Plattformunabhängige Modellanimation mit Hilfe von Vertexshadern

Studienarbeit

Vorgelegt von
Arne Claus

Institut für Computervisualistik
Arbeitsgruppe Computergraphik

Betreuer und Prüfer:
Prof. Dr.-Ing. Stefan Müller

Mai 2004

Inhaltsverzeichnis

1. Aufgabenstellung und Ziel der Arbeit 2

2. Skins and Bones 3

2.1. Konzept 3

2.2. Mathematische Grundlagen 5

2.3. Programmiertechnischer Ansatz 7

3. Maya 8

3.1. Der Dependency Graph 8

3.2. Animationskurven 9

3.3. MEL 11

4. Der Exporter 13

4.1. Das S3D Dateiformat 13

4.2. Der Exporter im Detail 16

4.2.1. Die Hauptfunktion getData() 17

4.2.2. Funktionen zur Ermittlung der Bonedaten 18

4.2.3. Funktionen zur Ermittlung der Polygondaten 19

4.2.4. Export der Animationskurven 21

4.2.5. Ausgabe der Daten 22

5. Cherubim 23

5.1. Interner Aufbau 23

5.2. Der Szenengraph 25

5.3. Anpassung des Szenegraphen 27

5.3.1. Shadercores 27

5.3.2. Modellcores und preShader 28

5.3.3. Animierte Transformationsknoten 30

5.4. Animationen 30

5.5. Der Importer 31

6. Demonstration 32

6.1. Export aus Maya 32

6.2. Import und Darstellung in Cherubim 34

7. Fazit und Ausblick 36

Abbildungsverzeichnis

Abb. 2.1.a Psygnosis, „Ecstatica“, 1994. 3
Abb. 2.1.b Eidos, „Tomb Raider III“, 1998. 3
Abb. 2.1.c Eidos, „Tomb Raider IV“, 2003. 3
Abb. 2.1.d Rigid-Body (links) vs. Skins and Bones 4
Abb. 2.2.a Faltung bei Rigid-Skining 6
Abb. 2.3.a Klassendiagramm eines möglichen programmiertechnischen Ansatzes 7
Abb. 3.1.a Ausschnitt aus dem Dependency Graph einer Maya Szene 9
Abb. 3.2.a Screenshot des Animationseditors 10
Abb. 4.1.a Eine einfache Animation als Szenengraph 13
Abb. 4.1.b Blockschema S3D 15
Abb. 4.2.a Screenshot des Exporter GUI 16
Abb. 4.2.4.a Hermite und Bezier Splines 21
Abb. 4.2.4.b Bezeichnungen 21
Abb. 5.1.a Cherubim Module 24
Abb. 5.2.a Übergang von Volume Graph zu Draw Graph 26
Abb. 5.3.2. Verwendung der Joint Orient Matrix 29
Abb. 6.1.a Modellierung der Demoszene 32
Abb. 6.1.a Animationskurven der gelben Kugel 33
Abb. 6.2.a Cherubim Demoszene unter MacOS X und Windows XP 34

1. Aufgabenstellung und Ziel der Arbeit

Diese Studienarbeit entstand im Rahmen des freien Spiele-Projektes Seraphim [1]. Dieses Spiel wird dem Genre des „Massive Multiplayer Online Role Play Game“ (MMORPG) zugeordnet, es handelt sich somit um eine Client / Server Anwendung. Diese Arbeit befasst sich jedoch ausschließlich mit dem Client bzw. dem Client-Framework „Cherubim“.
Zu Beginn dieser Studienarbeit bestand Cherubim aus einem rudimentären Szenengraphen sowie einem Eventmanager und einigen mathematischen Basisklassen. Aufgabe war es, den Szenegraphen so zu erweitern, dass das Animieren von Polygonobjekten mit Hilfe eines durch Vertexshader beschleunigten Skins and Bones-Systems möglich wurde.

Die Schwerpunkte dieser Arbeit lagen somit auf der Theorie und Implementierung eines Skins and Bones-Systems, der Implementierung eines Exporters für Vertex- und Bonedaten aus einer bestehenden kommerziellen Modellierungslösung, der Erweiterung Cherubims um Laderoutinen für die exportierten Daten sowie der Darstellung dieser Daten unter Anwendung des bestehenden Szenegraphs.
Die zusätzliche Prämisse der Plattformunabhängigkeit resultierte aus dem Umstand, dass eine der Hauptanforderungen Seraphims besagt: Cherubim muss auf verschiedenen Plattformen lauffähig sein. Während das bestehende System durch die Verwendung von plattformunabhängigen Bibliotheken bereits diesen Anforderungen entsprach, fielen Problematiken wie verschiedene Grafik-Hardware bzw. das Implementieren von Vertex-Shadern und eine mögliche Endian-Konvertierung im Dateiformat in den Bereich dieser Arbeit.

Die hier verwendeten Theorien zu Skins and Bones bauen auf der vorausgegangenen Studienarbeit von Florian Schulze [Schulze03] sowie den in von Möller und Haines vorgestellten Techniken des Vertexblending [MöllerHaines02] auf. Die Implementierung der Vertexshader für diese System – insbesondere unter der Verwendung der Sprache Cg [4] – basiert zu einem großen Teil auf den in „The Cg Tutorial“ [FernandoKilgard03] diskutierten Verfahren.

2. Skins and Bones

2.1. Konzept

„Skins and Bones“ ist eine von vielen Bezeichnungen für ein Verfahren, das oft auch als „Vertexblending“ bezeichnet wird [MöllerHaines02, FernandoKilgard03]. Ich ziehe jedoch die Bezeichnung „Skins and Bones“ vor, da sie das Konzept meiner Meinung nach am besten umschreibt.

Das Prinzip besteht darin, eine einfache Transformationshierarchie, das Skelett – bestehend aus Knochen („Bones“) – mit einer Polygonhaut („Skin“) zu verbinden und diese wiederum durch das Skelett über einen vorgegebenen Zeitraum hinweg oder durch Benutzereingaben zu animieren. Die Ursprünge dieses Verfahrens sind nicht genau geklärt, nach Möller und Haines lässt es sich jedoch bis in die 80er Jahre zurück datieren [MöllerHaines02].

Der Vorteil dieser Methode gegenüber der meist in älteren Spielen wie Tomb Raider oder Ecstatica (s.u.) verwendeten Rigid-Body Transformation besteht darin, dass die Animationen wesentlich realistischer und vor allem zusammenhängender erscheinen („Steckpuppeneffekt“ bei Rigid-Body Transformationen).

Abb. 2.1.a Psygnosis, „Ecstatica“, 1994.
Ein Versuch die Rigid-Body Transformationen durch Ellipsen realistischer erscheinen zu lassen.

Abb. 2.1.b Eidos, „Tomb Raider III“, 1998.
Probleme der Rigid-Body Transformation an den Arm- und Beingelenken.

Abb. 2.1.c Eidos, „Tomb Raider IV“, 2003.
Bessere Übergänge an den Gelenken durch Skins and Bones.

Bei der klassischen Rigid-Body Transformation wird der Körper einer Figur in mehrere Teile unterteilt, die z.B. durch eine Szenengraph-Hierarchie animiert werden. So wird z.B. der Unterarm einer Figur an den Oberarm „angehängt“, was dazu führt, dass der Unterarm automatisch mitbewegt wird wenn sich die Position oder Rotation des Oberarms ändert. Bewegt man nun aber den Unterarm alleine, so verweilt der Oberarm in seiner vorher eingenommenen Position, wird also nicht beeinflusst.
Der bei dieser Form von Animationen oft auftretende „Steckpuppeneffekt“ ist darauf zurückzuführen, dass sich die einzeln animierten Teile des Körpers an den Gelenken überschneiden und so bei extremen Bewegungen und/oder unsauberer Modellierung Löcher und unerwünschte Kanten entstehen lassen.
Dieses Problem tritt bei Skins and Bones nicht auf, da hier eine zusammenhängende Polygonoberfläche indirekt animiert wird.

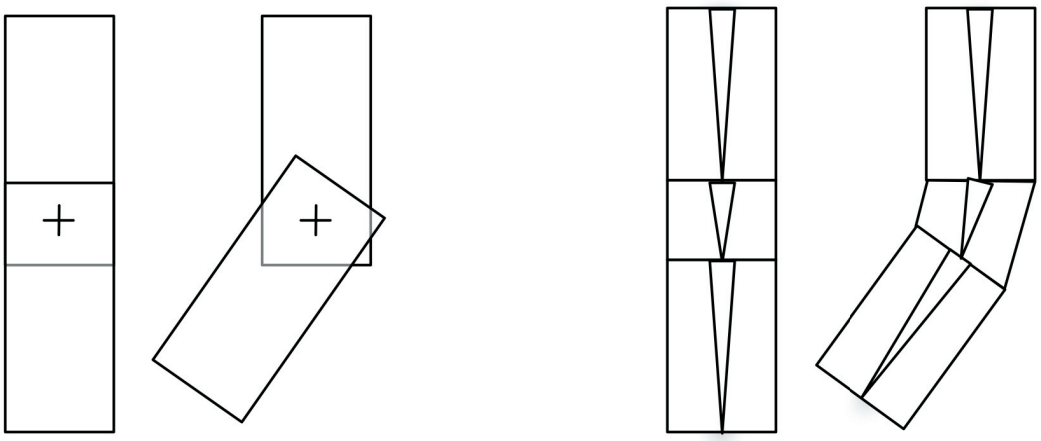


Abb. 2.1.d Rigid-Body (links) vs. Skins and Bones

Das Prinzip der Rigid-Body Animation wird bei Skins and Bones zwar übernommen, aber nur auf die i.d.R. nicht sichtbaren, dem Modell zugeordneten Bones angewendet. Die einzelnen Vertices der Polygonoberfläche werden nun in Relation zu dem Skelett bewegt, wodurch die Bewegungen der Bones auf die Vertices übertragen werden.
Weist man hier jedem Vertex genau einen Bone zu, so ergeben sich ähnliche Effekte wie bei der Rigid-Body Animation, ohne dass jedoch Löcher entstehen können. Gelöst wird das Problem schließlich dadurch, dass ein Vertex von mehreren, unterschiedlich gewichteten Bones beeinflusst werden kann. Dieses Verfahren wird inzwischen in fast allen aktuellen Spielen wie Halflife 2 von Valve oder Doom 3 von ID Software verwendet.

2.2. Mathematische Grundlagen

Die bei Skins and Bones verwendeten Vertices sollten in der Ausgangspose des zugehörigen Objektes vorliegen. Die Koordinaten sollten in Weltkoordinaten angegeben werden, da die Skelette in Maya relativ zu diesen und nicht den Objektkooridnaten angelegt werden.
Die verwendete Transformationshierarchie besteht i.d.R. aus Knoten mit einer Translationsmatrix T und einer durch z.B. Quaternione angegebenen Rotation R, ebenfalls in Matrixform. Skalierungen können verwendet werden, allerdings wird dadurch die Berechnung einer inversen Matrix z.B. für die Transformation der Normalen erschwert. Da Skalierungen für Skins and Bones nicht zwingend notwendig sind und in Spielen eher selten verwendet werden, betrachtet diese Arbeit daher nur den einfachen Fall ohne Skalierung.
Jedem Knoten der Transformationshierarchie lässt sich jeweils ein Elternknoten zuordnen, so dass der Pfad zum Ursprung der Transformationshierarchie immer eindeutig ist.
Die Transformationsmatrix eines Bones B_i zur Zeit t ergibt sich demnach wie folgt:

$$B_i(t) = \prod_{j=0}^{i} R(t)_j * T(t)_j$$

Die Bindung der Polygonoberfläche an dieses System kann nun auf zwei unterschiedlichen Wegen durchgeführt werden:
Beim sogenannten Rigid-Skinning [6] wird zunächst jedem Vertex auf der Polygonoberfläche genau ein Bone zugeordnet, wodurch sich die Position eines transformierten Vertex q (Weltkoordinatensystem) aus dem ursprünglichen Vertex p (Objektkoordinatensystem) wie folgt errechnen lässt:

$$q(t) = B(t) * M^{-1} * p$$

Bei der hier verwendeten Matrix M handelt es sich um die Transformation, die das Skelett in das Weltkoordinatensystem überführt. Um also einen Vertex aus dem Weltkoordinatensystem in Skelettkoordinaten zu transformieren, muss die Inverse dieser Matrix aufmultipliziert werden. M entspricht z.B. der Matrix des ersten Bones in Maya [6] wenn sich das Skelett in seiner Anfangspose befindet.
Bei dieser Inversen Matrix ist zu beachten, dass im Falle von Rotationen nach Euler

sich auch die Reihenfolge der Rotationen umkehrt. Dies ist wichtig, da es sich anbietet die Inverse durch entgegengesetzte Rotationen anstatt durch z.B. Determinanten zu ermitteln.

