– Aufgabenstellung–

1 Überblick

Ziel dieser Aufgabe ist es, eine Direct3D-Applikation zu entwickeln, die einen animierten und steuerbaren Avatar darstellt. Das animierfähige 3D-Modell des Avatars ist in Form von Vertex/Index Buffers mit entsprechenden Vertex-Attributen vorgegeben. Das komplette 3D-Modell besteht aus mehreren Teilobjekten (das Skin, die Zähne, die Augen). Die Teilobjekte stehen als einzelne Dreiecksnetze zur Verfügung. Für das Skin-Mesh sind zwei Blendshapes (Expressions), Bone Indices und Skin Weights, sowie ein kinematisches Skelett (Rig) vorgegeben. Rendering und Animation soll mit Hilfe von Shadern umgesetzt werden.

Abbildung 1: Beispiel Posen.

Die Aufgabenstellung teilt sich in zwei miteinander vernetzte Pakete, das Animation/Interaktion-Paket und das Rendering/Shader-Paket.

- Im Animationspaket werden durch Benutzereingaben komplexe Gesichtsausdrücke durch Kombination mehrerer elementarer Transformationen zusammengesetzt.

  Im Anhang der Aufgabenstellung sind die möglichen Transformationen des Modells vorgegeben (sowohl für die Expressions als auch für das kinematische Skelett). Die Skelett-Transformationen bestehen aus einer Hierarchie von Transformationsmatrizen. Die Expressions sind als Vertex- und Normal-Offsets vormodelliert.

- Das Rendering-Paket kümmert sich um die visuelle Darstellung des Avatars. Es besteht aus mehreren Shader-Programmen:
  - Der Vertex-Shader soll aus den Transformationsmatrizen und den Blend-Gewichten die animierte Geometrie berechnen.
  - Der Fragment Shader soll anschließend die Geometrie mit ihren Materialeigenschaften darstellen. Hierzu stehen mehrere Texturen zur Verfügung die beliebig angewandt werden können.
  - Ein Geometry Shader soll auf bestimmte Teile der Geometrie Haare in Form von Liniensegmenten aufbringen.

Die Aufgabenpakete werden in den folgenden Abschnitten genauer beschrieben.

1.1 Bewertung

Die Bewertungskriterien für die Teilaufgaben sind in den entsprechenden Abschnitten aufgeführt. Die Teilung des Projekts in zwei Teilprojekte bedeutet nicht, dass jeder ein Teilprojekt übernehmen soll, vielmehr wünsche ich mir, dass ihr in beiden Teilprojekten einige Aufgaben übernehmt. Einzelne Aufgaben (beispielsweise das OOD) können auch gemeinsam erledigt werden. Für die Abgabe wird neben dem Programmcode und den geforderten UML-Diagrammen ein kurzer, formloser Text erwartet, der die Tastenbelegung Eures Programms erklärt und der darstellt wer welche Aufgaben übernommen hat. Für ein Bestehen mit 4.0 sind mehr als 50% der Teilaufgabe erfolgreich zu bearbeiten.

Prozent der Aufgabenstellung	Note
90–100 %	1.0
85–89 %	1.3
80–84 %	1.7
75–79 %	2.0
70–74 %	2.3
65–69 %	2.7
60–64 %	3.3
55–59 %	3.7
51–55 %	4.0

2 Aufgabenpaket A: Softwaredesign Animation/Interaktion

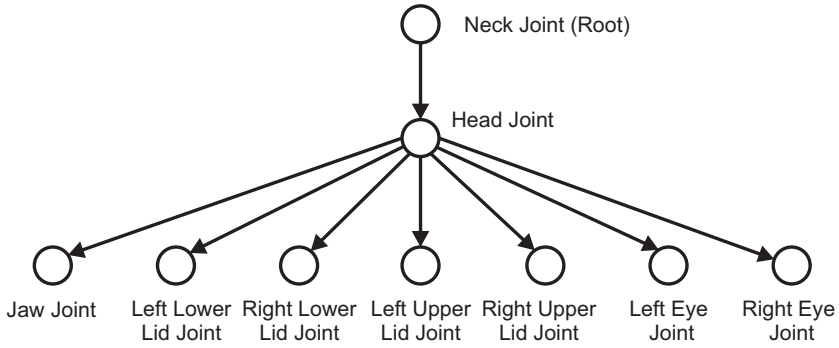

Abbildung 2: Die Transformations-Hierarchie des kinematischen Skeletts.

