Universität Koblenz-Landau
Fachbereich Informatik

Diplomarbeit

Modellierung und Photorealistische Darstellung mit dem RenderMan Interface

Andreas Pidde

Betreut von:
Prof. Dr.-Ing. H. Giesen
Dipl. Inform. Detlev Droege

Wintersemester 1994/95

Andreas Pidde

Universität Koblenz-Landau

Institut für Informatik

Rheinau 3–4
56075 Koblenz

Vorwort

Die Komplexität der Berechnung dreidimensionaler photorealistischer Darstellungen in der Computergrafik und die verschiedensten Ansätze hierzu legen die Verwendung einer einheitlichen Schnittstelle zwischen Erstellung und Berechnung dreidimensionaler Szenen nahe. Die amerikanische Computergrafik-Firma Pixar beansprucht für sich, mit ihrem Produkt RenderMan^{TM} und dem RenderMan Interface Bytestream (RIB) Dateiformat eine solche geräte- und methodenunabhängige Schnittstellenbeschreibung gefunden zu haben. Renderer, die dieser Schnittstellenbeschreibung genügen und Modellierungswerkzeuge mit einer RIB-Ausgabe wurden bereits auf einer Vielzahl von Geräten implementiert. In das Betriebssystem des NeXTs wurden zwei dieser Renderer integriert. Die Anbindung an Computergrafik-Anwendungen geschah unter der Verwendung des objektorientierten 3DKits. Die vorhandene Version 3.1 der Schnittstellenbeschreibung soll von Pixar in Zukunft noch erweitert werden — eine Version 4.0 ist im Gespräch.

Die vorliegende Diplomarbeit befaßt sich anhand eines in ihrem Rahmen erstellten Modellierwerkzeugs mit der Verwendung der Schnittstelle. In den beiden ersten Kapiteln kann eine kurze Einführung in die Thematik der photorealistischen Darstellung und den Funktionsumfang RenderMans und 3DKits gefunden werden. Im folgenden Kapitel wird die Verwendung des Interfaces in dem Modellierwerkzeug und die Bedienung des erstellten Modellierwerkzeugs behandelt. Im Anhang wurde die komplette Referenz der Interface- und 3DKit-Funktionen zusammengestellt. Eine umfangreiche tutorielle Einführung in das RenderMan Interface und eine Bibliographie für das Thema Rendern kann in [Upstill89], eine vollständige Referenz in der Beschreibung des Schnittstellenstandards [PixSpec] gefunden werden. Die Online-Handbücher des NeXTs geben den aktuellen Implementierungszustand der Renderer und die Definition der 3DKit Bibliothek wieder.

An dieser Stelle möchte ich meinen Dank an Prof. Dr.-Ing. H. Giesen und Dipl. Inform. Detlev Droege für die Betreuung der Diplomarbeit und an Frau Stefanie Gies für das Durchsehen der Arbeit aussprechen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Die photorealistische Darstellung ist eines der faszinierendsten und technisch aufwendigsten Teilgebiete der Computergrafik. Diese Disziplin befaßt sich damit, dreidimensionale Szenen möglichst realistisch, vorwiegend als zweidimensionale Pixelmatrizen, in photographisch wirkender Qualität darzustellen. Im Gegensatz zu gescannten Grafiken, braucht keine Vorlage des Resultats vorzuliegen. Es ist vielmehr möglich, eine dreidimensionale Szene numerisch vorzugeben und davon beliebige zweidimensionale oder stereoskopische Ansichten zu erstellen. Es kann natürlich von Vorteil sein, einzelne Komponenten eines Modells dreidimensional zu scannen, statt sie von Hand einzugeben. Mit *Rendern* wird im folgenden das Berechnen einer zweidimensionalen Darstellung ausgehend von einer Datenstruktur, die ein dreidimensionales Modell beinhaltet, bezeichnet. Zum Rendern können verschiedene Methoden wie Scanline, Radiosity, Raytracing oder gemischte Verfahren zum Einsatz kommen. Mit *Renderer* wird sinngemäß das Programm bezeichnet, das das Rendern vornimmt. Die 2D Darstellung kann in einem beliebigen Format ausgegeben werden und muß sich nicht an die physikalischen Grenzen eines Ausgabemediums (z.B. die eines bestimmten Bildschirms) orientieren. Ergänzend zum Rendern werden für das Erstellen photorealistischer Grafiken Werkzeuge, wie 3D-Scanner und Scansoftware, Modellierungsprogramme und Animationsprogramme, verwendet. Der Renderer bildet letztendlich die unterste Schicht bei der Berechnung der Grafiken.

Photographische Genauigkeit ist natürlich nicht immer das vorgegebene Ziel von Computergrafiken. In der Visualisierung von Daten oder Simulationen ist es mitunter wünschenswert, die Grafiken zu abstrahieren, damit die Zusammenhänge, die sichtbar gemacht werden sollen, nicht in einer Flut visueller Daten untergehen. Soll der Vorgang eines Vogelflugs modellhaft simuliert werden, wird sich die Visualisierung normalerweise nicht mit einem natürlich gefärbten Federkleid und der korrekten Schattierung des Vogels abgeben. Stattdessen könnte die Information über die Muskeltemperatur als Fehlfarben dargestellt werden. Interessant sind hier nur die Daten und wie sie dem Betrachter in einer integralen, verständlichen Form dargebracht werden.

Eine andere Möglichkeit des Einsatzes von Computergrafik ist die Nachbildung von künstlerischen Werkstoffen wie Pastellkreide und Ölfarbe bis hin zur Simulation von Malstilen wie Pointillismus, Impressionismus und Expressionismus. Diese Techniken werden vor allem in Echtfarb-Malprogrammen und speziellen Renderern verwendet ([Haeb90]).

Wo liegen die Anwendungsbereiche der photorealistischen Darstellung? Unter den kommerziellen Anwendern befinden sich vor allem die Filmbranche, die Produktentwicklung und die Werbeindustrie. In letzterer ist allerdings wohl weniger die natürliche Darstellung als vielmehr die Effekthascherei durch die, mit der Computergrafik möglich gewordenen, ungewohnten Seheindrücke ausschlaggebend.

In der Filmbranche tut sich ein weites Anwendungsfeld auf. Dort kommt bei den meisten Anwendungen neben der bloßen photorealistischen Darstellung noch die Dimension der Zeit in Form der Animationen hinzu. Anschießend muß diese im allgemeinen noch vertont werden. Animationen sind nicht nur eine Aneinanderreihung von photorealistischen Bildern (24 Bilder/Sek. im Film, bzw. 25 Bilder/Sek. für Videoaufnahmen). Bewegungsabläufe, wie der Gang eines Menschen, die Bewegung eines Blattes im Wind oder das Fließen von Wasser sollen natürlich wirken, ohne daß die Daten eines jeden Einzelbildes von Hand angepaßt werden müssen; Bewegungen sollen nach Möglichkeit berechnet werden (weniger Kosten, schnellere Produktion). Die Ausarbeitung von Modellen entsprechender Bewegungen kann als Übergang zur Simulation natürlicher Abläufe betrachtet werden. Diese ist mehr Gegenstand der Wissenschaft (z.B. Biologie, Physik) als der Filmindustrie. Alternativ oder zusätzlich zur Simulation können Bewegungen 'ferngesteuert' werden. Als aktuelles Beispiel hierzu kann eine neuere französische Verfilmung des Romans '20 000 Meilen unter dem Meer' von Jules Verne gelten, in der echte Schauspieler künstliche computergenerierte Modelle durch ihre Körperbewegungen steuern. Die Bewegungen der Schauspieler werden durch eine spezielle Kamera in Steuerinformationen für die Animation umgesetzt.

Durch die Computergrafik sind der Phantasie eines Regisseurs oder eines Drehbuchautors ähnlich wie bei einem Zeichentrickfilm im Prinzip keine Grenzen gesetzt. Dadurch werden vor allem für den phantastischen Film neue Möglichkeiten geschaffen. Es können z.B. reine Phantasiewelten geschaffen oder neue Lebewesen in Science Fiction Streifen zum Leben erweckt werden. Allerdings sind die Produktionskosten einer aufwendigen Animation, neben den Schwierigkeiten bei der Modellierung, momentan wohl noch höher als bei einem von Hand erstellten Zeichentrick oder einer gespielten Szene vor einer entsprechenden handgemachten Kulisse (zum Glück für die Schauspieler). Die Übergänge sind in diesem Bereich jedoch schon fließend, Schauspieler können in einer vom Computer errechneten Kulisse agieren oder mit computeranimierten Gestalten zusammenspielen. In den photorealistischen Bildern dieser Filme werden neben der berechneten Schattierung häufig vorgefertigte Texturen verwendet, die auf die Oberflächen abgebildet werden. Texturen bieten aber nicht immer ein zufriedenstellendes Ergebnis, oft sehen die Bilder unnatürlich, wie ausgeschnitten aus.

Damit Animationen natürlich wirken, ist es wichtig, bewegte Körper ein wenig unscharf darzustellen (Simulation der Öffnungszeit des Kameraobjektivs), um nicht einen stroboskopähnlichen Effekt zu erzeugen. Neben dieser Bewegungsunschärfe kann auch eine Tiefenunschärfe den realistischen Effekt einer Computergrafik erhöhen (Simulation der Brennweite eines Kameraobjektivs). Durch die Trägheit des menschlichen Auges bedingt ist es möglich, bewegte Körper weniger detailgetreu als ruhende darzustellen, ohne daß die Qualität der Animation darunter leidet. Zusätzlich können Körper, die nur eine kleine Oberfläche auf der 2D-Projektion einnehmen, weniger genau modelliert werden als die, die eine große Fläche verdecken.

Durch die natürlich wirkende Nachbildung von Szenen, bzw. der nachträglichen digitalen Bildbearbeitung, entsteht natürlich auch die Gefahr der beabsichtigten oder unbeabsichtigten Täuschung des Betrachters. Selbst bei genügender Aufklärung über die Möglichkeiten der Computergrafik — was soll geschehen wenn nicht einmal mehr den in den Nachrichten gezeigten Bildern Glauben geschenkt werden darf, wenn visueller Information kein Informationsgehalt mehr gehört, da sie möglicherweise täuschend echt gefälscht wurde? Die mögliche Manipulation oder Täuschung des Zuschauers, sei es auch 'nur' durch die Werbung, ist eine negative Auswir-

kung der Computergrafik. Allerdings sind diese Täuschungen sicherlich auch ohne Computergrafik möglich, wenn auch nicht in einem so hohen Grad der Perfektion ([Mitch94]), und gehen nicht von der photorealistischen Grafik an sich, sondern von denjenigen, die die Möglichkeiten für negative Zwecke ausnutzen könnten, aus. Durch eine gewisse Vielzahl von untereinander unabhängigen, kritischen Informationsgebern (Presse) kann die Gefahr einer Manipulation abgeschwächt werden. Die Betrachtung der Vor- und Nachteile der photorealistischen Darstellung kann sicherlich noch soweit ausgebreitet werden, daß sie den Rahmen dieser Arbeit sprengt, die sich ja vielmehr mit der Vorgehensweise bei der Modellierung und der technischen Realisierung durch das RenderMan-Interface befassen soll. Es macht allerdings noch Sinn, sich kurz über die weiteren Anwendungsmöglichkeiten der photorealistischen Computergrafik Gedanken zu machen, möchte man sich nicht nur die imposanten Bilder anschauen, was zugegebener Maßen auch recht amüsant sein kann. Also, was für Möglichkeiten bietet die photorealistische Grafik außer in Film und Werbung noch? Mir fallen dazu folgende Anwendungen ein:

Architektur: Modellierung von Gebäuden ([Green91]), deren Einbettung in die Umwelt, Walk-through, Anlegen einer Parkfläche

Design: Modellierung und Präsentation von neuen Fahrzeugen, Einrichtungs- und Gebrauchsgegenständen

Die photorealistische Darstellung kann zur visuellen Aufbereitung von Daten, die mit einem CAD System erstellt wurden, verwendet werden. Ein Designer kann auf diese Weise einen visuellen Eindruck von einem Modell bekommen und es mit anderen Körpern in Beziehung setzen.

Archäologie: Visuelle Modellierung von nur teilweise erhaltenen prähistorischen Fundstücken, Nachbildung von antiken Bauwerken nach gefundenen alten Plänen ([QuiM91]), Darstellung einer Animation von Urmenschen, Urtieren u.s.w.

Medizin: Aufbereitung von Computer- und Kernspinresonanz-Tomographien ([Meinzer93], [DreCH88])

In den Bereich der virtuellen Realität, in dem in Echtzeit 3D-Bilder animiert werden, kann die photorealistische Darstellung wohl erst Einzug halten, wenn die Rechner sehr viel leistungsfähiger werden. Momentan werden die Bilder einer photorealistischen Animation noch einzeln mit hohem Speicher- und vor allem Zeitaufwand, mitunter durch ganze Rechnernetze berechnet und anschließend zu einer Animation zusammengestellt. Für ein einziges Bild werden unter Umständen Stunden, laut [Adel94] sogar Tage und Wochen an Rechenzeit verbraucht. Sollen professionell Bilder erstellt werden, wird häufig (teurere) Spezialhardware verwendet, damit die Rechenzeit nicht alle Rahmen sprengt. Das RenderMan-Interface bietet eine hardwareunabhängige Schnittstelle zwischen einem Modellierungswerkzeug und einem Renderer, der z.B. auf einer speziellen Grafik-Maschine implementiert sein kann. Da Supercomputer oft das 100fache einer normalen Workstation kosten ([Adel94]), werden die endgültigen Bilder häufig durch verteiltes Rendern von vielen kleineren, aber leistungsstarken Computern berechnet.

Diese Arbeit soll anhand eines eigenen Modellierprogramms zeigen, wie Pixars RenderMan-Interface ([PixSpec]) zum Rendern von Grafiken verwendet werden kann, wie die Vorgehensweise der Modellierung einer Szene aussehen kann, welche Möglichkeiten das Interface für

Modellierungsprogramme bietet, und Einblicke in Implementierungsmöglichkeiten geben. Als Hardwareplattform diente ein NeXT, dessen Software Zugang zu dem Interface und damit zu den Rendereren mit Hilfe der 3DKit-Objektstruktur bietet ([NeXTDoc], [PixQRM], [Uter93a], [Uter93b]). Einführungen in die Programmierung von Grafiksystemen und Shading werden z.B. in [FolVDFH90], [NewS86], [Rogers85], [RogA90] und [RogE90] behandelt. Kleine Beispielprogramme können u.a. in [ClaP93], [ClaP91] und [HorP89] gefunden werden. Eine spezielle Behandlung der Modellierung mit Splines und Bézierkurven kann aus [BBB87], [RogA90] und [HosL91] entnommen werden. Ein Überblick der Anwendung der Computergrafik wird in [Will89] gegeben. Geometrische Probleme werden in Büchern wie [PreS85] behandelt.

2. Kurzbeschreibung von RenderMan und 3DKit

Die beiden folgenden Abschnitte sollen einen zusammenfassenden Überblick von dem RenderMan-Interface und seiner Einbettung in das 3DKit geben. Eine vollständige alphabetische Zusammenstellung der Interface-Funktionen kann im Anhang gefunden werden. Eine tutorielle Einführung ist in [Upstill89] zu finden, in [PixSpec] befindet sich die Definition der Version 3.1 des Interfaces. [PixQRM] behandelt die Abweichungen des *Quick RenderMan* vom 3.1 Standard. Die Klassenbeschreibungen von 3DKit sind vollständig in [NeXTDoc] enthalten.

2.1 Das RenderMan-Interface

RenderMan ist ein Interface zur Beschreibung von dreidimensionalen Szenen, das durch Funktionen einer Programmiersprache (*Language Binding*) oder durch den Befehlssatz des programmiersprachenähnlichen *RenderMan Interface Byte Streams* (RIB) gegeben ist. Eine RenderMan Eingabe kann aus mehreren einzelnen Szenenbeschreibungen (*Frames*) bestehen, die nacheinander gerendert werden. Auf diese Weise können Animationen hergestellt werden. RenderMan wird häufig mit dem für zweidimensionale Seitenbeschreibungen ausgelegten PostScript ([Adobe]) verglichen. Im Gegensatz zu PostScript bietet RIB nicht den Umfang einer ‘echten’ Programmiersprache.

Implementationen des Interfaces müssen den kompletten RenderMan Befehlssatz syntaktisch auswerten können, brauchen aber nicht unbedingt alle Funktionen der *Optional Capabilities*[1] ausführen. Andere Funktionen werden u.U. nur vereinfacht ausgeführt. Es gibt je nach Implementation Unterschiede in der Realisierung des Interfaces. Da aber alle Renderer die komplette Interface-Definition verstehen, braucht beim Programmieren eines 3D-CAD-Programms oder der Erstellung einer 3D-Szene nicht darauf eingegangen werden, welcher Renderer letztendlich verwendet werden soll. Die *Qualität* des Renderers wird durch die Anzahl der ausführbaren Funktionen und der Realitätsnähe des Ergebnisses bestimmt. Die Qualität ist höher, je mehr Funktionen entsprechend der Interface-Definition ausgeführt werden und je natürlicher die Resultate wirken. Sie nimmt entscheident auf die Geschwindigkeit des Renderns Einfluß. Renderer, die zur Unterstützung von Werkzeugen (Modeller) für den interaktiven Aufbau von Szenen entworfen wurden, werden, um schnelle Reaktionszeiten zu erzielen, eine geringere Qualität

[1] Optionale Teile des Interfaces, die durch die RenderMan Spezifikation [PixSpec] festgelegt sind

besitzen als solche, die photorealistische Bilder erzeugen, ohne daß der Benutzer interaktiv eingreifen kann. Die Qualität sagt im allgemeinen nichts über die Brauchbarkeit des Renderers aus, sondern soll auf seinen speziellen Einsatzbereich und die zur Verfügung stehende Hardware abgestimmt sein. Es gibt keine Maßeinheit für die Qualität in ihrer Gesamtheit. Die Qualität kann bestimmen, ob Splineapproximierungen von kurvigen Oberflächen durchgeführt werden oder ob diese nur als Polygone gerendert werden. Die Darstellung von Oberflächen als Punktwolke, Gitternetz, flache Schattierung oder als Schattierung mit weichen Übergängen ist ebenfalls eine Ausprägung der Qualität. Sie kann auch darüber entscheiden, ob benutzerdefinierte Schattierungsalgorithmen (sogen. *Shader*) oder nur eine kleine Auswahl von vordefinierten Shadern für die Objektdarstellung verwendet werden. Qualitätsmerkmale können meistens vom Benutzter beeinflußt werden; durch das Herabsetzen der Qualität können Ergebnisse schneller erzielt werden.

Eine weitere Variable des Renderers ist der Detaillierungsgrad, der sogenannte *Level of Detail* eines Objekts. Er wird als Pixelfläche, die die Hülle des Objekts im gerenderten Bild einnehmen würde, gemessen. Die Funktion, die die Berechnung der Pixelfäche vornimmt, kann vom Anwender verwendet werden, um die Genauigkeit des Renderns in Abhängigkeit des dargestellten Ausmaßes eines Objekts zu beeinflussen. Der Level of Detail kann zusätzlich mit einem, durch den Benutzer gegebenen Faktor relativiert werden. Je höher der Level of Detail ausfällt, desto detailreicher sollten die Objekte dargestellt werden, oder — mit anderen Worten ausgedrückt — Darstellungen, die nur wenig Pixelfläche füllen, brauchen normalerweise kein großes Detailreichtum zeigen. Das Rendern kann auf diese Weise beschleunigt werden, ohne daß ein Qualitätsverlust bemerkbar wird. Wenn ein Anwender von RenderMan für verschiedene Bereiche des Level of Detail ein Modell von einem darzustellenden Objekts definiert, kann eine RenderMan Implementation automatisch ein entsprechendes Modell auswählen. Auch fließende Übergänge von einem Modell zu einem anderen in einer einzigen Darstellung sind denkbar. Wird der Detaillierungsgrad in aufeinanderfolgenden Bildern relativiert, können, bei einer entsprechenden Implementierung des Interfaces, auch metamorphoseartige Effekte erzielt werden. Eine weitere Methode das Level of Detail zu verwenden, ist die Programmierung prozeduraler Modelle. Ein darzustellendes Objekt wird durch eine Prozedur gegeben, die vom Renderer zum Zeitpunkt der Darstellung mit dem aktuellen Level of Detail und einem Zeiger auf einen vorher definierten Datenblock als Parameter aufgerufen wird. Die Prozedur kann flexibel auf diese Variablen reagieren und entsprechende RenderMan Funktionen aufrufen. Für die Auswahl der Modelle aufgrund des Detaillierungsgrades werden Methoden der Fuzzy Logik verwendet.

Für RenderMan besteht eine Szenenbeschreibung aus Kamera, Lichtquellen, darzustellenden Objekten und der ‘Atmosphäre’ zwischen Lichtquelle und Objekt und zwischen den Objekten, Kamera und Objekt und dem Inneren eines Objekts. Jedem dieser Bildteile können Shader zugeteilt werden, die die Eigenschaften der Oberfläche oder des Volumens, die über deren grobe Geometrie hinausgehen, bestimmen (z.B. ‘Bump Mapping’, Oberflächentextur und -reflexionseigenschaften). Die Shader machen einen wesentlichen Anteil der photorealistischen Darstellung von RenderMan aus. Ein Shader ist ein Schattierungsalgorithmus, der für ein bestimmtes Objekt verwendet wird. Er kann vom Anwender als Prozedur in der *Shading Language* ([HanL90]) in einer C-ähnlichen Syntax entworfen werden. Die Sprache ist speziell für den Zweck der Schattierer ausgelegt. Zur Unterstüzung der Shader können zusätzliche Funktionen in der gleichen Sprache geschrieben werden. Durch diese Vorgehensweise wird der eigentliche

Schattierungsalgorithmus vom Renderer getrennt, wodurch eine beliebige Anzahl von Oberflächen- und Volumentypen in das Rendern integriert werden kann.

In der Abbildung 2.1 (s. [Upstill89], Shader from the Outside looking in, Seite 227) ist der Datenfluß beim Schattieren dargestellt. Die Abbildung 2.2 zeigt den vereinfachten Datenfluß von den auf dem NeXT implementierten Renderern. Diese Renderer verwenden einige der benutzerdefinierten Shader-Typen nicht.

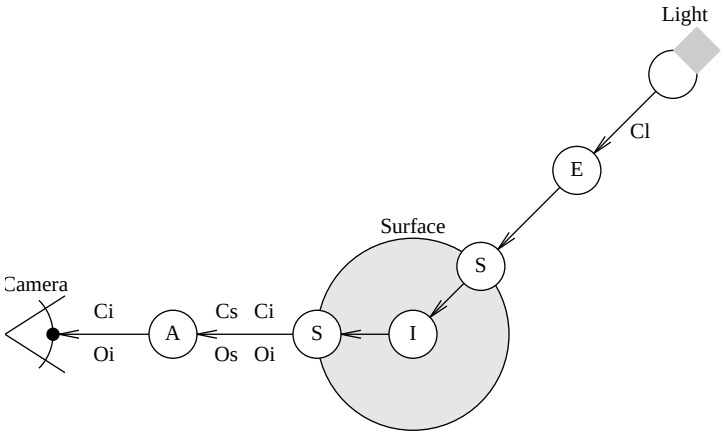

Abbildung 2.1: Einige Shader des RenderMan-Interfaces

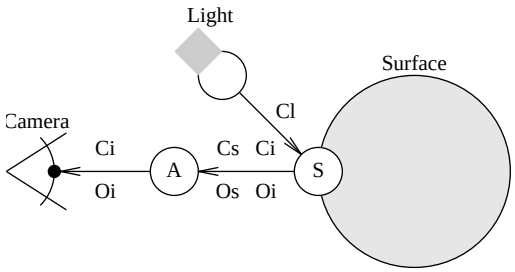

Abbildung 2.2: Einige Shader der NeXT Implementationen

Die Buchstaben in den Kreisen bezeichnen die wichtigsten, der für die Berechnung eines Punktes verwendeten Shader-Typen:

A Volumen-Shader für die Atmosphäre zwischen Kamera und darzustellenden Punkt, Shader für Nebel (fog) und Tiefenschnitt (depthcue) sind auf den meisten Systemen vorhanden

E Volumen-Shader für die Atmosphäre zwischen Lichtquelle und Oberflächen, in diesem ‘Exterior Shader’ kann reflektiertes Licht von anderen Körpern einbezogen werden

I Volumen-Shader für das Innere eines Körpers. Er bestimmt die Veränderung der Shaderparameter bei der Durchdringung eines Körpers.

S Oberflächen-Shader, dient haupsächlich zur Bestimmung der Oberflächenfarbe und Opazität.

Die Bezeichner an den Pfeilen in Richtung des Datenflusses sind eine Auswahl der Namen der globalen Variablen, die in einen Shader über einen Versorgungsblock eingegeben bzw. von einem Shader ausgegeben werden. Die Namen sind für alle Shader gleich (auch wenn nicht explizit im Schema angegeben):

Cl Lichtfarbe von einem Lichtquellen-Shader

Cs Eingegebene Oberflächenfarbe an einem bestimmten Punkt. Der Wert wird vom aktuellen Grafikzustand des Renderers, meist durch einen Aufruf der Funktion **RiColor()**, mitbestimmt, reflektiertes Licht kann eingerechnet sein

Os Opazität an einem bestimmten Punkt. Der Wert wird, meistens durch einen Aufruf der Funktion **RiOpacity()**, im Grafikzustand des Renderers festgelegt

Ci Berechnete (reflektierte) Oberflächenfarbe für einen bestimmten Punkt

Oi Berechnete Opazität für einen bestimmten Punkt

Eine vollständige Aufzählung der Variablen kann in der Beschreibung der ‘Shading Language’ ([Upstill89], Seite 293f, 295f) gefunden werden. Dort ist auch aufgeführt, in welchen Shadern bestimmte Variablen lesend oder schreibend verwendet werden können.

Das Diagramm 2.3 zeigt schematisch die Verwendung der verschiedenen Shader-Typen zur Berechnung der Oberflächenfarbe eines Punktes. Reflektierte Farben werden von dem ‘Exterior Shader’ bearbeitet und an den Oberflächen-Shader weitergeleitet. Zusätzlich erhält er die von transmittierten Licht hinzukommende Farbe über den ‘Internal Shader’ und die Farbe des von Lichtquellen emittierten Lichts. Der ‘Surface Shader’ berechnet von diesen Werten ausgehend die Farbe des reflektierten Lichts. Die Position des Oberflächenpunktes, dessen Farbe bestimmt werden soll, kann zuvor von einem ‘Displacement Shader’ verschoben worden sein. Die Farbe kann anschließend noch von einem ‘Atmosphere Shader’ verändert werden.

Es können gleichzeitig mehrere Shader unterschiedlichen Typs aktiv sein. Eine zerfurchte Metalloberfläche kann durch einen ‘Displacement Shader’ für die Furchen und einem ‘Surface Shader’ für die Reflexionseigenschaften des Metalls beschrieben werden. Eine andere Anwendung der Shader ist, Lichtquellen mittels eines ‘Texture Maps’ und eines entsprechenden ‘Lightsource Shaders’ in Diaprojektoren zu verwandeln. Es können Shader für Oberflächen (Textur, Reflexionseigenschaften), Lichtquellen (Scheinwerfer, Punktquelle, parallele Strahlen, . . .), Volumen, Farbkonvertierungen, Oberflächendeformationen und Transformationen eines Koordinatensystems geschrieben werden. Nicht alle dieser Shader-Typen werden notwendigerweise von einer konkreten RenderMan Implementation unterstützt. Jede RenderMan Realisierung wird aber zumindest über eine bestimmte Anzahl von fest eingebauten Standard-Shadern (z.B. einem Gouroud-Shader für Oberflächen) verfügen.

Viele Oberflächenarten werden durch das Aufrechnen von Bitmaps erzielt. Bitmaps können für einfache Oberflächentexturen (*Texture Mappings* [Smith87]), Schatten (*Shadow Mappings* [ReeSC87]), die von Lichtquellen-Shadern verwendet werden, Turbulenzen der Normalen (*Bump Mappings* [CabMS87]) und Reflexionen (*Environment Mappings* [RogE90]) eingesetzt

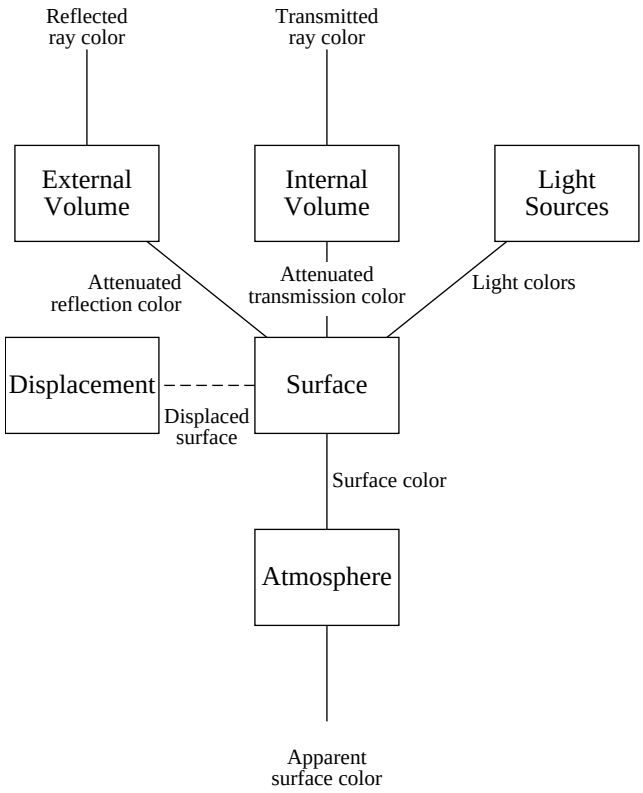

Abbildung 2.3: Schematische Darstellung der Berechnung der Oberflächenfarbe, aus [PixSpec]

werden. Zusammenfassungen der Mapping-Typen können in [Upstill89] und [FolVDFH90] gefunden werden. Raytracing kann das ‘Environment Mapping’ nicht immer ersetzen, obwohl es manchmal als ‘poor man’s ray tracing’ ([RogE90]) bezeichnet wird. In Computergrafiken, die mit realen Gegenständen (z.B. die Animationen zu dem Film ‘Abyss’, siehe Abb. Plate 26 in [Upstill89]) gemischt werden, ist ‘Environment Mapping’ eine geeignete Methode zur Darstellung von reflektierenden Oberflächen, will man nicht die Schauspieler und deren Umgebung modellieren. Für das ‘Shadow Mapping’ gilt ähnliches. Da für ‘Shadow Maps’ Tiefen-Bitmaps verwendet werden, die aus dem Blickwinkel einer Lichtquelle gerendert wurden, entstehen bei transparenten Körpern und bei nicht-punktförmigen Lichtquellen, die keinen Kernschatten besitzen, Probleme.

Neben ‘Texture Maps’ können Oberflächentexturen im Sinne des prozeduralen Schattierens ([Perlin85], [Peac85], [PerH89]) erzeugt werden. Berechnungsmodelle für Muster, die auf den chemischen Stoffaustausch in Lebewesen beruhen ([Turk91], [WitK91], [MeiK91]) können aufgrund der wirkenden Prozesse, in die, wie in zellulären Automaten, die Wechselwirkungen mit Nachbarbereichen einbezogen werden, nicht direkt in der ‘Shading Language’ ausgedrückt werden, da jeder Schattierer nur genau einmal pro sichtbaren Bildpunkt aufgerufen wird. Rekursio-

nen und komplexe Datentypen werden zudem nicht unterstützt.

Die Shader-Prozeduren werden vom System mit verschiedenen Variablen des Renderers und über zusätzliche, benutzerdefinierte Parameter versorgt. Zu den Variablen des Versorgungsblocks zählen u.a. der Vektor zur Kamera, die Normalen der Oberfläche und Vektoren von den Lichtquellen. In einigen Variablen, z.B. in der Ausgabevariable der Oberflächenfarbe, können dem Renderer Werte geliefern werden. Ein Shader besitzt keinen expliziten Rückgabewert. Um alle Lichtquellen in der Umgebung untersuchen zu können, stehen in der 'Shading Language' Iteratoren über alle aktuell gültigen Lichtquellen zur Verfügung. Mit dem Iterator **illuminance()** und der Funktion **trace()** kann eine Art Raytracing ([PixSpec]) simuliert werden. Ein globales Beleuchtungsmodell ([Kajiya86]) allein mit Shadern und 'Mappings' ist aber nur bedingt erreichbar. Wird ein Raytracing Oberflächen-Shader verwendet, kann nicht zusätzlich eine prozedurale Textur verwendet werden, es sei denn, sie ist im Raytracing Oberflächen-Shader einprogrammiert. Von einem Renderer standardmäßig eingesetzte, aufwendige Raytracing und Radiosity Methoden ([CohG85], [KayK86], [WalCG87]) können hier (teurere) Abhilfe schaffen.

An geometrischen Oberflächen stellt RenderMan Prozeduren für den Aufbau von quadratischen Oberflächen: Kugel, Paraboloid, Kegel, Zylinder, Torus, Hyperboloid und Scheibe bereit. Mit verschiedenen Parametern kann der Aufbau der Oberflächen gesteuert werden (z.B. gekappte Kugel). Mit Hilfe der optionalen *Constructive Solid Geometry* (CSG) können diese Flächen in 'Solid' Blöcken als solide Körper dargestellt und im Sinne dieser Modellierung durch die Operationen Vereinigung, Schnitt und Differenz miteinander kombiniert werden.

Beliebige Oberflächen können durch Polygonzüge und Gitternetze modelliert werden. Den Flächen, die durch ein Gitternetz gegeben sind, können vom Benutzer die Normalen zugeordnet werden. Die Gitternetze können auch als Hülle für verschiedene Splinesorten verwendet werden. Der Benutzer kann die 4×4 Basismatrizen für uniforme nicht-rationale B-Splines selbst spezifizieren. RenderMan kennt bereits die Basismatrizen der approximierenden B-Splines ([ForB88]) und Bézierkurven, sowie der interpolierenden Catmull-Rom- und Hermite-Splines. Desweiteren können nicht-uniforme nicht-rationale B-Splines und NURB's ([ShaC88]) für eine Oberflächendarstellung verwendet werden. Die Interpolierungsgenauigkeit der Oberflächen kann vom Benutzer beeinflußt werden. Eine ausführliche Behandlung der Splineinterpolation ist u.a. in [BBB87] und [RogA90] zu finden. Eine gute, knapp gehaltene Einführung in die Programmierung zweidimensionaler Bèzierkurven kann auch in [HosL91] gefunden werden.

RenderMan benutzt ein hierarchisch abgestuftes Koordinatensystem. Alle Körper haben ihre eigenen Objektkoordinaten und liegen in einem Weltkoordinatensystem eingebettet, dieses ist durch eine Transformation vom Kamerakoordinatensystem abhängig. Ein Teil des Weltkoordinatensystems (*View Cone*) wird beim Rendern auf ein genormtes zweidimensionales Rechteck projiziert. Dieses, oder ein Teil davon, kann als Pixelfeld ausgegeben werden. Der Benutzer kann auf die Ausführung praktisch aller Abbildungen Einfluß nehmen. Es existieren Funktionen, die das Umrechnen zwischen den Koordinatensystemen übernehmen. Standardmäßig werden linkshändige Koordinatensysteme benutzt.