Rigid-Skinning eignet sich nur und vor allem für Polygone mit geringer Auflösung. Wählt man ein Polygon mit einer höheren Auflösung, so treten unter Umständen unerwünschte Faltungseffekte im Bereich der Gelenke auf.

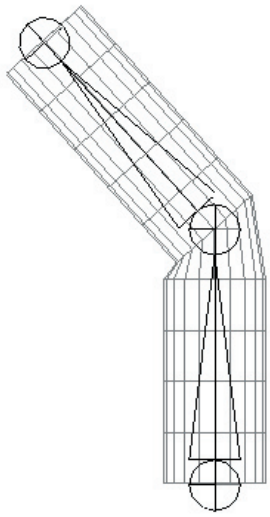


Abb. 2.2.a Faltung bei Rigid-Skining

Für höher aufgelöste Polygone eignet sich daher das sogenannte Soft-Skinning [6] wesentlich besser. Hierbei werden jedem Vertex beliebieg viele Bones unter Verwendung von Gewichtungen zugewiesen, deren Summe „1“ ergibt.
Die Position des transformierten Vertex q errechnet sich danach wie folgt:

$$q(t) = \sum_{i=0}^{n} w_i * B_i(t) * M^{-1} * p \qquad \text{wobei} \quad \sum_{i=0}^{n} w_i = 1 \text{ und } w_i > 0 \qquad \text{[MöllerHaines02]}$$

Auf die korrekte Ermittlung der Gewichtungen werde ich an dieser Stelle nicht eingehen, da diese Aufgabe in den Bereich der Modellierungssoftware fällt. Angaben hierzu wurden bereits in früheren Arbeiten gemacht [Schulze03].

2.3. Programmiertechnischer Ansatz

Ein Ansatz zur programmiertechnischen Umsetzung dieses Konzeptes muss im Wesentlichen zwei Dinge umfassen: Die Skelett-Hierarchie sowie die an einzelne Elemente dieser Hierarchie gebundenen Vertices.

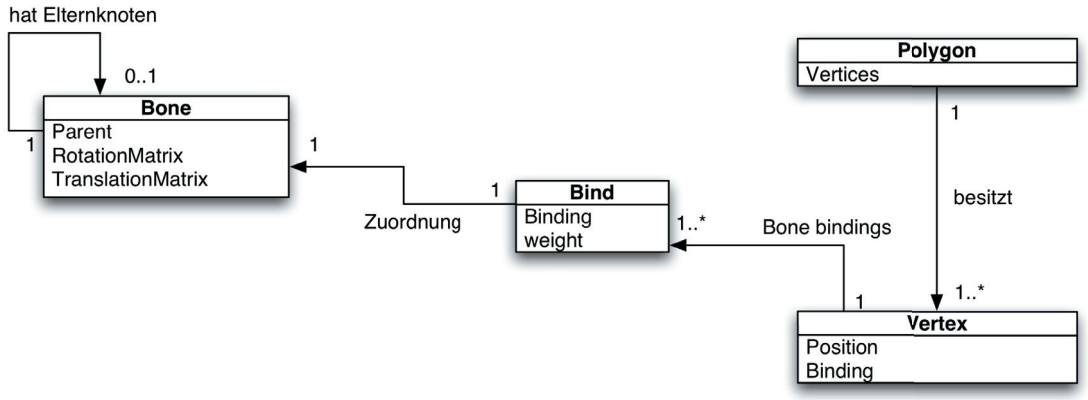


Abb. 2.3.a Klassendiagramm eines möglichen programmiertechnischen Ansatzes

Dieser Ansatz erfasst sowohl das Rigid- als auch das Soft-Skinning-Verfahren, da Rigid-Skinning als eine Untergruppe des Soft-Skinnings angesehen werden kann. In diesem Falle wäre nur eine mit „1“ gewichtete Verbindung zwischen Vertex und Bone gegeben.

Dieser Ansatz berücksichtigt jedoch weder die Animierbarkeit der einzelnen Elemente noch die Einschränkungen, die sich z.B. durch die Verwendung der GPU als Vertexprozessor ergeben. Dies hängt damit zusammen, dass Vertexshader nur eine bestimmte Anzahl an Parametern entgegennehmen können, wodurch sich die maximale Anzahl der möglichen Bindungen und auch der verfügbaren Bones reduziert. Im Falle des später verwendeten OpenGL ARB Vertex-Program [2] sind beispielsweise 96 vierelementige floatingpoint Parameter erlaubt, wodurch die Anzahl der verwendeten 4x4 Matrizen auf maximal 24 sinkt. Nutzt man die Tatsache, dass bei den verwendeten Matrizen keine Projektion vorliegt und somit die letzte Zeile einer Bone-Matrix immer gleich ist (0,0,0,1), kann man 4x3 Matrizen verwenden, womit sich die Anzahl der übergebenen Matrizen auf maximal 32 erhöht.
Da nun aber keine Parameter für die Indizierung und Gewichtung der Bones mehr zur Verfügung stehen, muss z.B. auf die Texturkoordinaten als Parameter ausgewichen werden. Somit beschränkt sich das Verfahren in diesem Fall sogar auf 4 bis maximal 64 Bindungen pro Vertex bei Verwendung aller 32 OpenGL Texturkoordinaten.
Diese Zahlen ändern sich natürlich je nach verwendetem Vertexshader, und so kann

z.B. ein Cg-Fx Shader wesentlich mehr Matrizen fassen.

Die Berechnung der einzelnen Vertices erfordert zudem Funktionen zur Multiplikation von mindestens dreielementigen Vektoren mit 4x4 bzw. 4x3 Matrizen auf der CPU. Diese werden vor allem dann benötigt, wenn nicht nur der Vertexshader, sondern auch die CPU als möglicher Vertexprocessor verwendet werden soll sowie um einmalige Vorberechnungen auf der CPU ausführen zu können.

Die Umsetzung dieses Ansatzes unter Berücksichtigung der oben genannten Problematiken wird in Punkt 5 genauer beschrieben.

3. Maya

3.1. Der Dependency Graph

Die Entscheidung, Maya als Modellierungslösung zu verwenden, wurde aus zwei verschiedenen Gründen getroffen. Zum einen ist Maya im Gegensatz zu z.B. 3ds-max [7] auf verschiedenen Plattformen verfügbar und passt somit in das Konzept der Plattformunabhängigkeit Cherubims. Zum anderen ist Maya aufgrund seiner internen Struktur sehr gut dazu geeignet, Szenen bzw. Objekte für einen Szenengaph zu exportieren.
Maya verwendet intern einen sogenannten Dependency Graph [Gould03]. Dieser Graph wird zwar wie eine Pipeline verwendet, kann jedoch aufgrund seiner Merkmale an einigen Stellen mit einem Szenegraphen verglichen werden. So findet man hier z.B. Szenengraph-typische Konzepte wie die Trennung von Transformations- und Objektdaten wieder.

Alle in Maya verwendeten Objekte sind im Dependency Graph abgelegt, wobei der Graph in Teilbereiche wie geometrische Objekte und Texturen unterteilt wird. Der Begriff des „Objekts“ ist daher eher im programmiertechnischen als im modellierungstechnischen Sinne zu sehen, da hier neben Polygonen auch Texturen, Animationskurven und andere Elemente des Systems abgelegt werden.

Das Konzept der Pipeline wird durch folgendes Beispiel deutlich:
Man beginnt z.B. mit einem Polygon oder einem anderem soliden Objekt. Auf die Daten dieses Objekts können nun verschiedene Knoten wie zum Beispiel eine Free

Form Deformation (FFD) zugreifen, indem sie die Objektdaten als Eingangsdaten verwenden. Die FFD würde nun ihrerseits wieder Daten als Ausgangsdaten zur Verfügung stellen, die dann wiederum von einem Rendering-Knoten verwendet werden könnten. Dieser Knoten könnte dann ein Bild liefern, welches wiederum in einen post-Processing Knoten wandern könnte, usw.

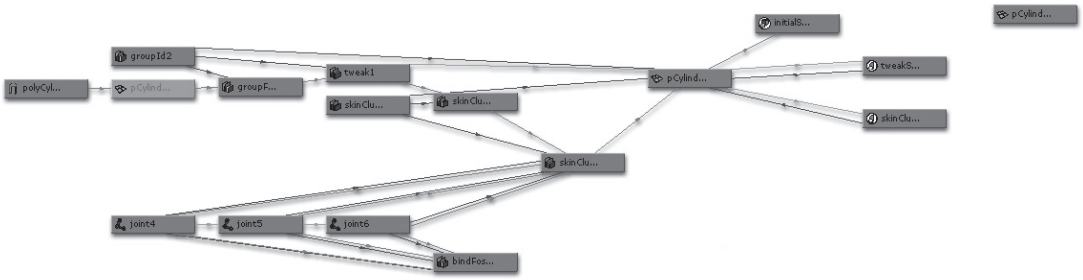


Abb. 3.1.a Ausschnitt aus dem Dependency Graph einer Maya Szene

Der Vorteil dieses Systems besteht darin, dass sich die Pipeline an beliebiger Stelle verändert und erweitern lässt. Außerdem kann auf Objektdaten jederzeit zugegriffen werden, ein Objekt kann also z.B. an einem beliebigen Punkt seiner Verarbeitungskette exportiert oder visuell dargestellt werden.

3.2. Animationskurven

Alle Knoten im Dependency Graph besitzen ein oder mehrere Attribute, um die Verarbeitung der Daten in den Knoten steuern zu können. Alle diese Attribute können in Maya animiert werden. Hierzu wird zu einer bestimmten Zeit ein Animatons-Key für das Attribut, bestehend aus dem Wert des Attributs zu der gewählten Zeit und zwei Tangenten, festgelegt.

Zur Interpolation zwischen den einzelnen Animations-Keys verwendet Maya eine leicht modifizierte Variante der Hermite Splines [Giberson01]. Diese zeichnen sich gegenüber Bezier Splines durch eine schnellere Berechnung und ein unterschiedliches Tangentenmodell aus.

Maya unterschiedet hierbei zusätzlich zwischen gewichteten und ungewichteten Kurven. Ungewichtete Kurven haben eine normierte Tangente der Länge „1“, während gewichtete Kurven eine Tangente beliebiger Länge besitzen können.

Der Wertebereich der Animationskurven ergibt sich aus der Zeit der Animation auf der x-Achse und dem Wertebereichs des Attributs auf der y-Achse.
Bei dem in Abb.3.2.a gezeigten Animationseditor ist darauf zu achten, dass x- und y-Achse nicht zwingend gleich skaliert sind, weswegen ein Tangentenwinkel von 45° nicht unbedingt auch als solcher dargestellt wird.
Zudem können Keys außerhalb der festgelegten Animationszeit definiert werden oder Kurven nicht den vollständigen Animationsbereich ausnutzen. Endet oder beginnt eine Animationskurve vor Ende bzw. nach Beginn der Animationszeit, so sind ihre y-Werte im angrenzenden Bereich durch die Attribute pre- und post-Infinity definiert [Gould03].