Das Animation-Paket kümmert sich im Detail um folgende Aufgaben:

A1: Hierarchische Transformationen: Das kinematische Skelett für unsere Anwendung ist in Abbildung 2 dargestellt. Die Aufgabe des Animationspakets ist es, die hierarchischen Transformationen als Objekte im Speicher abzubilden.

In einem kinematischen Skelett besitzt jeder Joint J zwei *lokale* Matrizen: eine lokale *Bind-Pose-Matrix* $\mathbf{B}_J$ und eine lokale Transformationsmatrix $\mathbf{L}_J$:

- Die Bind-Pose-Matrix $\mathbf{B}_J$ ist unveränderlich und gibt die Position und Orientierung des Joints im Bezug auf das Koordinatensystem seines Parent-Joints an (beim Root-Joint im Bezug auf das Weltkoordinatensystem). Die Bind-Pose-Matrizen für die einzelnen Joints sind in der Datei `Transformations.xml` gegeben.
- Die lokale Transformationsmatrix $\mathbf{L}_J$ wird mit der Einheitsmatrix initialisiert und zur Laufzeit kontinuierlich verändert um die aktuelle Pose einzunehmen. Es sollte daher eine Schnittstelle nötig sein, die es erlaubt die Transformationsmatrizen einzelner Joints zu setzen.

Aus diesen Matrizen muß das System zur Laufzeit die Transformationsmatrizen $\mathbf{X}_J$ für die einzelnen Joints berechnen. Diese Berechnung ist in Anhang 1 detailliert beschrieben. Lesen Sie Anhang 1 aufmerksam durch um ein sinnvolles objektorientiertes Konzept für die Transformationen zu erarbeiten. Beachten Sie dabei, dass sich eine lokale Transformationsmatrix $\mathbf{L}_j$ auch aus mehreren Elementarmatrizen $\mathbf{M}$ zusammensetzen kann!

Anmerkung: Für die Animation muß dem Vertex-Shader zur Laufzeit eine Liste der aktuellen Bone-Matrizen $\mathbf{X}_J$ übergeben werden. Es scheint dazu entweder eine einfache Objektverwaltung (Singleton-Pattern) oder ein Visitor Pattern für die Joints nötig zu sein.

A2: Unterstützung von Posen: Die Interaktion soll in der Applikation über Tastendrücken erfolgen. Implementieren Sie einen Keyboard-Callback (die DXUT darf verwendet werden, wie z.B. in Tutorial10). Auf Tastendruck sollen vordefinierte Posen automatisch eingenommen werden. Diese Posen setzen sich aus Parameterwerten für die Winkelstellungen der Gelenke (Anhang 3) und die Blendweights für die Expressions zusammen. Alle in Anhang 3 beschreibenen Elementartransformationen sollen möglich sein.

Die individuellen Tasten und Posen die Sie erstellen möchten, dürfen Sie sich selbst aussuchen (Beispiele finden Sie in Abbildung 1). Wenn eine Taste gedrückt wird, sollte allerdings die Pose nicht sofort, sondern allmählich eingenommen werden, d.h. die Parameterwerte der momentanen Pose sollen langsam in Zielpose überführt werden. Bei erneutem Tastendrücken während dieser Phase soll die Transition zur alten Pose abgebrochen werden und stattdessen die neue Pose angesteuert werden.

