aus den Verbindungslinien jeweils drei dieser Punkte bilden. Ein solches Mesh kann als ein Array von Vertices (Punkte im Koordinatensystem) und ein Array von Indizes definiert werden. Drei dieser Indizes, welche auf je einen Punkt des ersten Arrays verweisen, bilden jeweils ein Dreieck.

Mithilfe eines solchen Meshs kann nun eine Textur aufgespannt werden. Dazu wird zusätzlich ein Array von UVs (Bildkoordinaten) definiert. Je Punkt wird ein UV festgelegt. Unabhängig von den Koordinaten eines Punkts kann ihm jetzt eine Position in der Textur-Bilddatei zugewiesen werden. Mit den gegebenen Informationen kann jede gängige Grafikbibliothek die Farbe der Pixel innerhalb eines Dreiecks genau bestimmen.

Meshes werden oft verwendet, um 3D-Objekte darzustellen. Da ein 2D-Animationseditor implementiert wird, werden nur 2D-Meshes verwendet. Diese befinden sich ausschließlich auf einer Ebene. Die Verwendung von 2D-Meshes bringt einige Vorteile mit sich, darunter:

a) Bilder können nicht nur in ihrer ursprünglichen Erscheinung, sondern auch in verzerrter Form angezeigt werden. Das ist eine große Hilfe bei der Umsetzung des Animationsprinzips *Squash and Stretch*.

b) Beim Rendern einer PNG-Datei wird für gewöhnlich jedes Pixel gerendert, auch wenn dieses transparent ist. Ein Mesh hingegen kann so definiert werden, dass es sich nur über sichtbare Bereiche hin erstreckt, was die Performanz erhöht.

c) Meshes finden in der Physiksimulation Anwendung.

4.4.1 Mesh-Generierung

Je nach Verwendungszweck unterscheidet sich die Natur des am besten geeigneten Meshs. Es gilt, die Größe der vom Mesh bedeckten Fläche zu minimieren, um die Performanz zu maximieren. Das bedeutet, das Mesh sollte alle sichtbaren Bereiche bedecken, damit diese vollständig angezeigt werden, aber vollständig transparente Bereiche sollten möglichst ausgelassen werden.

Die Dichte von Dreiecken in einem Bereich des Bildes und die Form und Lage der Dreiecke beeinflusst die Art und Weise, auf welche sich die angezeigte Grafik bei Verzerrung des Meshs verändert. Übermäßig große Dreiecke, besonders in Bereichen, welche stark verzerrt werden, führen meist zu einem unnatürlich erscheinendem Ergebnis.

Eine einfache Methode ein Mesh zu generieren ist, ein gleichmäßiges Raster über die gesamte Länge des Bildes aufzuspannen. Diese Methode funktioniert problemlos, allerdings führt sie dazu, dass auch transparente Bereiche gerendert werden.

Marching Squares Algorithmus

Um ein enganliegendes Mesh zu generieren, wird zuerst eine Kontur für die Bilddatei, ein Bild im PNG-Format, generiert. Im Folgenden wird erklärt, wie diese Kontur mit dem *Marching Squares Algorithmus* (engl. für wandernde Quadrate) generiert werden kann. Die Erklärung ist basiert auf [GN13]: 1.

Abbildung 4.10: Originalbild, (Quelle: i.etsystatic.com)

Abbildung 4.11: Distanzfeld

Abbildung 4.12: Schwellwert

Abbildung 4.13: Kontur

Um den *Marching Squares Algorithmus* anwenden zu können, wird ein Binärfeld[1] benötigt. Zunächst wird aus dem Alpha-Kanal des Bildes ein Distanzfeld generiert. Dessen Dimension muss für den vorliegenden Anwendungsfall gegebenenfalls etwas größer sein als die Dimension der Bilddatei. Das Distanzfeld ist eine Grauwertmatrix, welche für jedes Pixel im Originalbild einen Wert enthält. Dieser Wert kann entweder positiv oder negativ sein. Ist er positiv, gibt er an, wie weit dieser Pixel vom nächsten sichtbaren Pixel entfernt ist. Ist er negativ, dann gibt dessen Betrag an, wie weit er vom nächsten unsichtbaren Pixel entfernt ist. Ab welchem Alphawert ein Pixel als sichtbar gilt, wird im Programmcode definiert. Als Nächstes wird auf dieses Distanzfeld ein Schwellwert angewandt, um das benötigte Binärfeld zu generieren. Abbildungen 4.10, 4.11, 4.12 und 4.13 visualisieren die soeben erklärten Schritte.

Auf das Binärfeld wird nun ein Konturraster gelegt, welches in x- und y-Richtung jeweils eine Zelle kleiner ist. Die Eckpunkte des Konturrasters liegen jeweils im Zentrum einer der Binärzellen. Jeder Zelle des Konturrasters wird ein Wert zugewiesen. Dieser ergibt sich daraus, ob die Werte der vier Ecken dieser Zelle $true$ oder $false$ sind. Den Ecken werden die Werte 2^0, 2^1, 2^3 und 2^4 zugewiesen (siehe Abb. 4.14).

Für jede der 16 möglichen Kombinationen wird jeweils ein entsprechendes Konturstück festgelegt (siehe Abb. 4.15). Diese 16 Fälle können in einer Lookup-Tabelle gespeichert werden. Fügt man all diese Stücke zusammen, ergibt sich die gesuchte Kontur. Abbildung 4.16 zeigt

[1] eine Matrix mit Elementen $true$ oder $false$

eine Beispiel-Rastergrafik. Abbildung 4.17 zeigt die Konturzellen (gelb) und den jeweils zutreffenden Fall aus der Lookup-Tabelle. Die generierte Kontur ist in Grün eingezeichnet.

Aus den Zellen wird nun ein ungerichteter Graph generiert. Ein Graph setzt sich aus Knoten und Kanten zusammen. Die Knoten sind hier die Schnittpunkte von Kanten mit dem Konturraster. Sie werden als 2D-Ortsvektoren in einem Array gespeichert. Die Kanten werden als Indexpaare in einem weiteren Array gespeichert.

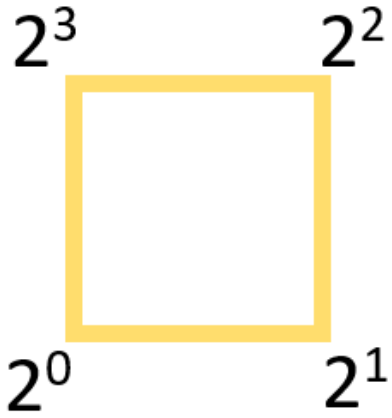

Abbildung 4.14: Konturzelle

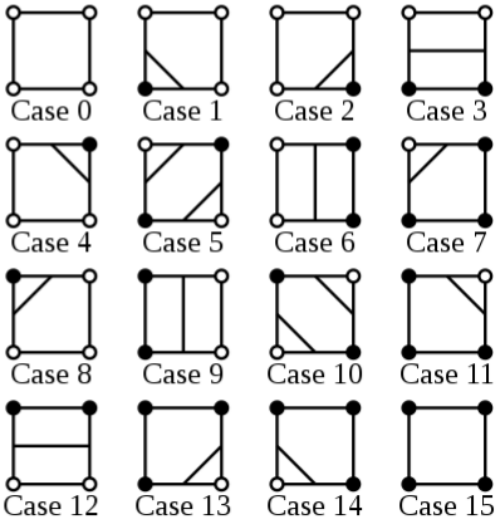

Abbildung 4.15: Mögliche Fälle [Map03]

