Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Implementierung eines 3D Animations-Systems mit Dual Quaternion Skinning

Florian Oetke
(Matrikel-Nr. 957795)

Projektdokumentation zur Vorlesung Advanced Game Technology

Betreuer: Prof. Dr. Christof Rezk-Salama

Trier, 31.08.2018

Kurzfassung

Diese Dokumentation beschreibt die Implementierung und Funktionsweise des, als Projektarbeit für das Modul Advanced Game Technology implementierten, skelettbasierten 3D Animationssystems. Zur Transformation der Vertices wurde sowohl Linear Blend als auch Dual Quaternion Skinning implementiert.

Inhaltsverzeichnis

1

Einleitung

Detaillierte Animation von 3D-Modellen stellen einen der wichtigsten Aspekte einer realistischen bewegten Szene dar. Somit bilden sie auch einen wichtigen Grundpfeiler für ein gutes Spielgefühl bei Computerspielen. Diese stellen allerdings auch sehr hohe Anforderungen an die Laufzeit der hierzu eingesetzten Verfahren, womit eine Abwägung zwischen Realismus und Laufzeitgeschwindigkeit notwendig ist.

Das bei 3D-Spielen verbreitetste Animationsverfahren ist die skelettbasierte Animation. Hierbei wird zusätzlich zu den Vertices des Modells, welche die Oberfläche[1] bilden, eine Hierarchie von Knochen[2] definiert. Anschließend werden bei einem Rigging genannten Prozess jedem Vertex eine bestimmte Anzahl Knochen zugeordnet, von denen er beeinflusst wird. Während diesem Prozess sollten sich das Modell und das Skelett in einer äquivalenten Pose befinden. Diese wird Bind-Pose genannt und bildet im folgenden die Basis für alle Animationen. [Rez16]

Zum Abspielen einer Animation wird dieses Skelett durch Modifikation der Knochen-Transformationen in die gewünschte Position gebracht und anschließend die Vertices transformiert, sodass sie ihre relative Position zu ihren verbundenen Knochen behalten. Als Quelle für die Zieltransformationen der Knochen können z.B. vorher aufgezeichnete Keyframes dienen, welche für bestimmte Zeitpunkte eine Transformation des Skeletts festlegen, zwischen denen dann interpoliert wird um den Eindruck einer flüssigen Animation zu erreichen. Alternativ kann die Bewegung des Skeletts z.B. auch über eine Physiksimulation oder einen IK-Solver[3] bestimmt oder manipuliert werden. [Rez16]

Das Animationsverfahren wurde auf Basis eines existierenden Vulkan Deferred Renderers implementiert. Diese Implementierung verfügt neben dem verbreiteten Linear Blend Skinning (LBS) Verfahrens außerdem über Dual Quaternion Skinning (DQS), welches einige der Artefakte und Probleme von LBS löst. Zur Demonstration wurde eine Beispielszene angelegt welche einen Vergleich beider Verfahren

[1] engl. Skin.

[2] engl. Bone oder häufig auch Joint. Bestehen jeweils aus Angaben zur lokalen Translation, Rotation und Skalierung des Knochens relativ zu ihrem Vater-Knochen.

[3] Ein Inverse-Kinematik Solver ist ein System, dass versucht eine festgelegte globale Transformation eines Bones unter Einhaltung gewisser Bedingungen zu erreichen. z.B.: die Hand in eine bestimmte Position bringen, ohne den Abstand zwischen einzelnen Bones zu verändern oder Gelenke unnatürlich zu rotieren.

bei einem einfachen Test-Modell und einem komplexeren Charakter-Modell[4] mit mehreren Animationen ermöglicht (siehe Abb. 1.1). Die abgespielten Animationen lassen sich dabei über eine Reihe von Optionen auf der rechten Seite steuern. Mittels dieser kann die aktuelle Animation des Charakter-Modells gewählt (inkl. blenden zur neuen Animation), die Zeitachse und Geschwindigkeit manipuliert, pausiert, umgekehrt sowie die Wiederholung aktiviert/deaktiviert werden. Beim einfachen Test-Modell kann die Animation lediglich pausiert werden.