Beim Aufbau einer Szene wird man zuerst die Kamera im Weltkoordinatensystem positionieren und globale Lichtquellen setzen (es können auch effizientere lokale Lichtquellen, die nur eine Gruppe von Körpern beleuchten, verwendet werden). Variablen, die das Kameramodell und die eigentlich darzustellenden Objekte in ihrer Gesamtheit beeinflussen, werden *Optionen* genannt. Die Abbildung, die durch die Kamera vorgenommen wird, kann durch ver-

schiedene Optionen beeinflußt werden: Verwendung paralleler oder perspektivischer Projektion, Tiefenschärfe u.s.w. Neben den Optionen gibt es die *Attribute*. Attribute sind Variablen, die einzelne Objekte verändern oder bestimmen. Zu ihnen gehört die aktuelle Transformationsmatrix (Körper können verschoben, gedreht, skaliert und gedehnt werden), die Oberflächenfarbe und die Opazität. Optionen und Attribute besitzen implementierungsabhängige Vorbelegungen. Optionen dürfen nicht mehr geändert werden, wenn begonnen wurde, Objekte im Weltkoordinatensystem zu plazieren. Die Attribute hingegen können und sollen während des Plazierens verändert werden. Sie gelten vom Zeitpunkt ihrer Definition im aktuellen *Attributblock*. Funktionen zur Definition von Attributblöcken werden analog zu den **save** und **restore**, **gsave** und bf grestore PostScript Operatoren verwendet. Die Menge der aktuellen Attributwerte wird als Grafikzustand bezeichnet. Wie bei PostScript wird dieser Zustand auf einem Grafik-Stack gehalten. Durch geeignet geschachtelte Attributblöcke kann eine Objekthierarchie aufgebaut werden. Die Transformationen werden in hierarchisch gruppierten Körpern als Pfad vom Blatt zur obersten Hierarchie-Ebene ausgeführt. Eine Hierarchie von Grafikobjekten wird also ‘top down’ aufgebaut und ‘bottom up’ ausgegeben. Die Koordinaten einer Ebene sind auf diese Weise lokal zu denen der hierarchisch übergeordneten. Grafikobjekte können dadurch unabhängig voneinander zu größeren Objekten gruppiert werden. Neben den Attributen, die die Transformation und aktuelle Oberflächen-Farbe beeinflussen, kann u.a. die Bewegungsunschärfe eines Körpers (‘*Motion Blur’*) durch ein weiteres Attribut spezifiziert werden. Es existiert eine große Vielzahl von Optionen und Attributen. Sie werden in [Upstill89] ausführlich beschrieben.

In [PixSpec] sind die Spezifikation der Version 3.1 des RenderMan Interfaces und Anmerkungen zu dem ‘RenderMan Interface Bytestream’ zu finden. Der *RIB-Code* dient RenderMan Renderern standardmäßig als Eingabe. Ist das RenderMan-Interface durch eine Schnittstelle zu einer anderen Sprache gegeben, muß diese die gleiche Funktionalität bieten wie das *RIB-Interface*. Die Funktionen können einen Renderer direkt steuern oder den entsprechenden RIB-Code ausgeben. Der RIB-Code ist standardisiert und spiegelt die Möglichkeiten des RenderMan-Interfaces vollständig wider. Im Gegensatz zu PostScript stehen RIB keine Rechenfunktionen und höheren Programmiersprachenkonstrukte wie Schleifen, bedingte Anweisungen und Prozeduren zur Verfügung. Der RIB-Code ist zwar eine standardisierte Eingabe und damit eine einheitliche Szenenbeschreibung für Renderer, durch die Einschränkungen, die eine Implementation im Rendervorgang durch Weglassen der optionalen Fähigkeiten machen kann (z.B. keine benutzerdefinierten Shader, keine CSG, keine Bewegungsunschärfe), sehen die Resultate von verschiedenen Renderern u.U. gravierend unterschiedlich aus, auch wenn mit der gleichen Auflösung gearbeitet wird. Neben der Auflösung können Rechengeschwindigkeit und Speicherkapazität erheblichen Einfluß auf die Realisierung eines Renderers nehmen. Mit anderen Worten: Ein Bild, daß von einem Renderer auf einer Indigo Workstation von SiliconGraphics berechnet wurde, wird sich in den meisten Fällen erheblich von einem unterscheiden, das auf einem simplen PC gerendert wurde. Etwas anderes wäre schließlich aus rein technischen Gründen nicht zu erwarten. Qualitativ gleichwertige Renderer auf gleichwertigen Rechnern hingegen werden auch ähnliche Resultate erzielen. Das von dem RenderMan Standard keine Vorgaben an die endgültige Qualität der Ausgabe gemacht wurden, sodaß diese voneinander abweichen werden, kann nicht als Nachteil gewertet werden, weil das mit sinnvollem Aufwand Machbare auf den unterschiedlichen Rechnerplattformen erheblich variieren kann. Ein wichtiger Vorteil ist auch, daß die Spezifikation unabhängig von der Rendermethode ist und daß die gleiche Szenenbeschreibung, die

interaktiv mit Modellierungswerkzeugen, die auf Geschwindigkeit getrimmte Renderer verwenden, bearbeitet wurde, von anderen, langsameren Renderern als qualitativ hochwertige photorealistische Grafik berechnet werden kann. Es kann allerdings vorkommen, daß Szenenbeschreibungen für bestimmte Implementationen zu komplex werden, sodaß Speichermangel zu einem Abbrechen des Rendervorgangs führen kann. Durch die stetig besser (und billiger) werdende Hardware wird die Qualität und Kapazität der Renderer steigen können. Neuere Methoden, z.B. der gemischte Ansatz von Raytracing und Radiosity ([WalCG87]), könnten so durch eine entsprechend angepaßte RenderMan Implementation geboten werden.

Der interaktive *Quick RenderMan*, eine RenderMan Implementation auf dem NeXT, die das schnelle Rendern auf einem Bildschirm erlaubt, wurde um *Kontexte* erweitert. Ein Kontext ist eine Umgebung die alle Renderinginformationen beinhaltet, z.B. die Information über den verwendeten Renderer, den Grafik-Zustand und das Ausgabemedium ([PixQRM]). Zu einem Zeitpunkt kann in einer Applikation immer nur ein Kontext aktiv sein. Alle Render-Kommandos, die ausgeführt werden, beeinflussen nur den jeweils aktiven Kontext. Ein einziger Renderprozeß kann so quasiparallel, durch geeignetes Wechseln zwischen den Kontexten in verschiedenen Fenstern mehrere Szenen oder eine Szene mit unterschiedlichen Methoden (z.B. Drahtgitter und geglättet), verschiedenen Projektionen oder anderen Bildausschnitten rendern. Natürlich können, falls genügend Ressourcen zur Verfügung stehen, in einem Multitaskingsystem, wie dem des NeXTs, trotzdem mehrere Renderprozesse laufen. Sinnvollerweise kann neben einem hochpriorisierten interaktiven Renderer ein niedrigpriorisierter photorealistischer Renderer seine Arbeit verrichten. Durch das Einsetzen eines anderen Renderers oder eines anderen Rendermodus im aktuellen Kontext, können von demselben Programm unterschiedliche Ausgaben (z.B. Pixelbild oder RIB-Code) gewonnen werden.

2.2 3DKit

Das 3DKit ([NeXTDoc]) ist eine Sammlung von Objektklassen und Hilfsfunktionen, die das RenderMan-Interface in die NeXTSTEP-Umgebung einbindet. Auf dem NeXT stehen zwei Implementationen des RenderMan-Interfaces zur Verfügung. Zum einen der photorealistische RenderMan (`prman`) für das Erzeugen hochqualitativer Bilder als TIFF- oder PostScript-Dateien und zum anderen der interaktive RenderMan (*Quick RenderMan*) zum schnellen Rendern am Bildschirm mit geringerer Qualität und zur Ausgabe von RIB-Code auf einem Stream. Der photorealistische Renderer nimmt als Eingabe eine Datei mit dem RIB-Code einer Szenenbeschreibung und kann aus 3DKit heraus, auch zum verteilten Rendern auf mehreren Hosts, gestartet werden. Er kann auch direkt von einem Benutzer durch das Kommando `prman` aus einer Shell heraus aufgerufen werden. Der *Quick RenderMan* kann in Verbindung mit 3DKit verwendet werden.

Das Interface wird je nach Aufgabenbereich auf verschiedene Objektklassen verteilt. Es existieren Objektklassen für Kamera, hierarchisch gruppierbare Grafikobjekte (*Shapes*), Shader und Lichtquellen, sowie eine Klasse für das Rotieren am Bildschirm, zur bildlichen Repräsentation von RIB-Code, eine Filmkameraklasse für Animationen (durch das Rendern mehrerer Frames nacheinander), eine Klasse zum Starten des verteilten Renderns auf mehreren Hosts und eine zur Behandlung der Kontexte von *Quick RenderMan*. 3DKit reicht in Methoden der Kamera die

Möglichkeit des Renderers, Objekte in einem Rechteck von Bildschirmkoordinaten zu lokalisieren (*Picken*), weiter. Neben den Renderern selbst, ist auf dem NeXT noch der Compiler `shade` für die Übersetzung der ‘Shading Language’ vorhanden.

Zum verteilten Rendern wird das Ausgangsbild standardmäßig in gleichgroße, horizontale Streifen unterteilt. Die Methode zur Bildaufteilung kann überdefiniert werden. Es können, durch das Interface bedingt, nur zusammenhängende Rechteckteile gerendert werden. Die in [Adel94] vorgeschlagene Methode zur besseren Rechnerauslastung abwechselnd zeilenweise auf die Renderer zu verteilen, kann nicht verwirklicht werden. Bei der Standardaufteilung kann es bei ungleichmäßiger räumlicher Objektverteilung durchaus vorkommen, daß einige Rechner im Vergleich zu anderen sehr viel mehr rechnen müssen, wodurch die Gesamtrechenzeit unnötig verlängert wird. Vorkehrungen zur Wiederaufnahme des Renderns oder Neuverteilung der Aufgaben, falls eine Host abstürzt, wurden nicht getroffen.

2.3 Schichtenmodell der 3D Komponenten

Der Zusammenhang der einzelnen Systembestandteile kann anhand eines Schichtenmodells verdeutlicht werden.

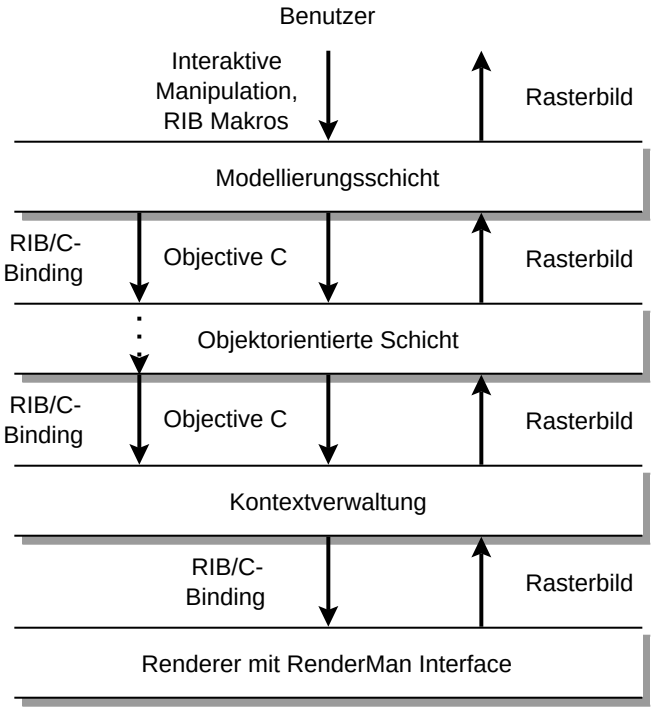

Abbildung 2.4: Schichtenmodell des 3D Systems

Die Modellierungsschicht soll dem Benutzer erlauben, mit Werkzeugen eine 3D-Szene zu

erstellen oder zu manipulieren. Die auf dem NeXT erhältlichen Tools **Perspective**, **3DReality** oder **intuitiv'3d** sowie das im Rahmen dieser Diplomarbeit erstellte Programm **ModelMan** sind in dieser Schicht angesiedelt. Der Benutzer kann mit den Werkzeugen dieser Schicht beispielsweise über ein grafisches Interface kommunizieren. RIB-Dateien können normalerweise als Makros eingebunden werden. Programmierer können die oberste Schicht umgehen und Szenen in Objective C für die objektorientierte Schicht programmieren. Die Renderer der untersten Schicht können auch direkt von einer UNIX-Shell gestartet werden. Verteiltes Rendern ist nur über eine der höheren Schichten möglich.

Unterhalb der Modellierungsschicht liegt die objektorientierte Schicht. Deren Funktionalität ist durch das 3DKit gegeben. Das RenderMan-Interface ist nicht vollständig in Objekte und Methoden der Objective C Klassenhierarchie abgebildet. Vor allem in der **renderSelf:** Methode von abgeleiteten **N3DShape** Objekten werden RenderMan-Interface Aufrufe direkt angegeben. Für die Verwendung von RIB-Makros steht keine vordefinierte Klasse zur Verfügung. Die Verbindung der Modellierungsschicht mit der folgenden Kontextverwaltung ist durch das **N3DContextManager** Objekt, das RenderMan C-Binding und den durchgereichten RIB-Makros realisiert.

Die Kontextverwaltung ist, wie der Name schon sagt, für die Renderkontexte des *Quick RenderMan*s zuständig. Eine Instanz der Klasse **N3DContextManager** verwaltet alle Kontexte einer Applikation. Alle Interface-Aufrufe beeinflussen nur den jeweils aktuellen Kontext. Die Interface-Aufrufe werden an das Frontend des *Quick RenderMan* geleitet, der entweder das Rendern direkt in ein Fenster oder die Ausgabe von RIB-Code einleitet. Auch die Steuerung für das verteilte Rendern kann in dieser Schicht angesiedelt werden. Zum verteilten Rendern werden zu jeder der ausgewählten Hosts eine oder mehrere RIB-Dateien gesendet, die dort von dem photorealistischen Renderer bearbeitet werden. Bei Animationen werden komplette Rahmen in nicht vorhersehbarer Reihenfolge gerendert, bei Einzelbildern standardmäßig Streifen, die durch Crop-Bereiche definiert werden. Einzelbilder werden nach dem Rendern wieder zu einer Gesamtgrafik zusammengestellt und einem eingesetzten 'Camera Delegate' (**N3DCamera**: **setDelegate:** und **delegate**) als Stream zur Verfügung gestellt.

Die Renderer der untersten Schicht bekommen ihre Eingabe entweder über das C-Interface (*Quick RenderMan*) oder als RIB-Datei (`prman`). Der interaktive Renderer kann seine Ausgabe direkt in einen Fensterpuffer schreiben, der photorealistische Renderer schreibt in eine Datei, die von den darüberliegenden Schichten an die Applikation als Stream weitergereicht wird. Momentan werden vom photorealistischen Renderer nur RGBA TIFF-Dateien unterstützt, in der objektorientierten Schicht können auch PostScript Dateien verwendet werden.

2.4 Einschränkungen der Implementationen auf dem NeXT

Die beiden Renderer des NeXTs realisieren nur eine beschränkte Anzahl der Fähigkeiten des Interfaces. In [PixQRM] ist der aktuelle Implementierungsstand des *Quick RenderMan* und Abweichungen zur Version 3.1 des Interfaces dokumentiert. Der auf Geschwindigkeit getrimmte *Quick RenderMan* besitzt keine der optionalen Fähigkeiten für das interaktive Rendern, kann aber zumeist den entsprechenden RIB-Code ausgeben, der vom `prman` weiterverarbeitet werden kann. Leider ließ sich keine Beschreibung des Implementierungsstands des photorealis-

tischen Renderers auftreiben. Nur der `prman` erlaubt das Erzeugen von ‘Texture Maps’ aus TIFF-Dateien, keiner der Renderer erlaubt das Erstellen von ‘Bump Maps’. Es gelang nicht, mit einem der beiden Renderer ein ‘Shadow Map’ zu erstellen — Schatten können erzeugt werden, wenn ein Renderer das Anlegen von ‘Depth Maps’ in Bezug zu einer Lichtquelle erlaubt. Ein allgemeines ‘Texture Mapping’ kann mit dem (*Quick RenderMan*) des NeXT s nicht eingesetzt werden. Die folgende Tabelle zeigt den derzeitigen Stand der Implementierung:

Optional Capabilities	`prman`	*Quick RenderMan*
Solid Modeling	ja	nein
Trim Curves	ja	nein
Level of Detail	ja	nein
Motion Blur	nein	nein
Depth of Field	ja	nein
Programmable Shading	ja	nein
Special Camera Projections	nein	nein
Deformations	nein	nein
Displacements	ja	nein
Spectral Colors	nein	nein
Texture Mapping	ja	nein
Environment Mapping	ja	nein
Bump Mapping	nein	nein
Shadow Depth Mapping	eingeschr.	nein
Ray Tracing	nein	nein
Radiosity	nein	nein
Area Lightsources	ja	nein

Das ‘Environment Mapping’ mit Hilfe des RIB-Kommandos **MakeCubeFaceEnvironment** ist sehr speicheraufwendig, weil sechs ‘Texture Maps’ pro ‘Environment Map’ verwendet werden müssen. ‘Area Lightsources’ ist eine optionale Fähigkeit, die das Verbinden einer beliebigen Oberfläche mit einer Lichtquelle erlaubt. Die Ausleuchtung von Leuchtstoffröhren kann auf diese Weise durch einen Zylinder simuliert werden. Neben den meisten der optionalen Fähigkeiten fehlt den Renderern des NeXTs bedauerlicherweise die Fähigkeit, Objekte zu definieren. Die gegebene Alternative der RIB-Makros ist nicht befriedigend, da sich vor allem beim interaktiven Rendern durch die Verwendung der RIB-Repräsentation erhebliche Geschwindigkeitsverluste bemerkbar machen.

Mit dem derzeitigen Entwicklungsstand des 3DKit können mit RenderMan Konstrukten wie dem ‘Solid’ Block nicht ohne weitere Vorkehrungen mehrere Instanzen des **N3DShape** Objekts innerhalb einer Shape-Hierarchie zusammengefasst werden. RenderMan Aufrufe werden üblicherweise in **N3DShape** Objekten nur in der **renderSelf:** Methode gemacht. Diese werden vom System immer durch ein **RiTransformBegin()** und ein **RiTransformEnd()** gekapselt. Das **RiTransformEnd()** wird zwar erst aufgerufen, wenn alle **render:** Methoden der hierarchisch untergeordneten Objekte ausgeführt wurden, der Benutzer kann jedoch direkt vor dem schließenden **RiTransformEnd()** keine eigenen Interface-Aufrufe mehr einbringen. Alle in einer **renderSelf:** Methode geöffneten Blöcke müssen also in der gleichen Methode wieder geschlossen

werden, damit sich die Blöcke nicht überschneiden. Mit einer geeigneten Überdefinition der Traversierung der Objekthierarchie **render:**, wie das Aufrufen einer gesonderten Methode zum Schließen des Attributblocks, oder dem direkten Aufruf von Interface-Routinen können allerdings entsprechende Möglichkeiten geschaffen werden.

Bei Tests mit unterschiedlichen Modellen hat sich herausgestellt, daß der photorealistische Renderer in einigen Fällen Schwierigkeiten mit Glanzpunkten hat, wenn der Gesamtwert der Lichtintensitäten den Wert 1 überschreitet. In der Dokumentation ist vermerkt, daß beim Rendern als PostScript-Datei immer ein schwarzer Hintergrund ausgegeben wird.

2.5 Die Modellierung

Unter Modellierung versteht man das zumeist interaktive Bearbeiten von geometrischen Daten und von Eigenschaften von Oberflächen und Körpern, kurz das Erstellen von Szenen. Die Modelle dienen einem Renderer als Eingabe. RenderMan definiert die Schnittstelle zwischen Modellierung und Renderern mit Hilfe des RIB-Dateiformats. Im allgemeinen ist es nützlich, geometrische Objekte oder Teilobjekte in Modellbibliotheken zusammenzufassen. Vorteilhaft ist sicherlich, die Modelldaten der Bibliothek in genormten Koordinaten abzulegen, z.B. in einer Rechteckhülle mit Seitenlänge 1. Eine andere Möglichkeit ist, zusätzlich zum Modell die kleinste umgebende dreidimensionale Rechteckhülle (*Bounding Box*) zu speichern, deren Daten durch das Modellierungswerkzeug ausgewertet werden können — trotzdem sollten die Modelle ähnliche oder aufeinander abgestimmte Größen besitzen. Die Modelle können beim Plazieren vom Benutzer auf ihre konkrete Größe skaliert werden. Sind die Modellgrößen zu unterschiedlich, kann es, wie man sich leicht vorstellen kann, zu Problemen kommen. Einen Rahmen für die Erstellung von Modellen als RIB-Dateien bilden die RIB-Konventionen, die entfernt an die Konventionen von PostScript- und EPS-Dateien erinnern ([PixSpec]).

Einem potentiellen Benutzer stehen mehrere Möglichkeiten zur Modellerstellung zur Verfügung. Am einfachsten ist häufig die Verwendung eines fertigen Modellers, der eine RIB-Ausgabe direkt oder indirekt, mittels eines Hilfsprogramms zur Umwandlung des eigenen Formats in eine gleichwertige RIB-Ausgabe, bietet. Nachteil dieser Möglichkeit ist, daß längst nicht alle Modeller den vollen Umfang des Interfaces ausschöpfen. Vor allem seien die benutzerdefinierten Shader, das prozedurale Modellieren, eigene Filterfunktionen für das Antialiasing, ‘Motion Blur’, der Detaillierungsgrad und die Verwendungsmöglichkeit der vielfältigen Spline-Oberflächen genannt. Die Bilder müssen also im allgemeinen nachbearbeitet werden. Es existiert zumindest ein Programm (ShowPlace$^{\mathrm{TM}}$ von Pixar [Uter93a]), das es ermöglicht, mit Hilfe von Shadern die Oberflächendaten einer Modellerausgabe um verschiedene Darstellungsformen zu erweitern.

Einen anderen gangbaren Weg bietet die Programmierung von Bildern unter der Verwendung des RenderMan C-Interfaces. Diese Möglichkeit verlangt dem Benutzer zwar mehr Arbeit ab, die Möglichkeit der direkten Verwendung des gesamten Interfaces und die algorithmische Modellerstellung bieten dafür eine Vielzahl interessanter Möglichkeiten (z.B. L-Systeme für Pflanzen, Fraktale, 3D-Plots). Die Programmierung schließt nicht aus, komplexe geometrische Daten (z.B. größere Gitternetze) mittels Hilsprogrammen zu erstellen und die Plazierung der Objekte mit Hilfe vom *Quick RenderMan* am Bildschirm vorzunehmen oder auf Modellbiblio-

theken zuzugreifen. Prozedurale Modelle können als C-Funktionen verwirklicht werden. Durch Parametrisierung der Funktionen, die die Modelle generieren, ist eine hohe Flexibilität der Modellerzeugung erreichbar.

Da der RIB-Code anders als PostScript keine höheren Programmiersprachenkonstrukte anbietet, ist die direkte Programmierung von RIB-Dateien eher nicht anzuraten. Möglich ist aber, Modelle als RIB-Dateien zu generieren und innerhalb einer Bibliothek anzubieten. Für interaktive Renderer können daraus jedoch sich unangenehm bemerkbar machende Geschwindigkeitsverluste erwachsen. Werden die Modelle mit möglichst wenigen Attributen ausgestattet, ist eine flexible Bilderzeugung durch nachträgliche Attributierung möglich. 3DKit und *Quick RenderMan* bieten Möglichkeiten, RIB-Dateien zu schreiben, zu laden und in eine Gesamtgrafik als RIB-Makros zu integrieren. Der Vorteil der Verwendung von RIB-Dateien für eine Objektbibliothek ist vor allem die Unabhängigkeit von einer Programmiersprache und einer bestimmten Maschine. Sowohl RIB-Code, als auch C-Interface sind von der Interpretierbarkeit her unabhängig von der aktuell verwendeten Implementation des Interfaces. Ein kleiner Wermutstropfen bei der Verwendung von RIB-Dateien für die Archivierung von Modellen ist, daß auf diese Weise keine prozeduralen Modelle (in deren Berechnung der aktuelle Detaillierungsgrad eingeht) möglich sind.

Bei der Modellierung einer Szene ist es sinnvoll, diese zuerst in einzelne voneinander unabhängige Körper zu zerlegen, diese wieder in ihre Teilkörper, bis man einzelne, sinnvoll nicht weiter aufteilbare, durch Interface-Funktionen modellierbare Primitive erhält. Zusätzlich sollte man sich überlegen, ob statt einer geometrischen Aufteilung die Verwendung eines Shaders (z.B. eines ‘Displacement Shaders’ oder ‘Bump Maps’) sinnvoller ist. Eine rauhe Wand wird man nicht durch eine Vielzahl kleiner Dreiecke sondern besser durch einen speziellen Shader modellieren. Beim hierarchischen Zusammenbauen der Teilkörper kann darauf geachtet werden, wiederverwendbare Teilkörper zu erhalten, die einer Objektbibliothek zugeführt werden können. Für komplexe Körper kann man versuchen, für kleinere Detaillierungsgrade durch das Weglassen von Details und Vereinfachen von Oberflächen einfachere Modelle zu finden. Stehende, von der Kamera weit entfernte Menschen können z.B. durch eine Kugel und einen Zylinder modelliert werden. Interessant ist die Verwendung von Fraktalen für prozedurale Modelle. Durch die direkte Einbeziehung des Detaillierungsgrades in die Iterationstiefe der Berechnung können fast beliebig komplexe Körper gebildet werden. Allerdings muß bei großen Körpern bedacht werden, daß der Detaillierungsgrad prinzipiell innerhalb eines einzigen Körpers variieren kann und auch Köper mit hohen Detaillierungsgrad weit entfernt sein können (z.B. Berge am Horizont). Ohne Kenntnis der aktuellen Lage einer Oberfläche relativ zur Kamera kann man diesen Sachverhalt nicht in die Modellierung einbeziehen. Einige RenderMan Implementationen können beim Rendern zwischen Modellen interpolieren (**RiDetailRange()**). Es gehört allerdings etwas Erfahrung dazu, die nötige Detaillierung einer Oberfläche abzuschätzen. Gegebenenfalls muß man sich Testbilder ansehen und die Detaillierung im Modell entsprechend anpassen. Das Auswerten von Testbildern macht insbesondere bei Animationen, für die sehr viele Einzelbilder (24–25 Bilder/Sek.) berechnet werden müssen, Sinn. Bei Animationen kommt noch hinzu, daß bewegte Oberflächen nicht so detailreich gerendert zu werden brauchen wie stehende. Das Auge nimmt an bewegten Körpern feine Details nicht so gut wahr wie an stehenden. Wird ‘Motion Blur’ verwendet, verwischen die Details, sodaß ihre Berechnung unter Umständen unnötigen Aufwand bedeutet. Auf dem NeXT wird für Animationen nur das nacheinander Rendern von

Frames durch das **N3DMovieCamera** Objekt automatisiert. Bewegungsskripte für die Steuerung von Kamerafahrten, Objektbewegungen und -veränderungen müssen selbst implementiert werden — sie sind auch mehr der Modellierungsschicht zuzuordnen. Es sei in diesem Zusammenhang darauf hingewiesen, daß der NeXT nicht mit einer hochgezüchteten Grafikmaschine von SilconGraphics, dem Pixarrechner oder ähnlichen verglichen werden kann. Der Komplexität von photorealischen Grafiken sind deshalb gewisse Grenzen gesetzt. Insbesondere Szenen mit mehreren Körpern wirken aufgrund der fehlenden Schatten nicht. Globale Beleuchtungsmodelle können nicht verwendet werden. Das Rendern von einzelnen Körpern (z.B. Schachfiguren), mathematischen dreidimensionalen Grafiken und anderen nicht zu komplexen Modellen kann aber mit durchaus ansprechenden Ergebnis durchgeführt werden.

Welche Möglichkeiten der Geometriedefinition bietet das RenderMan-Interface nun eigentlich? Eine Möglichkeit ist die Definition von Oberflächen anhand einer Anzahl von Quadriken: Kugel **RiSphere()**, Kegel **RiCone()**, Zylinder **RiCylinder()**, Paraboloid **RiParaboloid()**, Hyperboloid **RiHyperboloid()** und Torus **RiTorus()**. Auch ein Diskus **RiDiskus()** kann verwendet werden. Es werden nicht die Körper sondern nur die Mantelflächen generiert[2]. Mit Hilfe der Attribute kann die Generierung der Fläche je nach Typ des Drehkörpers gesteuert werden. Der Diskus eignet sich gut als Deckel für unvollständige Mantelflächen. Durch Verändern der Transformationsmatrix können auch Mantelfächen, wie die eines Ellipsoids, erzeugt werden. Quadriken die nicht durch einen einzigen Drehkörper erzeugt werden können (hyperbolisches Paraboloid, zweischaliges Hyperboloid) können nicht direkt durch RenderMan Befehle definiert werden.

Weitere Interface-Funktionen stehen zur Generierung von einfachen (konvexen) und komplexen (konkav mit Löchern) Polygonen (**RiPolygon()**, **RiGeneralPolygon()**) und Polyhedren (**RiPointsPolygon()**, **RiPointsGeneralPolygon()**) zur Verfügung. Glatte, gebogene Oberflächen können durch Gitter von bilinearen oder bikubischen 'Patches' (2×2, bzw. 4×4 Gitternetzen) erzeugt werden. Der Typ der Spline-Interpolierung bzw. -Approximierung der bikubischen Patches kann definiert werden. Die Annäherung von gebogenen Körpern durch Polygonnetze kann durch die Verwendung von Spline-Oberflächen der Qualität des Renderers überlassen werden. Durch die Verwendung einer steuernden Option kann eine gröbere Aufteilung der Patches und damit ein schnelleres Rendern mit geringerer Qualität erreicht werden. In Verbindung mit dem Detaillierungsgrad kann es, wie schon erwähnt sehr sinnvoll sein, verschiedene Modelle eines Körpers bereitzustellen: Einfache, schnell zu rendernde für Oberflächen einer kleinen Pixelfläche und komplexere, langsamer zu rendernde Oberflächen für die, die eine größere Fläche einnehmen und damit einen höheren Detaillierungsgrad besitzen. Durch eine geschickte Wahl der Modelle kann so mit Hilfe des Detaillierungsgrads die Geschwindigkeit des Renderns erhöht werden, ohne daß sich ein Qualitätsverlust bemerkbar macht. Eine weitere Geschwindigkeitssteigerung kann erreicht werden, wenn die Rückseiten der Oberflächen nicht in den Rendervorgang einbezogen werden und rechenintensive Funktionen wie Bewegungs- und Tiefenunschärfe global abgestellt werden. Da es RenderMan Implementationen oft erlauben, rechenintensive Qualitätsmerkmale global abzuschalten, können die Oberflächen lokal ohne Rücksicht auf die Rechenzeit implementiert werden.

[2] Innerhalb von **RiSolidBegin()**, **RiSolidEnd()** Blöcken können prinzipiell auch Körper definiert werden. Der *Quick RenderMan* des NeXT's unterstützt diese Möglichkeit aber nicht.

Das Zusammensetzen der Oberflächen geschieht durch ein hierarchisches Zusammenfügen der einzelnen Flächen in verschachtelten Attributblöcken. Immer wiederkehrende Teilflächen können prinzipiell durch Objektblöcke definiert werden. Die Flächenbeschreibung braucht dann nicht wiederholt dem Renderer übertragen werden (**RiBeginObject()**, **RiEndObject()**, **RiObjectInstance**). Die Renderer des NeXT verwenden hierfür allerdings nur RIB-Makros.

Durch die Unterstützung einer CSG Modellierung durch das Interface können Körper mit Hilfe der Mengenoperatoren Vereinigung, Durchschnitt und Differenz aus elementaren Körpern innerhalb von 'Solid' Blöcken definiert werden. Durch entsprechende Maßnahmen (s. [Upstill89], CSG composites, S. 128ff) können elementare Oberflächen wie unvollständige Drehkörper auch als Körper dargestellt werden. Die CSG Modellierung optional und wird auf dem NeXT nicht von dem *Quick RenderMan*, wohl aber von dem `prman` unterstützt.

2.6 Programmstruktur

2.6.1 Die Koordinatensysteme und das Rastern

Die Koordinatensysteme von RenderMan lassen sich grob in drei Klassen aufteilen:

- Parametrische Koordinaten
- Dreidimensionale Koordinaten
- Zweidimensionale Bildkoordinaten und Rasterkoordinaten

Je nach Einstellung kann mit links- oder mit rechtshändigen Koordinaten gerechnet werden. Auf dem NeXT wird derzeit nur das linkshändige Koordinatensystem unterstützt.

Die parametrischen (u,v)-Koordinaten sind an Oberflächen gebunden. Den Verlauf der linkshändigen (u,v)-Koordinaten kann man durch die Linke-Hand-Regel herausbekommen, indem man den Daumen in Richtung der Oberflächennormalen zeigen lässt. Der Zeigefinger weist in u-Richtung, der Mittelfinger in v-Richtung. Die Rotation der parametrischen (u,v)-Ebene ist vom jeweiligen Oberflächentyp abhängig. Die Parameter laufen zur Erzeugung der Oberfläche normalerweise von 0 bis 1. Die parametrischen Koordinaten werden häufig in Zusammenhang mit den Oberflächentexturen verwendet. Die Texturkoordinaten sind linear von den parametrischen Koordinaten abhängig. Zusätzlich können die Texturkoordinaten auf der Oberfläche periodisch aneinandergereiht werden (*texture wrapping*). Die Interface-Funktion **RiTextureCoordinates()** dient zur Definition des Texturkoordinatensystems. Mit der Funktion **RiMakeTexture()** kann aus einer Bitmap-Datei eine 'Texture map'-Datei erstellt werden. Texturkoordinaten werden auch für 'Bump Mapping' und 'Environment Mapping' verwendet. In programmierten Oberflächen-Shadern können die parametrischen Koordinaten einer Oberfläche zur Berechnung von Mustern verwendet werden.

Eine Oberfläche wird zunächst in ihrem eigenen dreidimensionalen (x,y,z)-Objektkoordinatensystem definiert. Dieses ist durch affine Abbildungen in das hierarchisch übergeordnete Koordinatensystem abgebildet. Das hierarchisch höchste Koordinatensystem, das Weltkoordinatensystem, wird wiederum in das dreidimensionale (s,t,u)-Kamerakoordinatensystem abgebildet.

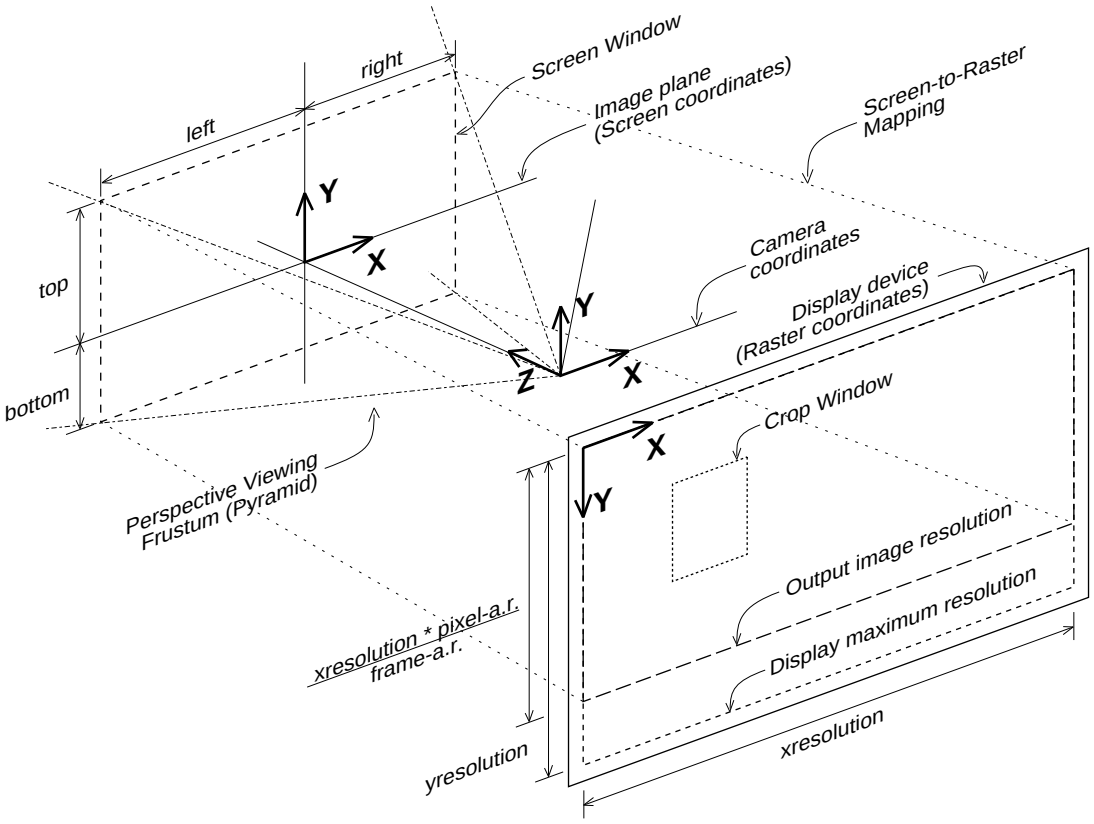

Abbildung 2.5: Die Abbildung auf das Bildrechteck (Camera to Raster Projection Geometry, aus [PixSpec])

Ein Ausschnitt aus dem Kamerakoordinatensystem, der '*View Cone*' wird auf ein normalisiertes zweidimensionales Bildkoordinatensystem abgebildet (s. 2.5). Ein Ausschnitt hiervon (*crop* Bereich) wird in Pixelkoordinaten überführt — diese wiederum werden durch Filter und andere Bildberabeitungsalgorithmen auf die eigentlichen Rasterkoordinaten (Abb. 2.6) abgebildet.