Entsprechende Klassen und Objekte für diese Funktionalität sollen entworfen und implementiert werden. Das Framework soll auf einfache Weise um zusätzliche Posen (auf dem vorgegebenen Mesh) erweiterbar sein.

2.1 Bewertungskriterien

Abgegeben werden soll neben dem funktonierenden Quellcode ein **UML Klassendiagramm (OOD) der Teilprojekte A1 und A2**. Ein komplettes UML-Diagramm für die gesamte Anwendung ist nicht erforderlich.

Bei Verwendung der DXUT-Klassen ist es nicht erforderlich diese im UML-Diagramm aufzuführen.
Die Bewertungskriterien für Teilaufgabe A sind wie folgt:

20% Die Teilaufgaben A1 und A2 sind mit UML sinnvoll modelliert. Evtl. sind besondere Designentscheidungen mit Kommentaren zu begründen.

50% Die gefragte Funktionalität ist korrekt implementiert

20% Der C++ Code ist sauber objektorientiert.

10% Die h-Dateien sind kommentiert. In den cpp-Dateien reicht es, die wichtigen Stellen zu kommentieren.
Anmerkung: Ich bin kein Fan von seitenweise Kommentaren, ein Satz der beschreibt, was die Funktion tut, sollte ausreichen!

Bonus Das Programm kompiliert ohne Warnings auf Level 3 (W3).
Anmerkung: Das Präprozessor-Define `_CRT_SECURE_NO_WARNINGS` darf dazu allerdings gesetzt sein, um Warnungen über *deprecated functions* (wie `fopen` etc.) abzuschalten.

3 Aufgabenpaket B : Rendering/Shader

Das Rendering-Paket kümmert sich um die Darstellung der Geometrie in Direc3D 10. Das Rahmenprogramm darf und soll einfach sein (die DXUT-Klassen dürfen verwendet werden). Es sollte allerdings im Gegensatz zu den Tutorials objektorientiert sein, d.h. keine globalen Variablen etc.)

B1: Vertex Shader: Der Vertex-Shader kümmert sich um die Berechnung der aktuellen Pose aus den Transformationsmatrizen und den Blend-Gewichten. Für die Skelettanimation enthält jeder Vertex Skin-Weights und Bone-Indices, die wie folgt angewandt werden sollen. Als uniforme Parameter sollen Blend-Gewichte und ein Array von Transformationsmatrizen übergeben werden.

$$\begin{aligned} \mathbf{v}_{\text{blend}} &= \mathbf{v}_{\text{in}} + b_0\mathbf{d}_0 + b_1\mathbf{d}_1 \\ \mathbf{v}_{\text{out}} &= \sum_{i=0}^{4} w_i \mathbf{X}_{b_i} \mathbf{v}_{\text{blend}} \end{aligned}$$

Dabei sind d_0 und d_1 die Blend-Gewichte zu den beiden Offset-Vektoren $\mathbf{d}_0$ und $\mathbf{d}_1$, w_i sind die vier Skin Weights des Vertex, b_i sind die vier Bone Indices und $\mathbf{X}_j$ ist die Transformationsmatrix des j-ten Joints.

Beachten Sie, das auch die Normalenvektoren auf transformiert werden müssen. Da alle Bone-Matrizen allerdings starre Transformationsmatrizen sind, ist es nicht notwendig die Transponiert-Inverse-Matrix für die Normalenvektoren zu verwenden.

Für die restlichen Meshes, außer dem Skin-Mesh, können Standard-Vertex-Shader zur Transformation verwendet werden.

B2: Fragment Shader: Der Fragment Shader kümmert sich um die Beleuchtungsberechnung und darf frei gestaltet werden. Dazu stehen eine Reihe Texturen zur Verfügung, zusätzliche Texturen dürfen bei Bedarf auch selbst erstellt bzw. besorgt werden.