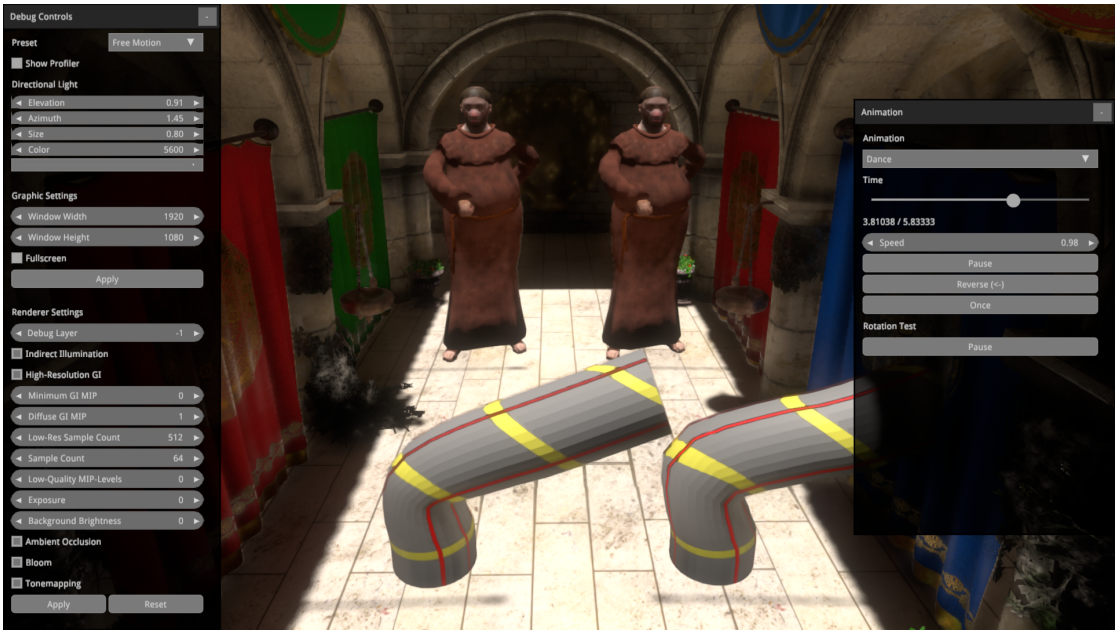


Abb. 1.1: Ein Screenshot der implementierten Demo-Anwendung. Auf der linken Seite befinden sich allgemeine Einstellungsmöglichkeiten, wie die Auflösung und Optionen zur Lichtquelle und Grafikqualität. Mit den Steuerelementen auf der rechten Seite können die abgespielten Animationen gesteuert werden. Die Szene enthält zwei animierte Modelle, ein einfaches Testmodell und ein komplexeres Modell mit mehreren unterschiedlichen Animationen. Diese werden jeweils mit den beiden implementierten Skinning-Verfahren Linear Blend Skinning (link) und Dual Quaternion Skinning (rechts) dargestellt.

[4] Das Modell steht unter CC0 und wurde von *hwoarangmy* mit Animationen unter CC-BY-SA 3.0 versehen: `https://opengameart.org/content/monk-animated`

2

Implementierung

Das Abspielen der eigentlichen Animation erfolgt anschließend in zwei Schritten. Als Erstes wird das Skelett über Änderung der lokalen Transformation in die gewünschte Position gebracht. Im zweiten Schritt werden anschließend alle Vertices so transformiert, dass sie ihre relative Position zu ihrem zugeordneten Knochen beibehalten. Bei mehreren Knochen pro Vertex muss ein gewichteter Mittelwert zwischen allen gebildet werden. Durch diesen Linear Blend Skinning genannten Ansatz entstehen allerdings u.U Artefakte, auf die in Kapitel 2.3 eingegangen wird. [Rez16]

Die 3D Modelle und Animationen werden zumeist in einem 3D-Modellierungs-Programm entworfen und in ein Dateiformat wie FBX exportiert. Der erste Schritt zum Abspielen der Animationen ist es also zuallererst alle benötigten Daten aus diesem Dateiformat in eine, zur Laufzeit effizient nutzbare, Datenstruktur zu laden. Aus Effizienzgründen erfolgt der aufwändigste Teil dieser Konvertierung in einem Offline-Verarbeitungsschritt, welcher Dateien generiert, die zur Laufzeit möglichst direkt in den Speicher geladen werden können. Zum Laden der externen Dateiformate kommt in dieser Implementierung die Assimp-Bibliothek zum Einsatz, welche eine Vielzahl unterschiedlicher Formate unterstützt.