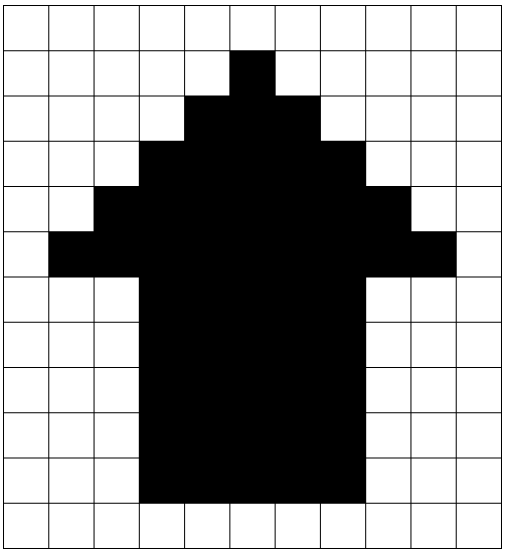

Abbildung 4.16: Binärfeld

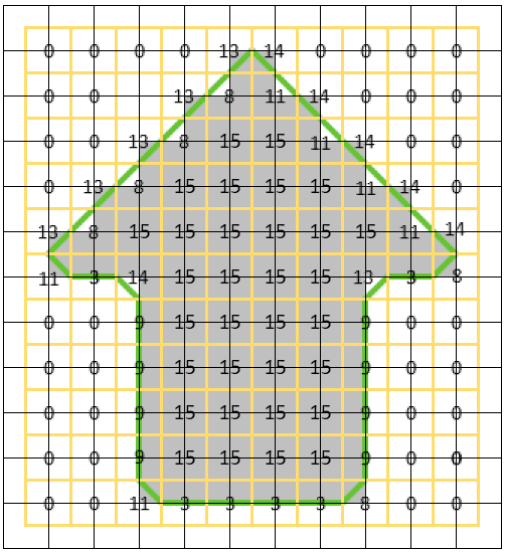

Abbildung 4.17: Konturraster

Umwandlung in ein Multi-Polygon

Der generierte Graph wird nun in ein Multi-Polygon[2] umgewandelt. Die Bibliothek *Geo* wird für Polygone verwendet. Geo bietet Funktionen, um Polygone und auch Multi-Polygone zu definieren und deren Lage relativ zu anderen Formen zu ermitteln. Nachher wird die Möglichkeit, zu ermitteln, ob sich ein Polygon innerhalb eines anderen befindet, von Nutzen sein. In Geo setzt sich ein Polygon aus einer *äußeren* und beliebig vielen *inneren Line-Strings* zusammen. Ein *Line-String* ist ein geordnetes Array von 2D-Ortsvektoren. Der *äußere Line-String* bildet die *konkave Hülle*[3], die *inneren Line-Strings* bilden Auslassungen im Polygon.

Es wird damit begonnen, alle fortlaufenden Linien aus dem Graphen in *Line-Strings* umzuwandeln. Ein Array von *Line-Strings*, genannt *line_strings*, wird erstellt. Vorerst enthält *line_strings* nur einen einzigen leeren *Line-String*. Dann wird der Graph, angefangen an beliebiger Stelle, traversiert. Jeder besuchte Knoten wird dem letzten Element von *line_strings* hinzugefügt. Sobald ein bereits besuchter Knoten erreicht wird, wird mit beliebigem noch nicht besuchtem Knoten fortgefahren. Dessen Position wird als erster Ortsvektor einem neuen leeren *Line-String* angehängt. Der neue *Line-String* wird *line_strings* hinzugefügt. Auf diese Weise wird der gesamte Graph traversiert.

Die Vektoren der *Line-Strings* liegen sehr eng beieinander. Um Speicher- und Rechenaufwand zu reduzieren, werden die *Line-Strings* vereinfacht, indem nur ein Anteil der Vektoren beibehalten wird. Um einen *Line-String* zu vereinfachen, wird ein Array namens *keep* angelegt und ihm wird der erste Vektor des *Line-Strings* hinzugefügt. Eine Maximalanzahl *max* von Kanten, welche zu einer Kante zusammengefasst werden sollen, wird festgelegt. Folgender Pseudocode zeigt, wie der *Line-String* vereinfacht wird:

```
let line_string sei der unvereinfachte LineString

// Maximale Anzahl an zu vereinenden Kanten
let max = 30;

// Erstelle Array mit Vektoren, die beibehalten werden sollen
// Und fuege keep den ersten Vektor des original Line-Strings hinzu
let mut keep = vec![];
keep.push(line_string[0]);

let mut i = 0;
let mut j = 1;
while j < line_string.len() - 1 {
    let position_1 = line_string[i];
    let position_2 = line_string[j+1];

    // Pruefe, ob in der Bilddatei auf der Linie zwischen
    // den 2 Positionen sichtbare Pixel liegen
    let is_collision = cuts_visible_pixels(position_1, position_2);
    if  is_collision || j - i == 30 {
```

[2] eine Form, nicht zwingend zusammenhängend, welche sich aus mehreren Polygonen zusammensetzt
[3] das Polygon, aber ohne dessen Löcher

```
21          // Behalte Vektor mit Index j
22          keep.push(line_string[j]);
23          i = j;
24      }
25
26      j += 1;
27  }
```

Listing 4.3: Multi-Polygon Pseudocode

Obwohl es sich sowohl um äußere als auch um innere *Line-Strings* handeln kann, wird aus jedem dieser *Line-Strings* zunächst ein Polygon ohne Löcher erstellt und all diese Polygone werden einem Array namens *all_polygons* hinzugefügt. Die Indizes dieses Arrays werden als Nummerierung der Polygone verwendet. Nun wird ein Array von Nummern namens *todo* erstellt, welches jeden der Indizes von *all_polygons* enthält. Für jedes Polygon wrid geprüft, von welchen anderen Polygonen es vollständig umschlossen wird und die Ergebnisse werden in dem Array *contained_by* gespeichert. Jedes Element dieses Arrays ist ein Set von Nummern und die Indizes von *contained_by* entsprechen denen von *all_polygons*. Es wird ein weiteres leeres Array von Polygonen erstellt, welches *result* genannt wird.

Folgender Prozess wird so lange wiederholt, bis *todo* leer ist: *todo* wird nach einer Nummer durchsucht, für welche in *contained_by* ein leeres Set gespeichert ist. Der gefundene Eintrag ist der Index eines Polygons, welches geklont und *result* hinzugefügt wird. Für alle Einträge in *todo* wird jeweils geprüft, ob der entsprechende Eintrag in *contained_by* ein Set mit einschließlich der Nummer des geklonten Polygons ist. Ist dies der Fall, wird das entsprechende Polygon als *innerer Line-String* dem geklonten Polygon hinzugefügt. Sobald alle Einträge in todo geprüft wurden, werden die Nummer des geklonten Polygons, sowie die Nummern aller Polygone, die als *innere Line-Strings* verwendet wurden, aus *todo* und aus allen Sets in *contained_by* entfernt. Nun wird der Prozess wiederholt. Sobald *todo* leer ist, wird ein Multi-Polygon aus den Polygonen in *result* erstellt.

Es folgt ein Pseudo-Code zur Erstellung des Multi-Polygons:

```
1   let all_polygons sei ein Array von Polygonen;
2
3   // Fuege [todo] alle Indizes aus all_polygons hinzu
4   let todo = vec![];
5   for i in 0..all_polygons.len() {
6     todo.push(i);
7   }
8
9   // Erstelle ein Array mit leeren Sets
10  let contained_by = vec![HashSet::new();all_polygons.len()];
11  for i in 0..all_polygons.len() {
12    // Ermittle Polygone, welche dieses umschliessen...
13    for j in 0..all_polygons.len() {
14      if i==j continue;
15      let self = &all_polygons[i];
16      let other = &all_polygons[j];
```

```
17      if other.contains(self) {
18        // ...und fuege sie dem Set hinzu.
19        contained_by.push(j);
20      }
21    }
22  }
23
24  //Ergebnis-Array
25  let result = vec![];
26
27  // Schleife mit Hauptlogik des Algorithmus
28  while !todo.is_empty() {
29
30    // Array mit Indizes aller abgehandelten Polygone
31    let done = vec![];
32
33    // Index des aeusseren Polygons
34    let index;
35
36    // Finde aeusseren Line-String
37    for i in 0..todo.len() {
38      index = todo[i];
39      if contained_by[index].is_empty() {
40        result.push(all_polygons[index]);
41        done.push(index);
42        break;
43      }
44    }
45
46    // Fuege innere Line-Strings hinzu
47    for i in 0..todo.len() {
48      // Ueberspringe Selbst-Check
49      if i == index continue;
50      // Set mit Nummern der Polyogene, welche das betrachtete enthalten
51      let set = contained_by[todo[i]];
52      if set.len() == 1 && set.contains(index) {
53        // Fuege dem Polygon ein Loch hinzu
54        result.last().interiors_push(all_polygons[i]);
55        done.push(i);
56      }
57    }
58
59    // Entferne soeben abgehandelte Polygone aus [todo]
60    for i in (0..todo.len()).rev() {
61      if done.contains(todo[i]) {
62        todo.swap_remove(i);
63      }
64    }
65
66    // Entferne soeben abgehandelte Polygone aus allen Sets in [
    contained_by]
67    for index in done {
68      for set in contained_by {
69        set.remove(index);
```

```
70          }
71      }
72
73  }
```

Listing 4.4: Multi-Polygon Pseudocode

Delaunay Triangulation

Um aus dem Multi-Polygon ein Mesh zu machen, wird eine *eingeschränkte Delaunay-Triangulation*[4] durchgeführt. Die *Delaunay-Triangulation* ist das Bilden eines Dreiecksnetzes aus einer Ansammlung von Punkten. Die Punkte werden verbunden, sodass der Umkreis jedes entstehenden Dreiecks keine weiteren Punkte enthält. Das gewährleistet, dass der kleinste Winkel jedes Dreiecks maximiert wird. Diese Eigenschaft ist von Bedeutung, da sie dazu beiträgt, dass Dreiecke nicht zu lang werden. Solche führen in der resultierenden Animation oft zu übermäßig starken Verzerrungen. Die *eingeschränkte Delaunay-Triangulation* hat als Zusatzbedingung, dass eine Menge vordefinierte Kanten im resultierenden Dreiecksnetz enthalten sind. Diese Zusatzbedingung führt dazu, dass die erste Bedingung teilweise nicht erfüllt werden kann. (vgl. [PC89]: 1-2)

Es wird das zuvor erstellte Multi-Polygon für den Algorithmus genutzt. Alle Kanten, welche im Multi-Polygon enthalten sind, sollen auch im resultierenden Dreiecksnetz enthalten sein. Als Punktemenge wird vorerst die Menge aller Eckpunkte des Multi-Polygons festgelegt. Zusätzlich werden dieser Menge Punkte im inneren des Multi-Polygons hinzugefügt, welche nachher eine feinere Deformierung ermöglichen. Die zusätzlichen Punkte werden in gleichmäßigen Abständen hinzugefügt. Wie eng die Punkte aneinanderliegen, hängt von der gewünschten Verformbarkeit des Meshs ab und wird vom Animator festgelegt.