Die *Imaging Pipline* stellt eine qualitätsverbessernde Weiterführung des geometrischen Kameramodells auf die Pixelebene dar. Nach dem der Inhalt eines Pixels[3] durch einen Hidden Surface Algorithmus bestimmt wurde, verzweigt die Imaging Pipeline in zwei Pfade zu der Berechnung der Farbwerte und der Tiefenwerte der Bildpunkte. Die Farbwerte durchlaufen zuerst den Sampler, der für ein Pixel einen einzigen Farbwert bestimmt. Die Samplerate und zugehörende Filteralgorithmen können durch Interface-Funktionen **RiPixelSamples()**, **RiPixelVariance()** und **RiPixelFilter()** bestimmt werden. Die Filter dienen der Vermeidung von 'Treppchenstufen' oder 'Jaggies', die durch zu geringe Sampleraten entstehen. Nach der ersten Stufe

[3] Pixel werden als kleine aneinandergrenzende Rechtecke in der Bildebene gesehen.

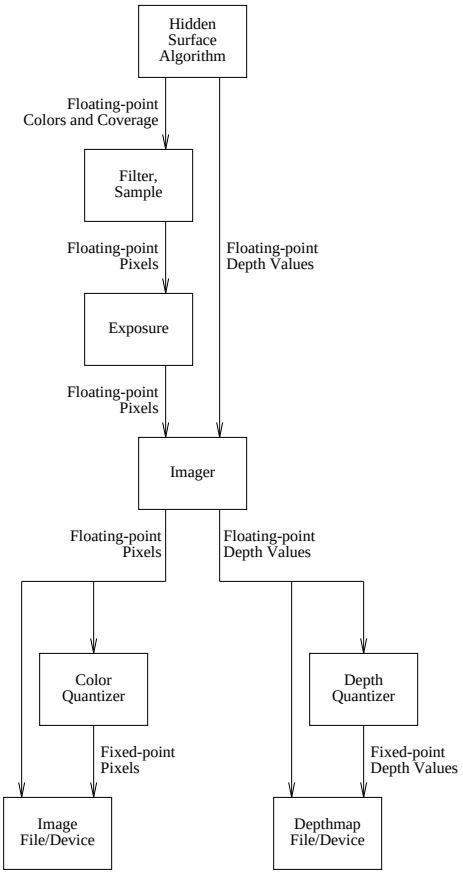

Abbildung 2.6: Die Berechnung des Rasterbildes (Imaging Pipeline, aus [Upstill89], Raster Output)

ist jedem Pixel eine eindeutige Farbe zugeordnet. Anschließend wird die Helligkeit der Farbe an die physikalischen Gegebenheiten des aktuellen Ausgabegeräts angepaßt. Die Helligkeitskorrektur kann durch die **RiExposure()** Funktion eingestellt werden. In der folgenden Stufe *Imager* kann ein vom Benutzer definierter 'Imager Shader' eingesetzt werden. Mit ihm kann der eingehende Farb- oder Tiefenwert beliebig verändert werden. Eine allgemeine Farbkorrektur ist hier möglich. Bisher wurden für die Werte Fließkommazahlen verwendet. Sie müssen üblicherweise in diskrete Werte umgewandelt werden, die dem Auflösungsvermögen der Ausgabe (z.B. 24Bit Pixeldateien) entsprechen. Diese Quantisierung kann in der nächsten Stufe vorgenommen werden. Da durch die Quantisierung ungewünschte Falschkonturen entstehen, werden die Werte zur Qualitätsverbesserung zusätzlich *gedithert*. Mit der Funktion **RiQuantize()** kann der Vorgang gesteuert werden.

Neben den in der Abbildung 2.6 gezeigten, auf räumliche Koordinaten beschränkten Filtern, kann auch ein Algorithmus für das 'Motion Blur', gegen das stroboskopartige *temporale Alia-*

sing eingesetzt werden. Ähnlich wie das ‘Motion Blur’ erzeugt die Steuerung der Tiefenschärfe **RiDepthOfField()** der Kamera ein Verwischen bestimmter Bildbereiche. Nicht direkt zum Kameramodell gehörend, kann auch der Detaillierungsgrad bis auf Pixelebene wirken (wenn zwischen mehreren Modellen interpoliert wird).

2.6.2 Die Blockstruktur von RenderMan

Das RenderMan-Interface verlangt, daß die Szenenbeschreibung in einer, die einzelnen Grafik-Objekte abkapselnden, Blockstruktur dargestellt wird. Die 3DKit Objektstruktur bildet die Attributblockstruktur in eine hierarchische Objektstruktur ab.

Die Interface-Definition kennt folgende Blocktypen:

- Programmblock **RiBegin()**, **RiEnd()**
- Animationsrahmen Blöcke (Frames) **RiFrameBegin()**, **RiFrameEnd()**
- Weltblöcke **RiWorldBegin()**, **RiWorldEnd()**
- Attributblöcke **RiAttributBegin()**, **RiAttributEnd()**
- Transformationsblöcke **RiTransformBegin()**, **RiTransformEnd()**
- CSG Blöcke **RiSolidBegin()**, **RiSolidEnd()**
- Objektdefinitions Blöcke **RiObjectBegin()**, **RiObjectEnd()**
- Bewegungsunschärfe Blöcke **RiMotionBegin()**, **RiMotionEnd()**

Schachtelreihenfolge und Schachtelbarkeit von Blocktypen sind durch das Interface in gewissen Grenzen eingeschränkt. Den obersten Block bildet immer ein einziger Programmblock (**RiBegin()**, **RiEnd()**). Dieser enthält einen oder mehrere Frames (**RiFrameBegin()**, **RiFrameEnd()**). Diese wiederum eine Sequenz von Weltblöcken (**RiWorldBegin()**, **RiWorldEnd()**). Existiert nur ein Frame, braucht dieser nicht extra durch **RiFrameBegin()** und **RiFrameEnd()** geklammert zu werden. Programm-, Welt- und Frameblöcke dürfen nicht anders als in der angegebenen Weise untereinander und nicht ineinander verschachtelt werden. Vor dem Öffnen des Weltblocks müssen sämtliche Optionen gesetzt sein. Innerhalb des Weltblocks dürfen nur Objekte definiert und Attribute verändert werden. Dazu können prinzipiell beliebig viele Attribut-, Transformations-, ‘Solid’- und ‘Motion Blur’-Blöcke beliebig tief geschachtelt werden. Die Blöcke dürfen sich, wie von Programmiersprachen gewohnt, nicht überlappen. Eine hierarchische Modellstruktur kann durch ineinandergeschachtelte Attribut- und Transformationsblöcke erreicht werden. Im Gegensatz zu den Attributblöcken speichern die Transformationsblöcke nicht den gesamten Attributzustand sondern nur die aktuelle Transformationsmatrix. In [Upstill89], Seite 56–58 wird in einer Tabelle zusammengefasst, in welchen Blöcken welche Interface-Aufrufe gemacht werden dürfen.

Durch die Blockstruktur bedingt haben alle RenderMan Anwendungen im großen und ganzen den gleichen Aufbau (s.a. [Upstill89], Seite 55):

```
Der Bereich ausserhalb des Programmblocks darf nur RiErrorHandler(),
RiDeclare() Aufrufe enthalten und Makros definieren.

RiBegin();

    Globale Optionen setzen: z.B. Bildausgabeoptionen, Kameraoptionen.
    Der Programmblock darf bereits alle Interface-Aufrufe ausser
    Objektdefinitionen (also nicht: RiPolygon(), RiSphere(), Solid Bloecke
    u.s.w.) enthalten.


    RiBeginFrame(1);

        // Frame-lokale Optionen und Attribute, globale Optionen duerfen
        // ueberdefiniert werden. Es duerfen noch keine Objekte definiert
        // werden.

        // Pixelmaps duerfen nur ausserhalb des Weltblocks generiert werden
        // RiMakeTexture(), RiMakeBump(), RiMakeLatLongEnvironment(),
        // RiMakeCubeFaceEnvironment() und RiMakeShadow(),

        RiWorldBegin();

            // Falls nur ein Frame vorhanden ist, darf der Weltblock auch
            // direkt unter dem Programmblock definiert sein.

            // Hier werden nur noch Attribute geaendert und Objekte
            // definiert. Optionen duerfen nicht mehr veraendert werden.

            // Ab dem Weltblock duerfen Solid Bloecke definiert werden.
            // Attribut- und Transformationsbloecke koennen beliebig
            // geschachtelt werden. Z.B.:

            // Beispiel einer Attributschachtelung
            RiAttributeBegin();
                RiTransformBegin();
                RiTransformEnd();

                RiTransformBegin();
                    RiAttributeBegin();
                    RiAttributeEnd();
                RiTransformEnd();
                RiAttributeBegin();
                RiAttributeEnd();
            RiAttributeEnd();

            RiTransformBeginBegin();
            RiTransformEnd();

        RiWorldEnd();

        // Nach einem RiWorldEnd() ist ein Einzelbild komplett und kann
        // gerendert werden.
```

```
        // Optionen und Attribute duerfen zwischen Weltbloecken
        // ueberdefiniert werden.

        // Weitere Welt Bloecke...

    RiEndFrame();

    // Zwischen Frames duerfen Optionen und Attribute ueberdefiniert werden.

    // Weitere Frame Bloecke...

RiEnd();
```

Die ‘Solid’ Blöcke zur CSG Modellierung (**RiSolidBegin()**, **RiSolidEnd()**) benötigen eine spezielle innere Struktur. Die am tiefsten geschachtelten Blöcke müssen immer vom Typ **RI_PRIMITIVE** sein, also elementar im Sinne der CSG. Dieser Elementarblock darf alle möglichen geometrischen Primitive enthalten. Die Oberflächen müssen so versiegelt werden, daß sie ein definiertes Inneres und Äußeres besitzen. Nach außen können diese Blöcke dann rekursiv in **RI_UNION** (Vereinigung), **RI_DIFFERENCE** (Differenz) und **RI_INTERSECTION** (Durchschnitt) ‘Solid’ Blöcke geschachtelt werden. Zwischen den ‘Solid’ Blöcken dürfen keine weiteren Obeflächen definiert werden.

Beispiel einer CSG Block Schachtelung:

```
RiSolidBegin(RI_DIFFERENCE); // Differenz d von i(u(A, B, C), D) und E

    RiSolidBegin(RI_INTERSECTION); // Schnitt i von u(A, B, C) und D
        RiSolidBegin(RI_UNION); // Vereinigung u von A und B und C
            RiSolidBegin(RI_PRIMITIVE);
                // Geschlossene Oberflaechen A
            RiSolidEnd(); // RI_PRIMITIVE

// zwischen den Bloecken duerfen Befehle wie RiRotate() stehen

            RiSolidBegin(RI_PRIMITIVE);
                // Geschlossene Oberflaechen B
            RiSolidEnd(); // RI_PRIMITIVE

            RiSolidBegin(RI_PRIMITIVE);
                // Geschlossene Oberflaechen C
            RiSolidEnd(); // RI_PRIMITIVE
        RiSolidEnd(); // RI_UNION

        RiSolidBegin(RI_PRIMITIVE);
            // Geschlossene Oberflaechen D
        RiSolidEnd(); // RI_PRIMITIVE
    RiSolidEnd(); // RI_INTERSECTION

    RiSolidBegin(RI_PRIMITIVE);
            // Geschlossene Oberflaechen E
    RiSolidEnd(); // RI_PRIMITIVE

RiSolidEnd(); // RI_DIFFERENCE
```

2.6.3 Vergleich der Objektstruktur von 3DKit mit der Blockstruktur

Damit die RenderMan Blockstruktur mit der objektorientierten AppKit-Struktur vom NeXT besser zusammengeht, wurde sie in die Objekthierarchie von 3DKit abgebildet. Die Funktionalität des Interfaces wurde jedoch nicht komplett umgesetzt, sodaß auch Funktionen des C-Bindings verwendet werden müssen. Für Transformationen und Änderungen der Rechteckhüllen sollten allerdings immer 3DKits Objektmethoden verwendet werden, da diese neben den nötigen Interface-Aufrufen zusätzlich über die Operationen Buch führen, um die aktuellen Ausmaße eines Objekts zu halten. Für Testzwecke oder aus Geschwindigkeitsgründen ist es möglich, statt der gesamten Oberfläche nur die Rechteckhüllen zu rendern.

Aus der Zusammenfassung der 3DKit Klassen in [NeXTDoc] ('3D Graphics Kit Classes') kann man entnehmen, daß die **N3DContextManager**-, **N3DCamera**- und **N3DShape**-Klassen die Blockstruktur des Interfaces realisieren. Der Programmblock wird durch **N3DContextManager**, der Frameblock und der Weltblock wird durch **N3DCamera**, Attribut- und Transformationsblöcke werden durch **N3DShape** implementiert. Die Verschachtelung der Attributblöcke kann durch eine hierarchische Anordnung von **N3DShape** Instanzen erreicht werden (Descendant, Peer). Die Darstellungsmethode **renderSelf:** von **N3DShape** soll die Interface-Aufrufe, mit denen ein Objekt dargestellt wird, enthalten. Der Methodenaufruf ist in einem Transformationsblock gekapselt. Sequenzen von Frame- bzw. Weltblöcken können durch wiederholtes Ausführen der **drawSelf::** oder **display** Methode einer **N3DMovieCamera** bzw. **N3DCamera** Instanz erreicht werden. 'Solid'- und 'Motion Blur'-Blöcke werden leider nicht explizit durch Objektklassen realisiert. Durch die Verwendung von Transformationsblöcken zur Kapselung der **renderSelf:** der **N3DShape**s, müssen, wenn nicht weitere Vorkehrungen getroffen werden, alle neuen Blöcke, die in dieser Methode geöffnet werden, dort auch wieder geschlossen werden. Andernfalls würden sich die Blöcke überschneiden.

Die Shape-Hierarchie, ein Baum mit dem ausgezeichneten Wurzelknoten 'Weltobjekt', wird top-down von links nach rechts traversiert. Die Objekte eines 'Unterbaums' liegen im gleichen Attributblock. Die Attribute einer hierarchisch übergeordneten Struktur gelten also bis zu ihrer Überdefinition. Nur die 'Blattknoten' sollten zum Rendern von Primitiven verwendet werden. 'Zwischenknoten' dienen nur zur logischen Gruppierung und zur Attributierung. Leider beziehen die Methoden des 3DKit zur Berechnung der Rechteckhülle auch die Daten aus den Zwischenknoten, die keinen Raum besitzen ein, sodaß eine Neuberechnung der Hülle der Zwischenknoten implementiert werden muß. Die Transformationen werden vom Blattknoten zum Wurzelknoten ausgeführt, die Punkte werden zur Transformation von links an die entsprechende Matrix multipliziert.

Die folgende Abbildung aus der NeXT Dokumentation der **N3DShape** Klasse zeigt eine Gegenüberstellung der Ausführung der Methode **render:** beim Rendern und den verwendeten RenderMan Funktionen.

Nach jeder Umstellung der virtuellen Kamera oder eines Objektes, wird durch einen Aufruf der **display** Methode einer Kamera ein komplett neuer Frame gerendert. Wird zur Anzeige nur in Drahtgittermodellen oder 'flach' gerendert, sind die Rechenzeiten erträglich. Durch die Struktur des Interfaces, die nur komplettes Rendern einer Szene zuläßt, ist es leider nicht möglich, Änderungen an Objekten lokal darstellen zu lassen. Bei komplexen Szenen sind Wartezeiten also schon vorprogrammiert.

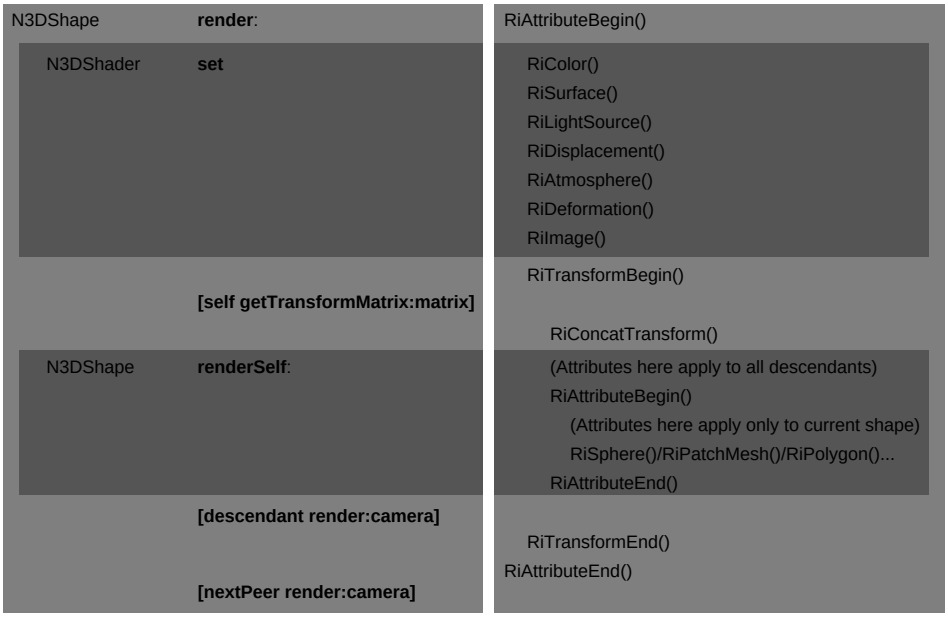

Abbildung 2.7: Ausführung der **N3DShape render:** Methode, aus [NeXTDoc]

2.7 Programmierung einer Miniapplikation

2.7.1 Erstellen eines Applikationsrahmens

Nachdem für das Programmierprojekt ein eigenes Verzeichnis (z.B. `FirstCamera`) angelegt wurde und mittels des *ProjectBuilders* in diesem Verzeichnis ein neues Projekt `PB.project` erzeugt wurde, kann durch einen Doppelklick auf `FirstCamera.nib` im Projektfenster der *InterfaceBuilder* gestartet werden. In das erscheinende Fenster ‘My Window’ kann nun eine ‘CustomView’ aus dem ‘Palettes’ Fenster gelegt werden. Sie kann anschließend soweit vergrößert werden, daß sie den gesamten Fensterinnenbereich ausfüllt. Im ‘CustomView Inspector’ können auf der ‘Size’ Seite die beiden inneren ‘Federn’ gesetzt werden, damit sich die View mit dem Fenster vergrößert. Anschließend kann ein Kameraobjekt von ‘Objekt/Responder/View/N3DCamera’ abgeleitet werden. Dazu aktiviert man in der ‘Classes’ Sektion des `FirstCamera.nib` Fensters, nachdem man im dortigem Browser ‘**N3DCamera**’ selektiert hat, die Operations-Pulldown-Aktion ‘Subclass’ — es wird die Klasse **MyN3DCamera** erzeugt. Durch die Operation ‘Unparse’ werden die Dateien `MyN3DCamera.h` und `MyN3DCamera.m` mit der Objetive-C Klassendefinition erzeugt. Beide Klassen sollen in das Projekt aufgenommen werden. Anschließend wird der ‘CustomView’ Klasse durch den Inspektor das Attribut ‘MyN3DCamera’ zugeteilt, die CustomView wird eine Instanz von **MyN3DCamera**. Abschließend sollte das Interface durch die Menüaktion ‘File/Save’ gespeichert werden. Der *InterfaceBuilder* kann danach verlassen werden. Das Programm kann nun schon mit der *ProjectBuilder* Aktion ‘Run’ übersetzt und gestartet wer-

den. Allerdings bekommt man nur ein schwarzes Fenster zu sehen. Um ein Objekt darzustellen muß man die **InitFrame:** Methode von **MyN3DCamera** überdefinieren. Dort kann die Kamera plaziert, eine Lichtquelle gesetzt und das Weltobjekt durch eine Instanz einer eigenen **MyN3DShape** Klasse ausgetauscht werden. Die `MyN3DCamera.h` Header-Datei und die `MyN3DCamera.m` Datei mit der überdefinierten Methode können wie folgt aussehen:

```
///////////////
// MyN3DCamera.h

#import <appkit/appkit.h>
#import <3Dkit/N3DCamera.h>

@interface MyN3DCamera:N3DCamera
{
}
- initFrame:(const NXRect *) theRect;
@end

///////////////
// MyN3DCamera.m

#import "MyN3DCamera.h"
#import <3Dkit/N3DLight.h>
#import "MyN3DShape.h"

@implementation MyN3DCamera
- initFrame:(const NXRect *) theRect
{
    // Kameraposition und -richtung
    RtPoint fromP = {0,0,-7.0}, toP = {0,0,0};

    // Vektor fuer Lichtquelle: DistantLight
    RtPoint fromDP = {0, 1, -5};
    RtPoint toDP = {0.0, 0.0, 0.0};

    id distantLight; // Zeiger auf Instanz von einer Lichtquelle

    // Initialisierung der Elternklasse aufrufen
    [super initFrame:theRect];

    // Kameraposition festlegen
    [self setEyeAt:fromP toward:toP roll:0.0];

    // Lichtquelle erzeugen, der Vektor gibt hier nur die
    // Strahlenrichtung, nicht die Lichtquellenposition an
    distantLight = [N3DLight new];
    [distantLight makeDistantFrom:fromDP to:toDP intensity:0.5];
    [self addLight:distantLight];
```

```
    // Erzeugen und Einsetzen des eigenen Shapes
    [[self setWorldShape:[MyN3DShape new]] free];

    // Die Oberflaeche soll schattiert werden, Standard ist das
    // Drahtrahmenmodell
    [self setSurfaceTypeForAll:N3D_SmoothSolids chooseHider:YES];

    return self;
}

@end
```

Nach dem Aufruf der **initFrame:** der Elternklasse kann mit Hilfe der eigenen Methode **setEyeAt:toward:roll:** die Kameraposition im (linkshändigen) Weltkoordinatensystem festgelegt werden. Der **roll** Parameter gibt die linkshändige Rotierung der Kamera um ihre u-Achse an (siehe Abb. 2.8).

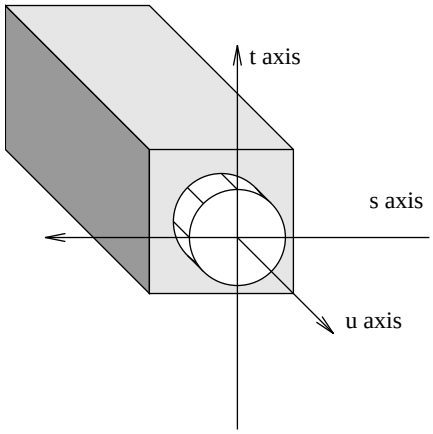

Abbildung 2.8: Das Kamerakoordinatensystem

Als Lichtquelle wird eine Instanz der **N3DLight** Klasse verwendet. Die Instanz kann die Rolle einer punktförmigen Lichtquelle, eines ambienten Lichts, eines Scheinwerfers oder, wie im Beispiel, einer Lichtquelle mit parallelen Strahlen (*Distant Light*) spielen. Die Lichtquelle wird anschließend noch mit **addLight:** in die Liste der globalen Lichtquellen der Kamera eingereiht. Die Deklaration des Lichtquellenobjekts steht in `3Dkit/N3DCamera.h` und muß importiert werden.

Da jede Kamera ein Standard-Weltobjekt besitzt, muß beim Einsetzen des eigenen mit **setWorldShape:** darauf geachtet werden, daß das Standard-Objekt freigegeben wird. Damit das Programm übersetzt werden kann, muß nun noch der Kopf der **initFrame:** Methode in die Headerdatei übertragen werden und ein **MyN3DShape** Objekt definiert werden, dessen Header in die `MyN3DCamera.m` Datei importiert werden sollte.

Die Dateien `MyN3DShape.h` und `MyN3DShape.m` werden zunächst mit dem Editor erzeugt und anschließend in das 'FirstCamera' Projekt als Klasse aufgenommen. Die beiden Dateien haben folgenden Inhalt:

```
///////////////
// MyN3DShape.h

#import <3Dkit/N3DCamera.h>
#import <3Dkit/N3DShape.h>

@interface MyN3DShape:N3DShape
{
}

- renderSelf:(N3DCamera *)theCamera;
@end

///////////////
// MyN3DShape.m

#import "MyN3DShape.h"

@implementation MyN3DShape

- renderSelf:(N3DCamera *)theCamera
{
    [super renderSelf:theCamera];
    RiRotate(-90.0, 1.0, 0.0, 0.0);
    RiCone(1.0, 0.5, 360.0, RI_NULL);
    return self;
}

@end
```

Die Methode **renderSelf:** wurde abgeleitet. In ihr stehen die RenderMan-Interface Aufrufe, die das Grafikobjekt definieren. Im Beispiel wird ein einfacher Kegel erzeugt, der vorher noch so gedreht wird, daß seine Spitze nach oben weist. In der Methode dürfen alle Interface-Aufrufe gemacht werden, die auch in einem Weltblock stehen dürfen. Es muß darauf geachtet werden, daß alle geöffneten Blöcke auch wieder geschlossen werden. Die **renderSelf:** Methode wird innerhalb eines Transformationsblocks aufgerufen. Nachdem das Programm übersetzt und gestartet wurde, sollte ein weißer Kegel oberhalb der Fenstermitte ausgegeben werden. Das Programm kann nun durch geeignete Methoden und Interface-Aufrufe erweitert werden. Denkbar sind eine Benutzungsoberfläche für die Kamera, das Einsetzen von Shadern durch **N3DShader** Instanzen oder die Darstellung einer ganzen Objekthierarchie unterhalb eines Weltobjekts. Bei Objekthierarchien sollten für einen logischen Aufbau nur in den Blattknoten Objekte erzeugt werden.

2.7.2 Minianimation

Die Programmierung einer Animation kann mit Hilfe des **N3DMovieCamera** Objekts geschehen. Dazu kann die vorige Miniapplikation erweitert werden. Zuerst kann der Objekttyp der Kamerainstanz im *InterfaceBuilder* zurück auf **View** gesetzt werden. Anschließend wird ein Menü-Item 'Animate' in das Hauptmenü eingefügt. Danach kann im Nib-Fenster in der 'Classes' Sektion von **N3DCamera** die Klasse **N3DMovieCamera** abgeleitet werden, der *InterfaceBuilder* kennt diese Klasse noch nicht. Dann sollte die Klasse **MyN3DCamera** gelöscht werden und eine neue Unterklasse mit dem gleichen Namen von **N3DMovieCamera** erzeugt werden. Die Aktion **startAnimate:** kann jetzt zu der neuen Klasse **MyN3DCamera** hinzugefügt werden. Im folgenden Schritt wird das Custom Objekt wieder als **MyN3DCamera** Instanz gesetzt: Custom Objekt anklicken, im Inspector auf der 'Attributes' Seite **MyN3DCamera** anklicken. Nun muß man noch zwischen dem Menü-Item 'Animate' und dem Customobjekt eine Verbindung zur Methode **startAnimate:** herstellen und die NIB-Datei kann gespeichert werden. Anschließend müssen die Programmdateien auf den neusten Stand gebracht werden. In `MyN3DCamera.h` muß anstelle der Datei `3Dkit/N3DCamera.h` der Header `3Dkit/N3DMovieCamera.h` 'included' werden, als Elternobjekt von **MyN3DCamera** muß **N3DMovieCamera** verwendet werden und der Kopf der Funktion **startAnimate:** muß eingesetzt werden. In der Implementierung `MyN3DCamera.m` muß diese Methode zugefügt werden:

```
- startAnimation:sender
{
    [self setStartFrame:0 endFrame:360 incrementFramesBy:10];
    [self displayMovie];
    return self;
}
```

Mit **setStartFrame:endFrame:incrementFramesBy:** werden die Grenzen und die Inkrementation der Animationsrahmennummern gesetzt. Mit **displayMovie** wird die Animation mit dem interaktiven RenderMan gestartet. Das photorealistische Rendern kann wie bei der **N3DCamera** mit **renderAsTIFF** oder **renderAsEPS** gestartet werden. Durch die Abfrage mit **frameNumber** kann während des Renderns die aktuelle Rahmennumer abgefragt und auf ihren Wert reagiert werden.

Durch eine Änderung der **renderSelf:** Methode von **MyN3DShape** kann erreicht werden, daß sich die Oberfläche in Abhängigkeit von der Animationsrahmennummer dreht:

```
- renderSelf:(N3DCamera *)theCamera
{
    [super renderSelf:theCamera];
    RiRotate(([theCamera frameNumber]), 1.0, 0.0, 0.0);
    RiCone(1.0, 0.5, 360.0, RI_NULL);
    return self;
}
```

Nach dem Übersetzen und Starten der Applikation kann durch die Aktivierung des Menü Items 'Animate' der Konus gedreht werden.

2.7.3 Einbinden von RIB-Makros

Um RIB-Makros einbinden und ausgeben zu können müssen die folgenden Interface-Funktionen verwendet werden: **RiResource()**, **RiCreateHandle()**, **RiReadArchive()**, **RiArchiveRecord()**, **RiMacroBegin()**, **RiMacroEnd()** und **RiMacroInstance()**. Zusätzlich spielt die Option **RI_ARCHIVE** mit den Parametern "`clipobject`", "`outputversion`" und "`expandmacros`" eine gewisse Rolle (s. [PixQRM]). Die Funktionen **QRMGetRIBHandlers(RtRIBHandlers *)** und **QRMSetRIBHandlers(RtRIBHandlers *)** können dazu verwendet werden, eigene Handler-Routinen für die Interface-Aufrufe, die aus dem RIB-Archiv gelesen werden, zu installieren.

Ein Makro darf zwar außerhalb eines Programmblocks deklariert werden, der Renderer muß aber initialisiert sein. Mit dem Erzeugen einer **N3DContextManager** Instanz in der Methode **appDidInit:** der Applikation kann man dies zur Not erreichen. Da kein expliziter Interface-Aufruf zum Zerstören eines Makro-Handles existiert, Makro-Handles aber nach dem Ende ihres umgebenden Blockes freigegeben werden, ist es besser Makros erst innerhalb eines Programmblocks zu erzeugen. Die Methode **worldHasBeginInCamera:** von **N3DCamera** ist ein guter Platz zur Initialisierung eines Makro-Handles (s. [NeXTDoc]), leider ist diese Vorgehensweise mit Geschwindigkeitsverlusten gegenüber den globalen Makros verbunden (liegt vor allem an **RiReadArchive()**). Das Laden der RIB-Datei in einen Puffer kann schon vorher geschehen (nicht jedesmal beim Rendern). Der Dateipuffer wird bei der Zerstörung des Handles nicht automatisch freigegeben. Vor dem Laden der Datei muß wegen der späteren Verwendung der Funktion **RiArchiveRecord()** beachtet werden, daß im Puffer vor dem Zeiger auf das erste Zeichen der Datei ein `'\n'` steht und hinter dem letzten Zeichen der Datei ein `'\0'`. Das Laden kann also folgendermaßen aussehen (**fileSize()** soll die Dateigröße liefern):

```
// RIB-Datei in einen Puffer lesen
//
char *readRibFile(const char *filename) {
    int fh, // Dateideskriptor der RIB-Datei
        fs; // Dateigroesse
    char *buffer;

    /* Datei oeffnen, Groesse bestimmen, fs <= 0 bei Fehler
    fh = open(filename, O_RDONLY, 0);
    fs = fh >= 0 ? fileSize(fh) : -1;

    if ( fs <= 0 ) {
        // Fehler beim Dateioeffnen oder leere Datei
        if ( fh >= 0 ) close(fh);
        return NULL;
    }

    buffer = malloc(fs+2);  // +2 wegen zusaetzlicher Zeichen (s.u.)
    if ( !buffer ) {
        close(fh);
        return NULL;
    }
```

```
    *buffer = '\n';          // -> Wegen RiArchiveRecord() !!!
    read(fh, buffer+1, fs);  // Einlesen der Datei
    buffer[fs+1] = '\0';     // -> Wegen RiArchiveRecord() !!!

    close(fh);
    return ++buffer;         // Zeigt auf erstes Zeichen der Datei
}

// Puffer der RIB-Datei wieder freigeben
//
void freeRibBuffer(char *buffer)
{
    free(--buffer);           // Nicht vergessen 1 abzuziehen
}
```

Die Erzeugung der Makro-Handles kann wie folgt geschehen:

```
// Erstellen von einem Makro-Handle. Die Funktion liefert 0 wenn
// kein Handle erzeugt wurden konnte. Ansonsten ist das Makro in
// *macro und das Handle auf das RIB-Archiv in *resource zu finden.
// Die beiden Rueckgabeparameter haben bei einem evtl. Fehler
// den Wert RI_NULL
//
int makeHandles(
    char *ident,       // Handlename, z.B. Basisname der RIB-Datei
    char *buffer,      // Puffer mit RIB-Datei
    int isEntity,      // Datei war eine RIB-Entity Datei
    RtToken *resource, // Rueckgabe Archiv-Ressource
    RtToken *macro     // Rueckgabe RIB-Makro
) {
    RtInt clipon = 1, clipoff=0;

    *resource = RiResource(ident, RI_ARCHIVE, RI_ADDRESS, &buffer,
                           RI_NULL);
    if ( *handle != RI_NULL ) {
        *macro = RiMacroBegin(ident, RI_NULL);
            if ( !isRibEntity )
                RiOption(RI_ARCHIVE, "clipobject", &clipon,
                         RI_NULL);
            RiReadArchive(*resource, NULL, RI_NULL);
            if ( !isRibEntity )
                RiOption(RI_ARCHIVE, "clipobject", &clipoff,
                         RI_NULL);
        RiMacroEnd();
        return *macro != RI_NULL;
    }
    *makro = RI_NULL;  // Kein Handle
    return 0;          // Fehler
}
```

Der zweite Parameter der **RiReadArchive()** Funktion kann einen Zeiger auf eine Callback-Funktion zur Behandlung der RIB-Data Records beinhalten: `callback(RtToken type, char *format, char *data)`. Eine Beschreibung über Sinn und Zweck dieser Callback Funktion konnte leider der Beschreibung in [PixQRM] nicht entnommen werden. Denkbar ist die Behandlung von Pseudokommentaren, den RIB Strukturkonventionen wie die Dateikennung (ähnlich den PostScript Strukturkonventionen) und eigenen Kommentaren.

Der Parameter **isEntity** soll `!=0` sein, wenn es sich bei der RIB-Datei um eine RIB-Entity-Datei handelt (s. [PixSpec]). RIB-Entity Dateien beinhalten eine einfache Objektdefinition ohne Weltblock, können also ohne weitere Behandlung in den Renderprozeß eingefügt werden. Entity-Dateien sind an ihrem Dateikopf erkenntlich:

```
##RenderMan RIB-Structure 1.0 Entity
```

Normale RIB-Dateien, mit Optionen und Weltblock haben die Kennung:

```
##RenderMan RIB-Structure 1.0
```

Von Dateien dieser Art darf nur das Innere des Weltblocks ausgegeben werden. Die Option ‘`RI_ARCHIVE, "clipobject", &clipon`’ schaltet einen entsprechenden Modus ein. Bei der Makrodefinition wird in diesem Fall alles vor dem **WorldBegin** stehend bis auf **Declare** Anweisungen überlesen. Bei der Instanzierung der Makros von RIB-Entity Dateien müssen laut [NeXTDoc] zwei Fälle unterschieden werden:

1. Beim interaktiven Rendern (`NXDrawingStatus == NX_DRAWING`) muß das Makro mit `RiMacroInstance(macro, RI_NULL)` ausgegeben werden, ein Makro-Handle muß zu diesem Zweck erzeugt worden sein.

2. Das Schreiben auf einen ‘RIB Stream’ geschieht mit einem Aufruf der Interface-Funktion `RiArchiveRecord(RI_COMMENT, buffer-1)` ohne vorheriges Erzeugen der Handles. Es muß gelten: `*(buffer-1)=='\n'`.