Die zentrale Datei, die von diesem Konvertierungsprozess generiert wird, ist das eigentliche 3D-Modell (.mmf). Diese Dateien enthalten neben der eigentlichen Geometrie – d.h. eine Liste von Vertices (optional mit Bone-Zuordnungen und -gewichten) und Indices, welche die Dreiecke des Modells bilden – auch einige zusätzliche Informationen, welche zum effizienten Zeichnen des Modells notwendig sind. Die wichtigste von diesen ist eine Partitionierung der Indices in eine Menge von Sub-Meshes, die jeweils das selbe Material verwenden. Dazu kommen weitere Optionen, wie z.B. ob die Vertices über eine Knochen-Zuordnung verfügen und eine Menge von Bounding-Spheres, die zur Laufzeit verwendet werden um die sichtbare Geometrie zu bestimmen (Culling). Der Quellcode für dieses Dateiformat befindet sich im Projekt in `src/mirrage/renderer/include/mirrage/renderer/model.hpp` und der Code zur Erstellung in `src/mesh_converter/model_parser.hpp`.

Die von den Sub-Meshes referenzierten Materialien werden in eigenen Dateien beschrieben (.msf). Diese enthalten neben der ID des zu verwendenden Material-Shaders auch die Asset-IDs der vom Material verwendeten Texturen. Diese Texturen sind zur Zeit zum einen eine Albedo-Textur, welche die Farbe des Objekts

festlegt, und zum anderen eine Material-Textur, welche materialspezifische Eigenschaften wie Normalen, Roughness und Metallic enthält. Des Weiteren werden die Texturen ebenfalls in ein Dateiformat konvertiert, das direkt auf die GPU hochgeladen werden kann (KTX). Der Quellcode für dieses Dateiformat befindet wie beim Modell-Format primär in `src/mirrage/renderer/include/mirrage/renderer/model.hpp`, mit dem Konverter-Teil in `src/mesh_converter/material_parser.hpp`.

Die erste für Animationen notwendige Datei enthält die Informationen eines Skeletts (.mbf). Diese besteht neben Optionen (z.B. zu verwendendes Skinning-Verfahren) aus den Daten der einzelnen Knochen, d.h. die Position, Rotation und Skalierung in Bind-Pose, sowie die inverse Bind-Pose-Matrix $B^{-1}_{(n)}$, welche vom Modell-Koordinatensystem in das des jeweiligen Knochens transformiert. Die Knochen werden dabei in Hauptreihenfolge (depth-first, pre-order) angegeben und enthalten einen Index in die Liste, mit dem sie ihren Vater-Knoten referenzieren und so die Baumstruktur des Skeletts abbilden.

Das letzte Dateiformat enthält die eigentlichen Animationsdaten (.maf). Diese bestehen primär aus einer Liste von Keyframe-Transformationen (getrennt nach Position, Rotation und Skalierung) mit Zeitstempeln pro Knochen und Angaben wie die Interpolation zwischen Keyframes durchzuführen ist.

Der Quellcode für die Skelett- und Animations-Dateien befindet sich in `src/mirrage/renderer/include/mirrage/renderer/animation.hpp` und der Konverter in `src/mesh_converter/animation_parser.hpp` sowie `src/mesh_converter/skeleton_parser.hpp`.

Zur Laufzeit verteilen sich die für die Animationen relevanten Daten im wesentlichen auf drei Komponenten[1]. Die Erste ist die `Pose_comp`, welche eine Referenz auf die geladenen Skelett-Informationen und die aktuellen Transformationen aller Knochen, getrennt in Position, Rotation und Skalierung enthält. Somit bildet diese Komponente die primäre Kommunikationsschnittstelle zwischen dem Code, der die Animationen abspielt und dem eigentlichen Rendering-Code.

Die anderen beiden wichtigen Komponenten sind `Animation_comp` sowie `Simple_animation_controller_comp`. Die `Animation_comp` hält dabei den Zustand [2] einer oder mehrere aktuell auf dem Modell abgespielten Animationen und wird von einem Animation-Controller gesteuert. Der einzige z.Z. implementierte Animation-Controller ist die `Simple_animation_controller_comp`-Komponente, welche lediglich eine aktive Animation unterstützt und bei einem Wechsel der Animation flüssig von der alten zur neuen interpoliert. In Zukunft sind aber auch komplexere Implementierungen – z.B. auf Basis eines Zustandsautomaten, Blend-Trees oder Skripten – denkbar.

Basierend auf diesen Komponenten werden in vier Schritten die transformierten Vertices des Modells bestimmt. Als erstes wird durch den Animation-Controller ggf. die laufende Animation der Modell modifiziert und im folgenden der Zeitstempel aller Animationen aktualisiert. Anschließend werden die in der `Pose_comp` gespeicherten Transformationen für alle sichtbaren Modelle, wie in Kapitel 2.1 beschrieben, neue berechnet. Diese werden dann in einem letzten Schritt verwendet um pro Knochen eine globale Transformationmatrix zu berechnen, die vom Model-

[1] Alle in `src/mirrage/renderer/include/mirrage/renderer/animation_comp.hpp`.
[2] u.a. Animations-Daten, Zeitpunkt und Geschwindigkeit.