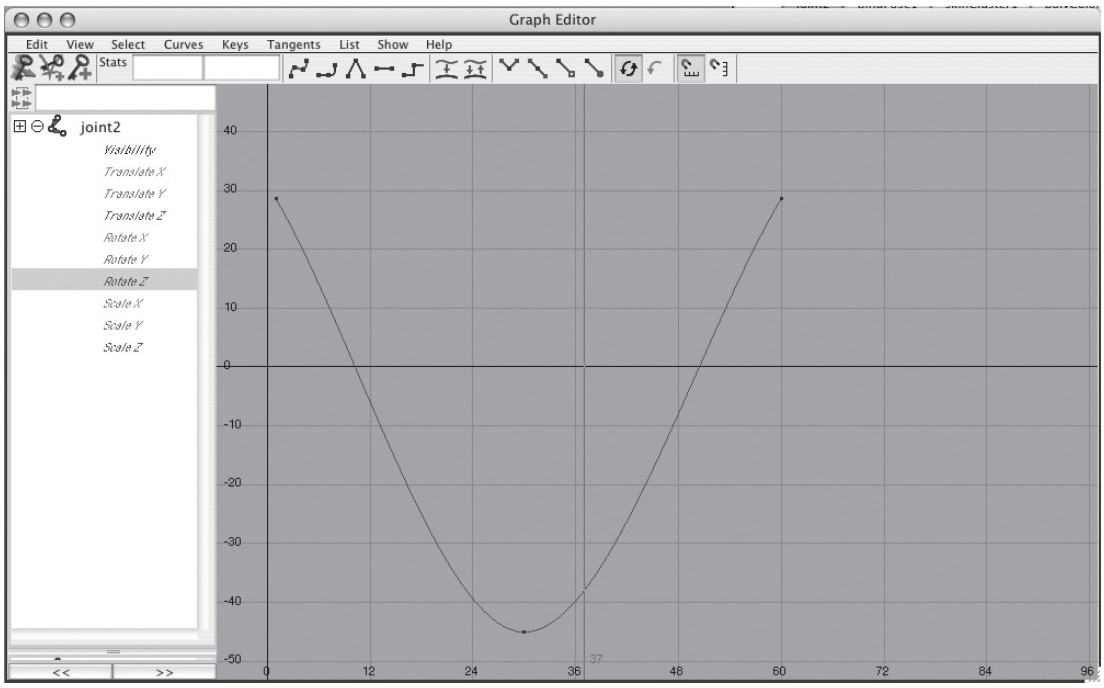


Abb. 3.2.a Screenshot des Animationseditors

3.3. MEL

MEL ist eine Abkürzung für „Maya Embedded Language“, dem Namen für die Maya eigenen Skripting-Sprache. Das User-Interface von Maya ist in MEL geschrieben, welches den Leistungsumfang der Sprache sehr gut verdeutlicht.
MEL hat Zugriff auf alle im Dependency Graph abgelegten Knoten und deren Attribute, was auch der Grund für die Entscheidung von MEL als Sprache für den Exporter war. Alternativ hätte noch die Möglichkeit bestanden, über die Maya C++ API auf die Szenendaten zuzugreifen; von dieser Möglichkeit wurde im Rahmen dieser Arbeit jedoch abgesehen, um möglichst keinen Einschränkung zu unterliegen.

Da der in MEL geschriebene Exporter in Punkt 4 im Detail erläutert wird, möchte ich an dieser Stelle auf ein paar Eigenarten der MEL Syntax hinweisen.

Der MEL Syntax ist an den der Sprache C angelehnt, hat jedoch einige signifikante Unterschiede vorzuweisen.
Es gibt in MEL insgesamt 3 Datentypen – `int`, `float` und `string` – sowie die Möglichkeit, Arrays, Vektoren und Matrizen zu nutzen. Im Gegensatz zu C gibt es jedoch keine Möglichkeit, Structs oder Klassen anzulegen, dafür ist es möglich, Arrays während der Laufzeit zu erweitern. Zudem sei erwähnt, dass `float` dem Wertebereich des C-Datentyps `double` entspricht und somit 64 Bit und nicht 32 Bit im Speicher bzw. einer Datei belegt.
Die eben genannten Datentypen können bei der Deklaration einer Variablen verwendet werden, MEL erlaubt jedoch auch zusätzlich automatische Variablen, deren Typ nicht angegeben werden muss und automatisch festgestellt wird.
Des Weiteren unterscheidet Maya genau wie C zwischen lokalen und globalen Variablen, allerdings mit dem Unterschied, dass globale Variablen als solche mit dem reservierten Wort „`global`“ gekennzeichnet werden müssen und nicht automatisch in Funktionen zur Verfügung stehen.

Der Zugriff auf die Attribute eines Knotens im Dependency Graph erfolgt wie in C-Structs üblich mit einem Punkt. Es sei hier allerdings erwähnt, dass es sich hier nicht um einen Elementzugriff im herkömmlichen Sinne handelt, sondern vielmehr um einen Bezeichner für dieses Attribut, bzw. einen Teil seines Pfad- bzw. Variablennamens.

So sind folgende Konstrukte in MEL möglich und äquivalent zueinander:

1. `getAttr joint1.tx;`
2. `string $nodename = „joint1“;`
   `getAttr $nodename.tx;`
3. `string $attribute = „joint1.tx“;`
   `getAttr $attribute;`

Der Befehl `getAttr` liefert hierbei den Wert eines Attributs zurück.

Funktionen können in MEL auf verschiedene Arten aufgerufen werden. Während die meisten mitgelieferten Funktionen in einer Kommandozeilen-ähnlichen Form aufgerufen werden, werden z.B. eigene Funktionen wie in C üblich verwendet:

```
getAttr –size pCubeShape1.pt;
meineGetAttrFunktion(„pCubeShape1“,“pt“);
```

Des Weiteren muss beachtet werden, dass Funktionen, die in einer Variablenzuweisung verwendet werden, in umgedrehte Hochkommata gesetzt werden müssen.

```
int $ptSize = `getAttr –size pCubeShape1.pt`;
```

Viele der mitgelieferten Funktionen unterscheiden zudem zwischen verschiedenen Modi, wie z.B. dem create und query Modus. So kann es vorkommen, dass eine Funktion sowohl zum Verändern als auch zum Abfragen von Attributen verwendet werden kann. Der query-Modus wird durch den Parameter –q eingeschaltet.

4. Der Exporter

4.1. Das S3D Dateiformat

Die Implementation des Exporters erforderte zunächst die Entwicklung eines Dateiformats, das diverse an das System gestellte Anforderungen erfüllt.
Zunächst musste das Format auch dann lesbar bleiben, wenn die Datei auf einem anderen System als dem aktuell verwendeten erstellt wurde. Probleme treten an dieser Stelle immer dann auf, wenn Systeme mit unterschiedlicher Endian-Formatierung (Anordnung der Bytes in zusammenhängenden Daten) verwendet werden, was z.B. bei einem Transfer zwischen Intel basierten PCs und Apple Macintoshs der Fall ist.
Eine gängige Vorgehensweise ist hier das Verwenden einer sogenannten „Magic-Number“, einem mindestens 2 Byte großen Datenfeld, dessen Wert bekannt ist und daher beim Einlesen der Datei mit dem Originalwert verglichen werden kann. Da als Name des Datenformats S3D (Seraphim Model 3D) gewählt wurde, bot sich hier die 4 Byte große Zahl 1397568324, bzw. der entsprechende C-String „SM3D“ an. Ein positiver Nebeneffekt dieser Darstellung ist die eindeutige Identifizierung der Datei als S3D Datei.
Die nächste Anforderung war es, in der Datei mehrere in ihren Transformationen animierbare Polygonobjekte zu sichern. Indirekt wurde damit vorausgesetzt, dass ein Objekt mitsamt seiner Transformationshierarchie abgespeichert wird, um z.B. einen um die Sonne kreisenden Planeten mit einem um den Planeten kreisenden Mond zu ermöglichen.

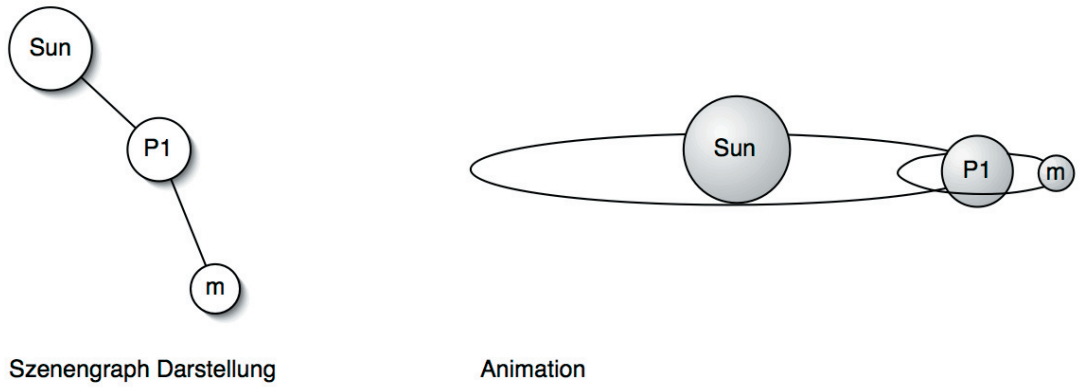


Abb. 4.1.a Eine einfache Animation als Szenengraph

Hieraus ergaben sich also eine Reihe von Problemen: Es musste möglich sein, zwischen verschiedenen evtl. hierarchisch verketteten Objekten mit eigenen Transformationen zu unterscheiden, und alle wichtigen Informationen zu diesen Objekten (wie Vertices, Normalen, Texturkoordinaten, usw.) zu sichern.
Diese Probleme ließen sich durch Blockbildung und Indizierung der Daten lösen. So wurden z.B. alle Vertices in einer Liste zusammengefasst, wobei jeder Vertex aus einer Position und Farbe besteht, also einen eigenen Datenblock bildet. Auf diese Weise erhält jeder Vertex durch seine Position in der übergeordneten Liste eine fortlaufende Nummer, die z.B. von einem Face genutzt werden kann, um dessen Eckpunkte zu definieren. Auf diese Weise kann ein Vertex auch in mehreren Faces verwendet werden, ohne neu definiert werden zu müssen, was die Datenmenge um ein Vielfaches reduziert.
Die Transformationshierarchie wird durch ein ähnliches Verfahren erfasst. Jedes Objekt verweist unter anderem auf eine Transformation, diese wiederum auf eine andere übergeordnete Transformation verweisen kann. Entsprechend sind Transformationen und Objekte ebenfalls in Listen gesichert.
Die letzten und wichtigsten Anforderungen im Rahmen dieser Arbeit waren es, Skelette abzuspeichern, diese an die Polygondaten zu binden sowie die Transformationen der Bones und Transformationsknoten zu animieren. Dies stellte durch das bereits angesprochene Verfahren der Blockbildung und Indizierung jedoch kein größeres Problem mehr dar.
Da Bones als Transformationen angesehen werden können, war es möglich, das bereits verwendete Schema anzuwenden, allerdings musste hier eine eigene Liste angelegt werden, da Bones auch zusätzliche bzw. andere Elemente als Transformationen enthalten. Die Bindung eines Bones an einen Vertex wurde durch 8 neue Parameter in den einzelnen Vertexblöcken erreicht, womit sich die Anzahl der möglichen Bindungen pro Vertex auf 4 reduziert (4 Index- / weight-Paare).
Diese Einschränkung wurde in Hinsicht auf den später verwendeten Vertexshader vorgenommen.
Letztendlich ergab sich das in Abb.4.1.b dargestellte Blockschema.

Die Anordnung der Blöcke wurde so gewählt, dass beim Ladevorgang keine Probleme entstehen und dieser möglichst schnell ablaufen kann.
So wurde außerdem darauf geachtet, dass indizierte Elemente möglichst vollständig vorhanden sind, bevor auf sie verwiesen wird. Aus diesem Grund müssen beispielsweise die Vertices vor den Faces definiert werden.