3DKit setzt nur beim interaktiven Rendern (Fall 1) die globale Variable **NXDrawingStatus** auf **NX_DRAWING**. Die beschriebene Methode hat den Nachteil, daß beim Schreiben in eine RIB-Datei das Makro bei jeder Instanzierung komplett ausgegeben wird. In *ModelMan* wurde deshalb eine alternative Vorgehensweise vorgezogen, die den Befehl **RiMacroInstance()**, bzw. **RiObjectInstance()** mit Hilfe von **RiArchiveRecord()** auch bei der RIB-Ausgabe verwendet.

3. Anwendung des RenderMan Interfaces

3.1 Programmierung des Modellierwerkzeugs ModelMan

3.1.1 Überblick

Die Klassen von 3DKit bieten einen objektorientierten Aufsatz auf Teile des RenderMan Interfaces, der um eine grafische Benutzungsoberfläche erweitert werden kann. Durch die Erstellung des Modellierwerkzeug **ModelMan** sollte es möglich werden, das RenderMan Interface auch ohne Programmierung zu verwenden. **ModelMan** versteht sich als Sicht und Kontrollinstrument für das RenderMan Interface mit einer möglichst breiten Unterstützung seiner Funktionalität. Durch die quasi beliebige Anzahl verschiedener Grafikobjekte (*Shapes*), die durch ein Modellierungsprogramm unterstützt werden können, ist eine einfache Erweiterung durch neue Objekttypen mit den dazugehörigen Kontrollelementen notwendig. Die Bundles des NeXT und das Binden von Objective C Objektdateien zur Laufzeit eines Programms, geben hierfür eine entsprechende Basis.

Der Aufbau des RenderMan Interfaces und 3DKits bietet eine Trennung zwischen einem Kameramodell und Grafikobjekten, die hierarchisch gruppiert werden können. Durch die Abkapselung der Shape-Hierarchie von einer darstellenden Kamera in Verbindung mit den Renderkontexten des *Quick RenderMan*s wurde es möglich, von der selben Hierarchie mehrere Projektionen gleichzeitig in verschiedenen Dokumentenfenstern darzustellen. Ebenso ist es möglich, verschiedene Szenen in verschiedenen Fenstern zu bearbeiten.

Neben der konzeptionellen Aufteilung einer 3D-Szene in Kamera und Shapes, wurde auf der Seite der Benutzungsoberfläche eine Trennung zwischen der Ausgabe des *Quick RenderMan*s in ein Dokumentenfenster und der Darstellung der aktuellen Objektzustände in verschiedenen Inspektoren vorgenommen. Die Inspektoren zeigen thematisch gegliedert den aktuellen Zustand von Objekten, die zum fokussierten Dokumentenfenster gehören. Innerhalb des Dokumentenfensters ist eine interaktive Manipulation und Selektion der Shapes direkt mit der Maus und die Änderung der Kameraposition durch weitere Kontrollelemente am Fensterrand möglich. Die Inspektoren verwenden zur Kontrolle der Objektzustände neben der numerischen Eingabe eine Vielzahl der von AppKit zur Verfügung gestellten Komponenten einer grafischen Benutzungsoberfläche.

3.1.2 Die Programmierung

Das Programm wurde mit Objective C erstellt. Ein gravierender Vorteil der Verwendung dieser Sprache ist, daß durch das dynamische Binden eine viel größere Unabhängigkeit unter den Objekten möglich ist als bei einer Programmiersprache wie C++, die nur ein eingeschränktes dynamisches Binden durch die virtuellen Elementfunktionen der Objektklassen erlaubt. Anders als bei dieser und ähnlichen Hybrid-Sprachen braucht beim 'Senden' von Nachrichten die Klassen-Deklaration des Empfängers (oder der Elternklasse, die die entsprechende virtuelle Methode besitzt) nicht bekannt zu sein. Durch die Verwendung informeller oder formeller Protokolle[1] braucht lediglich die Nachricht festgelegt zu werden, die gesendet werden soll — die Zuordnung von Nachricht zu Methode besorgt die Klasse zur Laufzeit. Anwendung fanden die informellen Protokolle vor allem in der Verbindung zwischen Objekt und seinem Inspektor, einem meist mehrseitigen Panel, mit dem der Zustand des Objekts dargestellt und geändert werden kann. Das Objekt muß den Inspektor (oder die Inspektoren) benachrichtigen, wenn es seinen Zustand geändert hat. Es deklariert dazu ein entsprechendes informelles Protokoll als 'Kategorie', kennt den Typ des Inspektors aber nicht. Bezogen auf das Model-View-Controller (MVC) Konzept ([Adams86]) ist der Inspektor ('Controller') unabhängig von dem kontrollierten Objekt ('Model'). Zwar ist in der Implementation von **ModelMan** (sowie der Klassen auf den die Applikation basiert) keine strikte Trennung von 'Model', 'View' und 'Controller' Objekten vorhanden, wohl aber eine Trennung der entsprechenden Methoden. Das Kameraobjekt besitzt Aspekte von einer 'View' (Darstellung der Shape-Hierarchie) und einem 'Controller' (Manipulation der Shapes mit der Maus). Die Shapes sind zwar 'Model' Klassen, besitzen aber auch 'Controller' Methoden zur Behandlung objekteigener Kontrollpunkte (**PatchShape**) und zur Darstellung, die von verschiedenen Kameras ('Views') aus ausgeführt werden können. Die Steuerung des 'Updates' der 'Views' der Shapes geschieht nach deren Änderung durch Methoden der Kamera. Die Shapes besitzen deshalb keine eigenen Zeiger auf ihre Kameras und Inspektoren.

Die Möglichkeit des Einbindens von Objektklassen zur Laufzeit eines Programms, ist mit vielen anderen Programmiersprachen als Objective C nicht in der Form und Einfachheit zu erreichen. Durch eine Abtrennung verschiedener Klassen, die zur Laufzeit geladen werden können, konnten in **ModelMan** konkrete Objekttypen wie Kugel, Rechteck und Torus mitsamt ihrer speziellen Steuerung vom Hauptprogramm abgekapselt werden. Andere Objekttypen können prinzipiell noch hinzugefügt werden ohne das Hauptprogramm zu übersetzen oder zu binden. Da die allgemeinen Methoden (Verschieben, Rotieren, Skalieren, ...) in einer allgemeinen Klasse implementiert wurden, konnten die spezielleren Klassen, die von dieser Klasse abgeleitet wurden, relativ klein gehalten werden.

Die Einbindung von Objective C in das System des NeXTs und die einfache Erlernbarkeit der wenigen neuen Sprachkonstrukte ist natürlich auch ein Grund für die Verwendung dieser Sprache. Der geringe Effiziensverlust durch die Methodensuche zur Laufzeit konnte in Kauf genommen werden.

[1] Durch die Mehrfachvererbung von C++ wird in dieser Sprache auch eine Art von Protokollen möglich. In der Datei mit der Deklaration einer Klasse, die für eines ihrer Attribute ein Protokoll definieren will, kann zusätzlich eine (pseudo-)abstrakte Klasse deklariert werden, die diese Methoden enthält. Das Attribut muß in diesem Fall als Zeiger auf eine Instanz dieser Klasse deklariert werden. Eine andere Klasse, die die Protokoll-Methoden implementieren soll, kann so von dieser Klasse erben und die gewünschten Methoden implementieren.

Für die Programmierung wurden die üblichen Werkzeuge **ProjectBuilder**, **InterfaceBuilder** und die daraus zugänglichen, bzw. aufgerufenen Programme wie **Edit**, **GDB**, **make** und **GCC** verwendet.

Programmbestandteile

Ein Überblick der Programmbestandteile kann aus den verwendeten Interface-Dateien (Abb. 3.1) gewonnen werden.

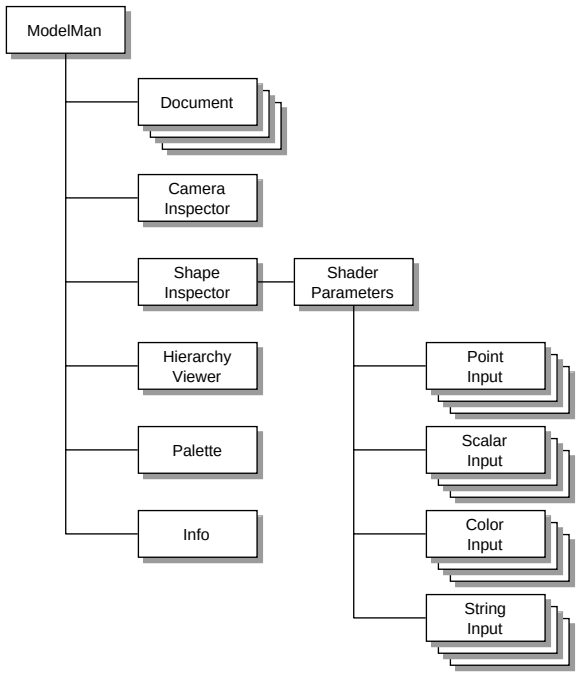

Abbildung 3.1: Die Interface Komponenten des **ModelMan**

Die Bezeichner in den Kästchen entsprechen den Klassennamen. Interface und Implementation sind in den gleichnamigen '`.h`' und '`.m`' Dateien definiert. In einer Datei mit dem Suffix '`.nib`' kann das zugehörige Interface gefunden werden. Die Verbindungen entsprechen den Verzeigerungen innerhalb der Objektstruktur.

ModelMan: NIB-Datei der Applikation, beinhaltet das Hauptmenü. Die Applikation wird in **main()** in der Datei '`ModelMan_main.m`' erzeugt.

Document: Das Dokumentenfenster mit einer View für die Kamera und einigen zusätzlichen Kontrollelementen zur Positionierung von Shapes und Kamera (Daumenräder, Umschalter). Die Applikation **ModelMan** verwaltet eine Liste von Dokumentinstanzen. Anfangs sind keine Dokumente vorhanden.

CameraInspector: Inspektor zur Darstellung des Kamerazustands und der RenderMan Optionen. Die Applikation besitzt maximal eine Instanz eines Kamera-Inspektors.

ShapeInspector: Der Shape-Inspektor stellt den Attributzustand eines selektierten Shapes dar. Die Attributwerte sind auch in den Instanzen der Shape-Klasse gespeichert. In dem Panel können Seiten in Abhängigkeit zum Type eines Shapes dargestellt werden.

ShaderParameters: Zum **ShapeInspector** gehörendes, eigenständiges Panel, mit dem die Shader für ein Objekt eingesetzt werden können. Die Shader-Parameter können in Eingabeobjekte (**...Input**), die in eine **ScrollView** einsortiert sind eingegeben werden.

PointInput: Eingabezeile für das **ShaderParameters** Panel für Punkte. Wird wie die anderen **Input** Interface-Objekte je nach Shader-Parameter in die **ScrollView** dieses Panels eingegliedert.

ScalarInput: Eingabezeile für einen **Float** Wert in das Parameter-Panel.

ColorInput: Eingabezeile mit **ColorWell**[2] und Eingabefelder für die RGB Farbwerte.

StringInput: Eingabezeile mit einem Texteingabefeld und einem Knopf zum Aktivieren eines Dateidialogs. Er wurde eingefügt, weil Textparameter häufig für Dateinamen verwendet werden.

HierarchyViewer: Zeigt im Gegensatz zum **ShapeInspector** nicht den Zustand eines einzelnen Shapes, sondern den Aufbau der Shape-Hierarchie innerhalb eines Browsers an. Der Name des Objekts soll an den **File Viewer** des NeXTs erinnern. Momentan ist nur der Browser im Panel implementiert. Ein 'Shelf' (Ablage für Shape-Icons) und eine **ScrollView** mit dem aktuellen Selektionspfad (Kamera, Weltobjekt, Shape,...) in der Hierarchie sollen folgen. Zusätzlich sind Knöpfe für die Gruppierung, Degruppierung, Gruppenumordnung und das Löschen von Shapes vorhanden.

Palette: Panel mit den Icons der aktuell geladenen Shape-Klassen und RIB-Makros. Im Panel wird für jedes Element ein spezieller 'Drag-Button' erzeugt, der es erlaubt, ein Icon vom Panel in ein Dokumentenfenster zu ziehen. Dort wird dann eine Instanz des Shapes oder des Makros erzeugt.

Info: Info-Panel mit dem Namen des Programms.

Anbindung des RenderMan Interfaces

Funktionen des RenderMan Interfaces werden in den Methoden der Klassen **RibArchive**, **InteractiveCamera**, **InteractiveShape** und den davon abgeleiteten Klassen verwendet. Die letzten beiden Klassen sind von den entsprechenden **N3D...** Klassen des 3DKit abgeleitet. In Abb. 3.2 ist ein Teil der Klassenhierarchie von AppKit mit allen, in **ModelMan** neu definierten Klassen zu sehen. Die neu eingeführten Klassen sind mit Schatten dargestellt.

[2] Es wird ein abgeleitetes **MyNXColorWell** verwendet, da das originale Objekt sich innerhalb einer **ScrollView** nicht korrekt zeichnet.

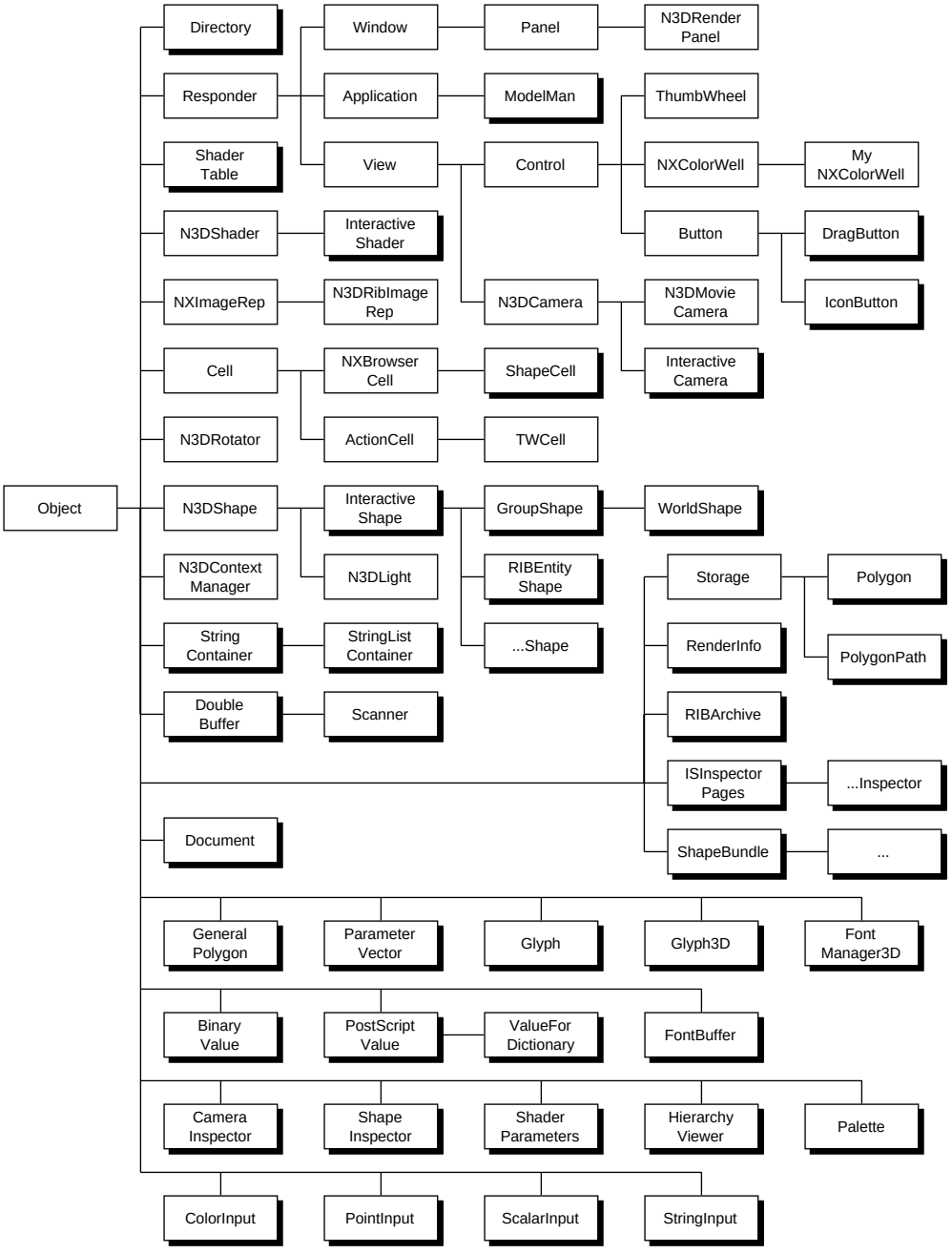

Abbildung 3.2: Die neuen Klassen von **ModelMan**

In der **InteractiveShape renderSelf:** Methode werden die Interface-Funktionen innerhalb des Weltblocks ausgerufen. Die **Interactive...** Klassen besitzen zusätzliche Methoden, um die Shapes über die Benutzungsoberfläche steuerbar und archivierbar zu machen. Die Kamera besitzt Methoden, die Interface-Aufrufe auch außerhalb des Weltblocks zulassen; in diesem Objekt können Optionen gesetzt werden. In der Klasse **RibArchive** wird eine RIB-Datei geladen und kann dem RenderMan als Makro zur Verfügung gestellt werden. Die Klasse **InteractiveShader** erlaubt das Speichern von Shader-Parametern in eine Datei und das Laden der Parameter von einer Datei. In den Hilfsklassen **Polygon**, **PolygonPath**, **GeneralPolygon** und **Glyph3D** kommen Methoden mit dem RenderMan-Interface in Berührung. In der **Polygon** können die Punkte eines einfachen Polygons (konvex ohne Löcher) gehalten werden, **PolygonPath** hält die Stützpunkte eines Pfades aus Linien- und Bézier-Segmenten, der in den *CharStrings* von Adobe Type 1 Fonts (eine Beschreibung des Font-Formats kann in [AdobeType1] gefunden werden) definiert wird. Die Klasse **GeneralPolygon** dient zur Aufnahme der Punkte von komplexen Polygonpfaden (Löcher, konvex oder konkav). Die Klasse enthält Methoden, um aus dem Polygonpfaden, die von Adobe Type 1 Zeichen gewonnen werden, RenderMan-Polygone zu generieren. Die Klasse **Glyph3D** dient zur Darstellung eines 3D-Zeichens, das einem Adobe Type 1 Zeichen (repräsentiert durch die Klasse **Glyph**) entspricht.

Die Inspektor-Panele

Es existieren zwei Inspektor-Panele **CameraInspector** und **ShapeInspector** für die Kontrolle von Instanzen der Klassen **InteractiveCamera** und **InteractiveShape** nebst Spezialisierungen. Der **CameraInspector** zeigt den Zustand der Kamera, die im aktuellen Dokumentenfenster dargestellt wird. Der **ShapeInspector** den Zustand des selektierten Shapes (bei Mehrfachselektionen wird das Panel abgeschaltet). Es brauchte kein spezielles Objekt für einen Inspektor oder Mehrseiten-Panel definiert werden. Die Umschaltung der Seiten ist trivial: Die einzelnen Seiten werden mit dem **InterfaceBuilder** in einem (nicht sichtbaren) Panel zusammengebaut und gruppiert. Nach dem Dearchvieren der NIB-Datei werden die Gruppen aus dem Panel herausgenommen und bei Bedarf links unten in das eigentliche Inspektor-Panel eingehängt. Die Seitenumschaltung kann der Benutzer durch eine einfache **PopUpList** vornehmen. Inspektoren sind nie modal. Modale Dialoge werden in **ModelMan** nur für Pagelayout-, Datei- und andere Standard-Panele verwendet.

Der CameraInspector

Der Inspektor besteht aus vier Seiten: Oberfläche der Shapes und Kamera-Projektion, Kameraposition, den Optionen zur Steuerung der Rasterabbildung und der Render-Kontrolle. Mit Hilfe der ‘Put’/‘Get’ Knöpfe lassen sich die aktuellen Einstellungen einer Kamera zwischenspeichern. Mit einem weiteren Knopf kann die Ausgabe eines 3D-Rahmens um selektierte Objekte ein- und ausgeschaltet werden.

Durch den Zeiger **camera** ist der Inspektor mit der aktuellen Kamera verbunden. Der Inspektor verwaltet auch die Liste aller Render-Jobs und dient der Kamera als ‘Delegate’. Der Inspektor wird deshalb vom System benachrichtigt, wenn eine Grafik vom photorealistischen Renderer fertiggestellt wurde.

Der ShapeInspector

Der Aufbau des **ShapeInspector** unterscheidet sich von dem des **CameraInspector** vor allem dadurch, daß die Anzahl seiner Seiten nicht fest ist. Standardmäßig enthält er nur die Kontrollen, die für das allgemeine **InteractiveShape** verwendet werden können: Steuerung von Oberfläche, Shader, Position, Skalierung, Rotation, Größe der Rechteckhülle und Dehnung. Je nach ausgewähltem Shape-Typ können neue Seiten hinzugeladen werden. Normalerweise gehört zu jedem Shape eine Seite 'Attributes' mit der z.B. bei einer Kugel der Durchmesser und bei einem Kegel die Höhe angegeben werden kann. Bei der Selektierung eines anderen Objekts wird nach Möglichkeit die Seite mit der gleichen Positionsnummer beibehalten. Bei der Programmierung neuer Inspektorseiten sollte deshalb darauf geachtet werden, daß die 'Attributes' Seite immer die erste Seite ist. Die Klasse der zugeladenen Seiten muß von **ISInspectorPages** abgeleitet sein. Diese Klasse gibt ein paar Methoden vor, die von spezielleren Inspektor-Seiten überdefiniert werden. Es ist in der momentanen Implementation noch nicht möglich, daß bestimmte Kontrollen für einige Shape-Typen ausgeschaltet werden, falls sie von diesen nicht verwendet werden.

Der Shape-Inspektor macht die zwei Arten der Vererbung deutlich: Die Vererbung über die Klassenhierarchie und die Vererbung über die Instanzhierarchie. Die Vererbung über die Klassenhierarchie ist für die Typen der Kontrollen der Inspektorseiten verantwortlich. Die Klasse **SphereShape** erbt von ihrer Elternklasse **InteractiveShape** die Attribute. Der Inspektor muß also zusätzlich zu den spezielleren Kontrollen, die für **SphereShape** notwendig werden (Radius, Schließwinkel,...) auch die allgemeineren Kontrollen für **InteractiveShape** (Transformationen, Shader,...) enthalten. Über die Instanzhierarchie, die einer Attribut-Blockstruktur entspricht, dagegen, werden die Werte der Attribute vererbt. Ein Shape kann z.B. den Wert 'Blau' ihrer Oberfläche von der Gruppe, in der sie eingeordnet ist erben. Die Gruppenstruktur (Instanzhierarchie) selbst wird in dem 'Browser' des 'Hierarchy Viewer'-Panels angezeigt.

Die Programmdateien

Die folgenden Programmdateien werden im **ModelMan**-Projekt übersetzt. Die Namen erhalten immer die üblichen Suffixe '`.h`' (für Klassen-Interface) und '`.m`' (für Klassen-Implementation), zu einigen gehört noch eine '`.nib`' Interface-Datei. Bis auf wenige Ausnahmen ist pro Datei je eine Klasse implementiert, einige Dateien enthalten Funktionssammlungen. Die Dateien sind zur besseren Übersicht thematisch zu Unterprojekten zusammengefaßt. Die Implementation der konkreten Shape-Objekte ist in Bundles geschehen. Es können so unabhängig vom Hauptprojekt neue Shape-Objekte programmiert und eingesetzt werden.

Hauptprojekt

Document: Objekt und Interface des Dokuments: Das Fenster mit der Kamera. Die Dokumente werden von der Applikation **ModelMan** verwaltet. Die Applikation fungiert als 'Distributor' und wird benachrichtigt, wenn sich ein Fenster ändert. d.h. geschlossen, zum Hauptfenster, miniaturisiert oder die Größe geändert wird. Die Benachrichtigungen sind durch die Kategorie **DocumentDistribution** von **Object** deklariert. Das Dokument ist für eine Archivierung auf einen 'Stream' vorgesehen.

InteractiveCamera: Kameraobjekt, das über einen Inspektor und die Dokumentenfenster-

Bedienelemente gesteuert werden kann. Die Optionen des RenderMan Interfaces können mit Hilfe der Kamera geändert werden. Benachrichtigungen bei Selektionsänderungen, Shape-Modifikationen und -Anforderungen werden als Nachrichten in Kategorien von **Object** deklariert. Auf diese Nachrichten müssen die eingesetzten Kontroll- und Verteiler-Objekte, momentan **ModelMan**, **HierarchyViewer**, **ShapeInspector** und **CameraInspector** reagieren. Die Kamera ist Ausgangspunkt für das Rendern, einige Operationen auf Shapes und für die Mausoperationen innerhalb des Dokumentenfensters. Sie verwendet einen α-Puffer der Applikation um 'Rubberbanding' bei der Selektion mit der Maus zuzulassen (s. [AdobeDPS]). Die Kamera stellt auch die Rechteckhüllen der Objekte dar. Eine Kamera kann eine Liste von Kindkameras besitzen, die alle dieselbe Shape-Hierarchie (möglicherweise auf verschiedene Art und Weise) darstellen. Die Kamera kann sich, die Kindkameras und die Shape-Hierarchie auf einen 'Stream' schreiben. Das 'Drag&Drop'-Protokoll ist für die **NXAsciiPboardType** Zwischenablage der **Palette** implementiert, die Operationen 'Cut'/'Copy'/'Paste' sind auf Shape möglich.

InteractiveShader: Wurde von **N3DShader** abgeleitet, um das Archivieren auf 'Streams' und das Schreiben von Shader-Parameterdateien zu implementieren. Es wurde eine **Object** Kategorie **ShaderNotification** deklariert, die eine Shader-Tabelle (s. **ShaderTable**) implementieren muß, wenn der Shader durch eine Parameterdatei aufgebaut werden soll. Die Tabelle wird von der Applikation verwaltet, sie liefert zu einem Shader-Namen den aktuellen Zugriffspfad (der von User zu User variieren kann). Leider sucht der photorealistische Renderer die Shader nur unter `/NextLibrary/Shaders`, sodaß bei lokalen Shadern der komplette Pfadnamen beim Rendern mitgegeben werden muß. Zusätzlich werden einige Hilfsfunktionen definiert, die u.a. Shader-Verzeichnisse verwalten:

shaderTypeNum(): Liefert zu einem Shader-Typbezeicher **SLO_TYPE...** die entsprechende numerische Konstante. Die Konstanten sind in dem RenderMan Header `ri/slo.h` definiert.

shaderTypeString(): Gegenfunktion zu **shaderTypeNum()**, liefert zu einer Konstante **SLO_TYPE...** den entsprechenden String

makeShaderBaseDir(): Liefert den vollen Pfadnamen des Shader-Basisverzeichnisses (momentan: `~/Library/Shaders`). Ist das Verzeichnis noch nicht vorhanden wird es erzeugt. In diesem Verzeichnis sollen die lokalen Shader mit der Endung '`.slo`' gehalten werden.

makeShaderParameterDir(): Erzeugt, falls noch nicht vorhanden, das Standardverzeichnis für die ('`.slp`') Shader-Parameterdateien:

`~/Library/Shaders/ShaderParameters`

Der volle Pfadname wird zurückgegeben.

makeShaderIconsDir(): Erzeugt, falls noch nicht vorhanden, das Standardverzeichnis für die ('`.tiff`') Icon-Dateien der Shader:

`~/Library/Shaders/ShaderIcons`

Der volle Pfadname wird zurückgegeben.

makeShaderSourceDir(): Erzeugt, falls noch nicht vorhanden, das Standardverzeichnis für die (‘`.sl`’) Shader-Quelldateien:

```
~/Library/Shaders/ShaderIcons
```

Der volle Pfadname wird zurückgegeben. Die Funktion wird nicht aufgerufen, weil Shader-Quellen von **ModelMan** nicht verwendet werden.

printShader(): Schreibt die Parameter eines Shader-Objekts als Parameterdatei im Klartext auf eine Datei.

InteractiveShape: Basisobjekt aller Shapes der Applikation. Die Datei enthält zusätzlich die Definion einiger von **InteractiveShape** abgeleiteten Objekte, die nicht in Bundles implementiert wurden: **RibEntityShape**, **GroupShape** und **WorldShape**. Die Kategorie **ArchiveHolder**, durch die Applikation **ModelMan** implementiert, deklariert das Protokoll zum Suchen eines RIB-Archivs. Es werden Funktionen definiert, die die Behandlung der Kontrollpunkte der Rechteckhülle der Shapes vereinfachen.

otherSide(): Liefert die gegenüberliegende Seite eines Anfaßpunktes. Die Anfaßpunkte sind als symbolische Konstanten in der Headerdatei aufgeführt.

hitHandleInRect(): Überprüft, ob ein Anfaßpunkt der Rechteckhülle mit der Maus getroffen wurde.

drawHandles() Zeichnen der acht Anfaßpunkte

drawRectWithHandles() Zeichnen Umrandung mit den Anfaßpunkten

drawCenteredHandle() Zeichnen eines (zentrierten) Anfaßpunktes

In der Headerdatei sind außerdem Konstanten definiert, die als Indizees für die **RtBound** Struktur verwendet werden können. Andere dienen als Indizees in ein **BbxHandles** Array von den acht Anfaßpunkten. In **InteractiveShape** sind anders als bei seinem 3DKit Vorgänger **N3DShape** Angaben zur Oberflächenfarbe und Opazität möglich. Die entsprechenden Werte der Shader-Objekte (Mehrdeutigkeit bei mehreren verwendeten Shader-Typen) werden nicht mehr beachtet.

MenuNum: Eine ‘Header’-Datei, die alle Tags des Hauptmenüs als Konstanten enthält.

ModelMan: Die Applikation selbst. Die Applikation ist für die Verwaltung diverser Daten zuständig und implementiert einige der, von den anderen Objekten geforderten, informellen Protokolle. Die Applikation verwaltet eine Liste der aktuellen Dokumente und die Haupt-Panele (s. Abb. 3.1). Zusätzlich werden die Bundles und die RIB-Makros (als **RiArchive** Instanzen) in Hashtabellen gehalten. Als Schlüssel dient der Name des Bundles bzw. der Basisname der RIB-Datei. Die Verwaltung der Shape-Klassen geschieht so, daß auch Shapes, die nicht in Bundles definiert sind im Programmverlauf auf die gleiche Art und Weise verwendet werden können. Beim Starten der Applikation werden die Tabellen aufgebaut und bei Bedarf die verschiedenen benötigten Verzeichnisse angelegt. RIB-Dateien für die Makros werden in dem Verzeichnis:

```
~/Library/ModelMan/ClipArt
```

gesucht, die dazugehörenden (nicht unbedingt notwendigen) Tiff-Icons in:

```
~/Library/ModelMan/ClipArtIcons
```

Icons und RIB-Dateien müssen den gleichen Basisnamen besitzen. Die Applikation besitzt auch die im Programm für die Kameras verwendeten Offscreen-Puffer und das Hauptmenü. Die Applikation besitzt die Methoden um als 'Open Delegate' fungieren zu können, Projektdateien (`.mms`, `.mmo` und `.rib` s.a. 3.2.2) können also vom **Workspace Manager**^{TM} aus geöffnet werden.

ModelMan_main: Eine Implementations-Datei (`.m`), die die vom **ProjectBuilder** generierte **main()** Funktion beinhaltet. Hier wird die Applikation instanziert und das Programm gestartet.

RibArchive: Das Objekt **RibArchive** zum Einlesen und Aufbewahren von RIB-Dateien ist in dieser Datei definiert. Das informelle Protokoll von **ArchiveHolderRemoval**, das zum Löschen nicht mehr benötigter RIB-Archive benötigt wird, wird von der Applikation implementiert.

ShapeBundle: Basisobjekt der Principal-Klassen[3] der Shape-Bundles. Eine Instanz wird mit ihrem **NXBundle** erzeugt (geschieht in **readBundlesFrom:** von **ModelMan**) und ist anschließend in der Lage, die Ressourcen des Bundles zu liefern. In allen Bundles muß die Principal-Klasse von **ShapeBundle** abgeleitet werden. Weil die gesamte Funktionalität schon in **ShapeBundle** implementiert ist, besitzt die Spezialisierung keine weiteren Methoden mehr.

CameraInspector

CameraInspector: Mehrseitiges Panel zur Darstellung des Zustands der aktuellen Kamera, eine Instanz der Klasse **InteractiveCamera**.

RenderInfo: Wird das photorealistische Rendern einer Szene gestartet, erzeugt der **CameraInspector** eine Instanz dieses Objekts und bewahrt es in einer Liste auf. Diese Instanzen helfen beim Darstellen der Identifikationen der laufenden Render-Jobs in einem Browser und nach Beendigung des Renderns bei der Identifizierung des 'Streams' mit dem Rasterbild. Nachdem die gerenderten Bilder weiterverarbeitet wurden, wird die zugehörende **RenderInfo** Instanz wieder gelöscht.

Font

FontBuffer: Puffer für einen Adobe Type 1 Font und die dazu benötigten Hilfsklassen **BinaryValue**, **PostScriptValue** und **ValueForDictionary**, sowie einige andere Deklarationen.

FontManager3D: Klasse zum Verwalten der Adobe Type 1 Fonts und Zeichen. Es existiert nur eine Instanz dieses Objekts. Momentan wird diese Instanz von einem **Text3D**-Shape erzeugt.

[3] Die Principal-Klasse wird als erstes geladen, ihr Klassenname muß mit dem Dateiname des Bundles (ohne Extension) übereinstimmen.

Glyph: Repräsentation eines Adobe Type 1 Zeichens

Glyph3D: Repräsentation eines für den RenderMan aufbereiteten Adobe Type 1 Zeichens

BuidChar Hilfsfunktionen zum Erzeugen einer **Glyph** Instanz aus einem Adobe Type 1 'Char-Path'.

buildCharName() Zeichen (**Glyph**) erzeugen, dessen Name bekannt ist

buildCharIndex() Zeichen (**Glyph**) erzeugen, dessen Index im 'Encoding-Vector' bekannt ist

HierarchyViewer

HierarchyViewer: Panel zur Darstellung der Shape-Hierarchie. Momentan ist nur ein **NXBrowser** zur Darstellung vorhanden. In der Datei ist auch eine Klasse **ShapeCell** als Kind von **NXBrowserCell** definiert. Instanzen dieser Klasse sind über einen Hilfszeiger mit dem zugehörenden Shape verbunden. Diese Zeiger wurden zur Identifizierung der Shapes benötigt, wenn der Benutzer innerhalb des Browsers selektiert. Der Browser zeigt auch die in einer Kamera-View gemachten Selektionen an. Innerhalb des Panels stehen zusätzlich Knöpfe zum Gruppieren, Degruppieren, Umgruppieren und Löschen von Shapes zur Verfügung.

Misc

Directory: Objekt zum Laden und Durchsuchen von Unix-Verzeichnissen. Zusätzlich werden einige Funktionen zur Dateibehandlung definiert:

basename(): Liefert den Basisnamen einer Datei in einem statischen Puffer (er wird bei jedem Aufruf der Funktion überschrieben). Dateiname, das abzutrennende Suffix und das Directory-Trennzeichen werden als Parameter übergeben.

cmpsuff(): Vergleicht den Suffix eines Dateinamens mit dem zweiten Parameter. Das Ergebnis entspricht dem von **strcmp()** .

fileSize(): Liefert die Größe der Datei, deren Deskriptor als Parameter übergeben wird.

home(): Liefert den Pfadnamen des Home-Verzeichnisses des aktuellen Users.

searchFile(): Sucht in den Verzeichnissen, deren Pfadnamen in einem String-Array übergeben werden, nach dem Auftreten einer Datei, deren Name ebenfalls als Parameter übergeben wird. Der komplette Pfadname (ein statisches Array innerhalb der Funktion) dient als Rückgabe.

DragButton: Es werden zwei Objekte definiert: **DragButton** und **IconButton**. **DragButton** wird in der **Palette** als Ausgangspunkt für das 'Drag&Drop' verwendet. Das Objekt legt einen frei wählbaren String in der **NXAsciiPboardType** Zwischenablage ab. Das Ziel der 'Drag&Drop' Operation der in der Palette verwendeten **DragButtons** ist eine **InteractiveCamera**.