Koordinatenraum in Bind-Pose in den Model-Koordinatenraum mit der angewendeten Animation transformiert. Abschließend wird diese dann im Vertex-Shader verwendet um die einzelnen Vertices zu transformieren. Dazu werden zuerst alle Transformationen zu Matrizen konvertiert und, wie in Gleichung 2.1 skizziert, mit der globalen Transformationsmatrix ihres übergeordneten Knochens sowie der inversen Bind-Pose-Matrix multipliziert um sie in den entsprechenden Koordinatenraum zu bringen. [Rez16]

Beim Entwurf dieser Implementierung wurde davon ausgegangen, dass Animationen lediglich für die Darstellung relevant sind und keine Auswirkungen auf die Physik oder das Gameplay haben. Dies ist zwar nicht bei allen Anwendungen der Fall, erlaubt aber deutlich mehr und komplexere Animationen, da lediglich jene aktualisiert werden müssen, die im aktuellen Frame sichtbar sind, d.h. die Culling-Tests bestanden haben. Hierdurch ist das Animations-System allerdings auch stärker mit dem Rest des Renderers verzahnt, da das Animations-Update zwischen dem Culling und dem Zeichnen der Modelle durchgeführt werden muss.

$$
\begin{aligned}
P_n^g &= P^g_{\text{parent}(n)} * P_n \\
M_n &= P_n^g * B_n^{-1}
\end{aligned}
$$

$$
\begin{aligned}
\text{mit } M_n &= \text{die globale Vertex-Transformationsmatrix für Knochen } n, \\
P_n &= \text{die lokale Transformationsmatrix für Knochen } n, \\
P_n^g &= \text{die globale Transformationsmatrix für Knochen } n, \\
P^g_{\text{parent}(n)} &= \text{die globale Transformationsmatrix für den Vater-Knochen von } n \text{ und} \\
B_n^{-1} &= \text{die inverse Bind-Pose Matrix für Knochen } n.
\end{aligned}
\tag{2.1}
$$

2.1 Berechnung der Pose

Zur Berechnung der aktuellen Pose (`Pose_comp`) werden in dieser Implementierung Animationsdaten ausgelesen und interpoliert. Diese Daten einer einzelnen Animation bestehen aus einer aufsteigend nach Zeitstempel sortierten Liste von Keyframes. Hierbei enthält jeder Keyframe optional eine lokale Position (XYZ-Vektor), Skalierung (XYZ-Vektor) und Rotation (Quaternion) pro Knochen.

Um die Anzahl an Allokationen zu reduzieren und den CPU-Cache optimal zu nutzen, liegen die Keyframe-Daten (Zeitstempel, Postionen, Skalierungen und Rotationen) einer Animation jeweils in kontinuierlichen Speicherbereichen, sortiert nach Knochen und Zeitstempel vor. Daneben gibt es pro Knochen eine weitere Datenstruktur welche Teile dieser Speicherbereiche referenziert und so die sechs benötigten Listen konstruiert[3].

Zum Abspielen einer Animation muss zuallererst basierend auf dem aktuellen Zeitpunkt pro Knochen und Eigenschaft der Keyframe direkt davor und danach

[3] Jeweils eine Liste von Zeitstempeln und Daten für die drei Teile der Transformation.

bestimmt werden. Hierzu werden die einzelnen Zeitstempel-Listen mittels Interpolationssuche durchsucht.

Eine Interpolationssuche arbeitet ähnlich wie eine Binärsuche über eine Schrittweise Aufteilung des Suchraums in zwei Hälften. Im Gegensatz zu dieser wird der Pivot-Punkt allerdings nicht in der Mitte zwischen dem aktuellen Minimum- und Maximum-Index gewählt sondern basierend auf den dort vorliegenden Werten interpoliert (siehe Abb. 2.1). Hierdurch weist das Suchverfahren bei gleichmäßig verteilten Daten eine Laufzeit von $O(\log \log N)$ auf. Die Worst-Case Laufzeitkomplexität liegt zwar bei $O(N)$, dieser Fall ist allerdings bei den Zeitstempeln einer Animation nicht zu erwarten, weswegen auf die Implementierung einer Alternative wie der quadratische Binärsuche verzichtet wurde. [Sed97]