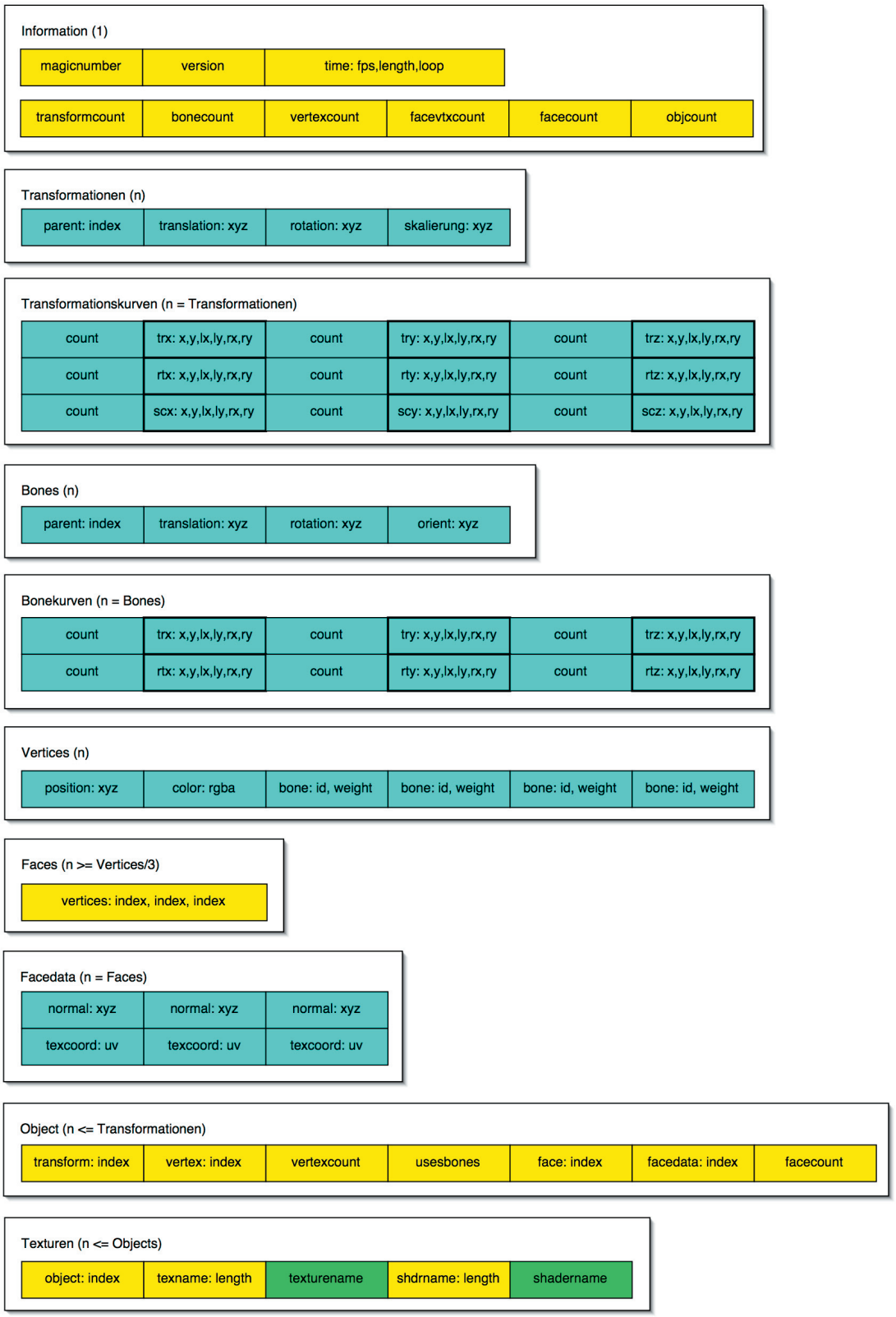


Abb. 4.1.b Blockschema S3D

4.2. Der Exporter im Detail

Der Ablauf des Exporters kann in 5 Schritte unterteilt werden.
Im ersten Schritt werden verschiedene Szeneninformationen wie Länge der Animation, Geschwindigkeit, etc. gesammelt. Als Grundlage werden hier die Einstellungen der Maya-Preview gewählt, da diese Vorschau am ehesten der Darstellung im Spiel entspricht und vor allem in Echtzeit zur Verfügung steht.
In den folgenden drei Schritten werden nun Informationen zu Bones, Polygonen und Objekten gesammelt, die im letzten Vorgang zusammen mit den Szeneninformationen in eine Datei geschrieben werden. Zur Steuerung des Ablaufs wird ein einfaches Userinterface genutzt.
Der Quellcode des Exporters liegt auf der im Anhang dieser Studienarbeit zu findenden CD bei.

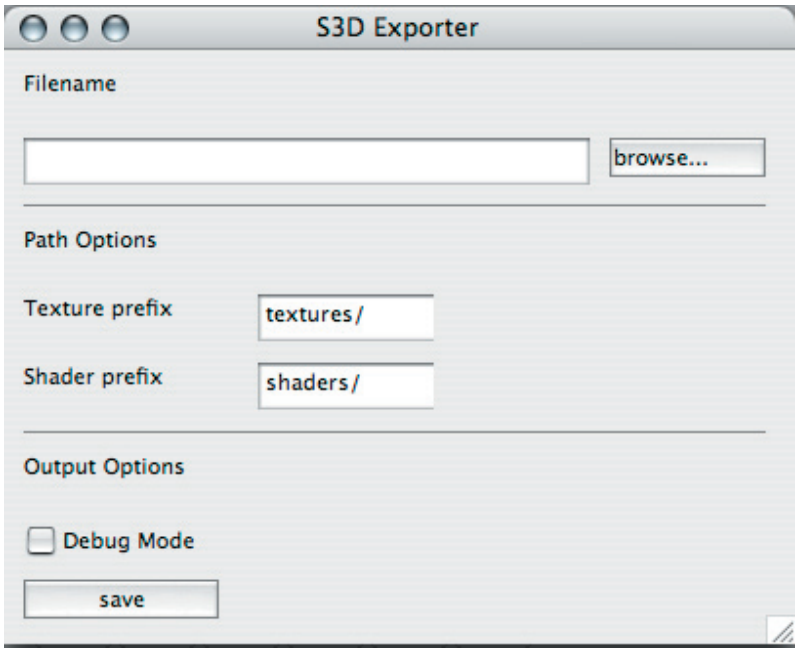


Abb. 4.2.a Screenshot des Exporter GUI

4.2.1. Die Hauptfunktion `getData()`

Die Funktion `getData()` dient der Initialisierung der Puffervariablen, der Schaffung bestimmter für den Exporter notwendiger Vorraussetzungen in Maya sowie dem Aufruf der einzelnen Unterfunktionen zur Informationsgewinnung in der richtigen Reihenfolge.

Die Initialisierung der Variablen findet zu Beginn statt und sorgt dafür, dass alle verwendeten Arrays wirklich leer sind. Da die verwendeten Arrays global definiert sind, bleiben ihre Inhalte nach Ablauf des Skriptes bestehen, was bei wiederholter Anwendung zu unnötig großen Dateien mit unerwünschten Daten führen würde.

Im nächsten Schritt wird die Zeit der Szene auf Frame 1 gestellt und die für den Export wichtigen Knoten im Dependency Graph gesucht.
Das Zurücksetzen der Zeit ist notwendig, da die verwendeten MEL-Funktionen immer nur den aktuellen Stand der Szene zurückliefern. Würde das Zurücksetzen nicht erfolgen, so könnten Fehler bei der Wiedergabe der Animation in Cherubim auftreten, da dort davon ausgegangen wird, dass Animationen bei Frame 1 beginnen. Alternativ könnte man hier sichern, in welchem Frame sich die Animation gerade befindet, dies würde den Import allerdings verkomplizieren.
Die gesuchten Knoten werden durch die beiden Befehle

```
string $sNodes[] = `ls -v -as`;
string $sTNodes[] = `ls -type shadingEngine`;
```

ermittelt. In `$sNodes` befinden sich nun die Namen aller sichtbaren Knoten (-v) auf der obersten Hierarchie-Ebene (-as), während sich in `$sTNodes` alle Textur-relevanten Knoten befinden.
Mit Hilfe dieser beiden Variablen wird nun zuerst nach Bones, dann nach Transformationen und Geometrie und letztendlich nach Texturen gesucht.

4.2.2. Funktionen zur Ermittlung der Bonedaten

Die Ermittlung der Bonedaten wird auf zwei Funktionen verteilt.
Die Funktion `traverseBones()` durchläuft die Skelett-Hierarchien rekursiv beginnend beim ersten Bone. Da `traverseBones()` die selben Eingabedaten wie `traverseMeshes()` erhält (`$sNode`, siehe 4.2.1.) muss zunächst durch die Funktion `objectType` ermittelt werden, ob es sich beim aktuell verwendeten Knoten um einen Bone, also den Maya-Typ „`joint`“ handelt. Anschließend wird Gebrauch von der Funktion `listRelatives` gemacht, um die Nachfolger des gefundenen Bones zu ermitteln.
Bevor nun die Daten durch `saveBone()` gesichert werden, wird der Bone durch den Befehl `gotoBindPose` in seine Ursprungspose zurückgesetzt. Dies ist wichtig, um später die verbundenen Polygondaten korrekt exportieren zu können, da diese ja bereits an das Skelett gebunden sind.

Die Funktion `saveBone()` arbeitet die Daten des übergebenen Bones ab und sichert diese in den entsprechenden globalen Arrays. Ein Bone verweist hierbei i.d.R. auf einen anderen Bone, der bereits gesichert sein muss. Da die Skelette der Szene jedoch immer vom ersten Bone beginnend durchlaufen werden und ein Bone immer nur einen übergeordneten Bone besitzen kann, kommt es hier zu keinerlei Problemen. Der erste Bone eines Skelettes verweist auf den Index –1.
Es sei zudem erwähnt, dass der Joint Orient des letzten Bones eines Skelettes nicht, d.h. als „0,0,0“ exportiert wird (siehe 5.3.2.).

4.2.3. Funktionen zur Ermittlung der Polygondaten

Polygondaten werden in einem ähnlichen Verfahren ermittelt, wie zuvor die Daten der Bones. In diesem Falle arbeitet die Funktion `traverseMeshes()` alle übrigen Knoten aus `$sNodes` rekursiv ab. Dies ist auch hier notwendig, um verschachtelte Hierarchien wie in 4.1. beschrieben zu exportieren.
Wichtig ist hier nun die Funktion `saveMesh()`.
Zunächst sichert `saveMesh()` die dem Shape-Knoten (Polygon) zugehörigen Transformationen ähnlich der in 4.2.2. beschriebenen Methode. Anschließend werden die Polygondaten ausgewertet.
An dieser Stelle ergeben sich nun einige Probleme.
Wird in Maya Skinning auf ein Objekt angewendet, so wird es „eingefroren“, was bedeutet, dass eine Kopie des Objekts in seinem aktuellen Zustand, also mit allen Transformationen und Änderungen auf Vertexebene angelegt wird. Es existieren also zwei gleiche Objekte, von denen allerdings nur das geskinnte exportiert werden darf, da es als einziges die Bindungsinformationen enthält.
Hier nutzt der Exporter die Tatsache, dass Maya das ursprüngliche Objekt mit dem Zusatz „Orig“ versieht, in der Hoffnung, dass der Benutzer nichts an diesem Namen verändert bzw. keine Objekte mit dem Postfix „Orig“ verwendet.
Das nächste Problem besteht darin, dass Maya Vertices in zwei Komponenten unterteilt – der ursprünglichen Position (`<name>.vt`) und einem Offset (`<name>.pt`).
Dieser Offset wird immer dann benutzt, wenn Vertices der Grundform durch den Benutzer verschoben wurden. Das Offset-Array ist nun aber nicht zwingend ebenso lang wie das Positions-Array, was aber – wenn man es weiß – kein Problem darstellt, da Maya auch Zugriffe auf nicht existente Indizes erlaubt und in diesem Falle 0 zurückliefert. Position und Offset können also gefahrlos addiert werden.
Das größte Problem entstand jedoch dadurch, dass viele Informationen in Maya nicht direkt durch Attribute zugänglich sind, sondern nur durch Funktionen ermittelt werden können. Diese Funktionen sind zwar in der Maya-Hilfe dokumentiert, aber nur selten mit den entsprechenden Knoten verlinkt. Hier eine kurze Übersicht der verwendeten Funktionen:

`polyColorPerVertex`	Liefert die Farbe zu einem Vertex mit dem Parameter –rgb sowie den Alphakanal mit dem Parameter –a. Man sollte hier darauf achten, dass eine nicht gesetzte Farbe der Farbe 0,0,0,0 entspricht, also nicht sichtbar ist.
`skinPercent`	Liefert die Verbindung zwischen einem Vertex und einem Bone sowie die Gewichtung der Verbindung.
`findRelatedSkinCluster`	Liefert den Skincluster zu einem Objekt. Der Skincluster muss als Parameter an skinPercent übergeben werden.
`polyNormalPerVertex`	Liefert die Normale zu einem Vertex in einem Face.
`polyEditUV`	Liefert die UV-Koordinaten zu einem Vertex in einer UV-Map.
`polyListComponentConversion`	Konvertiert eine Objektkomponente. Kann z.B. verwendet werden, um die UV-Map zu einem Face zu ermitteln.
`polyTriangulate`	Erstellt ein neues, aus dreieckigen Faces bestehendes Objekt.