IconButton dient als Zielobjekt für eine ‘Drag&Drop’-Operation. Das Objekt wird momentan im **ShaderParameters** Panel zum Aufnehmen eines Shaders oder einer Shader-Parameterdatei aus dem **File Viewer** eingesetzt und reagiert nur auf **NXFilenamePboard-Type** Zwischenablagen. Dateitypen und Zwischenablage-Typen sind jedoch einstellbar.

MyNXColorWell: Innerhalb von **ScrollViews** funktioniert das Update der normalen **NXColorWell** Objekte nicht richtig, deshalb wurde ein abgeleitetes Objekt, das sich immer vollständig zeichnet, verwendet. Das Objekt, das auf dem NeXTverfügbar ist, wird in **ColorInput** verwendet. Zusätzlich wurde die Archivierung überdefiniert, weil im Originalobjekt der Verweis auf das Zielobjekt einer Aktion nicht archiviert wurde.

StringContainer: Die Datei beinhaltet zwei einfache Hilfsklassen: **StringContainer** und davon abgeleitet **StringListContainer** zur Aufbewahrung von Strings bzw. String-Listen. Die String-Listen werden aufgebaut, indem ein String nach Trennzeichen abgesucht und an diesen Stellen aufgebrochen wird. Diese Art der String-Listen wird gebraucht, wenn sich mehrere Objekte in einer **NXFilenamePboardType** Zwischenablage befinden. Die Dateinamen sind in diesem Fall durch ein `TAB`-Zeichen voneinander getrennt. Beide Objekte fordern erst dann anderen Speicher an, wenn der String, den sie aufnehmen sollen zu groß wird, bzw. zu viele Zeiger für die Liste benötigt werden.

Utilities: Sammlung diverser Hilfsfunktionen, vor allem für Vektorarithmetik, Zeichenbehandlung und PostScript Zeichenroutinen.

Palette

Palette: Das Paletteobjekt ist zweigeteilt und dient zur Aufbewahrung von Icons von RIB-Makros und Shape-Bundles. Die Icons werden in **DragButtons** dargestellt. Der Benutzer kann ein Icon in eine **InteractiveCamera** View ziehen, die ein entsprechendes Objekt erzeugt. Die Kommunikation zwischen **DragButton** und **InteractiveCamera** erfolgt über eine **NXAsciiPboardType** Zwischenablage. Der Inhalt ist bei RIB-Makros der String: “`/RIBEntity/makroname`”, bei Shapes-Bundles einfach nur der Klassenname des Principal-Objekts. Die Namen werden unter dem Icon in den **DragButtons** dargestellt. Die **InteractiveCamera** interpretiert, nachdem ein Icon über der View fallen gelassen wurde, den String aus der Zwischenablage und fordert von ihrem Distributor, der Applikation, eine entsprechende Objektinstanz an. Das Objekt wird an der Stelle an der das Icon losgelassen wurde mit einem Standardabstand von der vorderen Clip-Ebene der Kamera dargestellt.

RenderUtilities

GeneralPolygon: Objekt zur Aufbewahrung von Polygonpunkten für eine Menge von **RiGeneralPolygon()**-Polygonen — gebraucht für die Verwendung von Adobe Type 1 Zeichen.

ParameterVector: Hilfsklasse zur Berechnung von Bézier–Kurven. Von dieser Klasse wird nie eine Instanz gebildet, sie besitzt nur Klassenmethoden. Eine Funktion **fillXYCurve-To()** dient zum Füllen eines **float**-Arrays mit den für eine Bézier-Kurve berechneten 3D-Punkten.

Polygon: Kapselung eines einfachen Polygons (knvex ohne Löcher) in eine Objektklasse. Die einfachen Polygone werden zum Abschließen unvollständiger Kugeln und Paraboloide beim Rendern mit dem *Quick RenderMan* gebraucht, falls der Diskus nicht verwendet werden kann.

PolyPath: Klasse zum Anlegen komplexer Polygonpfade aus Bézier- und Liniensegmenten mit 'moveTo', 'lineTo', 'curveTo' und 'closePath'. Diese Art von Pfad wird für Adobe Type 1 Zeichen verwendet. Zusätzlich sind die Funktionen **pscriptTransformX()** und **pscriptTransformY()** zur Transformation von 2D-PostScript Koordinaten (vom Typ **long**) vorhanden. Die Funktionen mit der Endung 'd' dienen zur Transformation von 2D-PostScript Koordinaten vom Typ **double**.

RenderUtilities: Momentan befindet sich hier nur eine Funktion: **closedCylinder()** zur Ausgabe der Interface-Funktionen für einen geschlossenen Zylinder. Dieser Zylinder wird bei der RIB-Ausgabe auch für geschlossene, unvollständige Kugeln und Paraboloide verwendet. Mit Hilfe der CSG Schnittmengen-Operation von Zylinder und Grafikobjekt, können diese Objekte auf einfache Weise in der Auflösung des jeweiligen Renderers erzeugt werden — werden die Objekte mit Polygonen geschlossen, muß eine Aufteilung der Fläche gewählt werden.

Scanner

DBuf: In dieser Datei ist ein Doppelpuffer-Objekt **DoubleBuffer** implementiert. Es wird von dem ASCII-Scanner **Scanner** als Basisobjekt verwendet. Es kann aus einem Speicherpuffer, einem 'Stream' oder einer Datei gelesen werden.

Scanner: Scanner für Ascii-Dateien, der Scanner teilt nacheinander eine Datei in Tokens auf, die weitergereicht werden. Der Scanner kann Real-Zahlen, Strings und Identifier auswerten. Kommentarzeilen, die mit einem '#' beginnen können automatisch überlesen werden. Der Scanner wird für das Auswerten der Shader-Parameterdateien verwendet. Durch das Setzen einer PostScript Option kann der Scanner auch für die Auswertung von Adobe Type 1 Fontdateien verwendet werden.

ShapeInspector

ColorInput: Farb-Eingabe, wird im **ShaderParameters** Panel verwendet. Alle **...Input** Objekte werden nur einmal von der korrespondierenden `.nib`-Datei geladen. Danach werden sie von einem 'Memory Stream' dearchiviert.

ISInspectorPages: Basisobjekt für die zusätzlichen **ShapeInspector** Seiten in den Bundles. Abgeleitete Objekte müssen bestimmte Methoden dieses Objekts überdefinieren, damit der **ShapeInspector** das Einfügen, die Anzahl und die Namen der Seiten behandeln kann. In dem Objekt ist als Feldvariable die aktuelle Kamera und Shape-Instanz eingetragen. In den abgeleiteten Objekten sollte immer eine Seite 'Attributes' die erste Seite sein.

PointInput: Eingabezeile für einen 3D Punkt im **ShaderParameters** Panel.

ScalarInput: Zahleneingabezeile für das **ShaderParameters** Panel.

ShaderParameters: Panel zur Darstellung und Eingabe der Parameter eines Shaders. Die Namen der Shader werden nach ihren Typ aufgeteilt in einer **PopUpList** dargestellt. Der Benutzer kann zwischen den Shader-Typen und den Shader-Namen auswählen. Je nach eingestelltem Shader können in einer **ScrollView** die Parameter eingegeben werden. Das **ShaderParameters** Objekt stellt Methoden zum Aufruf des Speicherns und Ladens der Parameter bereit. In das Panel ist ein **IconButton** integriert, in den ein Icon eines Shaders oder einer Parameterdatei vom **File Viewer** gezogen werden kann. Aus den eingegebenen Parametern kann eine **InteractiveShader** Instanz erzeugt werden. Damit ein **ShapeInspector** seine verwendeten Shader verwalten und Shader-Instanzen übernehmen kann, ist ein informelles Protokoll als **Object**-Kategorie **ShaderParameterDelegate** deklariert.

ShaderTable: Objekt, das alle verfügbaren Shader verwaltet. In einer Hashtabelle (Schlüssel ist der Basisnamen des Shaders) werden die Shader als **InteractiveShader** Instanzen verwaltet. Jedem Shader darf ein Icon zugeordnet sein, das im **ShaderParameters** Panel in dem **IconButton** dargestellt wird. Die Icons sind in einer anderen Hashtabelle aufbewahrt. Sie zeigen normalerweise einen Ausschnitt aus einer Oberfläche, die mit dem zugehörenden Shader gerendert wurden. Die Shader-Tabelle wird zum Programmstart von der Applikation aufgebaut. Der **ShapeInspector** verwaltet sie nach seiner Erzeugung. Die Tabelle wird von der Applikation aufgebaut.

ShapeInspector: Inspektor-Panel für einzelne Shapes. Das Panel kann je nach Shape-Typ neue Seiten vom Typ **ISInspectorPages** nachladen. Eine Kategorie **ShapeNotification** wurde deklariert, um ein informelles Protokoll zu definieren, das es ermöglicht, von einer verwaltenden Objektinstanz (die Applikation **ModelMan**) die **ISInspectorPages** Instanz und ein Icon der Shape-Klasse eines selektierten Shapes geliefert zu bekommen.

StringInput: Eingabezeile für Text, wird im **ShaderParameters** Panel verwendet.

ThumbWheel

ThumbWheel: Das Daumenrad-Eingabeobjekt von Jeff Martin. Zum Daumenrad gehört **TWCell** und die Kategorie **TWCellDraw**. Das Daumenrad wird immer dann verwendet, wenn eine ‘Delta-Eingabe’ (z.B. die Einstellung der Entfernung der Kamera vom Welturspung) gefordert ist. Unmodifizierte Slider sind für diese Eingaben nicht verwendbar, weil sie einen beschränkten Wertebereich besitzen. Das Objekt wurde so erweitert, daß einem **trackNotifier** eine Nachricht gesendet werden kann, wenn mit der Maus in das Innere der zugehörenden View geklickt wird. Die Kamera hat in diesem Fall die Möglichkeit ihre Interpolationsgenauigkeit herabzusetzen, wodurch schnellere Reaktionszeiten bei Kamera- und Shape-Bewegungen erreicht werden. Ein Fehler beim Umschalten zwischen den Darstellungsformen des Objekts wurde beseitigt.

TWCell: Inneres des Daumenrades, reagiert auf Mausbewegungen

TWCellDraw: Zeichenroutinen des Daumenrades

Zusätzlich zu den Programmdateien sind noch einige Images für die Icons und Hilfstexte vorhanden. Als Unterprojekte existieren einige Bundles (‘`.bproj`’), die im folgenden Abschnitt erläutert werden.

Die Bundles

Alle verwendeten Bundles ([NeXTDoc], Objective C) besitzen Dateien gleichen Types: Eine NIB-Datei mit dem Interface für die **ISInspectorPages** Seiten, ein abgeleitetes **ISInspectorPages** Objekt, das die Funktionalität der Inspektorseiten definiert, das Shape-Objekt (abgeleitet von **InteractiveShape**) mit der Schnittstelle zum RenderMan und ein Principal-Objekt (abgeleitet von **ShapeBundle**), das nichts tut, außer dem Bundle seinen Namen zu geben; die Funktionalität steckt schon im Elternobjekt. Zusätzlich sollte eine '`.tiff`' Icon-Datei für die Palette vorhanden sein. Alle Namen sollten mit dem Bundle-Namen beginnen. Im Beispiel **Cone** Bundle befinden sich die Dateien (außer den vom **ProjectBuilder** generierten):

ConeInspector.h/.m: Interface und Implementation der Inspektorseiten

ConeShape.h/.m: Das Shape-Objekt

Cone.h/.m: Das Principal-Objekt des Bundles

Cone.nib: Das Interface mit dem 'Owner' **ConeInspector**

Cone.tiff: Das Icon, max. 50×50

PB.project: Die Projektdatei für den **ProjectBuilder**, erzeugt durch die Funktion 'New Subproject...' 'Bundle'.

Möchte man ein eigenes Bundle erzeugen, müssen diese Dateien vorhanden sein und die benötigten Include-Pfade im `makefile.preamble` gesetzt werden. Die Schnittstellen zum Hauptprogramm sind über die Methodenvererbung definiert. Von **ShapeBundle** reicht es, eine 'leere' Klasse abzuleiten, die dem Bundle den Namen gibt, Bsp. **Cone**:

```
/////////
// Cone.h

#import "ShapeBundle.h"

@interface Cone:ShapeBundle
{
}
@end

/////////
// Cone.m

#import "Cone.h"

@implementation Cone
@end
```

Die Methoden der Klasse: Laden der Ressourcen und der anderen Klassen, werden schon von der Elternklasse **ShapeBundle** zur Verfügung gestellt.

Etwas mehr muß für die abgeleitete **ISInspectorPages** Klasse, im Beispiel **ConeInspector** getan werden, wenn das Shape eigene Inspektorseiten besitzen soll. Es muß ein entsprechendes Interface mit der **ConeInspector** Klasse als 'Owner' mit dem **InterfaceBuilder** definiert werden. Die Größen der Interface-Seiten (gruppierte Bedienelemente) müssen auf den **ShapeInspector** abgestimmt sein. Die Textfelder der abgeleiteten **ISInspectorPages** Klasse werden im *InterfaceBuilder* durch das 'Outlet' 'nextText' verknüpft, damit ein Benutzer mit Hilfe der Tabulator-Taste das jeweils nächste Feld selektieren kann. Die abgeleitete Klasse ist für alle Textfelder ein 'textDelegate'. Diese Verbindung wird ebenfalls für die Textfelder im *InterfaceBuilder* gesetzt. Die Delegate-Methoden **textDidChange:** (setzt den 'Edier-Status' des Fensters des Absenders) und **textDidEnd:endChar:** (ruft die eigene **usePage:** mit der aktuellen Seitennummer **currPage** auf und setzt den 'Edier-Status' des Fensters des Absenders zurück) sind schon in **ISInspectorPages** definiert. Die Beispiel-Klasse **ConeInspector** muß die Funktionalität des Interfaces implementieren und zusätzlich einige Methoden des Elternobjekts **ISInspectorPages** überdefinieren:

'nPages' liefert die Anzahl der vorhandenen Seiten. Wird kein eigenes Interface gewünscht, muß diese Methode 0 liefern oder nicht in der Klasse definiert sein (die Methode der Elternklasse liefert schon 0).

'page:' soll die Gruppe einer Seite ($0..nPages-1$) liefern.

'pageName:' soll den Namen einer Seite ($0..nPages-1$) liefern, der Name wird für die **PopUpList** des Inspektors benötigt.

'loadPage:' diese Methode wird aufgerufen, wenn eine Seite ($0..nPages-1$) mit Werten aus dem aktuellen Shape (**ISInspectorPages** Member: **shape**) aufgefüllt werden soll. Der abgeleitete Inspektor muß die Methode des Elternobjekts ausführen, weil sie die aktuelle Seitennummer (**currPage**) setzt.

'usePage:' diese Methode wird aufgerufen, wenn aus einer Seite ($0..nPages-1$) die Werte aus den Textfeldern auf das Shape angewendet werden sollen. Nach der Ausführung soll der 'Edier-Status' des Fensters (das ist das Shape-Inspektor Panel) der aktuellen Seite mit **setDocEdited:** aufgehoben werden.

'initWithResource:' Diese Methode braucht nur dann überdefiniert werden, wenn ein Bundle keine eigenen Inspektorseiten besitzt. In diesem Fall muß mit einem Aufruf von `[super init]` das Laden einer Interface-Datei umgangen werden (s. `Teapot.bproj`)

Das Shape selber wird als Beispiel von einer Klasse **ConeShape** beschrieben. Diese Klasse kann einige Methoden definieren, die bei Bedarf auch wegfallen können:

'init' Initialisierung des Objekts.

'read:' Lesen von einem typisierten 'Stream'.

'write:' Schreiben auf einen typisierten 'Stream'.

'awake' Durchführung von Initialisierungen nach dem Lesen von einem typisierten 'Stream'.

'renderSelf:' RenderMan Interface-Aufrufe zum Rendern in der Sicht der Kamera, die als Parameter mitgegeben wird. Ist `[self isClosed]` wahr, sollte dafür gesorgt werden, daß die Oberflächen zu Körpern geschlossen werden.

'calcBBX' Neuberechnung der Hülle des Objekts in Objektkoordinaten, falls diese aus Daten gewonnen werden kann.

Besitzt ein Shape eigene Kontrollpunkte (s. `Patch.bproj`), müssen die folgenden Methoden überdefiniert werden:

'hasOwnHandles' Soll **YES** liefern, wenn das Shape eigene Kontrollpunkte besitzt.

'hasSelectedHandles' Soll **YES** liefern, falls Kontrollpunkte selektiert sind.

'handleInteriorAt::camera:event:' Wird von der Kamera aufgerufen, wenn ein Shape Kontrollpunkte besitzt und in das Innere des Shapes geklickt wurde. Das Shape muß in dieser Methode den Mausklick in geeigneter Art und Weise behandeln. Dazu kann die Methode **moveSelectedHandlesFrom:** der **InteractiveCamera** nach der Durchführung einer Selektion aufgerufen werden.

'moveSelectedHandlesBy::camera:' Selektierte Kontrollpunkte sollen relativ um Pixelkoordinaten im Koordinatensystem einer Kamera verschoben werden, die Methode **getDistanceInWorldSpaceFrom:by::camera:** der Klasse **InteractiveShape** kann dazu verwendet werden.

'moveSelectedHandlesInWorld:' Selektierte Kontrollpunkte sollen relativ um eine Strecke im Weltkoordinatensystem verschoben werden.

'drawPS:::' Darstellung der Kontrollpunkte in einer Kamera-Sicht. Farbe und Größe der Kontrollpunkte werden von der Kamera bestimmt. Die Werte können durch die Methoden **selectionColor** (Grauwert der Kontrollpunkte) **invertedSelectionColor** (Grauwert der Umrandung der Kontrollpunkte) und **getHandleSize:** abgefragt werden. Die Methode wird nicht aufgerufen, wenn sich das Shape hinter der Kamera befindet.

Sollten in der **renderSelf:** verschachtelte Solid-Blöcke ausgegeben werden, muß die folgende Methode überdefiniert werden (geschehen in **SphereShape** und **ParaboloidShape**):

'getPrimitiveToken' Liefern des Tokens (**RI_UNION**, **RI_INTERSECTION**, **RI_DIFFERENCE**) für einen Solid-Block, mit dem der Aufruf der **renderSelf:** Methode von der Methode **render:** geschachtelt werden soll (s.a. 3.1.3).

Zusätzlich kann es nützlich sein, spezielle Methoden zur Kommunikation mit den eigenen Inspektorseiten der Art **setAttributes:** und **getAttributs:** zu implementieren, die die Daten einer Seite in einer dafür geeigneten Struktur im- und exportieren können.

In der momentanen Version von **ModelMan** sind folgende Bundles vorhanden:

- Cone

- Cube, ein aus Würfel aus Polygonen, die einzelnen Seiten lassen sich einzeln abschalten
- Cylinder
- Disk
- Hyperboloid
- Paraboloid
- Patch, linearer ‘Patch’, die vier Kontrollpunkte lassen sich verschieben, der Patch kann extrudiert werden
- Sphere
- Teapot, vordefiniertes Objekt des RenderMan Interfaces, kein Inspektor
- Text3D, 3D Text aus Adobe Type 1 Zeichen, zwei Inspektorseiten
- Torus

3.1.3 Einige Programmdetails

Das Selektieren

Die Software des NeXT unterstützt den Programmierer bei der Aufgabe Shapes an der Mausposition zu selektieren (picken). Die Kamera stellt die Methode **selectShapesIn:** zur Verfügung, die Zeiger auf alle Objekte, die in einem Rechteck von Bildschirmkoordinaten liegen, in einer Liste zurückgibt. Die Verwaltung dieser Liste übernimmt die **N3DCamera** selbst. Die Funktion reicht aber noch nicht aus, damit der Benutzer in gewohnter Art und Weise Shapes selektieren kann. Für eine Mehrfachselektion müssen Verweise auf schon gepickte Shapes in einer eigenen Liste gespeichert werden. Zusätzlich muß dafür Sorge getragen werden, daß sich der Benutzer die Objekthierachie ebenenweise ‘hinunterhangeln’ kann. Es sollen auch Gruppen, d.h. nicht nur primitive Objekte, selektiert werden können. Selektionen sollen auch wieder gelöscht werden können. Das Picken der Anfaßpunkte der Rechteckhüllen und der Shape-eigenen Kontrollpunkte muß gesondert vor der Shape-Selektion betrachtet werden. Wenn Objekte durch andere überdeckt werden, können diese nicht mit der Maus gepickt werden. Ein Selektieren über den **HierarchyViewer** und ein Manipulieren schon selektierter Shapes mit den Kontrollelementen ist aber immer möglich. Zur Not können die verdeckenden Objekte vor dem Picken auch unsichtbar geschaltet werden.

Das Picken selbst ist ein relativ komplexer Ablauf, der am besten grafisch dargestellt werden kann. In Abb. 3.3 ist deshalb ein Automat gezeigt, an dem das Picken erläutert wird. Die Funktionen zum Picken von Shapes sind in der **mouseDown:** Methode der **InteractiveCamera** implementiert. Dort wird zuerst überprüft, ob mit einem Mausklick ein Kontrollpunkt[4] eines schon selektierten Objekts getroffen wurde. Ist das der Fall, wird dem Benutzer ermöglicht, mittels Mausverschiebung die Größe der Bounding Box und damit die Größe des Objekts selbst zu

[4] Entweder einer der acht Anfaßpunkte der Objekthülle oder Shape-eigene Kontrollpunkte

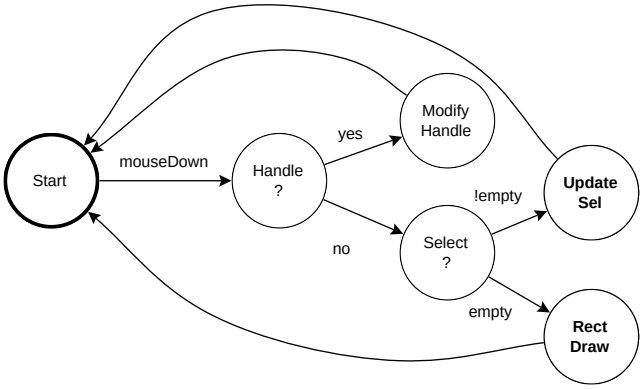

Abbildung 3.3: Das Picken mit der Maus

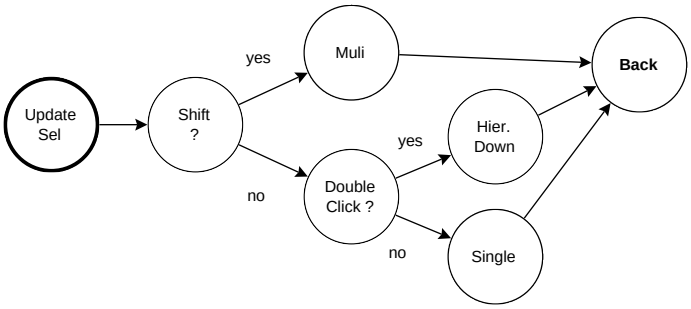

Abbildung 3.4: Update der Selektionsliste

ändern oder Punkte innerhalb des selektierten Shapes zu verschieben. War kein Kontrollpunkt getroffen, wird mit **selectShapesIn:** die Liste der Objekte bestimmt, die in einem Standardselektionsrechteck um den Mauszeiger liegen. Konnten keine Objekte gefunden werden, kann der Benutzer ein Selektionsrechteck aufziehen, andernfalls wird die Selektionsliste auf den neusten Stand gebracht. Ist im letzteren Fall zusätzlich zum Mausknopf die Shift-Taste gedrückt, geschieht eine Multiselektion: Die Objekte aus der Selektionsliste werden den schon selektierten Objekten in einer die Shape-Hierarchie beachtenden Weise hinzugefügt. Ist nur ein Objekt in der Liste und ist dieses schon selektiert, wird seine Selektion gelöscht (Toggle Select). War die Shift-Taste nicht gedrückt wird überprüft, ob es sich bei dem Mausknopfdruck um einen Doppelklick handelt, im positiven Fall wird nach Möglichkeit in der Shape-Hierarchie eine Ebene tiefer selektiert. War kein Doppelklick aufgetreten, wird die Selektionsliste, wenn ein nichtselektiertes Shape gepickt wurde, erneuert. Wurde ein selektiertes Shape gepickt, kann es mit der Maus verschoben werden. Die Rubberband-Selektion geschieht ähnlich wie die normale Selektion. Alle Objekte, die innerhalb eines Rechtecks selektiert wurden, werden behandelt. Sie werden, wenn die Shift-Taste gedrückt ist, den schon selektierten Objekten hinzugefügt. Das

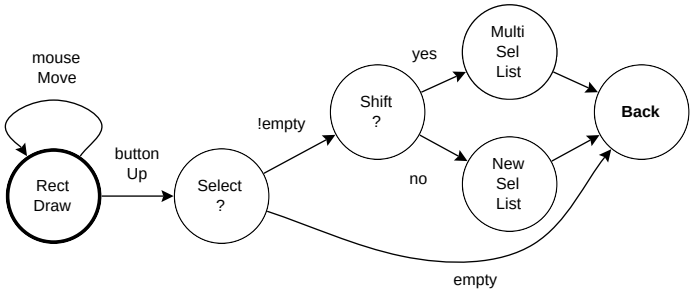

Abbildung 3.5: Selektieren mit Rubberbox

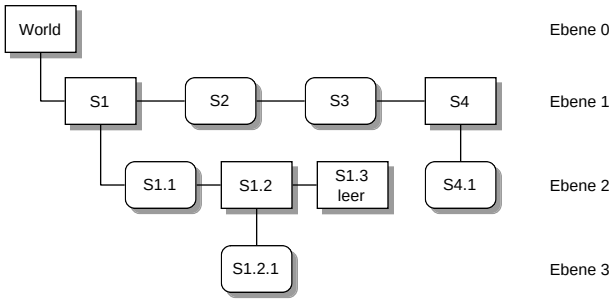

Abbildung 3.6: Shape-Hierarchie

‘Togglen’ von schon selektierten Objekten wurde als eher verwirrend empfunden und deshalb nicht ausgeführt. Ist die Shift-Taste nicht gedrückt wird die Selektionsliste durch die neue Liste erneuert.

Bei allen Selektionen muß beachtet werden, daß nur auf der gleichen Hierarchie-Ebene im gleichen Ast mehrfachselektiert werden kann (s. Abb. 3.6). Wird das nicht getan, kommt es zu Inkonsistenten mit dem ‘Browser’ des **HierarchyViewers**, in dem prinzipiell nur auf einer Ebene eines Astes (in einer Spalte) mehrfachselektiert werden kann. Mit Doppelklicks kann jeweils eine Ebene tiefer selektiert werden. Es können so auch Shape-Gruppen wie **S1** und **S4** selektiert werden, die selbst keine Oberflächen rendern. Alle Elternknoten von selektierten Knoten gelten auch als selektiert, sie können aber nicht mehr direkt manipuliert werden — sie bilden einen Selektionspfad zu den selektierten Objekten auf tieferer Ebene. Die Kamera verwendet zwei Listen für die Selektion: zum einem die Liste aller selektierten Objekte, die vor allem für das Zeichnen der Shape-Hüllen verwendet wird, zum anderen eine Liste der selektierten Objekte auf unterster Ebene, die manipuliert werden können. Das Weltobjekt wird als Ausnahme behandelt. Es gilt als immer selektiert — die Selektion wird aber nie dargestellt — und es kann im Kamerafenster nicht manipuliert werden. Ist ein Shape selektiert, hat es intern eine Selektions-Marke gesetzt. Selektierte Shapes können so auf einfache Weise auch in den möglicherweise vorhandenen zusätzlichen Views der Shape-Hierarchie dargestellt werden.

Es können leere Gruppen in der Shape-Hierarchie vorhanden sein. Gruppen werden nicht automatisch gelöscht, wenn alle ihre Mitglieder gelöscht sind. Der Benutzer kann eine Gruppe auf diese Weise nachträglich wieder auffüllen. Leere Gruppen können nur über den **HierarchyViewer** selektiert werden.

Das Einfügen von Objekten

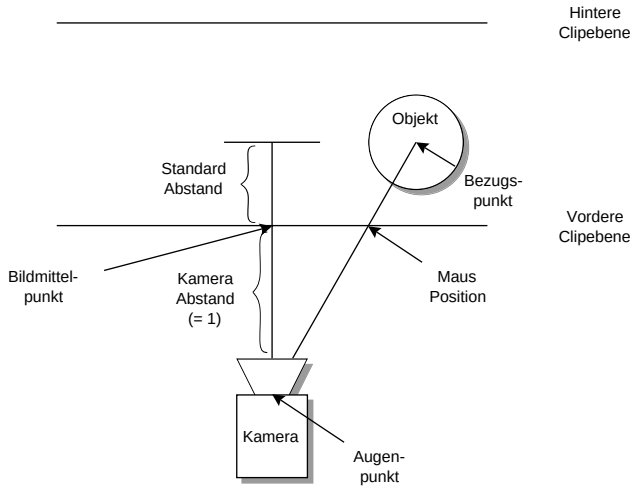

Abbildung 3.7: Das Einfügen von Objekten mit der Maus

Objekte werden immer unter der aktuell selektierten Gruppe an der Mausposition eingefügt. Die Methode **insertShape:at:** von **InteractiveCamera** wird zum Einfügen von Objekten verwendet. Die Methoden zur Projektion von den Bildschirmkoordinaten auf die vordere und hintere Clip-Ebene können als Ausgangspunkt zur Positionsbestimmung verwendet werden. Um Rechenfehler zu vermeiden wird die vordere Clip-Ebene vor der Berechnung auf den Wert 1.0 gesetzt. Damit alle Objekte mit ihrem Bezugspunkt an der Mausposition auf einer Ebene plaziert werden können, ist bei der perspektivischen Projektion eine kleine Berechnung notwendig (s. Abb. 3.7). Der Vektor vom Augenpunkt zur Projektion vom Bildmittelpunkt auf die vordere Clip-Ebene steht bei der verwendeten Projektion senkrecht auf dieser Ebene. Er kann also dazu verwendet werden, die zur vorderen Clip-Ebene parallele Ebene, auf der plaziert[5] werden soll, zu bestimmen. Diese Ebene hat zur vorderen Clip-Ebene den für das Programm fest gewählten, prinzipiell aber beliebigen, Abstand von 5.0. Das Verhältnis vom Abstand des Augenpunkts zur vorderen Clip-Ebene zu dem Abstand des Augenpunktes zur Plazierungsebene entspricht dem Verhältnis des Abstandes des Augenpunktes zur Projektion des Mauspunktes auf die vordere Clip-Ebene und dem Abstand des Augenpunktes zu dem Punkt auf der gedachten Plazierungsebene (s.a. [Mathe], Strahlensätze Abb. 7.8–3a). Der Punkt (in Weltkoordinaten) kann nach

[5] Es wurde nicht die vordere Clip-Ebene selbst verwendet, damit die Objekte immer vollständig sichtbar sind.

dem Auflösen der Verhältnisgleichung durch eine die entsprechende Verlängerung[6] des Vektors vom Augenpunkt zum projizierten Mauspunkt gefunden werden. Mit dem Einfügen des Shapes im Weltkoordinatensystem und dem anschließenden Gruppieren mit **group:** zur aktuellen Gruppe wird erreicht, daß die Welt-Position des Objekts in dem lokalen Koordinatensystem seiner Gruppe erhalten bleibt.

Das Verschieben von Objekten

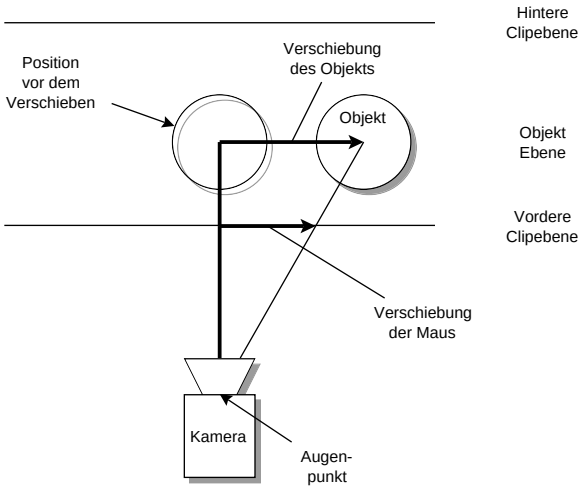

Abbildung 3.8: Das Verschieben von Objekten mit der Maus

Wurde ein selektiertes Objekt mit der Maus gepickt, kann es verschoben werden. Dazu muß, wie beim Plazieren zwischen paralleler und perspektivischer Projektion der Kamera unterschieden werden. Bei einer Parallelprojektion ist das Verschieben trivial, es wird der Vektor verwendet, der auf der vorderen Clip-Ebene ermittelt wurde. Bei der Parallelprojektion hilft, ähnlich wie beim Einsetzen eines Objekts, das Verwenden einer Verhältnisgleichung weiter. Die Verschiebung wird im Weltkoordinatensystem vorgenommen. Prinzipiell wird der Punkt-Richtungs-Vektor der Mausbewegung auf die Ebene des zu verschiebenden Objektes projiziert. Das hat zur Folge, daß weiter entfernte Objekte mehr verschoben werden als nähere[7] — außer sie werden innerhalb einer Gruppe verschoben. Die Länge des Verschiebungsvektors wird gewonnen, indem man das Verhältnis des Abstands vom Augenpunkt zum alten Mauspunkt zur Länge der Mausverschiebung ermittelt. Dieses ist äquivalent mit dem Verhältnis des Objektabstandes zur Länge der gewünschten Verschiebung (s.a. [Mathe], Strahlensätze Abb. 7.8–5), die Länge ist also: (Länge der Verschiebung auf der vorderen Clip-Ebene / Abstand des alten Mauspunktes) * Objektabstand. Der Verschiebungsvektor wird in der Methode **getDistanceInWorldSpaceFrom:by::camera:** der Klasse **InteractiveShape** in Weltkoordinaten berechnet: (Objektabstand

[6] Bei einem (vor dem Einfügen temporär gesetzten) Kameraabstand von 1 und einem ensprechend der aktuellen vorderen Clip-Ebene verlängertem Standardabstand einfach: Kameraabstand + Standardabstand

[7] Das möchte man auch.

/ Abstand des alten Mauspunkts) * Verschiebung auf der vorderen Clip-Ebene.

Transformationen

Bei Transformationen muß zwischen Transformationen im Objektkoordinatensystem und im Kamerakoordinatensystem unterschieden werden. Transformationen im Kamerakoordinatensystem werden über das Dokumentenfenster gesteuert. Die Transformationen im Objektkoordinatensystem können im **ShapeInspector** gemacht werden. Um direkt im Kamerakoordinatensystem transformieren zu können, werden die Shapes vor der Transformation in das Kamerakoordinatensystem transformiert und nach der Ausführung der gewünschten Transformation durch Rechtsmultiplikation der neuen Transformationsmatrix zurück in das Koordinatensystem ihrer Gruppe transformiert. Das Ausfügen und Wiedereinfügen eines Shapes hat zur Folge, daß die Transformationen einer Gruppe sich nicht auf Ausführung von Transformationen auf das Shape (bei einer Rechtsmultiplikation der Matrix) auswirken. Ist eine Gruppe z.B. in Y-Richtung vergrößert, würde eine Drehung eines ihrer Shapes dieses verzerren, wenn es nicht vor der Transformation aus der Gruppe genommen wird. Die Verzerrung träte auf, weil die Drehung vor der Transformation der Gruppe (der Skalierung) ausgeführt würde.

Die Koordinatensysteme der Shapes sind hierarchisch angeordnet. Die Matrix **transform** jedes Shapes transformiert in das jeweils hierarchisch übergeordnete Koordinatensystem. Jedes Shape enthält zusätzlich eine ‘Composite Transformation Matrix’ (CTM, **compositeTransform**), die vom Objektkoordinatensystem des Shapes in das Referenzkoordinatensystem transformiert. Die Inverse zur CTM (**inverseCompositeTransform**) zur Transformation vom Referenzkoordinatensystem in das Objektkoordinatensystem ist ebenfalls ein Feld des Objekts. Das Gültighalten dieser Matrizen übernimmt 3DKit. Die CTM ist durch Rechtsmultiplikation der Matrizen **transform** (Pfeilrichtung der Abbildung 3.9) erzeugt. Jedes Shape besitzt Methoden, um im eigenen Objektkoordinatensystem (Linksmultiplikation ‘premultiply’ der entsprechenden Transformationsmatrix an die aktuelle **transform**) und in dem Koordinatensystem seiner Gruppe (Rechtsmultiplikation der entsprechenden Transformationsmatrix an die aktuelle **transform**) transformiert zu werden.