Des Weiteren speichert das System die jeweils im letzten Frame verwendeten Indices pro Animation und Knochen. Durch einen Start der Suche bei diesen kann somit im durchschnittlichen Fall eine Laufzeit von $O(1)$ erreicht werden, da der tatsächliche Index idR. maximal um 1 vom vorherigen abweicht.

```
1  auto interpolation_search(gsl::span<const float> container, float value, std::int16_t i) -> std::int16_t {
2      auto high = std::int16_t(container.size() - 2);
3      auto low  = std::int16_t(0);
4      i         = std::min(high, std::max(low, i));
5
6      do {
7          if(container[i] > value) {
8              high = i - 1;
9          } else if(container[i + 1] <= value) {
10             low = i + 1;
11         } else {
12             return i;
13         }
14
15         auto new_i = low + ((value - container[low]) * (high - low)) / (container[high] - container[low]);
16         i = static_cast<std::int16_t>(std::min(float(high), new_i));
17     } while(high > low);
18
19     return low;
20 }
```

Abb. 2.1: C++-Code der zur Bestimmung der relevanten Keyframes verwendeten Interpolationssuche.

Sobald die beiden relevanten Keyframes identifiziert wurden, werden ihre Transformationen mit dem aus ihren Zeitstempeln abgeleiteten Interpolationsgewicht[4] linear interpoliert und in der `Pose_comp` gespeichert.

Wenn der aktuelle Zeitstempel außerhalb des von den Keyframes abgedeckten Zeitraums liegt, ist eine Interpolation in dieser Form allerdings nicht mehr möglich. Für diesen Fall kann in der Animationsdatei eines von drei Verfahren pro Knochen spezifiziert werden:

- Beibehalten der letzten spezifizierten Transformation
- Lineare Extrapolation basierend auf den letzten beiden Keyframes
- Animation wieder beim ersten Keyframe fortsetzen

[4] Die Formel für das Interpolationsgewicht bei aktuellem Zeitpunkt t und vorherigem Keyframe-Zeitpunkt t_i ist $\frac{t-t_i}{t_{i+1}-t_i}$.

2.2 Transformation der Vertices

Um auch komplexe Modelle mit vielen Vertices effizient animieren zu können, erfolgt die Transformation der Vertices auf der GPU. Hierzu werden die berechneten Transformationsmatrizen der einzelnen Knochen in einen Uniform-Buffer geschrieben, der im Vertex-Shader ausgelesen werden kann. Da diese Daten in jedem Frame zwischen der CPU und der GPU transferiert werden müssen, wird hierbei mit kompakteren 3x4-Matrizen gearbeitet. Bei diesen handelt es sich um 4x4-Matrizen, bei denen die letzte konstante Zeile verworfen und eine Transponierung durchgeführt wurde[5].

Für einen möglichst effizenten Transfer werden alle Transformationsmatrizen in einen zusammenhängenden Buffer geschrieben. Wobei ein Ringbuffer von Uniform-Buffern vorgehalten wird, so dass keine zusätzliche Synchronisation notwendig ist. Für jede in den Buffer geschriebene Animation wird der Start-Offset vermerkt und beim Zeichnen des Modells verwendet um dynamisch den relevanten Teil des Uniform-Buffers zu binden, ohne neue DescriptorSets generieren zu müssen (`VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`). [VK59]

Die Transformation der Vertices im Vertex-Shader erfolgt schließlich einfach über eine gewichtete Summe aller Produkte zwischen dem Vertex und den zuordneten Knochen wie in Abb. 2.2 (Linear Blend Skinning).