Einige dieser Funktionen erfordern weitere Erklärungen, um sie verwenden zu können.

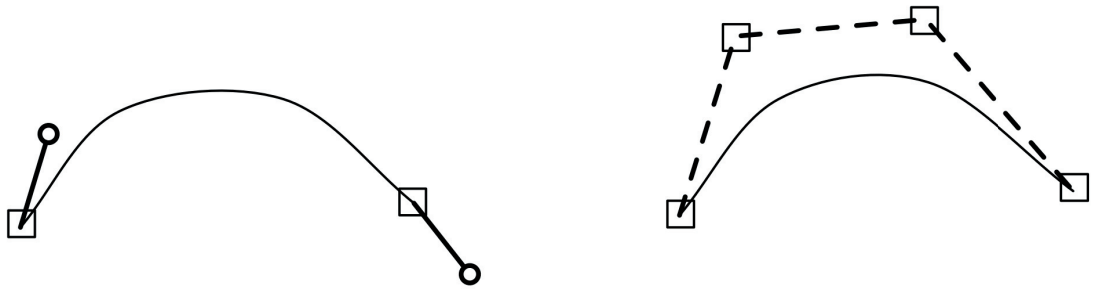
Zunächst wäre `findRelatedSkinCluster` zu erwähnen. Dieser Befehl funktioniert nur dann, wenn das Objekt mittels Soft-Skinning an ein Skelett gebunden wurde. Rigid-Skinning Bindungen verwenden keinen SkinCluster und können somit nicht über diesen Befehl ermittelt werden.

Der zweite wichtige Befehl ist `polyTriangulate`.

In Maya werden nicht alle Objekte mit dreieckigen, sondern i.d.R. mit viereckigen Faces angelegt. Da Cherubim jedoch mit dreieckigen Faces arbeitet, müssen die Objekte zunächst in dieses Format konvertiert werden. Der Befehl legt hierfür ein neues Objekt an, welches jedoch nach Verwendung durch den Exporter wieder gelöscht wird. Diese Funktion verarbeitet ausschließlich Polygonobjekte, weswegen NURBS-Objekte nicht exportiert werden können.

4.2.4. Export der Animationskurven

Wie bereits in 3.2. erwähnt, verwendet Maya zur Verarbeitung der Animationskurven eine leicht abgewandelte Version der Hermite-Splines. Da Cherubim jedoch Bezier-Splines verwendet, müssen die Animationskurven entsprechend konvertiert werden. Problematisch bei diesem Schritt ist, dass der von Maya zur Verfügung gestellte Befehl keyTangent nicht zwingend die erforderlichen und korrekten Werte liefert.
KeyTangent liefert alle Informationen zu den Tangenten eines Keyframes wie Eingangswinkel, Ausgangswinkel und Gewichtung, sowie beide Werte verrechnet als x- und y-Position relativ zu den Koordinaten des Keys. Letztere eigenen sich jedoch leider nicht direkt im Zusammenhang mit Bezierkurven.
Zwar kann man die zwei inneren Punkte eines kubischen Bezier-Splines als Tangenten zum Start- bzw. Endpunkt betrachten, diese wären aber im Vergleich zu den Tangenten eines Hermite-Splines nur ein Drittel so lang. Außerdem zeigen die Tangenten der Hermite-Splines immer in „Laufrichtung“ der Kurve, während die Tangenten der Bezier-Splines auch in die entgegengesetzte Richtung weisen.



Hermite Splines Bezier Splines

Abb. 4.2.4.a Hermite und Bezier Splines

Die Umrechnung erfolgt deshalb durch Verrechnung der Winkel mit dem Gewicht der Tangente:

$p_x = \cos(|\alpha|) * |t_1| / 3$

$p_y = \sin(\alpha) * |t_1| / 3$

$q_x = -\cos(|\beta|) * |t_2| / 3$

$q_y = -\sin(\beta) * |t_2| / 3$

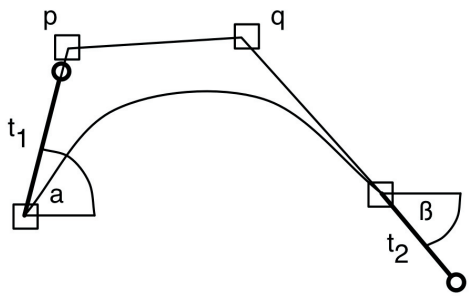


Abb. 4.2.4.b Bezeichnungen

Alle Winkel müssen im Bogenmaß vorliegen, d.h. im Falle der von keyTangent gelieferten Werte umgerechnet werden.

4.2.5. Ausgabe der Daten

Bei der Ausgabe muss nun nicht mehr viel beachtet werden. Die Daten liegen in den verschiedenen Arrays bereits so vor, dass sie später als Blöcke wieder gelesen werden können und werden daher direkt in die Datei geschrieben.
Zu beachten ist hier lediglich das Verhalten der Funktion fopen. Hier gibt es genau wie in C die Möglichkeit, Binär- und Textdateien zu schreiben. Da in dem hier vorliegenden Fall eine Binärdatei geschrieben wird, muss darauf geachtet werden, dass fopen mit dem Parameter „wb“ verwendet wird. Dies ist vor allem auf Unix-Systemen wie Mac OS X wichtig, da sonst die Zahlen 10 und 13 vertauscht werden, was vor allem bei der späteren Auswertung der Indizes ein größeres Problem darstellen kann.

5. Cherubim

5.1. Interner Aufbau

Cherubim besteht im Wesentlichen aus sechs verschiedenen Modulen: Subsystem, Input/Output, Eventmanagement, Netzwerk, Szene und Audio. Jedes Modul besteht aus einer Sammlung von Klassen, die i.d.R. von einer Managerklasse aus angesprochen werden. Diese Managerklassen sind durch den Eventmanager verbunden und können so untereinander kommunizieren.
Ein Beispiel hierfür wäre z.B. das Beenden des Clients über den Druck der Taste „Escape“:

1. Input/Output fängt die Tastatureingabe ab und ermittelt anhand der eingelesenen Konfiguration, dass es sich hierbei um den Befehl „exit“ handelt.
2. Input/Output wirft ein „SystemQuit“ Event.
3. Der Eventmanager verteilt dieses Event an alle registrierten Manager.
4. Subsystem erhält das Event, wertet es aus und beendet alle anderen Systeme.

Durch den Eventmanager können die einzelnen Module also vollkommen unabhängig voneinander arbeiten, ohne die Möglichkeit der Kommunikation untereinander auszuschließen.
Trotz der wichtigen Rolle des Eventmanagers stellt das Subsystem (Main) den eigentlichen Kern Cherubims dar und ist damit den anderen Modulen übergeordnet. Hier befinden sich Initialisierung und Destruktion des Gesamtsystems, also auch aller anderen Module wie dem Eventmanager und der Gameloop.

Das folgende Diagramm (Abb.5.1.a) verdeutlicht die Zusammenarbeit der einzelnen Subsysteme:

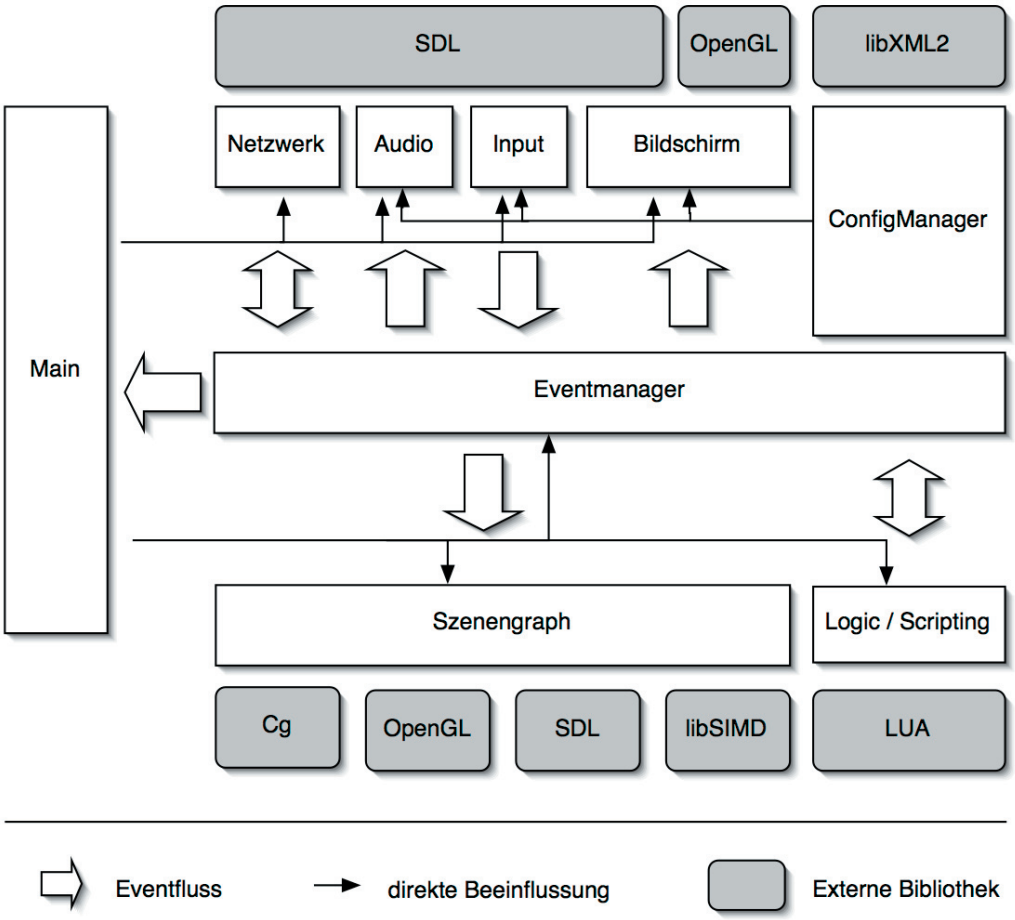


Abb. 5.1.a Cherubim Module

Das Subsystem und somit auch der Gameloop laufen im Haupt-Thread des Programmes, andere Modulmanager wie z.B. Netzwerk und Audio sind auf eigene Threads ausgelagert, wodurch sich der Gameloop auf einige wenige Funktionen reduziert:

```
while(!shutdown)
      Keyboard / Mouse abfragen
      Events für Subsysteme abfragen und auswerten
      Szenengraph -> Szene zeichnen
```

Die Keyboard/Mouse-Abfrage kann hier leider nicht ausgelagert werden, da SDL [3] hierfür einen eigenen Thread verwendet und die Dokumentation daher davon abrät.

Es mag seltsam erscheinen, dass der Gameloop Events für die einzelnen Module auswertet, obwohl diese selber Events erhalten und abarbeiten können.
Dieser Schritt ist jedoch notwendig, da ein Subsystem z.B. nicht über seine eigene Destruktion entscheiden kann. Events mit diesem Ziel – wie z.B. „System Quit“ fallen deshalb in den Bereich des Subsystems.

5.2. Der Szenengraph

Der Cherubim Szenengraph besteht im Wesentlichen aus zwei Bäumen, dem Volume Graph und dem Draw Graph. Der Volume Graph repräsentiert die geometrische Anordnung der Szene in Form einer axis aligned bounding box Hierarchie, während der Draw Graph als eine stateoptimierte Übersetzung des Volume Graphs verwendet wird.