Für die unterschiedlichen Teile der Geometrie (Skin, Zähne, Augäpfel, Glaskörper der Augen) sind möglicherweise mehrere, unterschiedliche Fragment-Programme notwendig. Die Glaskörper der Augäpfel sollen auf jeden Fall halb-transparent dargestellt werden (Alpha-Blending), egal ob ein eher photorealistisches oder non-photorealistisches Rendering implementiert wird.

B3: Geometry Shader: Der Geometry-Shader soll in einem Multi-Pass Verfahren zum Einsatz kommen und zusätzlich zu der polygonalen Geometrie Haare in Form von Liniensegmenten auf das Skin-Mesh berechnen. Die Haare sollen kontrollierbar auf bestimmte Stellen des Skins aufgebracht werden. Der Geometry-Shader darf frei gestaltet werden sofern er das Ziel erfüllt. Hier einige Vorschläge für eine mögliche Vorgehensweise:

- Der Geometry-Shader (oder auch der Vertex-Shader) liest für jeden Vertex einen Haardichte-Parameter aus einer Textur.
- Anschließend werden auf jedem Dreieck zufällige Punkte bestimmt. Die Anzahl der Punkte hängt dabei von dem Haardichte-Parameter der Eckpunkte ab (Hier kann man den Mittelwert der drei Punkte verwenden). Zufallswerte können vorberechnet werden und dem Shader in Form eines Buffers (siehe Tutorial 11, Implementation Step 2) oder einer Textur übergeben werden.
- Für jeden Zufallspunkt wird ein Liniensegment entlang des Normalenvektors ausgerichtet und ausgegeben.
- Überlegen Sie sich etwas nettes für die visuellen Attribute der Haare. Meist sieht es gut aus wenn der Haaransatz eine andere Farbe hat als die Spitze. Gekrümmte Haare aus mehreren Liniensegmenten sind nicht notwendig, aber cool.
- Da sich die zugrundeliegende Geometrie durch Animation deformiert, erscheint es nicht sinnvoll Stream-Out zu verwenden, sondern die Haare in jedem Frame neu zu erzeugen.

3.1 Bewertungskriterien

Abgegeben werden soll neben dem funktonierenden Quellcode ein FX-File mit den entsprechenden Shadern. UML-Diagramme sind in diesem Paket nicht erforderlich.
Die Bewertungskriterien für Teilaufgabe B sind wie folgt:

30% Vertex-Shader (geforderte Funktionalität ist sinnvoll umgesetzt)

10% Fragment Shader (möglicherweise mehrere für die unterschiedlichen Teilgeometrien)

50% Geometryshader (geforderte Funktionalität ist sinnvoll umgesetzt)

5% Das DirectX-Rahmenprogramm ist sauber objektorientiert und läuft auch im NVPerfHud.

5% Die Fx-Datei ist lesbar und an den wichtigen Stellen kommentiert.

Bonus Die visuelle Darstellung ist gelungen, insbesondere sind die Fragment Shader der einzelnen Meshes aufeinander abgestimmt und ergeben ein überzeugendes Gesamtbild.

Bonus Extrapunkte für die Umsetzung eigener Ideen!

Anhang 1: Geometrieformat

Das komplette 3D Modell des Avatars besteht aus 7 Teilobjekten. Das Skin ist das einzige Teilobjekt das mit Skinning animiert wird. Die anderen Teile werden starr mit einzelnen Joint Matrizen transformiert (als Model-Matrix). Die jeweilige Model-Matrix $\mathbf{X}_J$ ist in der unteren Tabelle angegeben. Die Berechnung der Matrizen steht in Anhang 2.