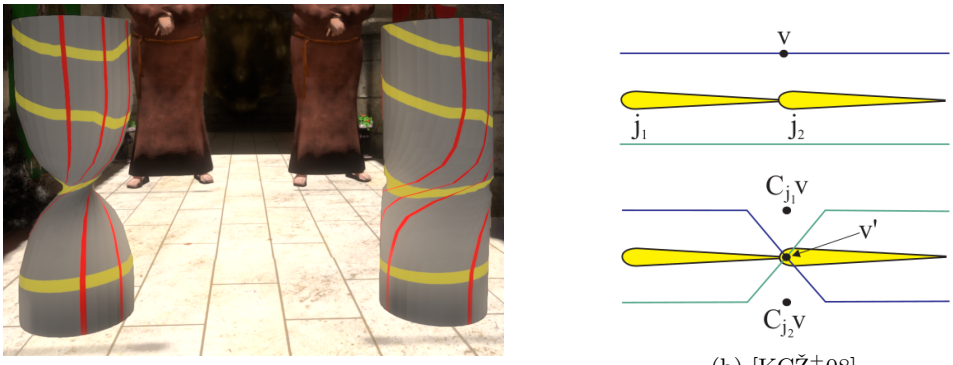
```
1  // ...
2  layout(location = 0) in vec3 position;
3  layout(location = 1) in vec3 normal;
4  layout(location = 2) in vec2 tex_coords;
5  layout(location = 3) in ivec4 bone_ids;
6  layout(location = 4) in vec4 bone_weights;
7
8  layout(set=2, binding = 0, std140) uniform Bone_uniforms {
9      mat3x4 offset[64];
10 } bones;
11 // ...
12
13 void main() {
14     float unused_weight = 1.0 - dot(bone_weights, vec4(1.0));
15
16     vec3 p = (vec4(position, 1.0) * bones.offset[bone_ids[0]]) * bone_weights[0]
17            + (vec4(position, 1.0) * bones.offset[bone_ids[1]]) * bone_weights[1]
18            + (vec4(position, 1.0) * bones.offset[bone_ids[2]]) * bone_weights[2]
19            + (vec4(position, 1.0) * bones.offset[bone_ids[3]]) * bone_weights[3]
20            + position * unused_weight;
21
22     vec3 n = (vec4(normal, 0.0) * bones.offset[bone_ids[0]]) * bone_weights[0]
23            + (vec4(normal, 0.0) * bones.offset[bone_ids[1]]) * bone_weights[1]
24            + (vec4(normal, 0.0) * bones.offset[bone_ids[2]]) * bone_weights[2]
25            + (vec4(normal, 0.0) * bones.offset[bone_ids[3]]) * bone_weights[3]
26            + normal * unused_weight;
27     // ...
28 }
29
```

Abb. 2.2: Ausschnitt aus dem Vertex-Shader mit der Implementierung des Linear-Blend-Skinnings. Es ist zu beachten, dass die Bone-Matrizen transponierte 4x4-Matrizen ohne die letzte Zeile sind, wodurch die Transformation in umgekehrter Reihenfolge erfolgen muss und lediglich einen XYZ-Vektor liefert.

[5] Die Matrizen müssen transponiert werden, da 4x3-Matrizen durch ihre höheren Alignment-Anforderungen genauso viel Speicher verbrauchen wie 4x4-Matrizen. [VK59]

2.3 Dual Quaternion Skinning

Dadurch das bei Linear Blend Skinning lediglich linear zwischen den transformierten Positionen geblendet wird, kommt es unter Umständen zu Artefakten, die sich zumeist durch einen Verlust an Volumen offenbaren. Das auffälligste dieser Artefakte tritt bei Rotationen eines einzelnen Gelenks um über 90 deg auf und sorgt im Extremfall dafür, dass mehrere Vertices auf einen zentralen Punkt zusammenfallen, wie in Abb. 2.3 dargestellt. [KCŽ+08]



(a) (b) [KCŽ+08]

Abb. 2.3: Beispiel für das beschriebene ”candy-wrapper”-Artefakt. Wie in (a) zu sehen ist, kommt es beim Linear Blend Skinning (links) zu einem deutlichen Volumenverlust, der bei Dual Quaternion Skinning (links) nicht zu beobachten ist. In (b) ist das Problem noch einmal in 2D für einen einzelnen Punkt verdeutlicht. Da sich die beiden transformierten Punkte genau auf gegenüberliegenden Seiten befinden, liegt ihr Mittelwert auf dem Knochen (Punktv').

Eine mögliche Lösung für diese Art von Problemen ist das Dual-Quaternion-Skinning Verfahren von Kavan, Collins, Žára und O’Sullivan [KCŽ+07]. Dieses basiert auf einem Trick, der in ähnlicher Form auch beim Blenden von Normalen zum Einsatz kommt und auf einer Normalisierung des geblendeten Ergebnisses beruht (siehe Abb. 2.4). Dazu kommen bei diesem Verfahren Dual-Quaternionen (DQ) zur Repräsentation der Knochen-Transformationen zum Einsatz. Im Shader werden dann nicht die bereits transformierten Punkte, sondern direkt die Transformationen in Form der DQ interpoliert. Die anschließende Normalisierung stellt dabei sicher, dass der mit diesem DQ transformierte Punkt auf der ”höherdimensionalen Kreisbahn” zwischen den Ausgangstransformationen, statt auf direkt Weg zwischen diesen liegt und somit kein Volumen verloren geht. [KCŽ+08]