Die nun vorhandenen Daten werden an eine Bibliotheksfunktion weitergegeben, welche die *eingeschränkte Delaunay-Triangulation* durchführt. Dem Ergebnis können die benötigten Indizes entnommen werden, von welchen jeweils drei ein Dreieck bilden. Einige dieser Dreiecke befinden sich allerdings außerhalb des Multi-Polygons. Diese Dreiecke werden aus dem Endergebnis entfernt. Die UVs werden bestimmt, indem die Vertex-Koordinaten auf den Wertebereich 0 bis 1 normalisiert werden. Alle benötigten Daten, um das Mesh zu erstellen, sind nun bekannt. Die *Delaunay-Triangulation* bietet den Vorteil, dass Vertices nun dynamisch ergänzt oder entfernt werden können. Somit ist es z.B. möglich, zusätzliche Vertices in Bereichen mit hohem Detailgrad zu erstellen.

4.4.2 Mesh-Deformierung

Die Deformierung des Meshs erfolgt durch die Verschiebung der dazugehörigen Punkte. Um eine solche Deformierung durch das Transformieren der Skelett-Knochen zu erreichen, wird der Einflussgrad jedes Knochens auf jeden Mesh-Vertex festgelegt. Dieser Einflussgrad wird,

[4] Constrained Delaunay Triangulation

abhängig von der Lage der Punkte zu den Knochen, automatisch generiert.

Es wird eine Maximaldistanz $max\_distance$ festgelegt. Für einen Vertex wird die Distanz zu allen Knochen berechnet. Ist diese Distanz kleiner als $max\_distance$, wird das Komplement der Distanz zu $max\_distance$ berechnet. Dieser Wert wird dann durch $max\_distance$ geteilt und in die zehnte Potenz erhoben, was den Einflussgrad $weight$ des Knochens auf den Vertex bestimmt. Die Einflussgrade aller Knochen auf diesen Vertex werden summiert. Nun werden alle Einflussgrade $weight$ durch diese Summe geteilt. Das sorgt dafür, dass die Summe aller Einflussgrade $weight$ auf einen Vertex 1 ist. Für alle Knochen, welche außerhalb des Radius liegen, wird der Einflussgrad auf 0 gesetzt. Es folgt ein Pseudocode zu dieser Rechnung:

```
let max_distance sei die Maximaldistanz;

let mut vertex_mappings: Vec<Vec<f32> = vec![];

for vertex in mesh.vertices.iter() {
    // Anzahl an Knochen im Skelett
    let bones_count = skeleton.bones.len();

    // Einflussgrad eines Knochens auf diesen Vertex
    let mut vertex_bone_weights: Vec<f32> = vec![0.0; bones_count];

    // Dintanzen aller Knochen zu diesem Vertex
    let mut bone_distances = vec![];

    for i in 0..bones_count {
        // Die Distanz  Punktes zu einem Knochen wird als
        // die Distanz eines Punktes zu einer Linie berechnet
        let distance = bone[i].distance(vertex);
        bone_distances.push(distance);
        if distance < max_distance {
            // Berechnung des Einflussgrads
            let dist_complement = max_distance - distance;
            let complement_normalized = dist_complement / max_distance;
            let weight = powi(complement_normalized, 10);
            vertex_bone_weights.push(weight);
        }
    }

    // Index des naechsten Knochens
    let index_of_closest = bone_distances.index_of_min();

    if bone_distances[index_of_closest] < max_distance {
        // Normalisiere Einflussgrade, sodass Summe = 1.0
        let weights_sum = vertex_bone_weights.sum();
        for weight in vertex_bone_weights.iter_mut() {
            *weight /= weights_sum;
        }
    } else {
        // Wenn keine Knochen innerhalb des Radius max_distance sind,
        // wird der Vertex allein vom naechsten Knochen beeinflusst
```

```
41            vertex_bone_weights[index_of_closest] = 1.0;
42        }
43
44        // Ergebnis speichern
45        vertex_mappings.push(vertex_bone_weights);
46    }
```

Listing 4.5: Einflussgrad von Knochen auf Vertices bestimmen

Es ist auch möglich, die generierten Werte, sollten diese nicht zufriedenstellend sein, manuell anzupassen. Zusätzlich zum Einflussgrad wird die Lage jedes Punktes relativ zur Lage, Rotation und Skalierung jedes auf ihn Einfluss nehmenden Knochens festgelegt. Für jeden Mesh-Vertex ergeben sich nun bei beliebiger Skelettpose eine Anzahl an globalen Koordinaten. Die endgültige Vertexkoordinaten ergeben sich zu festgelegten Anteilen aus diesen globalen Koordinaten.

Das angepasste Mesh wird an eine Grafikbibliothek weitergegeben, welche sich um das Rendern kümmert. Abbildung 4.18 zeigt eine Tulpe vor und nach Verformung. Abbildung 4.19 zeigt zusätzlich das dazugehörige Mesh in Orange. Die entsprechenden Dreiecke bilden vor und nach der Verformung die gleichen Bereiche der Pixelgrafik ab, auch wenn die Form und Lage des Dreiecks sich verändert hat.

Abbildung 4.18: Verformung einer Tulpe, (Verändert von: i.etsystatic.com)

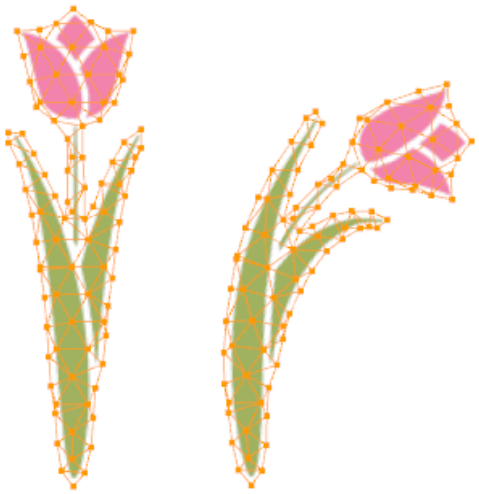

Abbildung 4.19: Verformung mit Mesh, (Verändert von: i.etsystatic.com)

Kapitel 5

Prozedurale Animation

Oft ist es von Vorteil, wenn Teile einer Animation nicht direkt vom Animator gestaltet werden müssen, sondern prozedural erstellt werden können. Soll z.B. eine humanoide Figur mit der Hand nach einem Objekt greifen, ist es mühselig und zeitaufwendig jeden Abschnitt des Arms einzeln zu animieren. *Inverse Kinematics* macht es möglich, mittels der Position und Rotation der Hand auf die Position und Rotation aller Armabschnitte zu schließen. Dies bedarf keiner weiteren Eingabe des Animators. Ein weiterer Anwendungsfall ist die genau Bestimmung der Trittposition einer Figur auf unebenem Terrain. Hier muss die Beinstellung so angepasst werden, dass die Fußsohle auf gleicher Höhe mit der Bodenoberfläche ist. (vgl. [Joh09]: 2)

Ein weiteres Beispiel für prozedurale Animation ist Physik-Simulation. Die automatische Animierung von Objekten, welche sich nach grundlegenden physikalischen Gesetzen verhalten, spart Zeit. Darüber hinaus ist sie meist genauer, als die Animierung von Hand. In interaktiven Anwendungen genügt oft ein annäherungsweise realistisches physikalisches Verhalten. Ein Anwendungsfall für Physik-Simulation ist ein herabhängendes Stück Stoff, das von einer Person getragen wird. (vgl. [MSJT08]: 5-6)

Ein weiterer Vorteil von prozeduraler Animation ist, dass nicht alle Details zur Erstellung der Animation bekannt sein müssen. So ist es z.B. möglich, dass eine animierte Figur mit der Umgebung interagiert, obwohl diese erst zur Laufzeit des Programms bekannt ist. Auch Nutzereingaben können in die Animation mit hineinspielen.