1. das Skin des Charakters mit Expressions, Bone Indices und Skin Weights
2. die Zahnreihe des Oberkiefers (transformiert mit $\mathbf{X}_{HeadJoint}$)
3. die Zahnreihe des Unterkiefers (transformiert mit $\mathbf{X}_{JawJoint}$)
4. der rechte Augapfel (transformiert mit $\mathbf{X}_{RightEyeJoint}$)
5. der linke Augapfel (transformiert mit $\mathbf{X}_{LeftEyeJoint}$)
6. der Glaskörper des rechten Auges (transformiert mit $\mathbf{X}_{RightEyeJoint}$)
7. der Glaskörper des linken Auges (transformiert mit $\mathbf{X}_{LeftEyeJoint}$)

Es folgt die detaillierte Beschreibung der einzelnen Geometriedateien:

3.2 Skin Mesh

Beschreibung	Dateiname	Dateityp	Größe in Byte
Vertex Buffer	`MarsienneSkin_VB.raw`	float	7872512
Index Buffer	`MarsienneSkin_IB.raw`	int32	367728

Primitive Type: TRIANGLES
Anzahl der Indices: 91932

Semantik	Datentyp	Beschreibung
Position	3x float	Vertex-Position
TexCoord	2x float	Textur-Koordinate
Normal	3x float	Normalenvektor
Bone Index	4x float	4 Indices i in den Array von Bone Matrizen $\mathbf{X}_i$
Skin Weights	4x float	4 Skin Weights s für die 4 Bone Matrizen
Blend Index	3x float	nicht benötigt *(nur aus Kompatibilitätsgründen falls mehr als 3 Offsets modelliert wurden)*
Position Offset 0	3x float	Position Offset für die SSmileËxpression
Normal Offset 0	3x float	Normal Offset für die SSmileËxpression
Position Offset 1	3x float	Position Offset für die FrownËxpression
Normal Offset 1	3x float	Normal Offset für die FrownËxpression

Die Bone Indices sind den Joints wie folgt zugeordnet:

0 = NeckJoint
1 = HeadJoint
2 = LtUpperLidJoint
3 = RtUpperLidJoint
4 = LtLowerLidJoint
5 = RtLowerLidJoint
6 = JawJoint

Die Bone-Matrizen für LtEyeJoint und RtEyeJoint werden für die Skin-Animation nicht benötigt und transformieren nur die Augäpfel.

Für dieses Mesh wird als Model-Matrix die Einheitsmatrix gesetzt (Eine Viewing-Matrix kann natürlich trotzdem noch angewendet werden!) Die Bone Matrizen $\mathbf{X}_J$ werden im Vertex-Shader für das Skinning angewendet.

3.3 Zahnreihe des Oberkiefers

Beschreibung	Dateiname	Dateityp	Größe in Byte
Vertex Buffer	`UpperJaw_VB.raw`	float	21088
Index Buffer	`UpperJaw_IB.raw`	int32	10200

Primitive Type: TRIANGLES
Anzahl der Indices: 2550

Semantik	Datentyp	Beschreibung
Position	3x float	Vertex-Position
TexCoord	2x float	Textur-Koordinate
Normal	3x float	Normalenvektor

Für dieses Mesh wird als Model-Matrix die Transformation $\mathbf{X}_{HeadJoint}$ gesetzt (siehe Anhang 2)

3.4 Zahnreihe des Unterkiefers

Beschreibung	Dateiname	Dateityp	Größe in Byte
Vertex Buffer	`LowerJaw_VB.raw`	float	13728
Index Buffer	`LowerJaw_IB.raw`	int32	6240

Primitive Type: TRIANGLES
Anzahl der Indices: 1560

Semantik	Datentyp	Beschreibung
Position	3x float	Vertex-Position
TexCoord	2x float	Textur-Koordinate
Normal	3x float	Normalenvektor

Für dieses Mesh wird als Model-Matrix die Transformation $\mathbf{X}_{JawJoint}$ gesetzt (siehe Anhang 2)