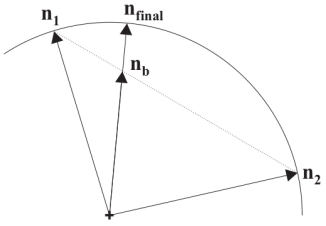


Abb. 2.4: Ein Trick zum Blenden zweier Normalen. Zuerst werden n_1 und n_2 zu n_b geblendet und dieser anschließend zu n_{final} normalisiert. [KCŽ+08]

Bei Dual-Quaternionen handelt es sich um eine Generalisierung der Quaternionen[6] analog zu den dualen Zahlen, bei der jedes Dual-Quaternion aus einem

[6] Ein Zahlenbereich, der die reellen Zahlen erweitert und mit dem sich Rotationen im dreidimensionalen Raum darstellen lassen. Ein Quaternion wird eindeutig durch vier reelle Zahlen beschreiben.

reellen und einem imaginären Quaternion-Anteil besteht. Mit diesen lassen sich neben Rotationen zusätzlich auch Translationen im dreidimensionalen Raum darstellen. [KCŽ+08]

Dual-Quaternionen bieten zwar eine gute Lösung für Rotationen und Translationen, viele Animationen bauen allerdings zusätzlich auch auf Skalierung und Scherung auf um das gewünschte Aussehen zu erreichen. Um auch diese Anteile der Transformationen zu unterstützen, erfolgt die Transformation der Vertices in zwei Schritten. Zuerst wird das Modell in Bind-Pose skaliert/geschert und erst danach die Rotation und Translation angewendet. Da Scherungen deutlich seltener notwendig sind und mehr Speicher benötigen würden, wurden in dieser Implementierung allerdings lediglich Skalierungen beachtet. [KCŽ+08]

Die notwendigen Anpassungen am Quellcode sind minimal und beschränken sich lediglich auf den Vertex-Shader zur Transformationen und den Code zum Transfer der Transformationsmatrizen an die GPU. Vor dem eigentliche Transfer müssen die Matrizen in ihre Bestandteile (Rotation, Translation und Skalierung) zerlegt und in ein Dual-Quaternion, sowie eine Skalierung konvertiert werden. Da die konvertierte Transformation kleiner ist als die Matrizen (11 vergl. mit 12 Fließkommazahlen), können diese in den selben Speicherbereich zurückgeschrieben werden, ohne das andere Teile des Codes geändert werden müssen. [KCŽ+08]

Im Vertex-Shader (siehe Abb. 2.6) wird zuerst zwischen den einzelnen zugeordneten Transformationen eines Vertex geblendet und das resultierende Dual-Quaternion wieder normalisiert. Anschließend erfolgt zuerst die Skalierung und danach die Rotation und Translation. [KCŽ+08]

Besondere Rücksicht muss hierbei auf die Dual-Quaternionen eines Vertex genommen werden, da diese für ein korrektes lineares Blenden das selbe Vorzeichen aufweisen müssen. Da auch für Dual-Quaternionen gilt, dass q und $-q$ die selbe Rotation beschreiben (Antipodality), werden diese in einem ersten Schritt analysiert und gegebenenfalls invertiert. [KCŽ+08]

Insgesamt wirken die Ergebnisse des Dual-Quaternion-Skinnings deutlich realistischer, bei einer ähnlichen Komplexität des Quellcodes und kaum messbar höheren Laufzeit. Allerdings kommt es durch den Erhalt des Ausgangsvolumens in einigen Fällen auch zu ggf. unerwünschten Effekten. So kommt es bei starken Rotationen wie in Abb. 2.5 zu einer Ausdehnung auf der gegenüberliegenden Seite. Es gibt Ansätze um dieses Problem zu reduzieren, da diese aber mit einer höheren Laufzeitbelastung und komplexeren Vertex-Transformation einher gehen, wurde hier auf eine Implementierung verzichtet. [KCŽ+08]

Durch die geringen Unterschiede im Quellcode zwischen den beiden Skinning-Verfahren, ist es problemlos möglich weiterhin beide Verfahren im Renderer zu unterstützen und ggf. in problematischen Fällen das jeweils bessere Verfahren zu wählen.