5.1 Inverse Kinematics

Der Begriff *Inverse Kinematics* (engl. für inverse Kinematik) kommt aus der Robotik. Er beschreibt das Problem, den Endeffektor eines Roboterarms mit n Gelenken und n Verbindungsstücken bekannter Länge an eine Zielposition zu bewegen. Die Zielposition liegt dabei in kartesischen Koordinaten vor, während die Lage des Roboterarms als ein Vektor von Rotationen vorliegt. Gesucht werden die passenden Einstellungen jedes beteiligten Ge-

lenks, sodass der Endeffektor die Zielposition erreicht. Beim Lösen von *Inverse Kinematics* wird zwischen direkten und iterativen Verfahren unterschieden. (vgl. [ALCS18]: 35-38)

Eine Kette von Gelenken und Knochen in der Skelettanimation hat die gleiche Struktur wie ein Roboterarm mit Gelenken und Verbindungsstücken. Die gleichen Techniken, welche zur Lösung von *Inverse Kinematics* in der Robotik eingesetzt werden, sind daher auch in der Skelettanimation anwendbar.

5.1.1 Direkte Verfahren

Die direkten Verfahren sind meist performant und liefern eine exakte Lösung. Für gewöhnlich haben sie keine Singularitätsprobleme, und sie erzeugen eine globale Lösung, unabhängig von der momentanen Lage der kinematischen Kette. Aufgrund dieser Eigenschaften werden sie oft für *Motion Planning* in der Robotik eingesetzt. (vgl. [ALCS18]: 38)

Kette mit zwei Gelenken

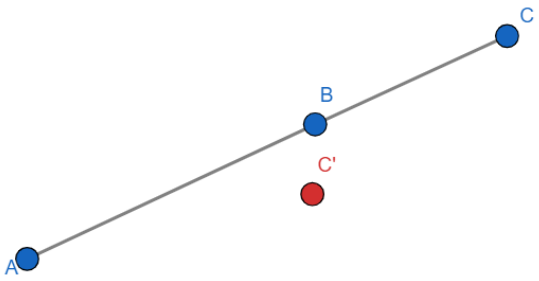

Abbildung 5.1: Ausgangsposition

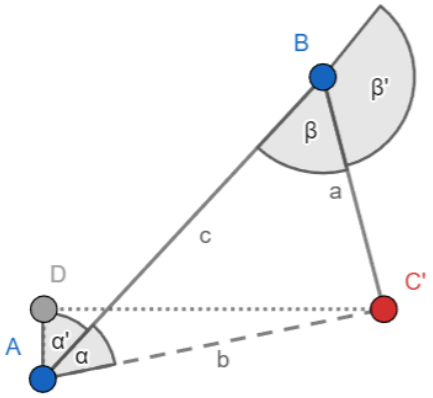

Abbildung 5.2: Zielposition

Es wird ein Roboterarm betrachtet. Der Arm hat zwei Gelenke, A und B. A ist fixiert. Am Ende des Arms ist der Endeffektor C befestigt (siehe Abb. 5.1). Nun soll C zur Zielposition C' bewegt werden. Dieses Problem kann mit dem Gesetz des Kosinus gelöst werden. a, b und c bilden zusammen ein Dreieck (siehe 5.2). Mit dem Gesetz des Kosinus (Gleichung 5.1) werden die Winkel α (Gleichung 5.2) und β ermittelt. (Gleichung 5.3)

$$c^2 = a^2 + b^2 - 2ab \cdot cos(\gamma)$$
$$\gamma = cos^{-1}\left(\frac{a^2 + b^2 - c^2}{2ab}\right) \tag{5.1}$$

$$\alpha = cos^{-1}\left(\frac{b^2 + c^2 - a^2}{2bc}\right) \tag{5.2}$$

$$\beta = cos^{-1}\left(\frac{a^2 + c^2 - b^2}{2ab}\right) \tag{5.3}$$

Die Winkel, welche gesucht werden, sind β' und α'. Diese sind die benötigten Winkeleinstellungen für die jeweiligen Gelenke[1]. β' und β ergeben den Gesamtwinkel π. Um β' zu bestimmen, wird β von π abgezogen. (Gleichung 5.4)

$$\beta' = \pi - \beta \tag{5.4}$$

Um den Gesamtwinkel $\alpha' + \alpha$ zu bestimmen, wird das Hilfsdreieck ADC konstruiert. (siehe Abb. 5.2) Auf dieses wird die trigonometrische Funktion Tangens angewandt. (Gleichungen 5.5 und 5.6) Nun wird nach α' aufgelöst. (Gleichung 5.7)

$$tan(\theta) = \frac{Gegenkathete}{Ankathete} \tag{5.5}$$

$$tan(\alpha + \alpha') = \frac{\vec{AC}_x}{\vec{AC}_y} \tag{5.6}$$

$$\alpha' = tan^{-1}\left(\frac{\vec{AC}_x}{\vec{AC}_y}\right) - \alpha \tag{5.7}$$

Bei zwei Gelenken gibt es meistens zwei mögliche Lösungen, welche sich nur darin unterscheiden, in welche Richtung sich der Arm beugt. Es kann aber auch vorkommen, dass es entweder keine Lösung gibt, wenn das Ziel unerreichbar ist, oder dass es genau eine Lösung gibt, wenn der Abstand zum Ziel genau der Länge des Arms entspricht.

5.1.2 Iterative Verfahren

Direkte IK-Verfahren sind nur schwer skalierbar. In der Computeranimation ist es allerdings nicht ungewöhnlich, dass eine Figur viele Freiheitsgrade hat. Deshalb werden in diesem Bereich oft iterative Verfahren angewandt, welche mit solchen Modellen umgehen können. Es existieren zahlreiche iterative Verfahren, mit verschiedenen Vor- und Nachteile. Abhängig vom Anwendungsfall sollte ein passendes Verfahren gewählt werden. Dabei sind wichtige Kriterien die Performanz, die Möglichkeit zur Definition von Constraints[2] und die Stabilität des Verfahrens. Als Stabilität wird die Eigenschaft verstanden, dass eine geringe Änderung der Zielposition auch nur zu einer geringen Änderung der Gelenkwinkel führt. (vgl. [ALCS18]: 38-39)

Es kann von Vorteil sein, vor dem Lösen von *Inverse Kinematics* festzustellen, ob eine Zielposition überhaupt erreichbar ist. Es ist offensichtlich, dass ein Punkt unerreichbar ist, wenn dessen Entfernung zum Befestigungspunkt der kinematischen Kette größer ist, als die summierte Länge der Kettenglieder. Bei unterschiedlich langen Gliedern ist es allerdings auch möglich, dass ein Bereich nahe dem Befestigungspunkt unerreichbar ist. Abbildung 5.3 veranschaulicht diesen Zusammenhang. Feststellung der Unerreichbarkeit eines Punktes im Voraus ermöglicht eine gezielte Handhabung dieser Fälle. (vgl. [ALCS18]: 37)

[1] Senkrecht nach oben wird als Nullwinkel definiert. Von dort aus weitet sich der Winkel im Uhrzeigersinn.
[2] engl. für Einschränkung

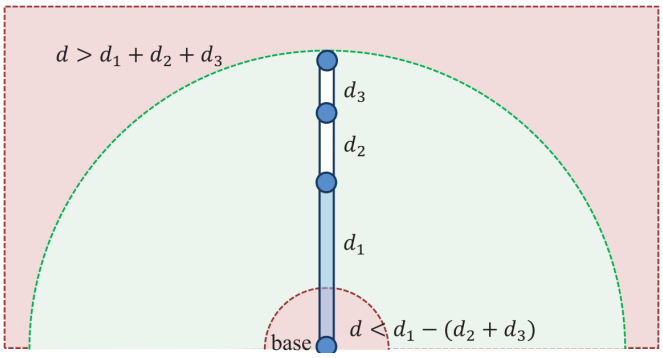

Abbildung 5.3: Reichweite einer Kette [ALCS18]

Bei Ketten mit mehr als zwei Gliedern, ist der Normalfall, dass es unendlich viele Lösungen gibt. Daher werden Methoden benötigt, um die am besten passende Lösung zu bestimmen. Im Bereich der Animation wird meist nach einer natürlich wirkenden Pose gesucht. Um zu verhindern, dass unnatürliche Posen generiert werden, können *Constraints* implementiert werden. Es wird zwischen *lokalen* und *globalen Constraints* unterschieden. *Lokale Constraints* beschränken die Beweglichkeit eines Gelenks. Ein Kniegelenk z.B. sollte sich niemals über 180° hinaus öffnen. *Globale Constraints* hingegen beschränken die Beweglichkeit der kinematischen Kette unter Berücksichtigung aller Glieder. Beispiele für globale Constraints sind die Umgehung eines Hindernisses oder die Berücksichtigung des Massenmittelpunkts zur Haltung des Gleichgewichts. Wir werden ausschließlich die Implementierung *lokaler Constraints* untersuchen, da diese für unsere Zwecke ausreichend sind. Auch die Stabilität spielt eine große Rolle bei der Wahl einer Lösung, da sie eine flüssige Animation garantiert, auch wenn jeder einzelne Frame mit *Inverse Kinematics* berechnet wird.