3.5 Rechtes und linkes Auge

Beschreibung	Dateiname	Dateityp	Größe in Byte
Vertex Buffer	`RtEyeBall_smtt_VB.raw`	float	123680
Index Buffer	`RtEyeBall_smtt_IB.raw`	int32	22860
Vertex Buffer	`LtEyeBall_smtt_VB.raw`	float	123680
Index Buffer	`LtEyeBall_smtt_IB.raw`	int32	22860

Primitive Type: TRIANGLES
Anzahl der Indices: 5715

Semantik	Datentyp	Beschreibung
Position	3x float	Vertex-Position
TexCoord	2x float	Textur-Koordinate
Normal	3x float	Normalenvektor

Für das rechte Auge wird als Model-Matrix die Transformation $\mathbf{X}_{RightEyeJoint}$ gesetzt (siehe Anhang 2) Für das linke Auge wird als Model-Matrix die Transformation $\mathbf{X}_{LeftEyeJoint}$ gesetzt (siehe Anhang 2)

3.6 Glaskörper für rechtes und linkes Auge

Beschreibung	Dateiname	Dateityp	Größe in Byte
Vertex Buffer	`RtEyeBall_smtt_VB.raw`	float	35200
Index Buffer	`RtEyeBall_smtt_IB.raw`	int32	6480
Vertex Buffer	`LtEyeBall_smtt_VB.raw`	float	35200
Index Buffer	`LtEyeBall_smtt_IB.raw`	int32	6480

Primitive Type: TRIANGLES
Anzahl der Indices: 1620

Semantik	Datentyp	Beschreibung
Position	3x float	Vertex-Position
TexCoord	2x float	Textur-Koordinate
Normal	3x float	Normalenvektor

Für das rechte Auge wird als Model-Matrix die Transformation $\mathbf{X}_{RightEyeJoint}$ gesetzt (siehe Anhang 2) Für das linke Auge wird als Model-Matrix die Transformation $\mathbf{X}_{LeftEyeJoint}$ gesetzt (siehe Anhang 2)

Anhang 2: Berechnung der Bone-Matrizen

Die Aufgabe des Animationspakets ist es, die hierarchische Transformationen als Objekte im Speicher abzubilden. Für die Animation muß dem Vertex-Shader zur Laufzeit eine Liste aktueller Bone-Matrizen übergeben werden, die berechnet werden müssen. Die Bone-Matrizen sind globale Matrizen (im Weltkoordinatensystem) die wie folgt berechnet werden können:

1. Zunächst muß rekursiv aus den lokalen Bind-Pose-Matrizen $\mathbf{B}_J$ eine globale World-Bind-Pose-Matrix $\mathbf{W}_J$ berechnet werden:

$$\begin{aligned} \mathbf{W}_J &= \mathbf{W}_{parent(J)} \cdot \mathbf{B}_J \qquad \text{(Rekursionsschritt)} \\ \mathbf{W}_{root} &= \mathbf{B}_{root} \qquad \text{(Rekursionsanker)} \end{aligned}$$

   In der Rekursion werden alle Bind-Pose-Matrizen der Parents aufmultipliziert: Als Beispiel für den JawJoint in Abbildung 2:

$$\begin{aligned} \mathbf{W}_{JawJoint} &= \mathbf{W}_{HeadJoint} \cdot \mathbf{B}_{JawJoint} \\ &= \mathbf{W}_{NeckJoint} \cdot \mathbf{B}_{HeadJoint} \cdot \mathbf{B}_{JawJoint} \\ &= \mathbf{B}_{NeckJoint} \cdot \mathbf{B}_{HeadJoint} \cdot \mathbf{B}_{JawJoint} \end{aligned}$$

2. Auf die gleiche rekursive Weise muß die globale Transformationsmatrix $\mathbf{T}_J$, die die lokalen Transformationen $\mathbf{L}_J$ berücksichtig, rekursiv berechnet werden

$$\begin{aligned} \mathbf{T}_J &= \mathbf{T}_{parent(J)} \cdot \mathbf{B}_J \cdot \mathbf{L}_J \qquad \text{(Rekursionsschritt)} \\ \mathbf{T}_{root} &= \mathbf{B}_{root} \cdot \mathbf{L}_{root} \qquad \text{(Rekursionsanker)} \end{aligned}$$