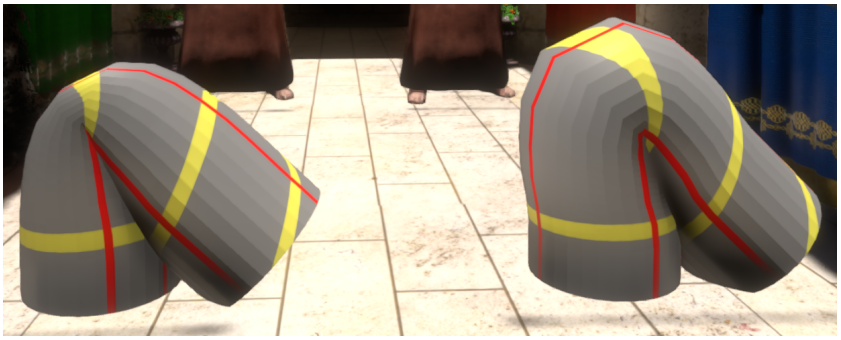


Abb. 2.5: In Folge der Volumenerhaltung kommt es bei starken Rotationen u.U. zu Ausbeulungen wie in dieser Abbildung rechts zu sehen.

```
1       // ...
2       layout(location = 0) in vec3 position;
3       layout(location = 1) in vec3 normal;
4       layout(location = 2) in vec2 tex_coords;
5       layout(location = 3) in ivec4 bone_ids;
6       layout(location = 4) in vec4 bone_weights;
7
8       layout(set=2, binding = 0, std140) uniform Bone_uniforms {
9           mat3x4 offset[64];
10      } bones;
11      // ...
12
13      vec3 transform_position(vec3 p, mat3x4 dq) {
14          p *= dq[2].xyz;
15          return p +
16              2 * cross(dq[0].xyz, cross(dq[0].xyz, p) + dq[0].w*p) +
17              2 * (dq[0].w * dq[1].xyz - dq[1].w * dq[0].xyz +
18          cross(dq[0].xyz, dq[1].xyz));
19      }
20
21      vec3 transform_normal(vec3 n, mat3x4 dq) {
22          return n + 2.0 * cross(dq[0].xyz, cross(dq[0].xyz, n) + dq[0].w * n);
23      }
24
25      void main() {
26          float unused_weight = 1.0 - dot(bone_weights, vec4(1.0));
27          mat3x4 identity_dqs = mat3x4(vec4(1,0,0,0), vec4(0,0,0,0), vec4(1,1,1,1));
28
29          mat3x4[] dq = mat3x4[](
30              bones.offset[bone_ids[0]],
31              bones.offset[bone_ids[1]],
32              bones.offset[bone_ids[2]],
33              bones.offset[bone_ids[3]]
34          );
35
36          // antipodality handling
37          for(uint i=1; i<=3; i++) {
38              if (dot(dq[0][0], dq[i][0]) < 0.0) {
39                  dq[i][0] *= -1.0;
40                  dq[i][1] *= -1.0;
41              }
42          }
43
44          mat3x4 bone = dq[0] * bone_weights[0]
45                      + dq[1] * bone_weights[1]
46                      + dq[2] * bone_weights[2]
47                      + dq[3] * bone_weights[3]
48                      + identity_dqs * unused_weight;
49
50          float dq_len = length(bone[0]);
51          bone[0] /= dq_len;
52          bone[1] /= dq_len;
53
54          vec3 p = transform_position(position, bone);
55          vec3 n = transform_normal  (normal,   bone);
56          // ...
57      }
58
```

Abb. 2.6: Ausschnitt aus dem Vertex-Shader mit der Implementierung des Dual-Quaternion-Skinnings. Die Knochen-Matrizen enthalten ein Dual-Quaternion in den ersten beiden und eine Skalierung in der dritten Zeilen.

Literaturverzeichnis

[KCŽ+07] Ladislav Kavan, Steven Collins, Jiří Žára und Carol O'Sullivan. Skinning with dual quaternions. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, Seiten 39–46, Seattle, Washington. ACM, 2007. ISBN: 978-1-59593-628-8. DOI: 10.1145/1230100.1230107. URL: http://doi.acm.org/10.1145/1230100.1230107.

[KCŽ+08] Ladislav Kavan, Steven Collins, Jiří Žára und Carol O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Trans. Graph.*, 27(4):105:1–105:23, November 2008. ISSN: 0730-0301. DOI: 10.1145/1409625.1409627. URL: http://doi.acm.org/10.1145/1409625.1409627.

[Rez16] Christof Rezk-Salama. Vorlesung 07 Tool- und Pluginprogrammierung: Animation Workflow, 2016.

[Sed97] Robert Sedgewick. *Algorithms in C: Parts 1-4, Fundamentals, Data Structures, Sorting, and Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd Auflage, 1997. ISBN: 0201314525.

[VK59] The Khronos Vulkan Working Group. Vulkan® 1.0.59 - A Specification, 2017. URL: https://www.khronos.org/registry/vulkan/specs/1.0-extensions/pdf/vkspec.pdf (besucht am 18. 09. 2017).