Cyclic Coordinate Descent

Cyclic Coordinate Descent (CCD) ist eine der bekanntesten iterativen Methoden zur Lösung von *Inverse Kinematics*. Alle Gelenke, welche Teil der betroffenen kinematischen Kette sind, werden nacheinander rotiert, um den Abstand zur Zielposition zu minimieren. Dabei werden alle Gelenke von der niedritsten bis zur höchsten Ebene in der Hierarchie abgearbeitet. Dieser Vorgang wird wiederholt, bis der Endeffektor annäherungsweise die gewünschte Position erreicht hat.

Abbildung 5.4 zeigt einen Roboterarm, welcher aus drei Knochen und drei Gelenken besteht. A ist fixiert, D ist der Endeffektor und soll zu D_{Ziel} bewegt werden. Es wird mit dem Knochen auf der untersten Ebene in der Hierarchie begonnen. Dieser wird um den Punkt C rotiert, sodass D auf $\vec{CD}_{Ziel}$ liegt, also auf der Linie, welche durch das betroffene Gelenk und den Zielpunkt geht (siehe Abb. 5.5). Eine Ebene weiter oben wird um den Punkt B rotiert (siehe Abb. 5.6), danach um den Punkt A (siehe Abb. 5.7). Werden diese Schritte wiederholt, nähert sich D mit jeder Iteration D_{Ziel} an. Der Algorithmus endet, sobald der Abstand von D zu D_{Ziel} einen vorher festgelegten Minimalabstand unterschreitet. Je nach

Aufstellung kann es sein, dass viele Iterationen benötigt werden, bis der Algorithmus auf diese Weise endet. Jedoch sind die Berechnungen und Transformationen nicht rechenintensiv, wodurch sich CCD für interaktive Echtzeitanwendungen eignet. (vgl. [FÛ3]: 207-208)

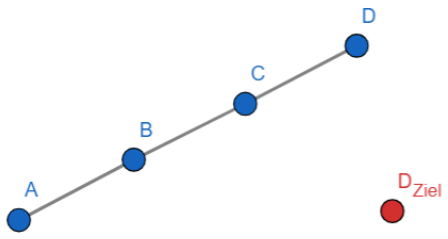

Abbildung 5.4: Ausgangsposition

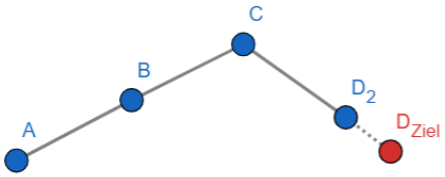

Abbildung 5.5: 1. Schritt

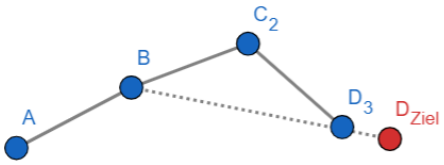

Abbildung 5.6: 2. Schritt

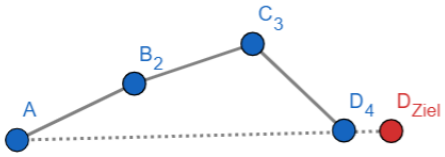

Abbildung 5.7: Zielposition

CCD ist einfach zu implementieren. Das Verfahren kann sehr einfach um die Einhaltung *lokaler Constraints* erweitert werden. Dazu wird die Rotationsfreiheit jedes Gelenks auf einen bestimmten Winkelbereich eingeschränkt. In jedem Schritt jeder Iteration wird das Gelenk, nach Anpassung des Winkels, auf die nächstliegende gültige Winkeleinstellung gesetzt. (vgl. [ALCS18]: 42)

Nachteile von CCD sind, dass sich vorwiegend die Gelenke bewegen, welche sich am Ende der kinematischen Kette, also nahe des Endeffektors, befinden. Das ist nicht immer erwünscht. Es ist schwer, *globale Constraints* zu definieren und je nach Gelenkeinstellungen und Zielposition kann es vorkommen, dass CCD nur sehr langsam konvergiert, besonders dann, wenn exakte Ergebnisse gebraucht werden. (vgl. [ALCS18]: 42)

Jacobi-Verfahren

Ein weiteres beliebtes Verfahren basiert auf der Berechnung der Jacobi-Matrix. Die Erklärung dieses Verfahrens basiert auf der Arbeit *Inverse Kinematics Techniques in Computer Graphics: A Survey* [ALCS18]. Die Jacobi-Matrix bildet die Veränderung der Winkeleinstellungen

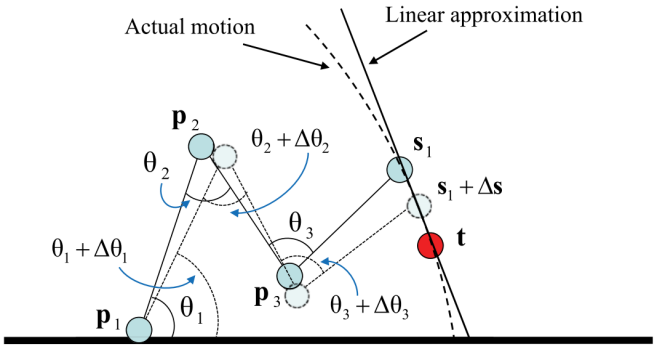

Abbildung 5.8: Lineare Annäherung [ALCS18]

$\boldsymbol{\delta\theta}$ auf die Positionsänderung der Endeffektoren $\boldsymbol{\delta s}$ als lineare Annäherung ab. Dabei wird die Tangente des eigentlichen Bewegungspfads beschrieben (siehe Abb. 5.8). Die Jacobi-Matrix J kann als Funktion der Gelenkwinkel $\boldsymbol{\theta}$ durch

$$J(\boldsymbol{\theta})_{ij} = \left(\frac{\delta s_i}{\delta \theta_j}\right) \tag{5.8}$$

beschrieben werden, wobei $i = 1, ..., k$ und $j = 1, ..., n$ (k ist die Anzahl an Endeffektoren, n ist die Anzahl an Gelenkwinkeln). J ist eine $k \times n$ Matrix, deren Elemente Positionsvektoren sind. Die Elemente von J können mit

$$\frac{\delta s_i}{\delta \theta_j} = v_j \times (s_j - p_j) \tag{5.9}$$

berechnet werden, wobei v_j Einheitsvektoren sind, welche entlang der Rotationsachse des j-ten Gelenks verlaufen. Gesucht sind nun die Winkeleinstellungen $\boldsymbol{\theta}$, für welche die Fehler $\boldsymbol{e}$ minimal sind. Dabei ist e_i der Vektor vom i-ten Endeffektor s_i zu dessen Zielposition t_i. Dieser Fehler wird definiert als

$$e_i = t_i - s_i(\theta). \tag{5.10}$$

Die Änderung der Endeffektorpositionen $\Delta\boldsymbol{s}$ bei Änderung der Gelenkwinkel $\Delta\boldsymbol{\theta}$ wird angenähert durch

$$\Delta\boldsymbol{s} \approx J\Delta\boldsymbol{\theta}. \tag{5.11}$$

Um den Fehler zu minimieren, suchen wir nach Werten $\Delta\boldsymbol{s}$, welche möglichst nah an $\boldsymbol{e}$ liegen. Darum schätzen wir die Veränderung der Winkelpositionen $\boldsymbol{\theta}$ mit

$$\Delta\boldsymbol{\theta} \approx J^{-1}\boldsymbol{e}. \tag{5.12}$$

Jedoch kann es sein, dass J nicht quadratisch und somit nicht invertierbar ist. Deshalb nutzen wir stattdessen die Pseudoinverse, auch *Moore-Penrose Inverse*, von J. Diese hat den Vorteil, dass sie im Gegensatz zur Inverse immer definiert ist. Die Pseudoinverse der Jacobi-Matrix wird wie folgt berechnet:

$$\Delta\boldsymbol{\theta} = J^T(JJ^T)^{-1}\boldsymbol{e} \tag{5.13}$$

Die Gelenke werden nun um die Winkel $\Delta\boldsymbol{\theta}$ rotiert, womit die erste Iteration beendet ist. Über mehrere Iterationen konvergiert der Endeffektor mit der Zielposition. Beim Jacobi-Verfahren, im Gegensatz zu CCD, verteilt sich die Transformation der kinematischen Kette auf alle beteiligten Glieder und es ist möglich mehrere Zielpositionen festzulegen. Allerdings hat das Verfahren das Problem, dass es in der Nähe einer Singularität fehleranfällig ist. Wenn der Zielpunkt sich außer Reichweite befindet, kann das dazu führen, dass das Modell oszilliert und nicht mit der Zielposition konvergiert. (vgl. [ALCS18]: 39-40)