Die Transformationen, die im Objektkoordinatensystem eines Shapes ausgeführt werden, können im ‘Shape Inspector’ **ModelMans** eingegeben werden. Das Programm baut aus diesen Angaben in der Reihenfolge: Skalieren, Dehnen und Rotieren um die X, Y, Z Achse des Shapes eine Transformationsmatrix auf (**N3DShape getLocalTransform:**), die mit der aktuellen **transform** kombiniert wird. Durch die Transformationen im Objektkoordinatensystem kann die Ausgangsform eines Objekts bestimmt werden, bevor es in eine Gruppe eingefügt wird.

Die 3D Fonts

ModelMan besitzt die Möglichkeit, aus Adobe Type 1 Zeichensätzen (im folgenden kurz *Fonts* genannt) durch Extrusion 3D-Zeichen zu erzeugen. Eine Zeichendefinition besteht jeweils aus einer Anzahl von Pfadbeschreibungen (Bézier- und Liniensegmente für Umrandungen gegen den Uhrzeigersinn und für ‘Löcher’ im Uhrzeigersinn) und einigen weiteren Angaben (vor allem der Dicke[8], die zur Aneinanderreihung der einzelnen Zeichen benötigt wird). Der Zeichensatz enthält neben den Zeichenbeschreibungen eine homogene 2D-Transformationsmatrix, die (nach der Anwendung einer Skalierung) dazu verwendet wird, aus den Zeichenbeschreibungen

[8] Breite der Zeichen von einem Bezugspunkt zum Nächsten

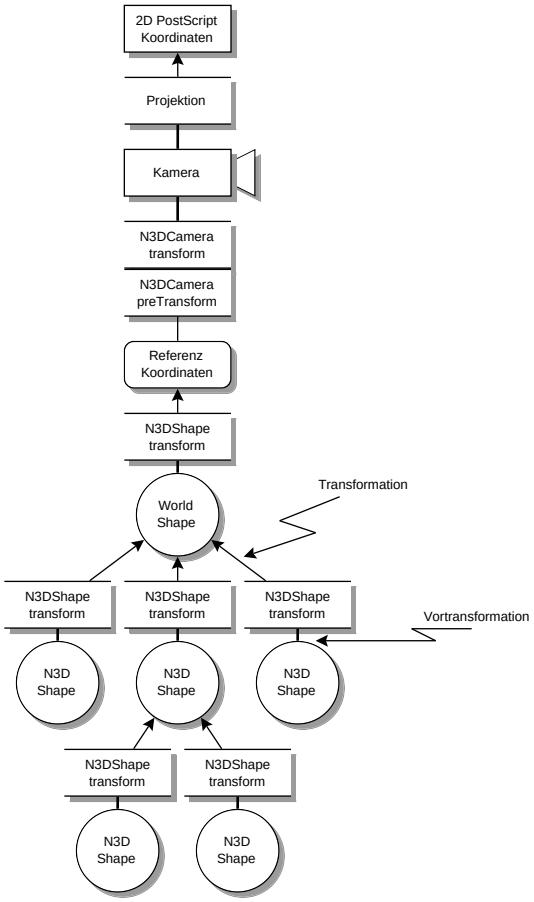

Abbildung 3.9: Transformationen zwischen den Koordinatensystemem

Zeichen zu generieren — schräggestellte (‘oblique’) Zeichensätze unterscheiden sich von den geraden Ausgangszeichensätzen oft nur durch die Transformationsmatrix, die eine Dehnung enthalten kann.

Um die Zeichensatzdateien verwenden zu können, war es zunächst notwendig, einen Scanner für das ‘Dateiformat’ (PostScript mit festgelegter Semantik der verwendeten Prozeduren) zu schreiben. Die Formatbeschreibung der Zeichensatzdateien kann in dem Buch [AdobeType1] nachgeschlagen werden. Die Dateien werden in Instanzen des Objekts **FontBuffer** geparst (**readFromFile:**), das auch die nötigen Zugriffsfunktionen (**useGlyph:**, **unuseGlyph:**) auf die einzelnen Zeichenbeschreibungen (hier Glyphs genannt) enthält. Die PostScript ‘Dictionary’ Strukturen wurden dazu in Hashtabellen abgebildet; als Schlüssel dienen die Namen; Werte sind Instanzen des Objekts **ValueForDictionary**. PostScript Vektoren (Arrays) werden in Listen gehalten. Der ‘Standard Encoding’ Vektor, mit dessen Hilfe zu einem Zeichenkode der Zeichenname (z.B. zu dezimal 48 der Name ”0”) gefunden wird, ist an den Zeichensatz des NeXTs

angepaßt.

Wird aus einer **FontBuffer**-Instanz ein Zeichen angefordert, wird (falls noch nicht geschehen) ein Instanz der Klasse **Glyph** erzeugt, die die Pfadbeschreibung **pathList** des Zeichens enthält. **pathList** ist eine Liste aus den Bézier- und Liniensegmenten, beinhaltet aber noch keine Polygondaten. Intern ist eine **Glyph**-Instanz (doppelt) mit dem zugehörigen PostScript-Wert (einem 'CharString' in einer **BinaryValue** Instanz) verkettet. **Glyph**-Instanzen werden, wenn sie nicht mehr verwendet werden, aus dem Speicher gelöscht.

Aus den **Glyph**-Instanzen können wiederum **Glyph3D** Instanzen gebildet werden, die die konkreten Polygondaten aus den Pfadbeschreibungen generieren und RenderMan Interface-Aufrufe zur Darstellung der Zeichen absetzen. Die Anwendung der PostScript Transformationsmatrix und die Extrusion erfolgt bei der Generierung der Interface-Aufrufe. Ein 3D-Zeichen, das aus mehreren Teilpfaden aufgebaut sein kann, besteht aus Umrandungen von bilinearen und bikubischen 'Patches' und optionalen 'Deckeln', die aus allgemeinen Polygonen **RiGeneralPolygon()**[9] aufgebaut werden. Die Polygonbeschreibungen werden von den **Glyph3D**s in einer Instanz der Klasse **GeneralPolygon** gehalten. Die Klasse besitzt die Methode **calcDirections:**, die die Polygone in Umrandungen und enthaltene Löcher aufgliedern kann. Zur Erzeugung der Polygondaten aus der Pfadbeschreibung wird die aktuelle Granularität **fidelity** herangezogen. Sie kann vom Benutzer allgemein für alle Shapes im Kamerainspektor gesetzt werden. Die Berechnungsformel für Bézierkurven hat folgenden Aufbau (s.a. [HosL91]):

$$B_i^n(t) = \begin{pmatrix} n \\ i \end{pmatrix} \cdot (1-t)^{n-i} \cdot t^i$$

$$X(t) = \sum_{i=0}^{n} B_i^n(t) \cdot \mathbf{P_i}$$

n : Grad des Polynoms, ist bei kubischen Bézierkurven 3

t : Parameter der Funktion, $0 \leq t \leq 1$

$B_i^n(t)$: Bernsteinpolynom i

$\mathbf{P_i}$: Bézierpunkt **i**

$X(t)$: Kurvenpunkt an der Stelle t

Bei der Erzeugung von Polygondaten aus Bézierkurven sind viele Berechnungen nur vom aktuellen Parameter t der Funktion abhängig — die Bernsteinpolynome können auch ohne die Kenntnis der Punkte, auf die sie angewendet werden, ausgewertet werden. Um die Berechnungen der Kurven, die nach jeder Neuwahl der Granularität nötig werden, zu beschleunigen, wurde eine Hilfsklasse **ParameterVector** implementiert, die die zu jeder Granularität (**ModelMan** verwendet Werte aus dem Bereich von 1 bis 25) gehörenden invarianten Faktoren der Berechnungsformel für kubische Bézierkurven $B_i^3(t)$ in einem Vektor liefern kann. Ein solcher Vektor besitzt jeweils $4 \cdot (\mathbf{fidelity} + 1)$ Komponenten:

[9] Allgemeine Polygone mit kurvigen Rändern können prinzipiell auch aus einem planen NURB-Patch mit Hilfe von Trimmkurven 'geschnitten' werden.

$$\left(B_0^3(0)\ B_1^3(0)\ B_2^3(0)\ B_3^3(0)\ B_0^3(\Delta t)\ \ldots B_3^3(1)\right)$$

Mit der Funktion **fillXYCurveTo()** kann suksessive ein Punkt-Array aus Kurvensegmenten aufgebaut werden.

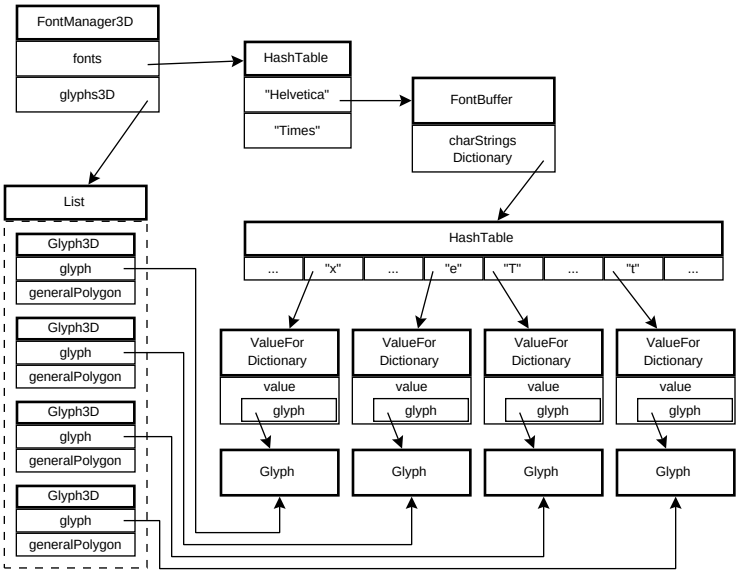

Abbildung 3.10: Verkettung der Font-Objekte

Die Verwaltung der **FontBuffer** und **Glyph3D** Instanzen obliegt einer Instanz der Klasse **FontManager3D**. Nichtverwendete Instanzen werden von diesem Verwaltungsobjekt gelöscht. Die Fonts können aufgrund ihres Fontnamens vom **FontManager3D** angefordert werden (**useFont:**, **unuseFont:**). **Glyph3D** Instanzen eines Fonts können aufgrund ihrers Namens oder Zeichenkodes gefunden werden (**getNamedGlyph3D:font:**, **getGlyph3D:font:**, **ungetGlyph3D:**). Es ist jeweils nur eine Instanz eines bestimmten Zeichens im Speicher. Diese Instanzen werden von **Text3D**-Shapes in eine Shape-Hierarchie eingegliedert. Das Schema 3.10 zeigt vereinfacht einen Ausschnitt einer Verkettung möglicher Instanzen.

Solid-Blöcke

Das RenderMan-Interface bietet die Möglichkeit, ‘Constructive Solid Geometry’ (*CSG*) Modelle mit Hilfe der Solid-Blöcken einzugliedern. CSG-Modelle können miteinander geschnitten oder vereinigt werden. Es kann ebenso die Differenz von Modellen gebildet werden. Wie in Abschnitt 2.6.2 schon ausgeführt ist die Schachtelung dieser Blöcke bestimmten Reglungen unterworfen. Die komplexen Blöcke **RI_INTERSECTION**, **RI_UNION** und **RI_DIFFERENCE** dürfen weitere komplexe Blöcke und ‘Primitive’-Blöcke (**RI_PRIMITIVE**) enthalten. Letztere sind immer die untersten Blöcke und enthalten die Grafikobjekte. Alle Grafikobjekte in einem solchen Block brauchen sich zwar nicht berühren, müssen aber geschlossen sein — sie müssen

ein wohldefiniertes Inneres und Äußeres besitzen. Die Methode **isClosed** von **InteractiveShape** liefert für diesen Fall (und wenn das Schließen explizit durch den Benutzer gesetzt wurde) logisch wahr. Weiterhin dürfen zwischen zwei Solid-Blöcken keine weiteren Oberflächen definiert werden.

Um die Solid-Blöcke in **ModelMan** integrieren zu können, mußte die **render:**-Methode von **N3DShape** in der Klasse **InteractiveShape** überdefiniert werden. Diese Methode steuert die Aufrufreihenfolge der **renderSelf:** Methoden der Shapes und führt zusätzlich eigene Interface-Aufrufe aus (siehe Abb. 2.7). Die überdefinierte Methode setzt die entsprechenden **RiSolidBegin()** und **RiSolidEnd()** Interface-Aufrufe ein. Der Typ des gesetzten Solid-Blocks wird beim Rendern in der Instanzvariablen **solid** bereitgestellt (**RI_NULL** steht für 'kein Solid-Block').

In der 3DKit Shape-Hierarchie sind die komplexen Solid-Blöcke an Gruppen-Knoten zu binden, da ihnen weitere Blöcke folgen können. 'Primitive'-Block zu sein hingegen, ist die Eigenschaft eines Blattknotens. Ist einer der Vorgänger ('Ancestor') eines Blattknotens ein Solid-Block, wird in **render:** deshalb dafür Sorge getragen, daß das Shape in diesem Knoten standardmäßig in einem 'Primitive'-Block ausgegeben wird. Sind in der **renderSelf:**-Methode eines Shapes weitere Solid-Blöcke vorhanden, darf der Typ des übergeordneten Blocks nicht **RI_PRIMITIVE** sein. Der benötigte Typ soll in diesem Fall von dem Shape durch eine überdefinierte Methode **getPrimitiveToken** geliefert werden.

RIB-Makros

Das Einlesen von RIB-Makros ist in **ModelMan** in der Klasse **RibArchive** implementiert. Die Applikationsklasse **ModelMan** verwaltet eine Liste von Instanzen dieser Klasse. Die Darstellung von RIB-Makros erfolgt in der Methode **renderSelf:** von **RIBEntityShape**. Ein Benutzer kann RIB-Makros entweder als RIB-Datei laden oder die vorgegebenen RIB-Makros aus der Palette verwenden.

Die Klasse **RibArchive** ist der Kern der RIB-Verwaltung. Das Archiv wird mit dem Pfadnamen der RIB-Datei initialisiert. Die RIB-Datei wird erst bei Bedarf geladen, normale RIB-Dateien werden in RIB-Entities (das Innere des Weltblocks) umgewandelt. Ein Zugriffszähler wird dazu benötigt festzustellen, ob das Makro noch von einem **RibEntityShape** verwendet wird. Wird der Zähler in der Methode **decAccessCount** 0, wird die Applikation mit **removeRIBArchive:** benachrichtigt, das Archiv zu löschen. RIB-Archive, auf die der Benutzer von der Palette aus zugreifen kann ('Galerie-Archive'), werden nie gelöscht. **RibArchive** besitzt neben den Methoden zum Laden von RIB-Entities die Methoden **defineMacro:** zum Definieren und **instanciateMacro:** zum Instanzieren des Makros. Das Makro bleibt solange definiert (es werden keine weiteren Makro-Definitionen ausgegeben) bis die Methode **clearHandles** ausgeführt wird. Der aktuelle Parameter der Methoden **defineMacro:** und **instanciateMacro:** soll logisch wahr sein, wenn auf einen RIB-Stream ausgegeben wird. Das Objekt generiert in diesem Fall andere Befehle als beim Rendern mit dem *Quick RenderMan.* Statt den **RiMacroBegin()**, **RiMacroEnd()** und **RiMacroInstance()** werden die entsprechenden **RiObject...()** Befehle generiert. **RiObjectInstance()** wird mit **RiArchiveRecord()** 'gedruckt', damit das Makro nicht expandiert wird. Auch die Makro-Definition wird, statt sie mit **RiReadArchive()** dem Renderer zu übergeben, mit **RiArchiveRecord()** geschrieben. Da **RiArchiveRecord()** (mit dem Typ **RI_COMMENT**) zu Beginn immer ein Kommentarzeichen '#' ausgibt, wird als erstes Zeichen immer ein Zeilenende übergeben. Es muß darauf geachtet werden, daß der *Quick RenderMan*

vor dem Erzeugen der Makro-Handles initialisiert wird. In der Methode **appDidInit:** wird aus diesem Grund `[N3DContextManager new]` ausgeführt.

Das Shape **RIBEntityShape** benötigt einen Zeiger auf das RIB-Archiv, dessen Makro es darstellen soll. Das Shape schickt dem Archiv bei einer Verbindung eine **incAccessCount** Nachricht und bei einem Lösen, bzw. Löschen des Shapes **decAccessCount**, damit nicht mehr verwendete Archive gelöscht werden können. Um das Archiv zu einem RIB-Namen (Pfadname einer Datei) anzufordern, wird der Applikation eine **getRIBArchive:** Nachricht gesendet. Wird das Shape gespeichert, wird nur der Pfadname gesichert und nicht das ganze Makro. Weil die Makro-Handles nicht explizit gelöscht werden können, wird zu Beginn eines Programm-Blocks in **worldWillBeginInCamera:** das Makro definiert. Da die Handles in diesem Fall nach dem Programm-Block wieder ungültig werden, werden sie in **worldHasEndedInCamera:** wieder gelöscht. Die Instanzierung des Makros geschieht in der **renderSelf:** Methode.

Die Verwaltung der RIB-Archive obliegt dem Applikations-Objekt **ModelMan**. Es erzeugt bei Programmstart eine Liste von RIB-Archiven, deren Icons in der Palette gezeigt werden. Das Flag **isGalleryArchive** ist bei diesen Objekten logisch wahr, damit sie in die Palette aufgenommen werden können, der Zugriffszähler wird von der Applikation erhöht, damit das Archiv nie gelöscht wird. Die Namen der Galerie-Archive sind durch eine Hash-Tabelle **ribEntities** mit der Archiv-Instanz verbunden. Zieht ein Benutzer ein Archiv-Icon in ein Kamerafenster, kann die Applikation in der Methode **getShapeInstanceOfClass:sender:** dem Sender der Nachricht (der Kamera des Fensters) eine entsprechende Shape-Instanz aus der Galerie liefern. Der erste Parameter beinhaltet die Bezeichnung des Archivs: */pfadname*[10]. Zum Suchen in der Hashtabelle wird nur der Basisname verwendet. Die Methode '**getRIBArchive:***(const char *)aPathname*' dient dazu, ein Archiv, das noch nicht vorhanden ist zu erzeugen und mit dem Pfadnamen zu initialisieren. Sämtliche RIB-Archive werden in der Liste **ribEntityList** aufbewahrt. Mit **removeRIBArchive:** werden die Archive, deren Zugriffszähler 0 geworden ist, gelöscht.

3.1.4 Erweiterungsmöglichkeiten

Neben der Erweiterung um neue Bundles können Funktionen eingeführt werden, die in andere Dateiformate schreiben und fremde Dateien importieren können. Ein Umwandeln von RIB-Dateien in eine 3DKit Objekthierarchie ist wegen der abgekapselten **renderSelf:** Methode der **N3DShapes** wohl nur mit einer tiefergreifenden Umstrukturierung des Rendervorgangs möglich. Komplexere Teil-Modelle können durch das Einfügen von Pseudokommentaren (wie BeginObject(Klassenname)/EndObject) sicherlich wieder in Instanzen der entsprechenden Klassen zurückgeführt werden.

Es können noch mehr Funktionen des RenderMan Interfaces (z.B. Level Of Detail, Depth Of Field, Polygone, Texturkoordinaten) übernommen werden. Allgemeine Hilfsfunktionen, z.B. das Erstellen von Rotationskörpern und das 'Extrude' von Patches und Polygonen, können eingeführt werden. Es fehlt noch die Möglichkeit, mehrere, aus **InteractiveShapes** zusammengesetzte Gruppen-Objekte, als Einheit in Bundles definieren zu können. Die Behandlung von Shape-eigenen Kontrollpunkten kann erweitert werden. Lichtquellen als Objekte und Polygoneditoren können noch implementiert werden. Es kann die Frage geklärt werden, wie

[10] Zum Anfordern vom Instanzen zu den Klassen aus den Shape-Bundles wird nur der Klassenname des Shapes (ohne führenden `"/"` als String verwendet

schon vorhandene Bundles in neuen Bundles wiederverwendet oder wie Bundles Hilfstexte und Menüpunkte im Hauptmenü zugeordnet werden können — einem Bundle ‘Polygon’ könnte zum Beispiel der sinnvolle Menüpunkt ‘Lade Polygondaten’ zugeführt werden. Die Verwendung der ‘Render-Delegates’, Shape-Links, die den Links in einem Dateisystem entsprechen, steht noch aus. Eine Unterstützung bei der Erstellung und Verwendung von den ‘Texture Maps’ (inkl. ‘Environment Maps’ und ‘Shadow Maps’) wäre eine weitere sinnvolle Programmerweiterung. Die Hilfstexte, das Drag&Drop-Prinzip und die Objektmanipulation im Dokumentenfenster können noch ausgebaut werden. Zur besseren Orientierung sollten Koordinatenkreuze in das Dokumentenfenster eingeblendet werden können. Eine Undo-Funktion würde die Benutzbarkeit des Programms verbessern. Die Bedienelemente können teilweise anders gestaltet werden (z.B. kann die **PopUpList** mit den Shader-Namen durch ein scrollbare Liste ersetzt werden). Beim Verschieben von Shapes mit der Maus aus dem Fenster heraus sollte die Kamera nachgeführt werden. Unterschiedliche Shader (z.B. Oberfläche und ‘Displacement’) können mitsamt Farbe zu einer Gruppe zusammengefaßt und in einer Palette zur Verfügung gestellt werden. Die Shape-Klasse kann so erweitert werden, daß mehrere lokale Lichtquellen-Shader pro Shape zugelassen sind. Eine Dokumentation der Typen, Funktionen und Klassen im Stil der NeXT Dokumentation sind sinnvoll, wenn die Applikation von anderen Autoren weiterentwickelt werden soll oder einzelne Klassen wiederverwendet werden sollen.

Da ein Modell Ressourcen aus mehreren Dateien verwendet (RIB-Makros, Shader, Texturen, eingebundene Shape-Dateien) könnten Modelle nicht als einzelne Datei sondern als Verzeichnis mit einer Hauptdatei (der eigentlichen Szene) und den verwendeten Ressourcen (zumindest den RIB-Makros) oder Datei-Links zu diesen (ähnlich den Rich Text Format `.rtf`-Dateien und `rtfd`-Verzeichnissen des NeXTs).

Zu einem Animationswerkzeug kann **ModelMan** nur mit sehr viel Aufwand ausgebaut werden. Die Komplexität eines solches Werkzeuges übertrifft die eines einfachen Modellers bei weitem. Für eine Verwirklichung wäre es notwendig, Erzeugung und Entfernung, Bewegungen, Zustandsänderungen (z.B. Kameraprojektion und Einschaltung der Beleuchtung), Verformungen von einzelnen Objekten und Objektgruppen und das Aufteilen einer Sequenz in Frames über ein Skript zu steuern. Automatische Interpolierungen zwischen verschiedenen Objektzuständen und -positionen (z.B. Bestimmung einer Objektposition aus linearer oder kubischer Interpolierung von vorgegebenen Bewegungskurven, Vorgeben von Beschleunigungen, Einbeziehung von Motion Blur und des Detaillierungsgrades) wären für die Brauchbarkeit eines solchen Werkzeuges unerläßlich. Wahrscheinlich reicht es nicht aus, ein einfaches Skript als Matrix zu implementieren: Eine Spalte pro Objekt (auch die Kamera gilt als Objekt, Objektgruppen sollten als Spalte fungieren können, nicht alle Objekte müssen in jeder Tabelle auftauchen), eine Zeile pro Zeitschritt und die, das Objekt betreffenden, Aktionen in den Zellen. Objekte müßten in Abhängigkeit des Zeitschritts generiert oder entfernt werden können. Ähnlich einer Programmiersprache müßten Schleifen für wiederholte Handlungen, sowie bedingte Aktionen für kleine Abweichungen in den wiederholten Anweisungen und Sprünge in der Tabelle möglich sein. Eine Art Unterprogramm für immer wiederkehrende Aktionsmuster kann eventuell durch eine Verzweigung in eine neue Tabelle realisiert werden. Die Aktionen müssen dann aber über die Zeitschritte in den verschiedenen Tabellen synchronisiert werden können. Die möglichen Aktionen und deren Ausführung sind von dem jeweiligen Objekttyp abhängig und können, ähnlich wie die Aktionen im Shape-Inspektor, vom Elternobjekt geerbt werden.

3.2 Benutzung des Modellierwerkzeugs ModelMan

In diesem Abschnitt wird vorausgesetzt, daß die NeXT Benutzungsoberfläche, die Grundbegriffe des RenderMan Interfaces bekannt und UNIX-Grundkenntnisse vorhanden sind.

3.2.1 Beispielsitzung

Das Programm 'ModelMan.app' kann durch einen Doppelklick auf sein Icon gestartet werden. Nach einer Initialisierungsphase wird das Hauptmenü sichtbar. Die Selektion 'File/New' in diesem Menü öffnet ein leeres Kamerafenster. 'Tools/Palette' macht das Paletten-Panel sichtbar. Mit der Maus kann aus der unteren 'ScrollView' der Palette ein Icon eines Shapes, z.B. der Teekanne, genommen und in das Dokumentenfenster gezogen werden. Nachdem der Mausknopf losgelassen wurde, sollte eine Teekanne im Fenster sichtbar werden. Mit dem Menüpunkt 'Preview…' kann nun das photorealistische Rendern gestartet werden. In dem erscheinenden Renderpanel des Systems können die 'Hosts', auf denen gerendert werden soll, selektiert werden (Mehrfachselektion durch Shift-Mausklick). Nach dem Fertigstellen der Grafik — **ModelMan** sollte in dieser Zeit nicht verlassen werden — wird die temporäre TIFF-Datei in der Standardapplikation zur Anzeige von TIFF-Bildern geöffnet. **ModelMan** kann anschließend über den Menüpunkt 'Quit' verlassen werden. Während der Verwendung von **ModelMan** kann, wie bei NeXTSTEP-Applikationen üblich, mittels des 'Help'-Menüpunktes im 'Info'-UnterMenü ein hypertextartiges Hilfesystem konsultiert werden.

3.2.2 Mitwirkende Dateien und Verzeichnisse

Verzeichnisse

ModelMan verwendet eine Anzahl von Verzeichnissen und Dateien. Das Programm sucht die übersetzten Shader ('`.slo`'-Dateien) in den Verzeichnissen `~/Library/Shaders` des Benutzers und in `/NextLibrary/Shaders`. Icons für die Shader werden in `~/Library/Shaders/ShaderIcons` und `/NextLibrary/Shaders/ShaderIcons` gesucht. Abgespeicherte Parameter sollten in `~/Library/Shaders/ShaderParameters` zu finden sein. Einige vorgefertigte Texturen ('`.tx`') können schon in `/NextLibrary/Textures` gefunden werden. Eigene Texturen können in `~/Library/Textures` abgelegt werden. RIB-Makros (RIB-Entity Dateien) der Palette werden in `~/Library/ModelMan/ClipArt`, die zugehörigen '`.tiff`' Icons in `~/Library/ModelMan/ClipArtIcons` gesucht. Die `~/Library` Unterverzeichnisse werden von **ModelMan**, falls noch nicht vorhanden, angelegt.

Dateitypen

ModelMan arbeitet mit den folgenden Dateitypen:

.mms ModelManScene, von **ModelMan** als typisierter 'Stream' gespeicherte Komplettszene mit allen Kindkameras. Die Datei beinhaltet außer der Shape-Hierarchie und den Kameraeinstellungen auch die Fensterpositionen.

.mmo ModelManObjects, als 'Stream' gespeicherte Shapes.

.rib RIB-Entities oder komplette RIB-Dateien.

.slp Shader-Parameterdatei von **ModelMan**. In dieser Datei steht in lesbarer Form eine Belegung der formalen Parameter eines Shaders.

.slo *Shading Language Object*, die Shader.

.sl *Shading Language Source*, wird von **ModelMan** nicht direkt verwendet, diese Dateien können mit Hilfe des Shader-Compilers **shade** in `.slo`-Dateien übersetzt werden.

.tx Texturen in einem Pixar-eigenen Format — Texturen können mit dem **MakeTexture** Interface-Funktion aus 24 Bit tiefen TIFF Dateien erzeugt werden.

.tiff Tiff Dateien für Icons oder vom `prman` gerendertes Bild.

.ps PostScript Dateien, sie können vom `prman` erzeugt werden.

.bundle Unterprojekte (Shape-Typen) stehen im Hauptbundle (`ModelMan.app`).

Ein Doppelklick auf ein `.mms`, `.mmo` oder `.rib` Datei-Icon in einem **File Viewer** Fenster, öffnet die zugehörende Datei in einem **ModelMan**-Fenster.

3.2.3 Die Dokumentenfenster

Das Dokumentenfenster entspricht der Sicht einer Kamera. Ein Dokumentenfenster kann Kindfenster besitzen, in denen die Objekthierarchie von einer anderen Kamera gezeigt werden kann. Wird das Hauptfenster geschlossen, werden automatisch auch alle Kindfenster geschlossen. Die Fenster besitzen neben den üblichen Bedienelementen zum Schließen und Miniaturisieren am Rand Daumenräder zur Positionierung und Rotierung der Kamera oder der selektierten Objekte. In der linken unteren Ecke befindet sich ein Umschalter zwischen dem Rotieren und dem Verschieben von Objekten. Der aktuelle Modus wird zusätzlich durch den Zeichenmodus der Daumenräder angezeigt: Radial beim Rotieren, flach beim Verschieben. Rechts unten im Fensterrand befindet sich ein Umschalter zwischen Kamera und selektierten Objekten. Mit Hilfe der Maus können die Grafikobjekte (Shapes) auch direkt im Fenster selektiert, verschoben und im Weltkoordinatensystem skaliert werden.

3.2.4 Die Palette

In der Palette sind Icons aller Shape-Typen und einer Anzahl von RIB-Makros zu finden, die von **ModelMan** verwendet werden können. Die Palette wird zum Programmstart aufgebaut und kann mit Hilfe des Menüpunktes ‘Tools/Palette...’ sichtbar gemacht werden. Sie besteht aus zwei Teilen, einer oberen ‘ScrollView’ mit den Icons der RIB-Makros aus dem `~/Library/ModelMan/ClipArt` Verzeichnis und einer unteren mit denen der Shapes. Die Größenaufteilung kann durch eine ‘SplitView’ verändert werden. Die Erzeugung eines Objekts in einem Fenster geschieht mit Hilfe von ‘Drag&Drop’. Das Icon eines Objekts wird mit der Maus in ein Dokumentenfenster verschoben und an der gewünschten Stelle ‘fallengelassen’.

3.2.5 Der Kamera-Inspektor

Der durch den Menüpunkt ‘Tools/Camera Inspector. . . ’ erreichbare Kamera-Inspektor dient zur Anzeige und Manipulation der Kameradaten. Sie entsprechen größtenteils den Optionen des RenderMan Interfaces. Der Inspektor teilt sich in vier Seiten auf: ‘Surface and Projection’, ‘Position’, ‘Screen Options’ und ‘Rendering’. Mit den immer sichtbaren Knöpfen ‘Put’ und ‘Get’ kann eine komplette Einstellung zwischengespeichert und zurückgeholt werden. Mit dem Knopf ‘3D Frame’ kann eine dreidimensionale Umrandung selektierter Shapes eingeschaltet werden. Die Seiten sind nur dann sichtbar, wenn ein Dokumentenfenster existiert und nicht miniaturisiert ist. Sie können mit Hilfe einer ‘PopUpList’ selektiert werden.

Surface and Projection

Auf dieser Seite läßt sich der Typ der Oberfläche: Glatte Interpolation, flache Flächenfüllung, schattiertes Drahtgitter[11], normales Drahtgitter und Punktwolke einstellen. Die Geschwindigkeit des Renderns wächst, je weniger komplex die Flächendarstellung ist: Die glatte Interpolation ist am langsamsten, die Darstellung als Punktwolke am schnellsten. Die gewählte Darstellung beeinflußt den gewählten Hider (Algorithmus für verdeckte Oberflächen). Auf Wunsch kann dieser auch von Hand gewählt werden. ‘Hidden’ kann bei Flächendarstellungen verwendet werden. Er ist langsamer als ‘InOrder’ rendert die sichtbaren Flächen aber korrekt. ‘InOrder’ rendert die Flächen in der Reihenfolge ihrer Erzeugung. Er kann sinnvoll bei der Gitternetz- und Punktwolken-Darstellung verwendet werden. Mit ‘No Rendering’ kann, wenn nötig, das Rendern ganz abgeschaltet werden.

Die Gruppe ‘Fidelity’ dient zur Einstellung der Genauigkeit der Oberflächeninterpolation. Je niedriger der eingestellte Wert ist, desto schneller ist das Rendern, mit dem Preis einer kantigeren Darstellung. Es können Werte zwischen 1 und 25 eingegeben werden.

Die Hintergrundfarbe der Kamera kann mit den Bedienelementen der Gruppe ‘Background’ bestimmt werden. Mit Hilfe des Schalters ‘Fill in’ kann das Neuzeichnen des Hintergrunds bei der Neudarstellung abgeschaltet werden.

Die bisher beschriebenen Einstellungen haben nur Einfluß auf die Darstellung im Fenster. Die photorealistische Ausgabe wird nicht beeinflußt. Die Einstellungen in der Gruppe ‘Projection’ hingegen wirken auch für den `prman`. Mit den Bedienelementen kann die Art der Projektion: Perspektivisch oder parallel und der Grad des Öffnungswinkels (Field of View) der perspektivischen Kamera bestimmt werden.

Position

Auf dieser Seite kann die Position und Rotierung der Kamera eingestellt werden. Knöpfe mit einstellbarer Schrittweite dienen der schnellen Positionierung der Kamera. Durch den Umschalter in der Mitte des ‘Shifter’ Knopffeldes kann zwischen Rotation und Positionierung gewählt werden. Der Richtungsvektor und die Position der Kamera kann auch direkt mit Hilfe des ‘Eye Points’ (Augenpunkt, Kameraposition) und des ‘View Points’ (Punkt auf den geschaut wird) eingegeben werden. Mit dem ‘Pivot Point’ kann der Punkt, um den mit dem Shifter gedreht wird, bestimmt werden. Bei einem Druck auf den Knopf ‘From View Point’ werden die Einstellungen aus ‘View Point’ übernommen. Die Drehung der Kamera um ihre Z-Achse (Azimuth) kann auch

[11] Das Drahtgitter wird von den Atmosphären-Shadern ‘fog’ und ‘depthcue’ schattiert

skalar eingegeben werden.

Screen Options

Auf der Seite ‘Screen Options’ sind einige das Fenster betreffende Einstellungen zusammengefaßt. Es kann mit einem Daumenrad oder skalar die Entfernung der vorderen und hinteren Clip-Ebene vom Augenpunkt bestimmt werden.

Mit den Umschalter ‘use’ kann festgelegt werden, ob das x-y Verhältnis der Darstellung (‘Aspect Ratio’) oder die genormten 2D-Koordinaten des Fensters (‘Projection Rectangle’) explizit explizit angegeben werden. Mit einem Verkleinern und Vergrößern der Fensterkoordinaten durch einen logarithmischen Slider kann der Bildausschnitt an eine Szenengröße angepaßt werden. ‘New’ übernimmt die aktuelle Einstellung als Standard damit auch Werte, die über die 0.01, 100.0 Grenzen hinausgehen einstellbar sind. ‘Reset’ setzt auf die Originaleinstellung zurück. In einem Knopffeld kann eingestellt werden, wie sich eine Fenster-Größenänderung auf die Kameraprojektion der Shapes auswirkt: Mit ‘Rubber’ wird das Innere des Fensters wie eine Gummimatte verzogen, mit ‘Variable shape size’ wird der Fensterausschnitt beibehalten und die Shapes entsprechend der Größe angepaßt und mit ‘Fixed shape size’ wird der Fensterausschnit entsprechend der Größenänderung angepaßt, die Shapes behalten ihre Größe und der Fenstermittelpunkt seine Position bei.