Die Knoten der beiden Graphen besitzen neben den Hierarchieinformationen sogenannte Cores. Diese bestehen aus einer überladenen Funktion, die verschiedene OpenGL Befehle ausführen kann, sowie einigen Zusatzinformationen zur Steuerung dieser Funktion, wie z.B. Textur- oder Transformationsdaten.
Das Konzept der Cores ist an das von OpenSG [5] angelehnt und wurde hier aufgrund zweier implementierungstechnischer Vorteile gewählt:

1. Beliebige Hierarchien (BSP, Octree, etc.) sind ohne aufwändige Neuimplementation oder Mehrfachvererbung möglich – Trennung von Form und Inhalt
2. Einfache Mehrfachverwendung eines Cores und damit u.a. schnelle Übertragung eines Cores bei Erstellung des Draw Graphen.

Speziell der zweite Punkt ist für den Aufbau des Draw Graphs sehr wichtig. So können Cores nach ihrer Priorität - in diesem Fall ihren Kosten - sortiert werden.
Zum Beispiel ist das Ausführen eines Texturcores bezogen auf die Ausführungszeit „teurer“ als das Ausführen eines Transformationscores. Tritt also eine Textur-/Transformationskombination zweimal auf, so ist ein Teilbaum mit einem Texturcore als Elternknoten und zwei Transformationscores als Nachfolgern „billiger“ als ein Teilbaum mit einem Transformations- und zwei Texturcores.
Um dieses Konzept zu realisieren, wird versucht die vorhandenen Pfade im Drawgraph - unter Verwendung der Sortierung - möglichst oft wiederzuverwenden. Die einzelnen Cores besitzen zu diesem Zwecke eine überladene Vergleichsfunktion.

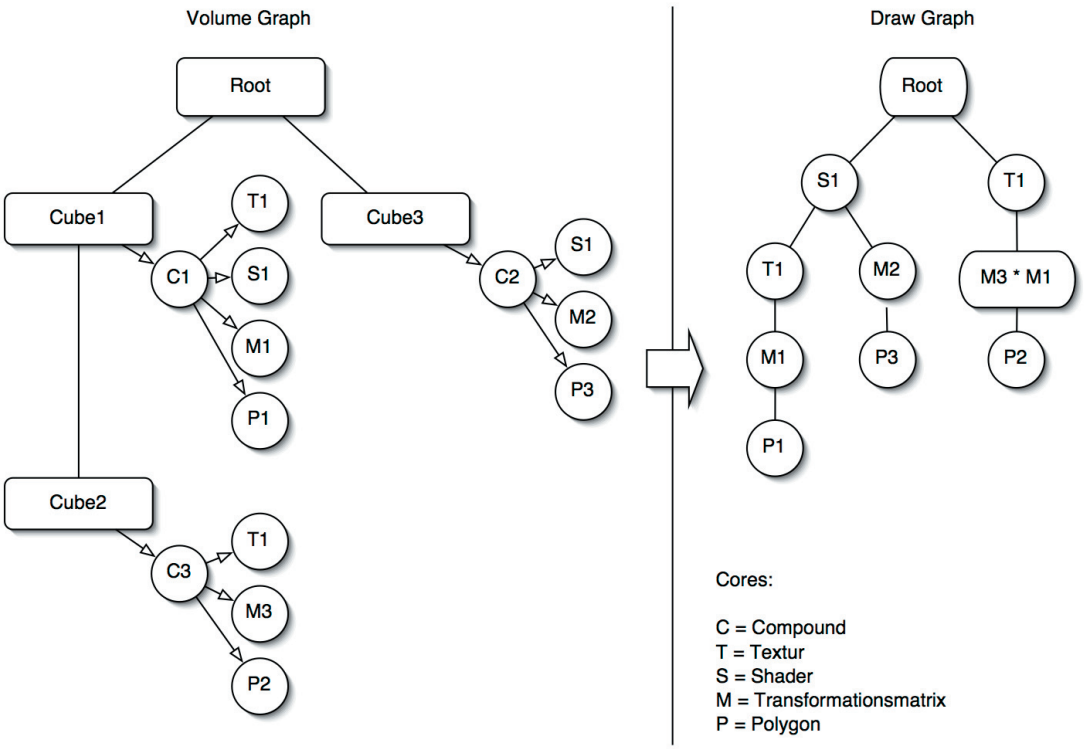


Abb. 5.2.a Übergang von Volume Graph zu Draw Graph

Bei diesem Konzept entsteht jedoch ein Problem: Da der Volume Graph nur für die Aufteilung der Geometrie in der Szene zuständig ist, lassen sich die einzelnen Cores nicht in einer 1:1 Beziehung zu den einzelnen Knoten verteilen. Eine solche Vorgehensweise würde den Baum unnötig verkomplizieren und so die Verarbeitung verlangsamen.

Um dies zu vermeiden, werden im Volume Graph die einzelnen der zur Darstellung eines Objekts benötigten Cores in sogenannten Compound Cores zusammengefasst. Dies hat den Vorteil, dass beim Traversieren des Baumes sofort, d.h. ohne Verwendung eines Stacks bekannt ist, welche Cores zu einem Objekt gehören. Diese können daher direkt einsortiert werden.

5.3. Anpassung des Szenegraphen

Zu Beginn dieser Studienarbeit umfasste der Szenengraph lediglich einige einfache Cores wie Transformationen, Texturen und fest definierte Körper. Es galt nun, Cores für verschiedene Shader, frei definierte Körper und animierte Transformationen zu schaffen.
Der Quellcode des Szenengraphen und des restlichen Frameworks liegen auf der im Anhang dieser Studienarbeit zu findenen CD bei.

5.3.1. Shadercores

Die Erstellung eines Cores für Vertex oder Fragmentshader stellte zunächst kein größeres Problem dar. Sowohl im Falle von Cg- als auch im Falle von ARB-Vertex Shadern muss hier im Wesentlichen nur eine Textdatei eingelesen, an die Pipeline übergeben und verwendet, also ein- und ausgeschaltet werden. Dieses Prinzip wurde bereits in ähnlicher Form bei Texturknoten verwendet.
Das eigentliche Problem stellte sich erst wesentlich später heraus.
Shader – vor allem Vertexshader benötigen Parameter. Diese Parameter beziehen sich allerdings meist nicht alleine auf diejenigen Daten, die durch Texturen und Vertexdaten ohnehin schon vorliegen, sondern auch auf zusätzliche Daten wie die Modelview / Projection Matrix, Position der Kamera und Lichter oder auch z.B. die Matritzen verwendeter Bones.
Man kann hier grob zwischen zwei Arten von Parametern unterscheiden:

1. Parameter, die bereits zur Zeit der Aktivierung des Shaders bekannt sind;
2. Parameter, die erst bei der Übergabe der Objektdaten feststehen.

Die erste Art von Parametern lässt sich durch Setzen von Zeiger auf die entsprechenden Daten realisieren, die zur Ausführungszeit des Shaders ausgewertet werden (Parameter-Registrierung).
Hier muss jedoch beachtet werden, dass ein Vertexshader sowohl die Modelview und Projektions Matrix aufrechnen als auch die Beleuchtung ausführen muss. Es ist also notwendig, dem Shader gewisse szenenabhängige Informationen wie die Position der Kamera und aller Lichtquellen zu übergeben.
Die zweite Art von Parametern stellt nun ein größeres Problem dar. Diese Daten müssen – unabhängig vom Shadercore – von dem Core gesetzt werden, der auch die Polygondaten verarbeitet. Ein Beispiel für solch einen Parameter wäre z.B. die Model-

view Matrix, die erst bei Übertragung der Polygondaten feststeht (siehe Abb.5.2.a).
Hier entsteht also eine Trennung von Shadercore und Parametern, was insofern schwer zu realisieren ist, da – vom objektorientierten Standpunkt aus – dem Modell-Core weder der Shader Core noch die Sprache des Shaders bekannt sein sollte.
Dieses Problem konnte im Rahmen dieser Arbeit noch nicht gelöst werden; das System geht lediglich davon aus, dass nur Cg-Shader verwendet werden. Eine mögliche Lösung wird in Punkt 7 diskutiert.

Eine weitere Schwierigkeit in diesem Bereich trat durch die `uniform` Parameter von Cg auf. Diese Parameter können nur einmal pro Vertexshader gesetzt werden und sich dementsprechend danach nicht mehr ändern. Skins and Bones Vertexshader müssen diese Art der Parameter verwenden, da nur hier Matritzen übergeben werden können.
Shader mit uniform Parametern sind schwer wiederzuverwenden und müssen daher i.d.R. einmal pro Objekt angewandt werden. Da der Szenengraph jedoch auf Mehrfachverwendung optimiert ist (s.5.2.) musste hier ein Schalter eingefügt werden, der dies für genau diese Art von Shadern verhindert.

5.3.2. Modellcores und preShader

Cores für frei definierte Geometrie stellten nur minimale Anforderungen: Eine beliebige Anzahl von Faces musste mit oder ohne Shader ausgegeben werden. Ein solcher Core muss also zwei Elemente enthalten – eine Liste mit den Daten der einzelnen Faces sowie eine Funktion zur Darstellung der Daten. Da nun die Darstellung je nach verwendetem Shader bzw. verwendeter Face-Form unterschiedlich erfolgen kann, wurde diese Funktion in ein sogenanntes preShader Objekt ausgelagert.
Ein preShader ist nun im Prinzip nichts anderes als eine Funktion, die ein Face als Eingabewert verlangt und dessen einzelne Elemente an die Pipeline weiterreicht. Der Vorteil dieses Verfahrens ist, dass man so z.B. den Vertexshader auf der CPU simulieren kann, indem lediglich der preShader ausgetauscht wird.
Im Rahmen dieses Konzeptes wurden zwei preShader entwickelt:
Der erste preShader reicht die Daten eines dreieckigen Faces als Displaylist an die Pipeline weiter und eignet sich daher ideal für starre Objekte.
Der zweite preShader nutzt ein in ihm gespeichertes Bone-System, um die übergebenen Vertices zu verformen. Dieser preShader ist somit das Kernsystem der Skins and Bones Implementation.

Der Skins and Bones preShader ist animiert, d.h. er wertet bei seinem Aufruf verschiedene Animationskurven abhängig von einem ihm zugeteilten Timer aus (siehe 5.4.). Seine Arbeitsweise bedarf einiger Erklärungen, da die verwendete Mathematik von der in 2.2. beschriebenen geringfügig abweicht.
In Maya besteht ein Bone nicht nur aus einer Transformation und Rotation, sondern aus folgenden Matrizen.

$S * Ro * R * Jo * S^{-1} * T$ [6]

S = Skalierung Ro = Orientierung der Rotation R = Rotation
Jo = Joint Orient T = Transformation

Dadurch, dass die Skalierung in dieser Arbeit wegfällt (siehe 2.2.), reduziert sich diese Formel auf folgende Form, wobei Ro und R zusammenfallen können:

$Ro * R * Jo * T$

Problematisch bei dieser Formel ist, dass Jo und R sich immer erst auf den jeweils nachfolgenden Bone bzw. die an den Bone gebundenen Vertices auswirken. Letzteres ist auch der Grund, warum die letzte Jo Matrix nicht mit eingerechnet werden darf. Diese Matrix entspricht immer dem invertierten Produkt aller Jo Matrizen des Skelettes. Wertet man diese Matrix aus, so würden alle an diesen Bone gebundenen Vertices falsch dargestellt werden. Um diesem Fehler vorzubeugen, werden die letzten Joint Orient Informationen bereits vom Exporter auf Null gesetzt.