5.2 Stoffsimulation

Ein Stück Stoff lässt sich mithilfe des *Verlet-Algorithmus* simulieren (vgl. [Cou]). Zur Vereinfachung wird ein Körper als eine Punktmasse betrachtet. In *Kinematics and Dynamics of Computer Animation* [REG+16] wird erklärt, dass *bei Betrachtung der Bewegung eines Objekts, dessen Größe im Vergleich zur zurückgelegten Strecke unerheblich ist und dessen Rotation und Verformung nicht von Interesse sind, das Objekt idealisiert als mathematischer Punkt mit einer Masse beschrieben werden kann. Die Bewegung dieser Punktmasse wird komplett von ihrer Laufbahn im Raum bestimmt, also durch dessen Positionsvektor* $x(t)$ *und dessen Geschwindigkeit* $v(t)$. ([REG+16]: 2)

Der *Verlet-Algorithmus* (vgl. [Sch]: 7f) bestimmt die Position x einer solchen Punktmasse zum Zeitpunkt t. v ist die Geschwindigkeit und a die Beschleunigung der Punktmasse. Zur Anwendung des *Verlet-Algorithmus* werden zwei Positionen benötigt, x_n und x_{n-1}. Ist die Position x_0 bekannt, muss zuerst die Position x_1 bestimmt werden. Dazu wird folgende Formel angewandt:

$$\vec{x}_1 = \vec{x}_0 + \vec{v}_0\Delta t + \frac{1}{2}\vec{a}_0\Delta t^2 \tag{5.14}$$

Durch Iteration des *Verlet-Algorithmus* können nun alle Positionen $\vec{x}_n$ bestimmt werden.

$$\vec{x}_{n+1} = 2\vec{x}_n - \vec{x}_{n-1} + \vec{a}_n\Delta t^2 \tag{5.15}$$

Es ist anzumerken, dass der *Verlet-Algorithmus* die Geschwindigkeitsberechnung aus der Formel für gleichmäßig beschleunigte Bewegung eliminiert. Das wird nachher von Nutzen sein.

Der Pseudo-Code in Listing 5.1 des *Verlet-Algorithmus* für eine Punktmasse wird wiederholt ausgeführt. Um bei variierender Framerate eine konsistente Simulation zu erreichen, wird $deltaTime$ in die Rechnung miteinbezogen. $deltaTime$ ist die seit dem letzten Schleifendurchgang verstrichene Zeit in Sekunden.

```
for pm in point_masses {
    // Berechne Geschwindigkeit [velocity]
    pm.velocity = (pm.position - pm.last_position) / deltaTime;

    // Speichere Position fuer naechsten Frame
    pm.last_position = pm.position;
```

```
7
8       // Zuweisung der neuen Position
9       pm.position = pm.position + pm.velocity * deltaTime
10              + (1/2 * pm.acceleration * deltaTime * deltaTime);
11
12      // Setze die Beschleunigung [acceleration] auf Fallbeschleunigung
13      pm.acceleration = vector(0., -9.81, 0.);
14  }
```

Listing 5.1: Verlet Pseudocode

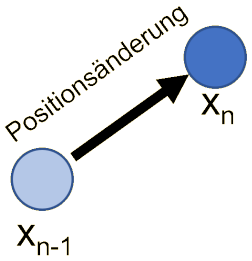

Abbildung 5.9: Positionsänderung

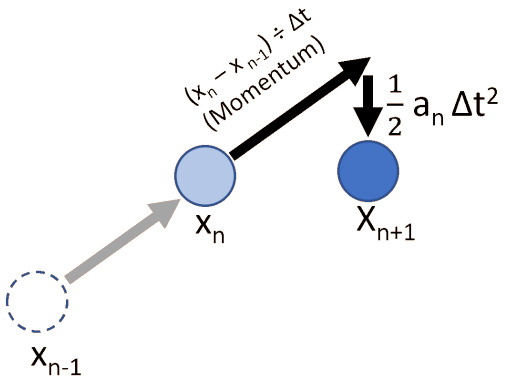

Abbildung 5.10: Momentum und Beschleunigung

Das Array $point\_masses$ enthält Instanzen der Klasse PointMass, mit den Feldern $velocity$, $last\_position$, $position$ und $acceleration$, welche Vektoren sind. $velocity$ ergibt sich aus der letzten und der derzeitigen Position. $last\_position$ enthält x_{n-1}, $position$ enthält x_n. In diesem Fall wird die y-Komponente der Beschleunigung $acceleration$ auf -9.81 gesetzt, um Schwerkraft zu simulieren.

Der *Verlet-Algorithmus* berechnet das Momentum positionsbasiert. Das Momentum ergibt sich aus der vorherigen Position, der derzeitigen Position und ggf. Beschleunigung aufgrund von äußeren Krafteinwirkungen wie z.B. der Schwerkraft. (siehe Abb. 5.9 und 5.10)

Ein Stück Stoff wird als eine Ansammlung von Punktmassen definiert, welche mit Gliedern verbunden sind. Die Glieder werden in Beziehung zu den jeweils zwei dazugehörigen Punktmassen definiert. Die Länge eines Glieds ist der Abstand der beiden Punktmassen zueinander. Nun werden Constraints[3] für das Stoffstück festgelegt: Jedes Glied hat eine Soll-Länge, welche angestrebt wird. Durch den Einfluss von Schwerkraft und von anderen Kräften verändert sich der Abstand zwischen den Punktmassen. Um dem entgegenzuwirken, werden die dem Stoffstück zugewiesenen Constraints gelöst[4]. Ist die Distanz zwischen

[3] engl. für Einschränkung, Als Constraint wird eine Bedingung bezeichnet, z.B. der minimale Abstand eines Objekts zu einem anderen.

[4] Ein Constraint wird als gelöst angesehen, wenn die ihm zugewiesene Bedingung erfüllt ist.

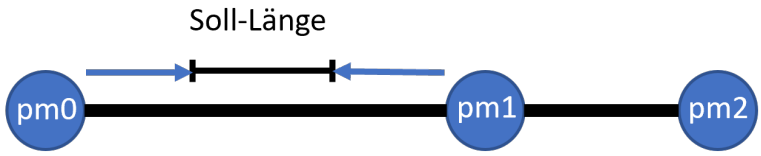

Abbildung 5.11: Ersten Constraint lösen

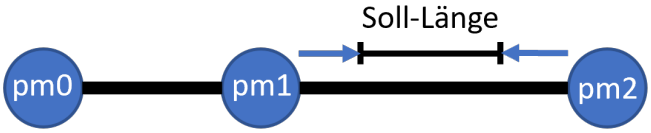

Abbildung 5.12: Zweiten Constraint lösen

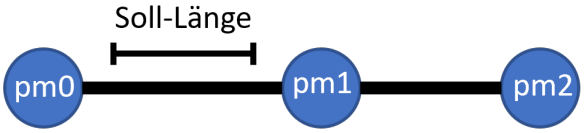

Abbildung 5.13: Erster Constraint ungelöst

Abbildung 5.14: Ausgangs- und Zielzustand und Momentum

zwei Punktmassen, welche mit einem Glied verbunden sind, kleiner oder größer als die Soll-Länge, dann werden die beiden Massen aufeinander zu- bzw. voneinander wegbewegt. Das wird nacheinander für alle Glieder getan. Durch das Lösen eines Constraints kommt es vor, dass ein anderer Constraint den gelösten Zustand verlässt. (siehe Abb. 5.11, 5.12, 5.13 und 5.14)

Das führt zu einem unglaubwürdigen Verhalten des Stoffstücks, da sich einige Bereiche dehnen, während andere steif wirken. Der Ablauf der Simulation ist zudem stark davon abhängig, welcher Constraint zuerst gelöst wird. Um diese Abweichungen zu minimieren, werden die Constraints nicht nur einmal gelöst, sondern über mehrere Iterationen hinweg. Je mehr Iterationen durchgeführt werden, desto geringer ist der Fehler und desto steifer wirkt das Stoffstück.

Da das Momentum positionsbasiert berechnet wird, bewegen sich Punktmassen, welche durch Contraints verschoben wurden, weiterhin in die Richtung der Verschiebung. Dadurch ergibt sich ein näherungsweise realistisches Verhalten. Die Positionen der Punktmassen werden gleichzeitig als Mesh-Vertices für das Rendern der Textur verwendet. Abbildungen 5.15, 5.16 und 5.17 zeigen ein Beispiel einer Stoffsimulation.