3. Um schließlich die globale Transformationsmatrix $\mathbf{X}_j$ zu berechnen wird die World-Bind-Pose-Matrix $\mathbf{W}_J$ invertiert (da sie starr ist, reicht auch eine Transponierung!) und mit der globale Transformationsmatrix $\mathbf{T}_J$ multipliziert:

$$\mathbf{X}_J = \mathbf{T}_J \cdot \mathbf{W}_J^{-1} \tag{1}$$

Anhang 3: Die lokalen Transformationen

Als elementare Transformationen sollen die folgenden Aktionen umgesetzt werden können:

3.7 Bewegung des Kopfes: Turn, Bend, Bow

Der Kopf soll sich nach links und rechts drehen können (turn). Außerdem soll er nicken (bow) und sich schräg stellen können (bend).

$$\begin{aligned} \mathbf{L}_{\text{HeadJoint}} &= \mathbf{M}_{\text{turn}} * \mathbf{M}_{\text{bow}}\mathbf{M}_{\text{bend}} \\ \mathbf{M}_{\text{turn}} &: \text{Rotation um (lokale) x-Achse} \\ \mathbf{M}_{\text{bow}} &: \text{Rotation um (lokale) y-Achse} \\ \mathbf{M}_{\text{bend}} &: \text{Rotation um (lokale) z-Achse} \end{aligned}$$

3.8 Bewegung des Unterkiefers: Open, Jut

Der Unterkiefer soll sich öffnen können (open). Außerdem sollte er sich ein kleines bisschen seitlich drehen können (jut)

$$\begin{aligned} \mathbf{L}_{\text{JawJoint}} &= \mathbf{M}_{\text{open}} * \mathbf{M}_{\text{jut}} \\ \mathbf{M}_{\text{open}} &: \text{Rotation um (lokale) z-Achse} \\ \mathbf{M}_{\text{jut}} &: \text{Rotation um (lokale) y-Achse} \end{aligned} \tag{2}$$

3.9 Bewegung der Augen

Für die beiden Augen werden die Transformationsmatrizen auf die gleiche Weise berechnet. Die Augen sollten sich allerdings unabhängig voneinander bewegen können (z.B. Schielen)

$$\begin{aligned} \mathbf{L}_{\text{LtEyeJoint}} &= \mathbf{M}_{\text{lookUD}} * \mathbf{M}_{\text{lookLR}} \\ \mathbf{L}_{\text{RtEyeJoint}} &= \mathbf{M}_{\text{lookUD}} * \mathbf{M}_{\text{lookLR}} \\ \mathbf{M}_{\text{lookLR}} &: \text{Rotation um (lokale) z-Achse} \\ \mathbf{M}_{\text{lookUD}} &: \text{Rotation um (lokale) y-Achse} \end{aligned} \tag{3}$$

3.10 Bewegung der Augenlider

Die Augenlider sollen sich nur nach oben und unten bewegen können:

$$\begin{aligned}
\mathbf{L}_{\text{LtUpperLidJoint}} &: \text{Rotation um (lokale) z-Achse} \\
\mathbf{L}_{\text{LtLowerLidJoint}} &: \text{Rotation um (lokale) z-Achse} \\
\mathbf{L}_{\text{RtUpperLidJoint}} &: \text{Rotation um (lokale) z-Achse} \\
\mathbf{L}_{\text{RtLowerLidJoint}} &: \text{Rotation um (lokale) z-Achse}
\end{aligned} \tag{4}$$

3.11 Gesichtsausdrücke *Smile* und *Frown*

Die beiden Gesichtsausdrücke *Smile* und *Frown* werden über Blend-Shapes realisiert (wie in Tutorial 09)