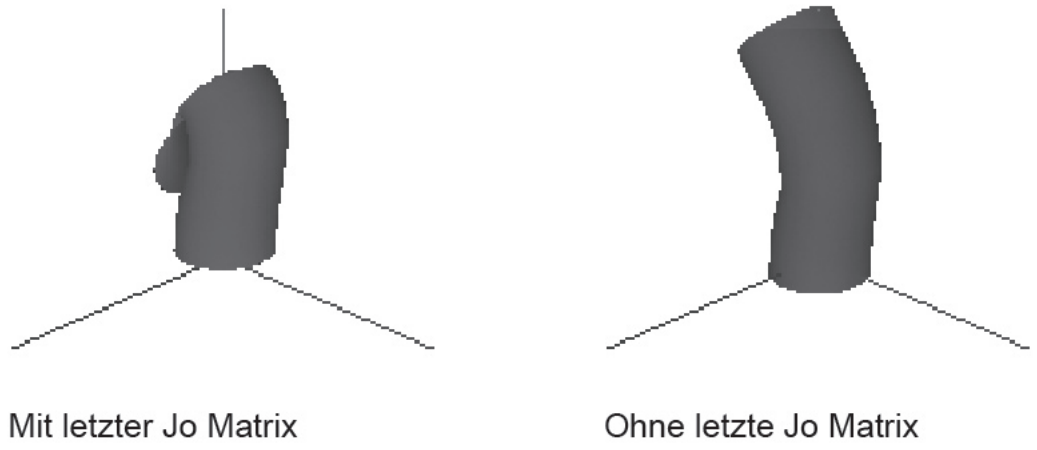


Abb. 5.3.2. Verwendung der Joint Orient Matrix

5.3.3. Animierte Transformationsknoten

Als Nebeneffekt des Exporters mussten nicht nur Animationen durch Skins and Bones möglich sein, sondern auch einfache Animation durch Veränderung der Transformationsdaten eines Objektes.
Hierzu wurde der bereits vorhandene Transformationcore abgeleitet und entsprechend angepasst. Genau wie beim Skins and Bones preShader hat auch dieser Core Zugriff auf diverse Animationskurven, die abhängig von einem übergebenen Timer ausgewertet werden (siehe 5.4.).

5.4. Animationen

Die Implementation bzw. Simulation des Maya-Animationsverhaltens erforderte zwei neue Klassen:
Zunächst wurde eine Timer Klasse benötigt, die abhängig von der Systemzeit und der Angabe „Frames pro Sekunde“ eine fortlaufende Zahl zur Verfügung stellt. Hierdurch sind parallel laufende Animationen mit verschiedenen Geschwindigkeiten möglich, wenn diese entsprechend auf verschiedene Timer-Objekte zugreifen.
Die zweite Klasse hatte die Aufgabe, Animationskurven zu verwalten und Werte für ein bestimmtes Frame zurückzuliefern. Dies erfordert eine Funktion, die den Y-Wert einer Bezierkurve zu einem bestimmten Zeitpunkt auf der X-Achse ermittelt. Da es sich in diesem Fall um kubische Bezierkurven handelt und die entsprechende Formel eine Tupel mit xy-Werten abhängig von einem Parameter u berechnen, müsste man also eine Polynom 4. Grades lösen, was ein nicht unerhebliches Problem darstellt, sollte es überhaupt zu lösen sein.

$$v_{xy} = \sum_{i=0}^{n} \binom{n}{i} * u^i * (1-u)^{n-i} * p_i$$ [SchreinerWooNeider97]

In der von Cherubim verwendeten Funktion wird daher nur eine Annäherung an den korrekten Wert vorgenommen:
Man skaliert zunächst den gesuchten x-Wert auf einen Bereich zwischen 0 und 1, basierend auf dem Start- und Endpunkte der Kurve.
Nun nimmt man an, dass $x = u$ gilt, und berechnet x anhand der Bezier Formel neu. Anschließend vergleicht man die beiden x-Werte und addiert deren Differenz zu 1/3 auf das zuletzt verwendete u.

Dieses Verfahren wird maximal zehn Mal wiederholt bzw. solange, bis die Differenz in einen Bereich < 0,001 liegt, der Fehler also gering ausfällt.
Vorteil dieser Methode ist, dass der gefunde Wert überhaupt berechenbar und exakt genug ist; der Nachteil liegt jedoch auf der Hand – es ist langsam.
Eine Möglichkeit, diese Methode zu beschleunigen, wird in Punkt 7 diskutiert.

5.5. Der Importer

Der Importer hatte sich, durch die auf den Import optimierte Strukturierung des Dateiformates, nur wenigen Problemen zu stellen.
Zu Beginn werden Magic Number und Versionsnummer eingelesen. Ist die Magic Number falsch, so wird hier versucht, diese und die Versionsnummer durch entsprechende Endian-Befehle zu korrigieren. Stimmen diese anschließend nicht mit den gesicherten Werten überein, so wird die Datei nicht weiter bearbeitet.
Anschließend werden die einzelnen Blöcke als Structs gelesen und, falls nötig, durch Endian-Befehle einzeln korrigiert. Es folgt das Erstellen der einzelnen Cores und eines Volume Graph Teilbaums, der bei Erfolg zurückgeliefert wird.
Der Loader kann anhand der Daten aus dem Configmanager entscheiden, ob zur Darstellung von Skins and Bones ein Vertex Shader oder die CPU verwendet wird. Entsprechend dieser Einstellungen wird der preShader für das Modell gewählt und ein evtl. nötiger Vertex Shader Core erstellt.
Zusätzlich beinhaltet der Importer verschiedene Caches, die an die entsprechenden Cores weitergeleitet werden, um das Laden von z.B. oft genutzten Texturen oder Shadern zu beschleunigen.
Diese Caches werden in späteren Versionen von Cherubim sicherlich aus dem Importer-Objekt ausgelagert werden, da nach einem Löschen des Importers auch diese Caches gelöscht werden und somit Zugriffsverletzungen vorprogrammiert sind. Als Übergangslösung funktioniert diese Variante jedoch schon recht gut.

6. Demonstration

6.1. Export aus Maya

Zur Demonstration des Exporters hab ich eine einfache Szene in Maya erstellt.
In dieser Szene werden alle Aspekte des Exporters berücksichtigt:
Sie enthält mehrere Objekte, wovon eines nur in seiner Transformation, und zwei weitere durch Bones animiert sind. Zusätzlich ist mit dem Boden auch ein texturiertes Objekt vorhanden.

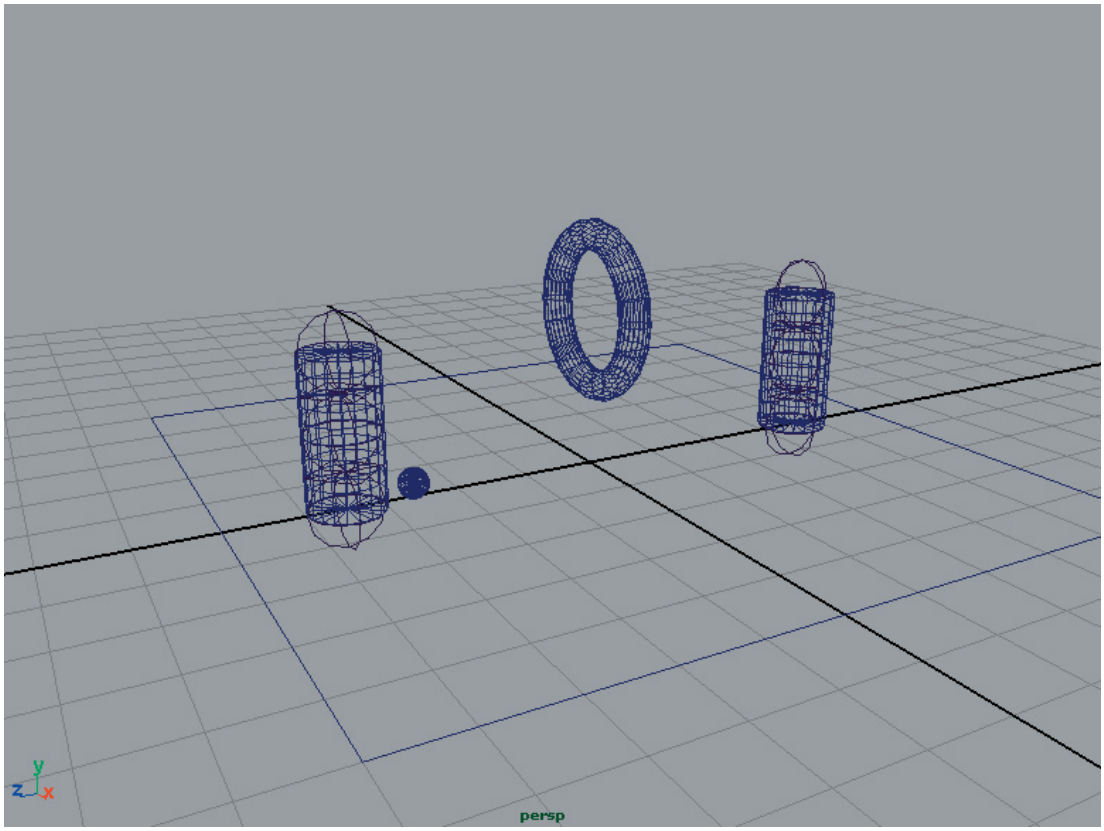


Abb. 6.1.a Modellierung der Demoszene

Wie in Abb.6.1.a zu sehen beinhaltet die Szene zwei Zylinder, einen Ball, einen Ring und eine Bodenfläche. Die beiden Zylinder sollen sich in der Animation nun gegenseitig den Ball durch den Ring zuspielen.
Der Auf Abb.6.1.a zu sehende linke Zylinder spielt dabei mit dem „Fuß“, während der rechte Zylinder mit dem „Kopf“ spielt.

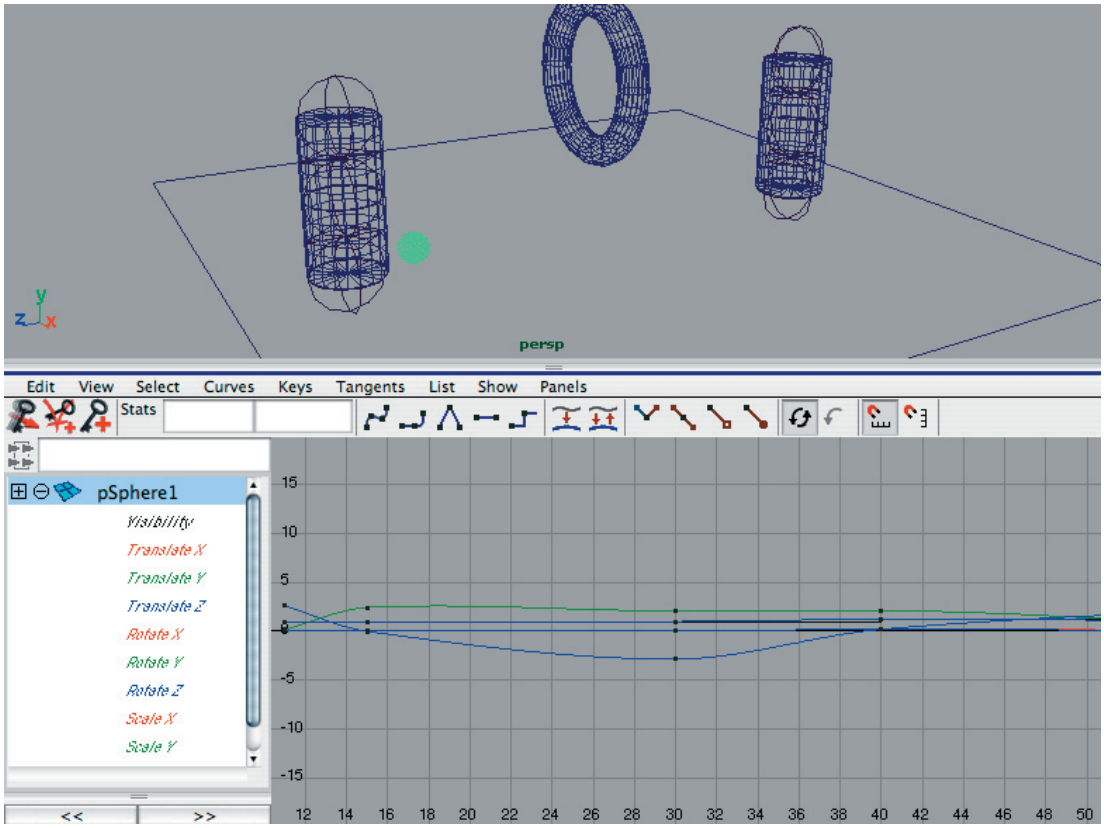


Abb. 6.1.a Animationskurven der gelben Kugel

Diese Szene ist deshalb als problematisch anzusehen, weil zum einen der Ball durch seine Beschleunigung eine modifizierte Animationskurve besitzt und zum anderen die beiden Zylinder zwei unterschiedlich erstellte Skelette besitzen. Das Skelett des linken Zylinders wurde von oben nach unten, das Skelett rechten Zylinders von unten nach oben erstellt. Diese Vorgehensweise führte zu verschiedenen Joint Orients und stark unterschiedlichen Koordinatensystemen bei Skelett und Modell.
Der Exporter konnte diese Szene ohne Probleme abarbeiten.