Rendering

Seite zur Steuerung des photorealistischen Renderns. In einem Browser ‘Files to Render’ werden die Namen der Fenster aufgeführt, deren Inhalt momentan photorealistisch gerendert wird. Bevor photorealistisch gerendert werden kann, müssen in dem Renderpanel des Systems die Hosts ausgewählt werden, auf denen gerendert werden soll. Es sollten möglichst nicht ‘Localhost’ und der Name der eigenen Host gleichzeitig angewählt werden. Die Knöpfe ‘Preview…’, ‘Render as TIFF…’, ‘Render as EPS…’ und ‘Print…’ dienen zum Anstoßen des photorealistischen Renderers. Der Dateiname der TIFF oder PostScript Datei muß für die Funktionen ‘Render as TIFF…’ und ‘Render as EPS…’ eingegeben werden. Diese Funktionen werden auch von dem Zustand des Umschaltknopfs ‘Open in previewer’ im Render-Panel beeinflußt. Ist er zum Zeitpunkt des Aktivierens einer der beiden Funktionen gesetzt, wird die Ergebnisgrafik in einem Fensters des Standardpreviewers des Systems dargestellt. ‘Preview…’ rendert auf eine temporäre TIFF-Datei. Das Ergebnis wird immer in einem Previewer dargestellt. Die Verwendung der ‘Preview’ Funktion macht Sinn, wenn keine Ergebnisdatei gebraucht wird. Mit dem Knopf ‘Save to RIB…’ kann die aktuelle Szenenbeschreibung als RIB-Datei geschrieben werden. Diese Dateien können nachträglich durch den `prman` gerendert werden. Der Knopf ‘Render Manager…’ dient zum Aufrufen des Programms **Render Manager** des NeXTs. Mit Hilfe dieses Programms kann sich der Benutzer ansehen, welche Render-Jobs auf den verschiedenen Hosts aktiv sind.

3.2.6 Der Shape-Inspektor

Der Shape-Inspektor, ein zweiter Inspektor **ModelMans** kann mit dem Menüpunkt ‘**Tools/Shape Inspector…**’ sichtbar gemacht werden. Er zeigt die Attribute eines im Dokumentenfenster selektierten Shapes an. Die für ein Welt- oder Gruppenobjekt gemachten Angaben gelten für hierarchisch untergeordnete Shapes als Standardeinstellung.

Im Panel wird oben das Icon, der Typname und der änderbare Instanzname des Objekts angezeigt. Darunter befindet sich die ‘PopUpList’ zur Seitenauswahl. Im Gegensatz zum Kamera-Inspektor kann der Shape-Inspektor je nach selektiertem Objekt unterschiedliche Seiten besitzen. Standardmäßig sind nur die Seiten ‘Surface and Shaders’, ‘Position, Rotation, Size and Scale’ und ‘Skew’ vorhanden. Die meisten Objekte besitzen zusätzlich eine Seite ‘Attributs’ mit typabhängigen Attributeinstellungen, das Text-Objekt besitzt zusätzlich die Seite ‘Font’ für die Zeichensatz-Einstellungen. Bei der Selektierung von Objekten wird nach Möglichkeit versucht, die aktuelle Seite beizubehalten.

Surface and Shaders

Auf dieser Seite können die Oberflächeneigenschaften eines Objekts eingestellt werden. In der Gruppe ‘Visibility’ kann eingestellt werden, ob das Shape sichtbar ist und ob es aus Geschwindigkeitsgründen nur als Rechteck gerendert werden soll, ‘close’ dient zum Abschließen der Drehkörper. In ‘Color and transparency’ kann, nachdem der Knopf ‘use’ aktiviert wurde, mit den auf dem NeXT üblichen ‘Colors’-Panel Oberflächengrundfarbe und Transparenz bestimmt werden. Ist der Knopf nicht aktiv, werden die Werte des hierarchischen Vorgängers übernommen. Das ‘Color Well’ ist in diesem Fall nicht selektierbar und zeigt den aktuellen Farbwert des Vorgängers. Unten auf der Seite ist eine Liste mit Knöpfen für die unterschiedlichen Shader-Typen vorhanden. Der Aufdruck der Knöpfe zeigt den Namen des eingesetzten Shaders oder ‘**empty…**’, falls kein Shader eingesetzt ist. Die Knöpfe sind nicht selektierbar wenn Shader eines bestimmten Typs nicht vorhanden sind. Auf dem NeXT können keine ‘Transformation’ und ‘Imager’ Shader eingesetzt werden. Die Volumen-Shader sind auf die Atmosphären-Shader ‘fog’ und ‘depthcue’ beschränkt. Durch einen Druck auf einen der Knöpfe wird im sichtbar gemachten ‘Shader Parameters’ Panel eine entsprechende Anzeige gemacht.

Position, Rotation, Size and Scale

Links werden die numerischen Werte für Position, (skalierte) Größe der Rechteckhülle im Welt- und Rotierung im Objektkoordinatensystem gezeigt. Rechts können die Werte mit Hilfe eines ‘Shifter’ Knopf-Feldes und eines auf die Achsen anwendbaren Daumenrads eingestellt werden. Die Achsen, auf die die Einstellungen des Daumenrads wirken, können über ‘Radioknöpfe’ selektiert werden. Die Art der Modifikation ist über eine ‘PopUpList’ einstellbar. Die Schrittweite der Knöpfe kann eingestellt werden. Möchte man ein Objekt gleichzeitig in alle drei Achsenrichtungen vergrößern, wählt man in der ‘PopUpList’ ‘Scale’, selektiert mit den ‘Radioknöpfe’ alle Achsen und stellt die gewünschte Größe mit dem Daumenrad oder den beiden Knöpfe links und rechts davon ein. Bei den Transformationen (bis auf die Positionierung) bleibt der Ursprung des Shapes erhalten.

Skew

Mit den Kontrollen dieser Seite kann das Grafikobjekt in seinem Koordinatensystem in verschiedene Richtungen gedehnt werden.

Die optionalen Seiten

Die Attributseite ‘Attributes’ ist optional. Ihr Inhalt variiert je nach selektiertem Objekttyp. Bei den derzeit vorhandenen Bundles sind folgende Einstellungen möglich:

Cube: Sichtbare Seiten, Höhe, Tiefe, Breite

Hyperboloid: Das Hyperboloid wird als Drehkörper einer Linie um die Y-Achse des Objektkoordinatensystems erzeugt. Es läßt sich getrennt für die beiden Enden des Hyperboloids die Höhe, der Radius des Drehkreises und die Winkelposition des Linienendpunktes bestimmen. Der Drehwinkel 'thetamax' zur Vervollständigung des Mantels kann bestimmt werden.

Disk: Höhenverschiebung (senkrecht zur Fläche), Radius und Drehwinkel

Patch: Die Extrusion

Paraboloid: Maximaler Radius, oberes und unteres Ende, Drehwinkel

Cylinder: Radius, obere und untere Kante und Drehwinkel

Cone: Maximaler Radius, Höhe und Drehwinkel

Sphere: Radius der Kugel, obere und untere Kappe (kann auf Radius gesetzt werden, um immer eine vollständige Kugel zu erhalten), Drehwinkel

Text3D: Auf der Attributseite kann der Text eingegeben werden, auf einer zusätzlichen Seite 'Font' kann der Zeichensatz, seine Ausgangsgröße, die Extrusion und die Font-Transformationsmatrix geändert werden. In der Texteingabe darf ein oktaler Zeichenkode oder ein PostScript-Zeichenname hinter einem Gegenschrägstrich (\) folgen.

Torus: Äußerer und innerer Radius, Schließwinkel ('phimin' und 'phimax') des Ringes, Drehwinkel der Mantelerzeugung

3.2.7 Das Shader-Parameters Panel

Das Shader-Parameters Panel ist vom Shape-Inspektor erreichbar (Shader-Knopf). Die erste Gruppe enthält eine 'PopUpList' zur Auswahl des Shadertyps. Für jeden Shadertyp existieren unterschiedliche Shader. Darunter befinden sich die Knöpfe 'Put' und 'Get' zum Zwischenspeichern und Zurückholen einer Parametereinstellung. Der Knopf 'Revert' holt die Shader-Einstellung des aktuell selektierten Shapes zurück. Eine Betätigung des Knopfs 'Use' überträgt die aktuelle Einstellung an das selektierte Shape. 'Use' und 'Revert' sind nur anwählbar, wenn ein Shape selektiert ist. In diesem Fall steht rechts von den Knöpfen Typ und Name des Shapes. In der zweiten, darunterliegenden Gruppe ist der Bezeichner eines Shaders aus einer 'PopUpList' auswählbar. Darunter befindet sich der, wenn nötig um den Pfadnamen erweiterte Bezeichner des Shaders. Rechts ist eine Iconbox mit dem Icon des Shaders zu sehen. In die Box kann das Icon eines '`.slo`' Shaders oder einer '`.slp`' Shader-Parameterdatei gezogen werden. Die entsprechende Datei wird dann geladen und falls noch nicht vorhanden, dem 'Shaderpool' lokal in `~/Library/Shaders` zugefügt; das Panel wird entsprechend umgeschaltet. Neue Shader können auch durch einen Druck auf den Knopf 'Add...' kopiert und geladen werden. Die Knöpfe in der Zeile 'Parameters' werden für das Speichern und Laden von '`.slp`' Parameterdateien gebraucht. 'Load...' zum Laden, 'Save' zum Speichern der aktuellen Parameterdatei, 'Save as...' zum Anlegen einer neuen Parameterdatei und 'Default' zum Zurücksetzen auf die Standard-Parameterbelegung.

Ganz unten befindet sich eine ‘ScrollView’ mit Eingabefeldern für alle Parameter, die dem selektierten Shader übergeben werden können. Der Titel enthält, falls definiert, den Namen der aktuellen Parameterdatei. Für jeden Parametertyp: ‘Scalar’, ‘Text’, ‘Color’ und ‘Point’ existieren unterschiedliche Eingabezeilen. Links in der Eingabezeile steht der Bezeichner des jeweiligen Parameters gefolgt von seinem Typ und den entsprechenden Eingabefeldern. Da Texte größtenteils zur Angabe von ‘`.tx`’ Texturdateien verwendet werden, kann ein ‘Open Panel’ in den Texteingabezeilen durch einen Knopf erreicht werden.

3.2.8 Der Hierarchy Viewer

Der ‘Hierarchy Viewer’ wird für die Darstellung der Shape-Hierarchie innerhalb eines Browsers verwendet. Mit dem Browser können Shapes selektiert werden. Mit Hilfe von Knöpfen können selektierte Shapes gruppiert, degruppiert, umgruppiert und gelöscht werden. Das Panel ist noch nicht fertig implementiert. Wie in dem NeXT **File Viewer** soll in einer ‘ScrollView’ der aktuelle Selektionspfad dargestellt, ein ‘Shelf’ zur Ablage von Shapes implementiert und der Name der Shapes eingebbar gemacht werden.

3.2.9 Das Menü

In den folgenden Zeilen werden die Funktionen des Hauptmenüs beschrieben. Die von den Namen abgetrennten Buchstaben bezeichenen die Command-Hotkeys.

Info

Information, Voreinstellung und Hilfe.

Info/Info Panel...
Darstellen des Informationspanels mit dem Programmnamen und Copyrightvermerken.

Help...
Die Programmhilfe wird gestartet.

File

Menüpunkte, die die Dateien betreffen.

File/Open... — o
Öffnen einer ‘`.mms`’ Datei und Darstellung des Inhalts in neuen Dokumentenfenstern.

File/New Scene — n
Öffnen eines neuen, leeren Dokumentenfensters.

File/Include — i
Nachladen einer ‘`.mmo`’ oder ‘`.rib`’ Datei (gespeicherte Shapes) in das aktuelle Fenster.

File/Save — s
Speichern des Hauptfensters und seiner Kindfenster in der aktuellen ‘`.mms`’ Datei. Bei Bedarf kann eine neue ‘`.mms`’ Datei angelegt werden.

File/Save As... — S
Speichern des Hauptfensters und seiner Kindfenster in einer neuen '`.mms`' Datei.

File/Save To RIB... — R
Speichert die aktuelle Szene in einer RIB-Datei. Es wird die entsprechende Funktion des Kamera-Inspektors verwendet.

File/Save All
Speichert alle offenen, noch nicht gespeicherten Fenster. Sind einige Dokumente keiner Datei zugeordnet, erscheint für jedes dieser Fenster nacheinander ein Save-Panel.

File/Render as TIFF... — T
Benutzt die Funktion des Kamera-Inspektors zum Rendern der aktuellen Szene als TIFF-Datei.

File/Render as EPS... — E
Benutzt die Funktion des Kamera-Inspektors zum Rendern der aktuellen Szene als EPS-Datei.

File/Miniaturize
Miniaturisiert das aktuelle Dokumentenfenster.

File/Close
Schließt das aktuelle Dokumentenfenster.

Child Camera

Menüpunkte für die Erzeugung von Kindfenstern eines Dokumentenfensters.

Child Camera/New Perspective
Erzeugt eine neue perspektivische Projektion des aktuellen Dokuments.

Child Camera/New Parallel
Erzeugt eine neue Parallelprojektion des aktuellen Dokuments.

Child Camera/New three Orthographic
Erzeugt drei orthographische Projektionen des aktuellen Dokuments. Eine in Richtung der negativen X-Achse (von rechts, die Y-Achse zeigt im Fenster nach oben), eine in Rictung der negativen Y-Achse (von oben, die X-Achse zeigt im Fenster nach unten) und eine in Richtung der positiven Z-Achse (von vorne, die X-Achse zeigt im Fenster nach rechts)

Edit

Menüpunkte zur Behandlung selektierter Shapes. Es werden RIB- und ASCII-Zwischenablagen unterstützt. Intern werden kopierte Shapes in 'Memorystreams' gehalten.

Edit/ Cut – x
Kopiert die selektierten Shapes in die Zwischenablage und löscht sie anschließend.

Edit/Copy — c
Kopiert die selektierten Shapes in die Zwischenablage.

Edit/Paste — v
Kopiert Shapes aus der Zwischenablage in das aktuelle Dokument. Funktioniert momentan nur mit **ModelMans** eigenen Objekten.

Edit/Delete — r
Löscht die selektierten Shapes.

Edit/Clear
Löscht alle Shapes eines Dokuments.

Edit/Select All
Selektiert alle Shapes der aktuellen Hierarchie-Ebene eines Dokuments.

Edit/Save Selection...
Speichert die selektierten Shapes in eine '`.mmo`' Datei.

Edit/Save Selection to RIB...
Speichert die selektierten Shapes als RIB-Datei.

Format

Menüpunkte für das Ändern des Ausgabeformats.

Format/Font Panel... — t
Öffnet ein Auswahlpanel für den Zeichensatz.

Format/Layout
Betrifft die Gruppierung der Shapes.

Format/Layout/Group — g
Gruppiert die selektierten Shapes.

Format/Layout/Unroup — G
Hebt eine Gruppierung von Shapes auf.

Format/Page Layout... — P
Einstellungspanel für das Seitenlayout beim Drucken.

Tools

Menüpunkte zum Zeigen der verfügbaren Panele.

Tools/Camera Inspector...
Zeigt den Kamera-Inspektor.

Tools/Shape Inspector...
Zeigt den Shape-Inspektor.

Tools/Hierarchy Viewer...
Zeigt den Hierarchy Viewer.

Tools/Palette...
Zeigt die Auswahlpalette mit den Shapes und RIB-Makros.

Tools/Colors...
Zeigt den Farbselektor des NeXTs.

Tools/RenderManager...
Startet das Programm **Render Manager** des NeXTs.

Windows

Standardmenü zur Fensterbehandlung.

Windows/Arrange in Front
Ordnet alle Dokumentenfenster im Vordergrund an.

Windows/(Window List)
Hier werden alle Namen der Dokumentenfenster aufgeführt. Die Fenster können durch Menüselektion aktiviert werden.

Windows/Miniaturize Window — m
Miniaturisiert das aktuelle Fenster oder Panel.

Windows/Close Window — w
Schließt das aktuelle Fenster oder Panel.

Preview...

Benutzt die Funktion des Kamera-Inspektors zum Rendern der aktuellen Szene als temporäre TIFF-Datei.

Services

Standardmenü des NeXTs.

Hide — h

Versteckt alle Fenster von **ModelMan**.

Quit — q

Beendet **ModelMan**.

4. Überlegungen zur Implementierung des Interfaces

4.1 Die Rendering-Pipeline

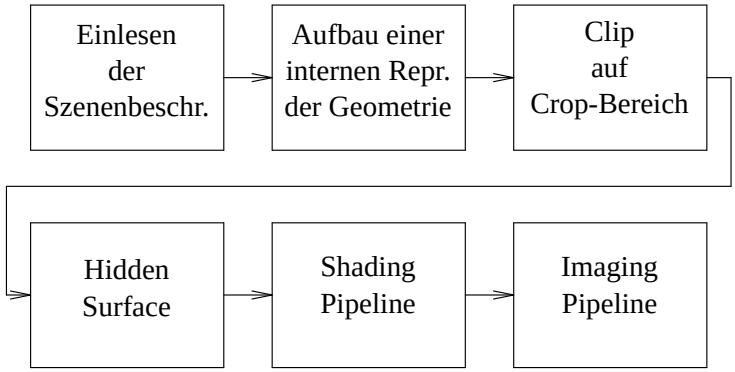

Abbildung 4.1: Mögliche Rendering-Pipeline

In Anlehnung an die Rendering-Pipelines aus [FolVDFH90], die den Ablauf des Renderns verdeutlichen, zeigt Abb. 4.1 einen möglichen Ablauf beim Rendern einer Szene. Am Anfang wird die Szenenbeschreibung über ein Language- oder RIB-Binding in den Renderer eingegeben. Der Renderer wird aus der Eingabe eine interne Repräsentation des Kameramodells und der Geometrie der Grafikobjekte unter Berücksichtigung der Optionen, Transformationen und Attribute aufbauen. Anschließend werden die Grafikobjekte auf den Crop-Bereich geclippt. Den durch den Hider bestimmten sichtbaren Punkten wird danach in der Shading-Pipeline (s. Abb. 2.3) eine Farbe zugeordnet. In der folgenden nachbearbeitenden Imaging-Pipeline (s. Abb. 2.6) wird aus der Punktmatrix das endgültige Rasterbild berechnet.

Es kann angenommen werden, daß der Aufbau der internen Repräsentation und das Clippen vollständig für eine Szene ausgeführt wurde, bevor mit dem Rendern begonnen werden kann. Natürlich kann es für einzelne Renderertypen und Hider Ausnahmen geben. Bei der Verwendung eines Drahtgitter-Renderers und des ‘InOrder’ Hiders, der die Objekte in der Reihenfolge ihrer Eingabe rendert, können die Grafikobjekte direkt nach ihrem Einlesen dargestellt werden. Durch

die Blockstruktur der Interface-Eingabe stehen zum Zeitpunkt des Auftretens eines Primitivs alle relevanten Attribute fest. Renderer, die keine globalen Beleuchtungsmodelle verwenden, brauchen die Objekte, die keine Lichtquellen sind und nicht innerhalb des Viewing-Frustums liegen, nicht betrachten. Bei der Verwendung von Raytracing- und Radiosity-Methoden werden alle Objekte eines Weltblocks in die Berechnung des Shadings eingehen.

4.2 Die Rendering-Gleichung

Mit Hilfe der Shading-Pipeline wird versucht, für die Rendering-Gleichung von Kajiya für die darzustellende Szene eine Lösung zu finden. Der verwendete Rendering-Algorithmus wird vom Interface nicht vorgeschrieben. Kajiyas Gleichung wird dazu auf die Shadertypen aufgeteilt (s. [HanL90]). Die Gleichung und ihre Terme werden in [Kajiya86] folgendermaßen definiert:

$$I(x,x') = g(x,x')\left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right]$$

$I(x,x')$: Intensität des Lichts, das von x' nach x gelangt. Der Intensität wird ein entsprechender Farbwert zugeordnet.

$g(x,x')$: 'Geometrie-Term', wird durch Körper und Volumen zwischen x und x' beeinflußt.

$\epsilon(x,x')$: Intensität des Lichts, das von x' direkt in Richtung von x emittiert wird.

$\rho(x,x',x'')$: 'Bidirektionale Reflexionsfunktion', relativiert die Intensität des Lichts, das von x'' über x' in Richtung von x gelenkt wird.

In dem Integral werden alle Punkte aller Oberflächen betrachtet. In der Praxis werden statt Punkten z.B. Mikrofacetten verwendet und es werden i.a. auch nicht alle Facetten betrachtet.

Mit der *Shading Language* und den Shader-Typen wurde versucht, eine von einem Renderalgorithmus unabhängige Modellierungsmöglichkeit zu schaffen. Die Berechnung der Terme wird auf die Shader-Typen verteilt:

Light Source Shader: Die Shader für die Lichtquellen berechnen den Anteil des Lichts, der direkt von einem Punkt einer Lichtquelle in eine bestimmte Richtung abgestrahlt wird: $\epsilon(x,x')$. Schatten können durch 'Shadow Maps' gerendert werden. Die 'Shadow Maps' werden in 'Lightsource Shadern' verwendet und aus Tiefen-Bitmaps, die von der Lichtquelle ausgehen gerendert wurden, erzeugt.

Surface Shader: Shader, durch den die Reflexionseigenschaften einer Oberfläche bestimmt werden. Er berechnet das Integral der bidirektionalen Reflexionsfunktion mit dem weitergeleiteten Licht: $\int_S \rho(x,x',x'')I(x',x'')dx''$. Reflektierende Oberflächen können durch 'Environment Maps' simuliert werden. Die Reflexion kann durch 'Bump Maps' verändert werden.

Atmosphere Shader: Volumen zwischen Kamera und Oberfläche

Interior Shader: Volumen, das von einer Oberfläche umschlossen wird

Exterior Shader: (Volumen zwischen Lichtquelle und Oberfläche und zwischen Oberflächen)

Die Volumen-Shader beeinflussen den Geometrie-Term $g(x, x')$. Es wird jeweils die Veränderung des Lichts, das ein Volumen durchstrahlt berechnet. Die Berechnung der Schnittpunkte von Strahlen und Oberflächen wird vom Renderer vorgenommen, Reflexion und Refraktion hingegen von den Oberflächen-Shadern.

Displacement Shader: Ein 'Displacement Shader' beeinflußt die Position der Oberflächenpunkte und damit die Reflexion.

Transformation Shader: Der 'Transformation Shader' beeinflußt die Abbildung einer Oberfläche in das Kamerakoordinatensystem oder transformiert das Objektkoordinatensystem.

Imager Shader: Dient der Nachbearbeitung der fertig berechneten Punkte und hat deshalb keinen Einfluß auf die Rendering-Gleichung.

Man kann sich fragen, warum das Interface eine Unterscheidung zwischen dem 'Atmosphere Shader' und dem 'Exterior Shader' macht. Beide behandeln dasselbe physikalische Medium. Beim Modellieren wird jedoch nicht die Dicke der Atmosphäre und die Entfernung der 'Distant Lightsources' (der Sonne) angegeben. Will man einen Nebeleffekt erreichen, wird die Mischung der Nebelfarbe mit dem abgeschwächten reflektierten Lichts nur durch die Entfernung der Oberflächen von der Kamera bestimmt. Die Intensität des eintreffenden Lichts von einer 'Distant Lightsource' hingegen ist konstant, da die Entfernung aller modellierten Oberflächen zur Sonne als gleich angenommen werden kann.

4.3 Raytracing und die Shading Language

Das Raytracing geht zurück auf Arbeiten von Arthur Apple (1968), die bereits die Behandlung von Schatten beinhalteten. Es wurde von Whitted (1980) und Kay (1979) um spekulare Reflexion und Transmission erweitert. Bei dem rekursiven Algorithmus von Turner wird ein Strahl vom Augenpunkt ausgehend durch einen Punkt in der Bildebene auf die darzustellende Szene geschickt. Der nahste Schnittpunkt mit einer Oberfläche wird bestimmt; an diesem Punkt wird der Strahl je nach Reflexions- und Brechungsseigenschaften der Oberfläche in einen reflektierten und einen transmittierten Strahl aufgeteilt. Die Strahlen werden rekursiv weiter verfolgt, bis eine vorgegebene maximale Rekursionstiefe erreicht wird, die Oberfläche nicht mehr reflektierend oder transparent ist oder der Strahl aus der Szene läuft. Die Oberflächenfarbe kann danach 'bottom up' bestimmt werden. Für jeden Schnittpunkt kann vor dem Hinabsteigen in die Rekursion die ambiente Farbe berechnet werden und, unter Berücksichtigung zwischenliegender Oberflächen, mit Hilfe der spekularen und diffusen Koeffizienten die Intensität des Lichts, das direkt von den Lichtquellen kommt und an der Oberfläche reflektiert wird, mit konventionellen Shading-Algorithmen und Schattenstrahlen bestimmt werden. Zu der so berechneten Farbe wird die reflektierte Farbe, die aus der Rekursion erhalten wurde, hinzugemischt. Da Schatten und reflektierende Oberflächen von dem Raytracing mit berechnet werden, kann es Sinn machen, die Fähigkeit des 'Shadow Mappings' und 'Environment Mappings' abzuschalten. Bei gemischten, photografischen und computergenerierten Szenen aber, können diese 'Mappings'

durchaus noch sinnvoll eingesetzt werden. Auch können durch die Verwendung verschiedener Oberflächen-Shader Raytracing und konventionelle Shader in einer Szene gemischt werden.

Eine Möglichkeit, das Raytracing in einem RenderMan Renderer zu implementieren ist die **trace()** Funktion der *Shading Language*. Diese Funktion liefert die Intensität des Lichts, das aus einer bestimmten Richtung kommt. Oberflächen-Shader können diese Funktion verwenden, um ein Raytracing in Richtung der reflektierten und transmittierten Strahlen durchzuführen. Die erhaltenen Werte müssen anschließend noch mit den Reflektions- und Transmissionseigenschaften der Oberfläche relativiert werden. Die diffusen und ambienten Anteile können zuvor in einem **illuminace()** Konstrukt berechnet werden. Das **illuminance()** Konstrukt führt bei einem Renderer mit Raytracing-Eigenschaften normalerweise auch die Berechnung der Schatten durch. Nachteil dieser Methode ist, daß bei der Verwendung des prozeduralen Shadings jeder Shader, der Raytracing implementieren will, die Aufrufe der **trace()** Funktionen beinhalten muß; die Schattierung wurde in der *Shading Language* nicht von der Farbgebung durch prozedurales Shading oder 'Texture Mapping' getrennt. Eine Möglichkeit, die Attribute Reflexions- und Refraktionseigenschaften einer Oberfläche innerhalb eines Oberflächen-Shaders dem Renderer zurückzugeben, der dann eigenständig die durch Raytracing erhaltenen Lichtintensitäten zur Farbe des Oberflächenpunkts hinzurechnen kann, würde die Raytracing-Eigenschaft des Renderers verbergen. Dieses Verfahren wäre dafür aber weniger flexibel als die Lösung mit der **trace()** Funktion.

In [PixSpec] ist der Quellcode eines einfachen Turner-Whitted Raytracers in der *Shading Language* abgedruckt (hier in leicht veränderter Form wiedergegeben):

```
surface
whitted(
    float Kt = .2;  /* Transmissionskoeffizient */
    float Ka = .1;  /* Ambienter Koeffizient */
    float Kd = .7;  /* Diffuser Koeffizient */
    float Ks = .2;  /* Spekularer Koeffizient */
    float Kss = 2;) /* Spekularer Exponent */
{
    point   Nn, /* Oberflaechen Normale aus Richtung I */
            H,  /* 'Halfway' Vektor */
            R,  /* Reflexions-Vektor */
            T;  /* Transmissions-Vektor */

    float eta, eta2;    /* Brechungsindizees */

    /* Brechzahl aus den Volumenshadern holen */
    if ( incident("eta", eta) && opposite("eta", eta2) ) {
            eta /= eta2;
    } else
        eta = 1.0; /* Anm.: Indizees sind gleich (keine Brechung) */

    Nn = faceforward(normalize(N), I);

    Ci = Ka * ambient();    /* ambienter Anteil */

    /* Diffuse und spekulare Anteile */
```

```
    illuminance( P, Nn, PI/2 ) {
        /* Diffus */
        Ci += Kd * Cl * L.Nn;

        /* Spekular */
        H = normalize(L+I);
        Ci += Ks * Cl * pow(max(0.0, Nn.H), Kss);
    }

    /* Reflexion */
    if ( Ks != 0.0 ) {
        R = reflect(I, Nn);
        Ci += Ks * trace(R);
    }

    /* Durchgang */
    if ( Kt != 0.0 ) {
        T = refract(I, Nn, eta);
        if ( length(T) != 0.0 )
            /* falls keine totale Reflexion */
            Ci += Kt * trace(T);
    }
}
```

Mit Renderern, die die **trace()** Funktion ausführen, kann auf diese Weise für die Oberflächen, für die der Shader eingesetzt wurde, ein Raytracing durchgeführt werden. Es wird im Prinzip die folgende Formel berechnet:

$$\overline{Ci} = Ka \cdot \text{ambient}() + \sum_{k=1}^{m} G_k \overline{Cl_k} \cdot \left[Kd \cdot (\overrightarrow{Nn} \cdot \overrightarrow{L_k}) + Ks \cdot (\overrightarrow{Nn} \cdot \overrightarrow{H_k})^{Kss} \right] + Ks \cdot \text{trace}(\vec{R}) + Kt \cdot \text{trace}(\vec{\text{T}})$$

Die Funktion $\text{trace}(\vec{V})$ berechnet rekursiv die $\overline{Ci}$ der Lichtstrahlen, die von der aktuellen Shading-Position aus der Richtung $\vec{V}$ einfallen. Die Summenbildung über alle Lichtquellen geschieht im **illuminance()** Konstrukt. Der Geometrie-Term G_k wird vom System an das Licht $\overline{Cl_k}$ multipliziert und taucht deshalb nicht explizit im Shader auf. Die Farben sind als Vektor aller Farb-Koeffizienten des jeweiligen Farbraums gegeben.

4.4 Radiosity und die Shading Language

Beim Raytracing wird versucht, die spiegelnden Anteile der Oberflächenreflexion möglichst genau nachzubilden, während die diffusen und ambienten Anteile der Reflexion mit herkömmlichen Mitteln angenähert werden. Mit dem Radiosity-Verfahren wird versucht, ein genaueres Rechenverfahren für diese beiden Komponenten bereitzustellen. Alle Körper werden als ideal diffus reflektierend (‘Lambertsche Oberflächen’) angenommen; sie reflektieren gleichförmig in alle Richtungen. Das hat zur Folge, daß im Gegensatz zum Raytracing, in der die Blickrichtung

eine große Rolle spielt, beim Radiosity-Verfahren lediglich die Geometrie der Szene von Bedeutung ist. Auch die Geschichte eines Strahls geht im Gegensatz zum Raytracing nicht in die Berechnung der Energieverhältnisse ein. Im Fall von Radiosity spricht man deshalb gegenüber dem blickpunktabhängigen Raytracing von einem blickpunktunabhängigen Verfahren.

Mit Radiosity wird die Summe der reflektierten und emittierten Strahlungsenergie (Licht) in einer Szene bezeichnet. Die Energie einer Szene ändert sich nicht mit der Zeit. Mit folgender Formel (s. [CohG85]) kann der Energieaustausch für ideal diffus reflektierende Oberflächen berechnet werden:

$$B_i = E_i + \rho_i \cdot \sum_{j=1}^{N} B_j \cdot F_{i,j}$$

B_i : Radiosity der Oberfläche i, Gesamtheit der Energie, die eine Oberfläche i verläßt (Einheit: Energie/Zeiteinheit/Flächeneinheit, W/m^2, Anm.: Leistung W: Energie/Zeiteinheit)

E_i : Emittierte Energie, von einer Oberfläche i selbständig abgestrahlte Energie (Einheit: Energie/Zeiteinheit/Flächeneinheit, W/m^2)

ρ_i : Reflektivität einer Oberfläche i, Anteil der Energie, die zurück in die Umgebung gestrahlt wird (< 1, keine Einheit)

$F_{i,j}$: (Berechneter) Anteil der Energie, die von einer Oberfläche j zu der Oberfläche i gelangt (keine Einheit)

Nach einer Umstellung der Formel:

$$E_i = B_i - \rho_i \cdot \sum_{j=1}^{N} B_j \cdot F_{i,j}$$

erhält man leicht das Gleichungssystem in der Matrixschreibweise:

$$\begin{bmatrix} 1-\rho_i \cdot F_{1,1} & -\rho_i \cdot F_{1,2} & \cdots & -\rho_i \cdot F_{1,N} \\ -\rho_i \cdot F_{2,1} & 1-\rho_i \cdot F_{2,2} & \cdots & -\rho_i \cdot F_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_i \cdot F_{N,1} & -\rho_i \cdot F_{N,2} & \cdots & 1-\rho_i \cdot F_{N,N} \end{bmatrix} \times \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{bmatrix}$$

Dieses Gleichungssystem kann z.B. mit Hilfe des Gauß-Seidelschen Iterationsverfahrens (s. [EngR91]) für jede Komponente des Farbsystems einzeln gelöst werden. Vorher müssen jedoch die Formfaktoren bestimmt werden.

Die Formfaktoren sind unabhängig von den verwendeten Farben und der Blickrichtung. Sie werden ausschließlich von der Geometrie der Szene beeinflußt. Wird die Szene nicht umgestellt, können die berechneten Formfaktoren in Bildsequenzen wiederverwendet werden. Vor allen in Kamera-Animationen und bei Lichtfarbänderungen kommt dieses Verhalten positiv zum Tragen. Die Berechnung der Formfaktoren kann mit Hilfe des in [CohG85] vorgestellten Hemi-Cube-Algorithmus geschehen. Durch die Verwendung von zwei Hemi-Cubes, einen obereren und einen unteren, könnten auch transparente Oberflächen modelliert werden (s. [WalCG87]).

Bezogen auf die Render Pipeline des RenderMan Interfaces muß bei der Verwendung eines Radiosity-Verfahrens die Berechnung der Formfaktoren nach der Aufstellung der internen Datenstruktur erfolgen. Da mit jedem neuen Frame eine komplett neue Geometrie aufgebaut wird, ist es ratsam, durch eine Option zu steuern, ob sich in einem neuen Frame die Geometrie ändert oder nicht. Im Fall einer gleichgebliebenen Geometrie brauchen die Formfaktoren nicht mehr berechnet werden, auch braucht in den folgenden Anweisungen bei einer geeigneten internen Darstellung nur noch die Kameraposition und der Zustand der Lichtquellen ausgewertet werden. Die Integration (bzw. Summation) des Lichts geschieht in dem **illuminate()** Konstrukt in den Oberflächen-Shadern durch die Iteration aller Oberflächen als Lichtquellen. In den Oberflächen, die zu einer 'Area Lightsource' gehören, wird zusätzlich das emittierte Licht addiert. Anders als das Raytracing hat das Radiosity-Verfahren keinen Einfluß auf das Inventar der *Shading Language*. Die Reflektivität ρ_i einer Oberfläche sollte als Instanzvariable (Parameter) in einem Oberflächen-Shader bereitgestellt werden. Der ambiente Anteil der Reflexion braucht prinzipiell nicht mehr gesondert aufaddiert zu werden, da er, durch das Radiosity-Verfahren bedingt, schon in der auf einer Oberfläche eintreffenden Energie enthalten ist. Durch die Verwendung von 'Environment Mapping' und einem spekularen Anteil können, mit dem Preis der Verfälschung der Energiebilanz, auch spiegelnde und glänzende Oberflächen dargestellt werden.