Abbildung 5.15:
1. Verformung,
(Verändert von: t0.gstatic.com)

Abbildung 5.16:
2. Verformung,
(Verändert von: t0.gstatic.com)

Abbildung 5.17:
2. Verformung mit Mesh,
(Verändert von: t0.gstatic.com)

Obwohl diese Anwendung ein 2D-Animationseditor ist, werden für die Stoffsimulation 3D-Vektoren verwendet. Die Stoffsimulation läuft also im 3D-Raum ab, die perspektivische Verzerrung wird beim Rendern allerdings vernachlässigt. Weil ein Stoffstück sich für gewöhnlich immer im 3D-Raum bewegt, erhöht das den Realismus. Des Weiteren kann die z-Koordinate an die Grafikpipeline weitergegeben werden, um die Tiefe korrekt darzustellen, wenn es zu Überlappungen des Stoffstücks kommt. In diesem Fall muss entschieden werden, welcher Teil des Stoffstücks am nächsten an der Kamera und somit sichtbar ist. Der Algorithmus der Stoffsimulation kann angepasst werden, um nicht nur Stoff, sondern auch andere Textilien wie beispielsweise Haar oder Seile darzustellen. Die physikalische Simulation dieser Teile einer Figur helfen bei der Umsetzung des Animationsprinzips *Follow Through*. Gleichzeitig berücksichtigt sie das Prinzip *Slow In & Slow Out*.

Kapitel 6

Fazit

6.1 Ergebnisse

Der im Rahmen dieser Arbeit implementierte Animationseditor erfüllt die anfangs aufgestellten Anforderungen größtenteils. Die Anwendung kann lokal installiert[1] oder im Browser ausgeführt[2] werden. Der Editor benutzt Forward Kinematics, um ein Skelett in beliebige Pose zu bringen. Basierend auf der Pose des Skeletts wird die Skin manipuliert. Zwischen manuell vom Nutzer festgelegten Keyframes wird interpoliert, um eine flüssige Animation zu erhalten.

Meshes werden automatisch generiert. Entsprechend der Bewegung des Skeletts verformen sich die Meshes und passen sich der momentanen Skelettpose an. Inverse Kinematics ermöglichen es, die Figur zu animieren, indem allein die Position des Endeffektors vom Animator bestimmt wird. Stoffähnliche Textilien können mit dem Verlet-Algorithmus automatisch simuliert werden. Animation Blending ermöglicht es, nicht nur zwischen Keyframes, sondern auch zwischen Animationen zu interpolieren. Dadurch entfällt das Erstellen von Übergangsanimationen, womit ein Arbeitsschritt des Animators eingespart wird.

Im Folgenden wird evaluiert, inwiefern der Editor die Anwendung der 12 Animationsprinzipien [TJ81] ermöglicht und fördert. Die Natur von Skelettanimationen erleichtert das Einhalten des Prinzips *Solid Drawing*, da die Herausforderung, jeden Frame von Hand zu zeichnen, eliminiert wird. Der gezielte Einsatz von passenden Interpolationsfunktionen hilft beim Befolgen der Prinzipien *Anticipation* und *Slow In & Slow Out*. Uneinheitliche Skalierung kann eingesetzt werden, um *Squash & Stretch* umzusetzen. Der Verlet-Algorithmus kann verwendet werden, um *Follow Through & Overlapping Action* für stoffähnliche Textilien umzusetzen. *Secondary Action* kann unter Umständen durch Animation-Layering vereinfacht werden. Das *Timing* kann mit dem Editor auch nach Fertigstellung einer Animation leicht angepasst werden, indem der Abstand zwischen Keyframes verändert wird. Die Anwendung der übrigen Prinzipien wird durch den Editor weder gefördert noch verhindert.

[1] https://github.com/lordbenedikt/skeletal_animation_2d_remake
[2] https://lordbenedikt.github.io/skeletal_animation_2d_remake/

Um alle Funktionen des Editors verständlich zu machen, liegt eine Nutzeranleitung vor. Diese ist über den Link in der Anwendung zugänglich.

6.2 Ausblick

Die in dieser Arbeit untersuchten Techniken sind grundlegend und schöpfen nicht alle vorhandenen Möglichkeiten aus. Es gibt nennenswerte Techniken, welche auf dem bereits Erklärten aufbauen. Auf der Website der Software *Spine* werden einige Techniken demonstriert[3]. Einige nützliche Techniken werden im Folgenden kurz erläutert.

6.2.1 Kombinierung mit der Einzelbild-Animation

Wie bereits erwähnt, ist eine große Einschränkung der Skelettanimation in 2D, dass Figuren immer nur von der gleichen Perspektive abgebildet werden können. In der Einzelbild-Animation gibt es diese Einschränkung nicht. Da ohnehin jedes Bild von Hand gezeichnet wird, kann der Animator die Perspektive jedes Mal frei wählen. Um die Vorteile der Skelettanimation und der Einzelbild-Animation zusammenzuführen, ist es möglich, die Bilder in der Skelettanimation auszutauschen. Das Bild eines Elements in der Skelettanimation kann z.B. im Moment seiner Drehung mit einem Bild aus einer anderen Perspektive ausgetauscht werden. Diese Vorgehensweise ist zwar aufwändiger, doch da eine prominente Drehung meist nur in bestimmten Teilen eines Bewegungsablaufs stattfindet, ist es nicht notwendig, die Bilder ständig auszutauschen.

6.2.2 Transformation Constraints

Es kommt oft vor, dass zwei Objekte sich in Abhängigkeit voneinander bewegen. Das ist beispielsweise bei Rädern eines fahrenden Autos der Fall. Alle vier Räder drehen sich mit der gleichen Geschwindigkeit. Um Zeit und Arbeit einzusparen, ist es möglich, sogenannte Tranformation Constraints zu definieren. Dreht sich eines der vier Räder, so drehen sich die drei anderen automatisch. Dabei müssen die voneinander abhängigen Größen nicht direkt proportional zueinander sein. Befinden sich ein großes und ein kleines Zahnrad nebeneinander, welche 10 bzw. 20 Zähne haben, so bewegt sich das kleine Rad mit der doppelten Geschwindigkeit des großen Rads und in entgegengesetzte Richtung.

6.2.3 Path Constraints

Path Constrainsts[4] ermöglichen das Animieren eines Objekts entlang eines kurvigen Pfads. Betrachten wir folgendes Beispiel: Die Translation eines Objekts verändert sich über zwei Keyframes hinweg. Wird Interpolation so durchgeführt, wie sie in dieser Arbeit beschrieben wird, hat das die folgende Konsequenz: In jedem Frame, welcher zwischen den beiden

[3] http://en.esotericsoftware.com/spine-demos

[4] engl. für Pfad-Bedingung, Ein *Path Constraint* beschränkt die Bewegung eines Objekts so, dass es sich ausschließlich auf einem definierten Pfad bewegen kann.

Keyframes liegt, befindet sich das Objekt irgendwo auf einer Linie, welche die Positionen des Objekts in den beiden Keyframes verbindet (siehe Abb. 6.1). Nach diesem Vorgehen ist es nicht möglich, eine kurvenförmige Bewegung zu beschreiben. Pfade ermöglichen es, diese Einschränkung zu überwinden (siehe. Abb. 6.2). Path Constraints unterstützen das Animationsprinzip *Arcs* und tragen damit nennenswert zur Erstellung einer lebendig wirkenden Animation bei.