6.2. Import und Darstellung in Cherubim

Der Import der Daten aus 6.1. in den Cherubim Szenengraphen wurde unter Windows XP und Mac OS X getestet. Der Export erfolgte auf Mac OS X, so dass unter Windows XP eine Endian-Konvertierung durchgeführt werden musste.
Auf beiden Systemen wurde die Datei ohne größere Probleme geladen und in den Szenengraph übernommen.
Probleme werden erst bei der Darstellung sichtbar:

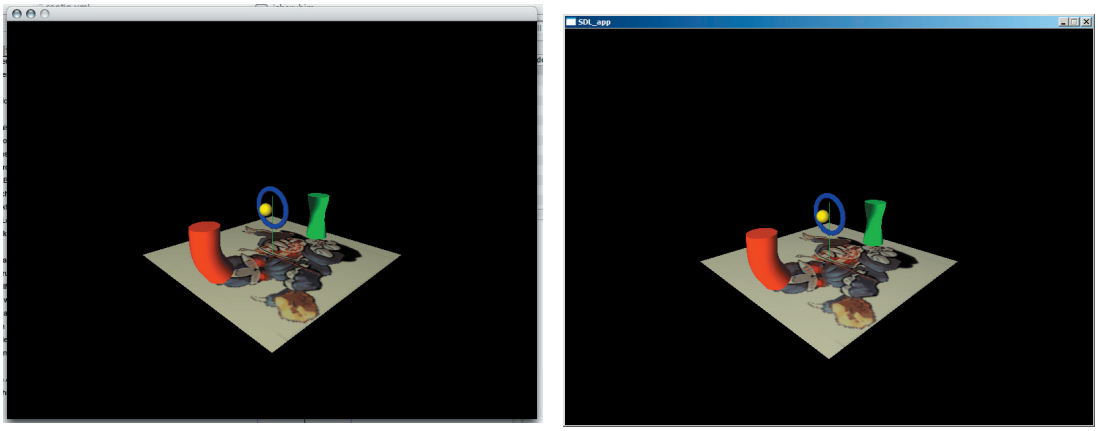


Abb. 6.2.a Cherubim Demoszene unter MacOS X und Windows XP

Die falsche Darstellung des rechten Bones kommt durch die aus Maya exportierten Joint Orients zustande. Der erste Bone des unterliegenden Skelettes ist in der Y-Achse (Weltkoordinaten) verdreht. Dies wird im zweiten Bone wieder korrigiert, wodurch die unteren Vertices gegenüber den oberen Vertices verdreht werden. Der erste Bone ist also - im Sinne der Darstellung in Maya - falsch orientiert, wird aber gemäß der von Maya übergebenen Daten - im Sinne des in dieser Arbeit verwendeten Verfahrens - dennoch korrekt angezeigt.
Eine Lösung für dieses Problem wurde noch nicht gefunden, es kann jedoch durch Korrekturen der Joint Orients in Maya umgangen werden.

Die in Abb.6.2.a zu sehende Szene wurde auf verschiedenen Systemen mit verschiedenen Grafikkarten getestet. Um das System zu beschleunigen wurden neben den Vertexshadern auch SIMD-Erweiterungen für die Berechnung der Vektoren und Matritzen genutzt.

System	FPU	SIMD	GPU	Frames per Second
G4 1.5 Ghz, ATI Radeon 9600 Mobility MacOS X 10.3	x			124 fps
		x		160 fps
	x		x	82 fps
		x	x	83 fps
G5 2x2.0 Ghz, ATI Radeon 9800 Pro MacOS X 10.3	x			197 fps
		x		310 fps
	x		x	141 fps
		x	x	142 fps
Athlon XP 3000+, GeForce FX 5600 Windows XP	x			402 fps
		x		569 fps
	x		x	347 fps
		x	x	348 fps

Die SIMD-Operation brachten eine Geschwindigkeitssteigerung von etwas mehr als 20%, wohingegen Cherubim bei Verwenden eines Vertexshader um etwas mehr als 30% langsamer lief als die FPU-Variante. Dieses Verhalten ist auch bei höher aufgelösten Modellen im gleichen Maße wiederzufinden.

Der Geschwindigkeitseinbruch bei Verwendung von Vertexshadern war nicht zu erwarten und lässt auf ein Problem in der Verarbeitungsweise des Szenengraphens und/oder der Skins and Bones Algorithmen schliessen. Da der Vertexshader aber im Wesentlichen die gleichen Funktionen wie die CPU-Simulation ausführt, scheint das Problem allerdings eher beim Szenengraphen, speziell beim Aufruf des Shaders zu liegen. Auch hier wurd noch keine Lösung gefunden.

7. Fazit und Ausblick

Im Großen und Ganzen wurden alle Ziele erreicht, wenn auch nicht alle Probleme gelöst. Vor allem während der Implementationsphase wurde klar, wie komplex die gesamte Thematik ist. Dadurch, dass Maya für mich Neuland war habe ich Maya durch Mel gelernt und konnte daher gegen Ende der Arbeit auch nicht alle Möglichkeiten und Wege ein Modell zu erstellen ausprobieren bzw. mit dem Exporter testen. So habe ich zum Beispiel noch nicht testen können ob und wie der Exporter auf angewandte Modifier wie z.B. FFD reagiert.
In Zukunft muss der Exporter zudem um einige Features erweitert werden. Eine Idee wäre es z.B. das Exportieren verschiedener Animationsdaten zu einem Objekt zu ermöglichen, um z.B. Laufen und Rennen zu ermöglichen ohne eine neue Datei laden zu müssen. Anschließend könnte man in Cherubim z.B. Motionblending, also das Überlagern von Animationen problemlos implementieren.
Vorgesehen, aber noch nicht umgesetzt ist der Export von eigenen CgFX Fragment-Shadern als Teil oder Unterstützung der Textur bzw. des Beleuchtungsmodells. Dies wäre zwar Prinzipiell schon möglich gewesen, da aber Maya 5 hierfür noch ein CgFX-Plugin benötigt, dieses aber von NVidia nur für Windows zur Verfügung gestellt wird, hoffe ich hier auf Maya 6. Sollte es auch in dieser Version noch Probleme geben, so könnte man versuchen das Maya-eigene Shading-Network durch entsprechende eigene Shader nachzubilden, wie es Michael Nikelsky in seiner Arbeit beschrieben hat [Nikelsky03].

Bei der Erweiterung Cherubims wurden ebenfalls alle gesteckten Ziele erreicht, allerdings mussten einige Probleme durch Kompromisse gelöst werden, für die in Zukunft bessere Wege gefunden werden müssen.
Als größtes Problem stellte sich die Weiterleitung von Parametern an einen Shader-Core heraus (siehe 5.3.1.). Hier wäre es sinnvoller gewesen ein Informationsobjekt zu verwenden, welches allen Cores bekannt ist und z.B. den aktuellen Shader enthält. Auf diese Weise könnte man auch die im preShader verwendeten Parameter wieder durch Funktionen im Shadercore verarbeiten lassen. Ein weiterer Vorteil dieser Vorgehensweise wäre es, dass man globale Objekte wie Lichter und Kameras an einer allgemein zugänglichen Stelle unterbringen könnte, wovon Shader und z.B. auch Billboards profitieren würden.
Dieses Konzept wurde noch nicht umgesetzt, da die Shader erst zu einem recht späten Zeitpunkt eingeführt wurden und die Implementierung dieses Verfahrens größere Veränderungen im Szenengraph und damit mehr Zeit erfordert hätte.

Das zweite große Problem ist die Geschwindigkeit des Systems. 14 Frames pro Sekunde bei einer Szene mit 6400 animierten Vertices (ermittelt auf einem 1.5 Ghz G4) ist definitiv zu wenig. Um dieses Problem zu beheben könnten nun verschiedene Ansätze umgesetzt werden:
Ein sicherlich nicht unerheblicher Teil der Rechenzeit pro Frame fließt in die Berechnung der Matrizen und Animationskurven. Die Matritzen werden zwar bereits nur einmal pro Objekt und nicht bei jedem Vertex neu berechnet, dies kann aber noch weiter optimiert werden.
Nach dem Sampling-Theorem könnte man die Animationskurven eines Objektes initial bei doppelter Framerate abtasten und die so entstandenen Matrizen anstelle der bei jedem Frame berechneten verwenden.
Ein Vorteil dieser Methode wäre, dass die Matrizen nur einmal beim Laden eines Objektes berechnet werden und sich so die Berechnung pro Frame auf eine Anzahl von Matrix-Vektor-Multiplikation reduzieren würde. Diese Variante ist allerdings mit einem erhöhten Speicherbedarf verbunden.

Auch wenn dieser Ansatz umgesetzt werden sollte, werden immer noch besser optimierte Klassen zur Verwaltung von Vektoren und Matrizen benötigt. Die bisher verwendeten Klassen sind zwar schon schnell, aber sicherlich noch nicht vollständig ausgereizt.
Das größte Problem in dieser Hinsicht stellt jedoch der verwendete Vertexshader dar. Hier sollte durch z.B. Profiling getestet werden warum diese Variante gegenüber SIMD und FPU so viel langsamer ist. Eine Beschleunigung dieses Verfahrens wird sicherlich die beste Optimieriung sein.

Abschließend bleibt zu sagen, dass es sich als recht einfach erwies, den in Cherubim verwendeten Szenengraphen zu erweitern. Da das System sehr modular gehalten und einfach aufgebaut ist, bereitete es - neben den bereits angesprochenen Problemen - kaum Schwierigkeiten bei der Implementation. Ob sich das Verfahren der sortierten Cores und die Trennung von Volume Graph und Draw Graph in großen Szenen bewährt, bleibt jedoch abzuwarten. Im momentanen Stadium des Frameworks sind Tests dieser Form leider noch nicht möglich gewesen, da z.B. noch keinerlei optimierte Hierarchien vorliegen.

Beide Elemente dieser Studienarbeit, Cherubim und der Exporter, werden in Zukunft noch an einigen Stellen erweitert werden müssen, bilden aber bereits ein gute Grundlage auf der weiter aufgebaut werden kann.

Literaturverzeichnis

[1] „Seraphim“, http://ww.seraphim.info, 2004.

[2] OpenGL Architecture Review Board, „OpenGL“, Version 1.4, http://ww.opengl.org, 2003.

[3] „SDL“, Version 1.2.7, http://www.libsdl.org, 2004.

[4] Nvidia, „Cg“, Version 2.0, http://www.nvidia.com/cg, 2004.

[5] „OpenSG“, Version 1.3, http://www.opensg.org, 2004.

[6] Alias | Wavefront, „Maya“, Version 5.0, http://www.alias.com, 2003.

[7] Discreet, „3dsmax“, Version 6.0, http://www.discreet.com, 2003.

[FernandoKilgard03] Randima Fernando, Mark J. Kilgard, „The Cg Tutorial“, 1. Ausgabe, Addison Wesley, 2003.

[Giberson01] Dean Giberson, „Programming Maya 3.0“, Game Developers Conference, 2001.

[Gould03] David A.D. Gould, „Complete Maya Programming“, 1. Ausgabe, Morgan Kaufmann, 2003.

[Nikelsky03] Michael Nikelsky, „Entwicklung einer attraktiven Beispielanimation zur Demonstration der Möglichkeiten modernster Graphikhardware“, Universität Koblenz, 2003.

[MöllerHaines02] Tomas Akenine-Möller, Eric Haines, „Real-Time Rendering“, 2. Ausgabe, A.K. Peters, 2002.

[Schulze03] Florian Schulze, „Entwicklung einer Programmbibliothek zur Simulation von Hautdeformation durch Knochen und Muskeln“, Universität Koblenz-Landau, 2003.

[SchreinerWooNeider97] Dave Schreiner, Mason Woo, Jackie Neider, „OpenGL Programming Guide, Version 1.1“, second edition, Addison Wesely, 1997.