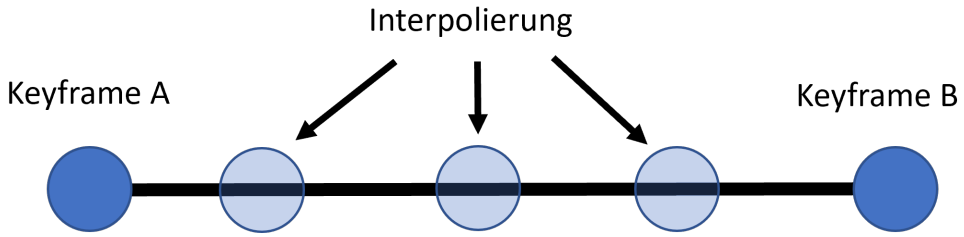

Abbildung 6.1: Ohne Path Constraint

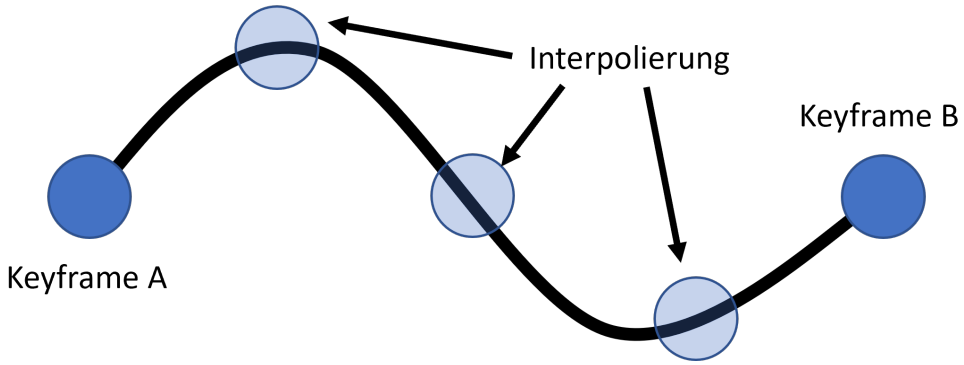

Abbildung 6.2: Mit Path Constraint

6.2.4 Normal-Mapping

Abbildung 6.3: Normal Map Beispiel[5]

Eine *Normal-Map* liefert detaillierte Informationen zur Oberfläche, welche bei der Schattierung berücksichtigt werden. Sie ist eine weitere Textur, welche zusätzlich zu der *Color-Map*[6] verwendet wird. Anders als die *Color-Map* speichert die *Normal-Map* nicht die Farbe, sondern die Normale der Oberfläche am entsprechenden Punkt. Der Einsatz von *Normal-Mapping* in der 2D-Skelettanimation verbessert die Möglichkeit, Plastizität zu vermitteln. Objekte reagieren dynamisch auf die Lichtverhältnisse und integrieren sich dadurch besser in die Umgebung. Abbildung 6.3 zeigt ein Beispiel. Links oben ist das Originalbild der Figur zu sehen. Rechts daneben ist die *Normal-Map* abgebildet. Die restlichen Bilder zeigen die Figur bei unterschiedlichen Belichtungen. (vgl. [Slo06])

[5] Games, DYA, 2022, https://www.kickstarter.com/projects/2dee/ sprite-dlight-instant-normal-maps-for-2d-graphics

[6] die primäre Textur eines Meshs, welche die Farbe bestimmt

Glossar

Skelett	Logisches Konstrukt, welches der Transformation einer animierten Figur zugrundeliegt. Ein Skelett besteht aus hierarchisch angeordneten Knochen.
Knochen	Elemente, welche ein Skelett bilden. Einem Knochen wird für gewöhnlich ein zusammenhängender Bereich eines Körpers zugeordnet.
Skin	Die Skin wird vom Skelett aufgespannt. Sie bestimmt die Erscheinung einer Figur. In der 2D-Skelettanimation besteht die Skin aus einer oder mehreren Bilddateien.
Figur	Eine Person oder anderes Lebewesen, ein Mechanismus oder jegliches zu animierendes Objekt mit einer vergleichbaren Struktur
Prinzipien der Animation	Die 12 Animationsprinzipien zur Animation lebendiger Figuren und Welten nach Ollie Johnston und Frank Thomas [TJ81]
Forward Kinematics	Berechnung der Position und Orientierung des Endeffektors einer kinematischen Kette
Inverse Kinematics	Berechnung der Rotation aller beteiligter Glieder einer kinematischen Kette, zur Erreichung einer gewünschten Position, vorliegend als kartesische Koordinaten, und Orientierung des Endeffektors
Mesh	Ein Mesh ist eine Ansammlung von Vertices (engl. für Eckpunkte) und Indizes. Jeweils drei Indizes definieren die Eckpunkte eines Dreiecks. Die Gesamtheit aller Dreiecke bildet eine Oberfläche. Soll das Mesh textuiert werden, muss zusätzlich ein Array von UVs festgelegt werden. Je Vertex wird ein UV festgelegt. Meshes werden in der Computergrafik sowohl im 3D- als auch im 2D-Bereich verwendet.
Textur	Eine Textur ist eine Bilddatei, mit welcher ein Mesh textuiert werden kann. Ein Array von UVs (Bildkoordinaten) bestimmt dabei, welche Stellen der Bilddatei, an welchen Stellen auf dem Mesh angezeigt werden.

Literaturverzeichnis

[ALCS18] Aristidou, Andreas ; Lasenby, Joan ; Chrysanthou, Yiorgos ; Shamir, Ariel: Inverse kinematics techniques in computer graphics: A survey. In: *Computer graphics forum* Bd. 37 Wiley Online Library, 2018, S. 35–58

[bev] *The Bevy Book*. https://bevyengine.org/learn/book

[Cor17] Corrigan, Nicholas C.: *Outta This World: Merging Classic Animation Styles with Modern Technologies and Designs*, Ohio University, Diss., 2017

[Cou] Counts, Jared: Simulate tearable Cloth. https://gamedevelopment.tutsplus.com/tutorials/simulate-tearable-cloth-and-ragdolls-with-simple-verlet-integration--gamedev-519

[DKL98] Dam, Erik B. ; Koch, Martin ; Lillholm, Martin: *Quaternions, interpolation and animation*. Bd. 2. Citeseer, 1998

[FÔ3] Fêdor, Martin: Application of Inverse Kinematics for Skeleton Manipulation in Real-Time. In: *Proceedings of the 19th Spring Conference on Computer Graphics*. New York, NY, USA : Association for Computing Machinery, 2003 (SCCG '03). – ISBN 158113861X, 203–212

[Feb18] Febucci: Easing Functions for Animations. (2018). https://www.febucci.com/2018/08/easing-functions/

[FHKS12] Feng, Andrew ; Huang, Yazhou ; Kallmann, Marcelo ; Shapiro, Ari: An analysis of motion blending techniques. In: *International Conference on Motion in Games* Springer, 2012, S. 232–243

[GN13] Gong, Sui ; Newman, Timothy S.: A corner feature sensitive marching squares. In: *2013 Proceedings of IEEE Southeastcon* IEEE, 2013, S. 1–6

[HCSZ21] Hatledal, Lars I. ; Chu, Yingguang ; Styve, Arne ; Zhang, Houxiang: Vico: An entity-component-system based co-simulation framework. In: *Simulation Modelling Practice and Theory* 108 (2021), S. 102243. – ISSN 1569–190X

[Joh09] Johansen, Rune S.: Automated semi-procedural animation for character locomotion. In: *Aarhus Universitet, Institut for Informations Medievidenskab* (2009)

[Jun20] Jung, Ralf: Understanding and Evolving the Rust Programming Language. (2020). https://www.ralfj.de/research/phd/thesis-screen.pdf

[KN19] Klabnik, Steve ; Nichols, Carol: *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019

[Kun19] Kunze, Erhart G.: Mathematische Grundlagen der Quaternionen. (2019). https://ekunzeweb.de/PAPERS/Mathematische_Grundlagen_der_Quaternionen.pdf

[Map03] Maple, C.: Geometric design and space planning using the marching squares and marching cube algorithms. In: *2003 International Conference on Geometric Modeling and Graphics, 2003. Proceedings*, 2003, S. 90–95

[MSJT08] Müller, Matthias ; Stam, Jos ; James, Doug ; Thürey, Nils: Real Time Physics: Class Notes. In: *ACM SIGGRAPH 2008 Classes*. New York, NY, USA : Association for Computing Machinery, 2008 (SIGGRAPH '08). – ISBN 9781450378451

[Muk02] Mukundan, Ramakrishnan: Quaternions: From classical mechanics to computer graphics, and beyond. In: *Proceedings of the 7th Asian Technology conference in Mathematics*, 2002, S. 97–105

[PC89] Paul Chew, L: Constrained delaunay triangulations. In: *Algorithmica* 4 (1989), Nr. 1, S. 97–108

[PW94] Patterson, John W. ; Willis, Philip J.: Computer assisted animation: 2D or not 2D? In: *The Computer Journal* 37 (1994), Nr. 10, S. 829–839

[REG+16] Ruder, Hanns ; Ertl, Thomas ; Gruber, Karin ; Günther, Michael ; Hospach, Frank ; Subke, Jörg ; Widmayer, Karin: *Kinematics and dynamics for computer animation*. http://nbn-resolving.de/urn:nbn:de:bsz:93-opus-76860. Version: 2016

[Sch] Schwermann, Alexander Christian und F. Christian und Fulst: Molekulardynamiksimulation.

[Sho85] Shoemake, Ken: Animating rotation with quaternion curves. In: *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, 1985, S. 245–254

[Sit] Sitnik, Andrey: Easings Functions. https://easings.net/

[Slo06] Sloan, Peter-Pike: Normal mapping for precomputed radiance transfer. In: *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 2006, S. 23–26

[TJ81] Thomas, Frank ; Johnston, Ollie: *The illusion of life: Disney animation*. Abbeville Press, 1981