4.5 Gemischtes Raytracing und Radiosity und die Shading Language

Weder Raytracing noch Radiosity bieten für sich allein gesehen eine umfassende Simulation der Beleuchtung. Raytracing beschreibt zwar recht gut die blickrichtungsabhängige spekulare Komponente, verwendet aber für die diffuse herkömmliche Verfahren. Auch die spekularen Interobjektreflexionen, die nicht durch das Standard-Strahlen-Rückverfolgungsverfahren gefunden werden, werden nicht beachtet. Radiosity hingegen befaßt sich ausschließlich mit der diffusen Komponente. Es liegt daher nahe, die beiden Verfahren so zu kombinieren, daß sie sich gegenseitig ergänzen. Die einfachste Möglichkeit ist, beide Verfahren hintereinander auszuführen: Zuerst wird mit dem Radiosity-Verfahren die diffuse Komponente berechnet und anschließend mit dem Raytracing die spekulare. Wie in [WalCG87] dargestellt, deckt eine einfache Hintereinanderausführung der Algorithmen nicht alle Aspekte der Beleuchtung ab, obwohl sie für eine Vielzahl von Anwendungen ausreichend ist. Da sich die photorealistische Darstellung zur Aufgabe gemacht hat, computergenerierte Grafiken möglichst genau zu erzeugen, wird versucht, die Methoden so zu erweitern und zu verfeinern, daß eine physikalisch adäquate Darstellung des Lichttransports näherrückt. Radiosity behandelt für sich allein genommen den diffusen zu diffusen Transport, während Raytracing den spekularen zu spekularen und den diffusen zu spekularen Transport behandelt. Der spekulare zu diffuse Transport wird von keinem der beiden Verfahren beachtet — der Strahlen zurückverfolgende Standard-Raytracer stoppt, wenn er auf eine diffuse Oberfläche stößt, beachtet also die spekularen Reflexionen (und Transmissionen), die diese Oberfläche zusätzlich beleuchten können nicht mehr. Das auf einen Spiegel treffende Licht wird also von keinem der beiden Verfahren oder von einem bloßen Hintereinanderschalten der beiden Verfahren korrekt modelliert. Mit dem Zwei-Pass Verfahren von Wallace, Cohen und Greenberg wird versucht, auch den letzten 'Transportmechanismus' zu integrieren:

Lichttransport = diffuser zu diffuser + spekularer zu diffuser +
diffuser zu spekularer + spekularer zu spekularer Transport

Beschrieben wird dieser Sachverhalt durch die Formel:

$$I_{out}(\theta_{out}) = E(\theta_{out}) + I_{d,out} + I_{s,out}(\theta_{out})$$

$$I_{d,out} = k_d \rho_d \int_{\Omega} I_{in}(\theta_{in}) \cos(\theta) d\omega$$

$$I_{s,out}(\theta_{out}) = k_s \int_{\Omega} \rho_s(\theta_{out}, \theta_{in}) I_{in}(\theta_{in}) \cos(\theta) d\omega$$

$$k_s + k_d = 1$$

mit

θ_{in}	:	Richtung der einfallenden Energie
θ_{out}	:	Richtung der ausfallenden Energie
θ	:	Differenz der Oberflächennormalen und der einfallenden Richtung θ_{in}
Ω	:	Sphäre der einfallenden Richtungen
$d\omega$	:	Differentieller ‘Solid Angle’ durch den die Intensitäten einfallen (s. ‘Physically based illumination models’ in [FolVDFH90])
$E(\theta_{out})$	:	Energie, die in Richtung θ_{out} abgegeben wird
$I_{out}(\theta_{out})$	:	Gesamtenergie, die in Richtung θ_{out} abgegeben wird. Die einfallende Energie wird über die Richtungen Ω integriert und entsprechend der Reflektivität der Oberfläche weitergegeben.
$I_{in}(\theta_{in})$	:	Gesamtenergie, die aus Richtung θ_{out} einfällt
$I_{d,out}$	:	Diffuse Komponente der abgegebenen Energie.
$I_{s,out}(\theta_{out})$	:	Spekulare Komponente der in Richtung θ_{out} abgegebenen Energie
k_d	:	Diffuser Koeffizient
k_s	:	Spekularer Koeffizient
ρ_d	:	Diffuse Reflektivität einer Oberfläche
$\rho_s(\theta_{out}, \theta_{in})$	:	Bidirektionale Reflexionsfunktion, Relativierung der Energie, die aus Richtung θ_{in} einfällt und in Richtung θ_{out} abgegeben wird

Anm.: Im Gegensatz zur Kajiya Formel werden Richtungen statt Punkte verwendet, deshalb die Abweichung der Schreibweise der bidirektionalen Reflexionsfunktion ρ_s.

Das Verfahren teilt sich in einen blickpunktunabhängigen Präprozeß, der alle vier Lichttransportarten zwischen den Objekten behandelt und einen sichtabhängigen Postprozeß, der, von den Ergebnissen des ersten Prozesses ausgehend, die kamerapositionsabhängigen spekularen Komponenten der Reflexionen aufrechnet und das Ausgabebild erzeugt.

Die Basis für den Präprozeß bildet der Hemi-Cube Radiosity Algorithmus von Cohen und Greenberg. Der Algorithmus wurde um die Behandlung der Transmission von Licht durch einen zweiten Hemi-Cube auf der Rückseite der Oberfläche erweitert. Durch einen 'Trick' wurde auch die spekulare Reflexion in den Algorithmus integriert. 'Sehen' sich zwei Oberflächen über eine weitere spiegelnde, wird eine Pseudo-Oberfläche an der gegenüberliegenden Seite der spiegelnden Oberfläche eingefügt. Zu jeder spiegelnden Oberfläche wird also eine gespiegelte Pseudowelt erzeugt, die mit der zu rendernden Welt mit der spiegelnden Oberfläche als Fenster verbunden ist. Durch die Erzeugung dieser Spiegelwelten werden zusätzliche Formfaktoren in die Berechnung der Energieverteilung eingefügt, sodaß auch die spekularen Energietransporte zwischen den Objekten behandelt werden. Durch eine rekursive Anwendung der Erzeugung der Spiegelwelten und der symmetrischen Natur der Energieverteilung zwischen zwei Oberflächen können alle vier Transportarten integriert werden. Mit dem Algorithmus können momentan nur planare, ideal spiegelnde Oberflächen modelliert werden. Nicht-planare Oberflächen beeinflussen die Geometrie der Pseudowelten. Für nicht ideale Spiegel wird eine Wichtung der Formfaktoren nötig.

Die Ausgabe des Präprozesses sind die physikalisch adäquat berechneten diffusen Komponenten der Intensität von geeignet gewählten Sample-Punkten. Sie besteht nicht nur aus den Formfaktoren wie beim Standard-Radiosity. Die Intensitäten der zwischenliegenden Punkte können durch Interpolation bestimmt werden. Der auf den Ergebnissen des Präprozesses aufbauende Postprozeß berechnet die Intensität der Pixel für eine gegebene Blickrichtung durch Einbeziehung der spekularen Komponente. Um sie berechnen zu können, wird in [WalCG87] eine Methode vorgeschlagen, die ein sogenanntes 'Reflection Frustum' (bzw. 'Transmisson Frustum') verwendet. Obwohl auch die spekulare Komponente von den Intensitäten abhängt, die über die ganze Hemisphäre über einen Punkt einfallen, wird sie ausschlaggebend von den Intensitäten, die durch einen Kegel in Richtung der Reflexion einfallen, beeinflußt. Mit Hilfe des Frustums wird eine (Z-Buffer) Matrix von einfallenden, gewichteten Intensitäten (aus den spekularen und diffusen Interobjekt-Reflexionen, die im Präprozeß berechnet wurden) aufgebaut, die untereinander addiert die spekulare Komponente ergeben. Spiegelt eine Oberfläche, deren Intensität in die Berechnung eingeht, wird Raytracing-ähnlich rekursiv die entsprechende richtungsabhängige spekulare Komponente durch ein weiteres Frustum eingerechnet. Die Auflösung der Frusten (bzw. der Z-Buffer) kann mit steigender Rekursionstiefe verkleinert werden. Durch zufällige Drehungen der verwendeteten Frusten kann der Aliaseffekt, der durch die Verwendung des diskretisierenden Z-Buffers entsteht, vermindert werden. Je kleiner der Öffnungswinkel des Frustums gewählt wird, desto spiegelartiger erscheint eine Oberfläche. Der Z-Buffer hat normalerweise eine geringe Auflösung von 10×10 Pixeln, durch die die betrachteten Richtungen in einen diskreten Rahmen gepreßt werden. Das Integral in der Berechnung von $I_{s,out}(\theta_{out})$ wird also durch zwei verschachtelte Summen für einen $n \times n$ Z-Puffer approximiert:

$$I_{s,out}(\theta_{out}) = k_s \sum_{i=1}^{n} \sum_{j=1}^{n} W_{i,j} I_{in}(\theta_{i,j}) \cos(\theta_{i,j}) \Delta\omega_{i,j}$$

Die Wichtungen $W_{i,j}$, die die bidirektionale Reflexionsfunktion repräsentieren, können durch die Phong-Reflexionsfunktion oder komplexeren physikalischen Reflexionsmodellen eines Oberflächenmaterials vorbestimmt werden. $\theta_{i,j}$ ist die Richtung, die durch das Pixel (i, j) bestimmt wird. Die Änderungen von $\cos(\theta_{i,j})\Delta\omega_{i,j}$ über die Pixel des Z-Buffers sind gering, so daß ein konstanter Faktor angenommen werden kann.

Möchte man dieses Verfahren für einen Renderer des RenderMan Interfaces verwenden, stößt man auf die Schwierigkeit, daß die Intensitäten von Pixeln durch die Shader nur komplett berechnet werden können und nicht nach diffusen und spekularen Interobjektreflexionen und spekularen blickpunktabhängigen Reflexionen getrennt. Es kann kein virtuelles Spiegelobjekt erzeugt werden, weil die Shader nicht die Reflexionseigenschaften sondern die Farbe von Oberflächenpunkten liefern. Die Spiegelwelt muß also schon beim Modellieren erzeugt werden, was nur in Grenzfällen funktioniert. Die Summation von Intensitäten innerhalb des ‘Reflection Frustums’ kann nur vereinfacht durch die **trace()** Funktion geschehen. Es müssen von dem Shader selbst mehrere **trace()** Funktionen abgesetzt und gewichtet werden, ohne daß die Anzahl der Aufrufe mit steigender Rekursionstiefe vermindert werden kann — der Wert der Rekursionstiefe ist von der ‘Shading Language’ aus nicht zugänglich.

A. Die RenderMan Befehle

Da die Interface-Beschreibung an verschiedene Sprachen ‘gebunden’ werden kann, spricht man von Language-Binding. Standardmäßig wird das C-Binding und das RIB-Binding unterstützt. Neben C- und RIB- sind auch CommonLisp-, Pascal-, Smalltalk- o.ä. Bindings denkbar.

Im folgenden werden die Typen, Konstanten, Variablen und Funktionen des C-Bindings von RenderMan aufgeführt. Die Deklarationen sind normalerweise in der Header-Datei `ri.h` zu finden. Anschließend folgt eine zusammenfassende Übersicht der Shading Language.

Das C-Binding und die Shading Language wird ausführlich in dem Buch [Upstill89] beschrieben. In der RenderMan Interface-Beschreibung [PixSpec] finden sich zusätzlich Hinweise zu den RIB Konventionen.

Der auf dem NeXT zur Verfügung stehende *Quick RenderMan* benutzt eine zur 3.1 Spezifikation leicht abweichende Interface-Definition (s. [PixQRM]). Neben den hinzugekommenen Renderkontexten ist auch der Scopemechanismus einiger Attribute zu dem der Version 3.1 verschieden. Auf Handles von Datenstrukturen oder externen Ressourcen kann nach dem Beenden des sie umgebenden Attribut-Blocks nicht mehr zugegriffen werden. Insbesondere kann auf die mit **RiLightSource()** und **RiAreaLightSoure()** erzeugten Handles von Lichtquellen außerhalb des Blocks, in dem sie definiert wurden nicht mehr zugegriffen werden. Möchte man auf die Lichtquellen auch außerhalb des Attribut-Blocks, in dem sie instanziert wurden, zugreifen, muß ihr Handle schon zuvor mit **RiCreateHandle()** vordefiniert werden. Handles von Shadern und anderen externen Ressourcen müssen mit **RiResource()** erzeugt werden. In den Interface-Funktionen (**RiSurface()**, **RiDeformation()**, ...), in denen sie instanziert werden, muß dann anstelle des Namens das entsprechende Token stehen, statt ‘`char *name`’ wird ein Parameter ‘`RtToken name`’ verwendet.

A.1 Das Bytestream Protokoll (RIB)

Für das RIB-Binding werden die **Ri...** Präfixe in den Namen der Interface-Funktionen (**request**) weggelassen (z.B. **RiLightSource()** vs. **LightSource**). Dem Befehl entsprechend folgt eine Parameterliste:

request *parameter1 parameter2 ... parameter2*

Es brauchen nicht immer Parameter vorhanden zu sein (angezeigt durch ein ‘–’). Sind die Typen der Parameter nicht explizit vorgegeben (Parameterliste am Ende der vorgegebenen Parameter) werden Parameternamen (sie stehen in Doppelhäckchen) gefolgt von den Werten: Nume-

rische Konstanten, Matrizen und Arrays in eckigen Klammern ([,]) und Strings in Doppelhäckchen entsprechend der C-Syntax verwendet. Integers werden automatisch in Fließkommazahlen umgewandelt. 'White Spaces' wie Zeilenende, Leerzeichen und Tabulatoren werden überlesen und dienen nur zur Trennung der lexikalischen Einheiten. Als Namen werden alle Einheiten erkannt, die keine Zahlen sind und keine der begrenzenden, speziellen Zeichen (", #, [,], 'White Spaces') besitzen. Kommentare können hinter einem einzelnen '#' eingefügt werden. Neben dem ASCII-Format können RIB-Dateien auch binär kodiert werden (s. [PixSpec]).

Wie für PostScript existieren für RIB auch Strukturkonventionen. Sie werden hinter einem doppelten Nummerzeichen eingefügt. Eine RIB-Datei, die eine komplette Szene enthält und kompatibel zur Version 3.1 des Interfaces ist, sollte folgenden Kopf besitzen:

```
##RenderMan RIB-Structure 1.0
```

Neben den vollständigen RIB-Dateien können RIB-Entity-Dateien ('clip-art') existieren, die in etwa den EPS-Dateien von PostScript entsprechen. Sie sind durch den folgenden Kopf gekennzeichnet:

```
##RenderMan RIB-Structure 1.0 Entity
```

Eine RIB-Entity-Datei sollte als Attribut-Block gekapselt sein und anfangs die Boundingbox in einem **Bound**-Befehl definieren. Das Objektkoordinatensystem muß um den Koordinatenursprung zentriert sein. Eine vollständige Auflistung der Strukturkonventionen kann in [PixSpec] nachgeschlagen werden.

A.2 Das C-Binding

In der Aufzählung der Funktionen steht die Ellipse (**...**) für eine Token-Array-Liste aus einer Sequenz von Paaren bestehend aus einem Token (**RI_...**-Konstante vom Typ **RtToken** oder dem Namen des Tokens als String), das den Wert-Typ bestimmt, gefolgt von einem Zeiger auf den entsprechenden Wert, z.B. '`RI_P, (RtFloat *) pointList`'. Der Wert entspricht einem (im Grenzfall einelementigen) C-Array. Die Liste wird durch das **RI_NULL** Token (ohne folgenden Wert) abgeschlossen. Die Tokens sind als konstante **extern** Variablen in der Include-Datei `ri.h` deklariert. Der durch sie bestimmte Typ des Arrays steht in der entsprechenden Konstantenbeschreibung. In der Funktionsbeschreibung ist zu finden, welche Token in der Liste verwendet werden dürfen. Wenn in der Funktionsbeschreibung nichts vermerkt ist, besitzt der Interface-Aufruf keine Standard-Attribute und die Parameterliste besteht nur aus dem obligatorischen Eintrag **RI_NULL**. Token können auch als Strings übergeben werden: '`(RtToken)"name", (RtPointer)&Wert`'. Die Inhalte der Strings entsprechen den **RI_...** Tokens. Der String wird in Kleinbuchstaben, ohne vorheriges **RI_** angegeben. Das 'bound by name' ist wichtig, weil die Namen der Instanzvariablen (deklariert als Parameterliste) von selbstdefinierten Shadern und die Parameter für spezielle Geometrien (**RiGeometry()**), Optionen und Attribute, nicht von vornherein feststehen (s.a. **RiDeclare()**, zur Definition von varying und

uniform[1] Variablen als Tokens). Nicht unterstützte Token-Array Paare werden normalerweise von den Interface-Funktionen und den Shadern ignoriert. Variablen sind immer mit Defaultwerten vorbelegt. Ähnliches gilt für etwaige Nichtstandard-Variablen. Die (interpolierten) Werte der Parameter können vom Renderer einem Shader in seinen Instanz- und Steuervariablen übergeben werden, Bsp.: Die aktuelle interpolierte Oberflächenfarbe aus den **RI_C** Parametern der Parameterliste eines Grafikobjekt-Befehls wird in der (varying) Steuervariablen **Cs** des Shaders bereitgestellt.

Alternativ zu den hier aufgeführten Funktionen mit den veränderlichen Parameterlisten existieren alternative Funktionen mit einer festen Anzahl formaler Parameter. Die Funktionen enden mit einem **V** und bekommen die Token-Wert Liste in Form einer Feldgröße und zwei getrennter Felder der Typen **RtToken** und **RtPointer** als letzte Parameter übergeben. Das erste der beiden Felder enthält die Token als Elemente, das zweite die Zeiger auf die Werte. Beispiel: **RiHider(RtToken type,...)** vs. **RiHiderV(RtToken type, RtInt n, RtToken tokens[], RtPointer params[])**.

A.2.1 Die Typen

RtBoolean: Typ zur Darstellung der logischen Werte **RI_TRUE** und **RI_FALSE**

RtInt: Ganzzahlen

RtFloat: Realzahlen

RtString: Zeichenketten (in C-Form, d.h. abschließende 0)

RtPointer: Nichttypisierter Zeiger auf Daten

RtVoid: 'Leere' Rückgabe einer Funktion

RtToken: Tokentyp für die Parameterlisten der Funktionen, alle Standard-Token sind als **RI_...** Konstanten deklariert. Das spezielle Token **RI_NULL** beendet die Parameterlisten.

RtColor[3]: **RtFloat** Farbvektor, normalerweise 'rgb'. Die Dimension des Vektors ist standardmäßig drei. Anders dimensionale Farbräume (z.B. der eindimensionale schwarz/weiß Raum) können durch die **RiColorSamples()** Interface-Funktion definiert werden. Anschließend werden für **RtColor** entsprechend n-dimensionale Vektoren verwendet. **RtColor** wird auch zur Darstellung der Opazität verwendet. Die Werte in den Feldern entsprechen dann der Opazität der entsprechenden Farbkomponenten.

RtPoint[3]: **RtFloat** Punkt im dreidimensionalen Raum

RtMatrix[4][4]: Transformationsmatrix: Zeile $\times$ Spalte, Row-Major. Der Typ der Felder ist **RtFloat.**

[1] Die *varying* (Wert aus einem Array mit je einem Wert pro definierten Oberflächenpunkt, der Index ändert sich beim Schattieren über die Oberfläche, die Werte der Bildpunkte, die zwischen den definierten Punkten liegen können interpoliert werden) und *uniform* (Wert bleibt über die Oberfläche konstant) Speicherklassen sind in dem Kapitel über die Shading Language beschrieben.

RtBasis[4] [4]**:** Spline-Basismatrix

RtBound[6]**:** dreidimensionale Bounding-Box (quaderförmige Hülle)
$[x_{min}\ x_{max}\ y_{min}\ y_{max}\ z_{min}\ z_{max}]$.

RtFloatFunc: Zeiger auf eine Funktion, die einen **RtFloat**-Typ liefert:
`RtFloat (* RtFloatFunc)()`

RtFunc: Zeiger auf eine Funktion ohne Rückgabewert: `RtVoid (*RtFunc)()`

RtObjectHandle: Zeiger auf ein (zusammengesetztes) Grafikobjekt (s.a. **RiObjectBegin()**, **RiObjectEnd()** und **RiObjectInstance()** zur Objektdefinition bzw. Instanzierung).

RtLightHandle: Zeiger auf ein (zusammengesetztes) Beleuchtungsobjekt zum späteren Ein- und Ausschalten der Lichtquelle.

A.2.2 Die Konstanten und Variablen

Nicht alle der folgenden **RI_...** Konstanten sind Tokens. Bei den Tokens wird der Typ der Elemente des folgenden Arrays angegeben. Die Tokens mit Ausnahme von **RI_NULL**, denen kein Wert folgt, werden im C-Interface ausschließlich als Werte verwendet. Im RIB-Code sind ihre Entsprechungen in der Parameterliste eingereiht. Die folgende Zusammenstellung ist nur eine Auflistung der Namen.

Basiskonstanten

RI_FALSE: **RtBoolean** Wert für logisch falsch

RI_TRUE: **RtBoolean** Wert für logisch wahr

RI_INFINITY: **RtFloat** größt mögliche Fließkommazahl, die das Interface bearbeiten kann, z.B. größt mögliche Entfernung der Rückseite des abzubildenden Raumes zu der Kamera

RI_EPSILON: **RtFloat** kleinste Zahl > 0, z.B. kleinst mögliche Entfernung der Vorderseite des abzubildenden Raumes zu der Kamera.

Es folgen die Tokenbezeichner vom Typ RtToken

RI_NULL: Null-Zeiger, Ende einer Parameterliste, dient als Wert.

Token für die Kamera, werden vorzugzweise in RiDisplay() verwendet

RI_FRAMEBUFFER: Logisches Ausgabegerät ist Framebuffer, kann in der **RiDisplay()** Funktion verwendet werde. Der Name des Buffers folgt als String.

RI_FILE: Logisches Ausgabegerät ist eine Datei, kann in der **RiDisplay()** Funktion verwendet werden. als Wert folgt als **RtString** der Name der Datei.

RI_RGB: Nur RGB Farbe ausgeben, dient als Wert

RI_RGBA: RGB und Transparenz (Alpha-Kanal) ausgeben, dient als Wert

RI_RGBZ: RGB und Entfernungsinformation ausgeben, dient als Wert

RI_RGBAZ: RGB, Transparenz und Tiefe ausgeben, dient als Wert

RI_A: Nur Transparenz ausgeben, dient als Wert

RI_Z: Nur Tiefenwerte (z.B. für 'Depth Maps') ausgeben, dient als Wert

RI_AZ: Transparenz und Tiefe ausgeben, dient als Wert

RI_ORIGIN: Ursprung eines Koordinatensystems vom Typ **RtPoint**. In der Parameterliste der **RiDisplay()**-Funktion kann dieser Parameter dazu verwendet werden den Ursprung des Bitmaps neu zu setzen.

RI_PERSPECTIVE: Kennzeichnet perspektivische 3D-2D Transformation, dient als Wert

RI_ORTHOGRAPHIC: Kennzeichnet parallele 3D-2D Transformation, dient als Wert

RI_HIDDEN: Verdeckte Flächen mit Standard-Algorithmus behandeln, dient als Wert

RI_PAINT: Flächen nur nach dem 'Painters algorithm' in der Reihenfolge ihrer Definition ausgeben, dient als Wert. In den NeXT Renderern wird die Bezeichnung 'InOrder' verwendet.

RI_CONSTANT: Konstantes Shading, keine Interpolation von Farbwerten innerhalb und zwischen Flächen durchführen, dient als Wert

RI_SMOOTH: Interpolation zulassen, dient als Wert

RI_FLATNESS: Grad der Annäherung gekrümmter Oberflächen durch Polygone approximieren, s.a. **RiGeometricApproximation()**

RI_FOV: Gesichtsfeld ('Field Of View') der Kamera als **RtFloat**

Beleuchtungsmodell

RI_AMBIENTLIGHT: Standard ambiente Lichtquelle, dient als Wert

RI_POINTLIGHT: Standard Punktlichtquelle, dient als Wert

RI_DISTANTLIGHT: Standard Lichtquelle mit parallelen Strahlen, dient als Wert

RI_SPOTLIGHT: Standard Scheinwerfer, dient als Wert

RI_INTENSITY: Helligkeit einer Lichtquelle (**RiFloat**)

RI_LIGHTCOLOR: Lichtfarbe (**RiColor**)

RI_FROM: Position der Lichtquelle (**RiPoint**)

RI_TO: (**RI_TO – RI_FROM**) ergibt den Vektor für die Strahlenrichtung (**RiPoint**)

RI_CONEANGLE: Winkel der Spitze des Beleuchtungskonus beim Scheinwerfer zur Simulation der Klappen (**RiFloat**)

RI_CONEDELTAANGLE: Winkel ab dem Helligkeit im Beleuchtungskonus eines Scheinwerfers abfallen soll (**RiFloat**)

RI_BEAMDISTRIBUTION: Strahlenverteilung des Scheinwerfers, ohne Verwendung der Klappen (**RtFloat**)

RI_MATTE: Shading abschalten, dient als Wert

RI_METAL: Standard-Shader für metalartige Oberflächen, dient als Wert

RI_SHINYMETAL: Standard-Shader für metalartige Oberflächen mit 'Environment Maps', dient als Wert

RI_PLASTIC: Standard-Shader für plastikartige Oberflächen, dient als Wert

RI_KA: Shaderparameter, Koeffizient der ambienten Reflexion (**RtFloat**)

RI_KD: Shaderparameter, Koeffizient der diffusen Reflexion (**RtFloat**)

RI_KS: Shaderparameter, Koeffizient der spekularen Reflexion (**RtFloat**)

RI_ROUGHNESS: Shaderparameter, Rauheit einer Oberfläche (**RtFloat**, beeinflußt bei der spekularen Reflexion den Rand und die Größe von Glanzpunkten)

RI_SPECULARCOLOR: Gesonderte Farbe der Glanzpunkte der spekularen Reflexion (**RtColor** z.B. für Plasikoberflächen)

RI_DEPTHCUE: Standard Atmosphären-Shader für Depth Cue (Tiefenschnitt), kein Wert

RI_FOG: Standard Atmosphären-Shader für Nebeleffekt, kein Wert

RI_BUMPY: Standard 'Displacement'-Shader für 'Bump Maps', kein Wert

RI_MINDISTANCE: Mindestabstand der Körper von der Kamera **RtFloat**,

$$\mathbf{RI\_EPSILON} \leq Wert \leq \mathbf{RI\_INFINITY}$$

zur Einbeziehung in einen Atmosphären-Shader

RI_MAXDISTANCE: Maximalabstand der Körper von der Kamera **RtFloat**,

$$\mathbf{RI\_EPSILON} \leq Wert \leq \mathbf{RI\_INFINITY}$$

zur Einbeziehung in einen Atmosphären-Shader

RI_BACKGROUND: Farbe des Hintergrunds **RtColor**

RI_DISTANCE: Abstand des Fokuspunktes von der Kamera **RtFloat**

RI_AMPLITUDE: Dither-Amplitude, **RtFloat**

Koordinatensysteme, Rendering-Prozeß, vordefinierte Parameter

RI_RASTER: Rasterkoordinaten, dient als Wert

RI_SCREEN: (normiertes) Bildkoordinaten, dient als Wert

RI_CAMERA: Kamerakoordinaten, dient als Wert

RI_WORLD: Weltkoordinaten, dient als Wert

RI_OBJECT: Objektkoordinaten, dient als Wert

RI_INSIDE: Inneres einer Oberfläche, dient als Wert

RI_OUTSIDE: Äußeres einer Oberfläche, dient als Wert

RI_LH: Linkshändige Orientierung, dient als Wert

RI_RH: Rechtshändige Orientierung, dient als Wert

RI_P: Knotenliste, Array von **RtPoint**

RI_PZ: Array von **RtFloat**, beinhaltet nur Tiefenwerte

RI_PW: vierspaltiges **RtFloat** Array (homogener Vektor für rationale B-Splines)

RI_N: Array von Normalen **RtPoint**

RI_NP: einzelner **RtFloat** Wert (Normale)

RI_CS: Array von Farben **RtColor**

RI_OS: Array von Opazitäten **RtColor**

RI_S: Array von **RtFloat** Texturkoordinaten (s-Richtung, entspr. parametrischer u-Richtung)

RI_T: Array von **RtFloat** Texturkoordinaten (t-Richtung, entspr. parametrischer v-Richtung)

RI_ST: zweispaltiges **RtFloat** Array von Texturkoordinaten (abwechselnd s und t)

RI_BILINEAR: Bilineare Interpolation einer 2×2 Oberfläche, dient als Wert

RI_BICUBIC: Bikubische (Spline) Interpolation oder Approximation einer 4×4 Oberfläche, dient als Wert

RI_PRIMITIVE: CSG-Typ, Definition eines soliden Körpers

RI_INTERSECTION: CSG-Typ, Körper wird aus der Schnittmenge von anderen Körpern berechnet

RI_UNION: CSG-Typ, Körper wird aus der Vereinigungsmenge von anderen Körpern berechnet

RI_DIFFERENCE: CSG-Typ, Körper wird aus der Differenzmenge von Körpern berechnet (Differenz vom ersten angegebenen Körper)

RI_WRAP: Oberflächen werden zirkulär geschlossen (z.B. Splineflächen), dient als Wert

RI_NOWRAP: Oberflächen bleiben offen, dient als Wert

RI_PERIODIC: Periodisches 'wrappen' vom Texturen außerhalb des Einheitsquadrates des Texturkoordinaten-Raums, dient als Wert

RI_CLAMP: Außerhalb des Einheitsquadrates des Texturkoordinaten-Raums wird die Farbe von dem jeweiligen Textur-Rand verwendet, dient als Wert

RI_BLACK: Verwendung von schwarzer Farbe außerhalb des Einheitsquadrates des Texturkoordinaten-Raums, dient als Wert

Vordefinierte Spline-Knoteninkrements für Knotengitter

RI_BEZIERSTEP: dient als Wert

RI_BSPLINESTEP: dient als Wert

RI_CATMULLROMSTEP: dient als Wert

RI_HERMITESTEP: dient als Wert

RI_POWERSTEP: dient als Wert

Vordefinierte Spline-Basismatrizen

RiBezierBasis: dient als Wert

RiBSplineBasis: dient als Wert

RiCatmullRomBasis: dient als Wert

RiHermiteBasis: dient als Wert

RiPowerBasis: dient als Wert

Fehlerstufen

RI_IGNORE: Fehler wird ignoriert, dient als Wert

RI_PRINT: Text zum Fehler soll ausgegeben werden, dient als Wert

RI_ABORT: Fehlertext soll ausgegeben werden und der Fehler soll zum Abbruch des Renderprozesses führen, dient als Wert

RI_HANDLER: Prozedur zur Fehlerbehandlung **RtFunc**

RiLastError: Variable mit der letzten Fehlernummer

Fehlerstufen

Fehler Code	Bereich
$01 - 10$	System- und Dateifehler
$11 - 20$	Programm Einschränkungen
$21 - 40$	Zustandsfehler
$41 - 60$	Parameter- und Protokollfehler
$61 - 80$	Fehler bei der Ausführung

RIE_NOERROR: Kein Fehler aufgetreten

System- und Dateifehler

RIE_NOMEM: Kein Speicher mehr vorhanden

RIE_SYSTEM: Verschiedene Systemfehler

RIE_NOFILE: Datei nicht gefunden

RIE_BADFILE: Datei hat falsches Format, ist zerstört o.ä.

RIE_VERSION: Falsche Dateiversion

Programm Einschränkungen

RIE_INCAPABLE: Optionale RI Funktion

RIE_UNIMPLEMENTED: Nicht implementierte Funktion

RIE_LIMIT: Grenzwertüberschreitung

RIE_BUG: Programmfehler im Renderer

Zustandsfehler

RIE_NOTSTARTED: **RiBegin()** nicht aufgerufen

RIE_NESTING: Blöcke überschneiden sich, Blockende nicht gefunden, usw.

RIE_NOTOPTIONS: Falscher Optionenzustand

RIE_NOTATTRIBS: Falscher Attributzustand

RIE_NOTPRIMS: Falscher Primitivzustand

RIE_ILLSTATE: Anderer falscher Zustand

RIE_BADMOTION: Falsch geformter ‘Motion Blur’-Block

RIE_BADSOLID: Falsch geformter Solid-Block

Parameter- und Protokollfehler

RIE_BADTOKEN: Token in Parameterliste nicht verwendbar

RIE_RANGE: Parameter außerhalb der zulässigen Grenzen

RIE_CONSISTENCY: Inkonsistente Parameter (z.B. hintere Klippebene näher als die vordere)

RIE_BADHANDLE: Kaputtes Licht- oder Objekthandle

RIE_NOSHADER: Shader konnte nicht geladen werden

RIE_MISSINGDATA: Unbedingt benötigte Parameter wurden nicht übergeben

RIE_SYNTAX: Syntaxfehler bei der Deklaration von Token **RiDeclare()**

Ausführungsfehler

RIE_MATH: Fehler bei Berechnung, z.B. durch Null dividiert

A.2.3 Die Funktionen

Alle Funktionen, die ohne Rückgabetyp aufgeführt sind, besitzen implizit die Rückgabe **RtVoid**.

RiArchiveRecord (RtToken type, RtString format...):

Schreiben eines 'user data records' (s. [PixQRM]) in ein RIB-Archiv. Der Typ ist entweder `"comment"` oder `"structure"`. Wird das Rib-Archive mit **RiReadArchive()** aus einer Ressource (s. **RiResource()**) zurückgelesen, kann beim Lesen eines Rekords eine benutzerdefinierte Callback-Routine aufgerufen werden. Sie bekommt die gleichen Parameter übergeben wie **RiArchiveRecord()**. Die Übergabe des Formatstrings und der Argumente entspricht der **printf()** C-Funktion. Die `"comment"` Rekords beginnen immer mit einem Kommentarzeichen '#' und enden mit einem Zeilenende, `"structure "` Rekords werden mit den RIB-Struktur-Konventionen eingeleitet und auch mit einem Zeilenende beendet.

RIB BINDING

ArchiveRecord type format parameterlist

RtLightHandle RiAreaLightSource (char *handle...):

3.1 Interface-Funktion zum Setzen eines Umgebungslichts als Attribut. Alle nach **RiAreaLightSource()** in dem gleichen Attribut-Block definierten geometrischen Primitive werden zu diesem Umgebungslicht hinzugefügt und erscheinen so insgesamt als Lichtquelle. Ein Zylinder kann als Modell für die Ausleuchtung einer Leuchtstoffröhre dienen. Ein Renderer braucht diese Art von Lichtquellen nicht notwendigerweise zu unterstützen. *name* ist der (Datei-)Name des Shaders, der auf das Umgebungslicht angewendet werden soll. Der Shader ist normalerweise ein 'Lightsource'-Shader, der keine Position, d.h. keine *from* Instanzvariable enthält. Die Position wird durch die Positionen der geometrischen Elemente bestimmt. Nach ihrer Definition können die Lichtquellen wie alle anderen

Lichtquellen (**RiLightSource()**) behandelt werden. Sie können mit **RiIlluminate()** aus-/ und eingeschaltet werden und sind nur in der 3.1 Version des Interfaces auch nach Beendigung der sie umgebenden Blöcke bis zum Ende des umgebenen Frame-Blocks, bzw. Welt-Blocks definiert. Die Lichtquelle wird am Ende des Attribut-Blocks, indem sie eingesetzt wurde, ausgeschaltet.

Die Parameterliste beinhaltet die Parameter-Wert-Paare, die dem Shader zum Zeitpunkt des Renderns als Instanzvariablen übergeben werden sollen. Die Parameternamen hängen von den Instanzvariablen des verwendeten Shaders *name* ab. Typisch ist das Paar `((RtToken)"intensity", (RtPointer)&doubleVar)`.

Der *Quick RenderMan* des NeXT implementiert die 'Area Lightsource' noch nicht. Sie entsprechen den normalen Lichtquellen. Es wird die folgende zur 3.1 Spezifikation abweichende Deklaration (s. [PixQRM]) verwendet:

RtToken RiAreaLightSource(RtToken handle, RtToken shader...)

shader ist das Handle eines mit **RiCreateHandle()** vordefinierten 'Area Lightsource'-Shaders oder das Handle eines mit **RiResource()** vom Benutzer definierten. *handle* ist von dem vom Benutzer vergebenen Namen der Lichtquelle abhängig. Die Lichtquelle ist sofort nach ihrer Definition eingeschaltet. Die folgenden Oberflächen, die die Lichtquelle definierten, sind also selbstbeleuchtend. Am Ende des umgebenden Attribut-Blocks wird die Lichtquelle automatisch wieder ausgeschaltet. Wird kein vordefiniertes Handle verwendet, kann auf die Lichtquelle nach dem Beenden des sie umgebenden Attribut-Blocks nicht mehr zugegriffen werden. Wird als Shader der Default-Shader `"null"` eingesetzt, wird kein Licht abgestrahlt.

In [PixQRM] wird das folgende Beispiel verwendet:

```
RtToken myarealight, amb;
RtFloat intensity = 0.5;

// Ressource definieren
amb = RiResource("ambientlight", RI_LIGHTSOURCE, RI_NULL);
// Lichtquelle erzeugen
myarealight = RiAreaLightSource("myalight", amb,
    "intensity", &intensity, RI_NULL);
```

RIB BINDING

AreaLightSource shader sequencenumber parameterlist (3.1)

Die *sequencenumber* ist eine eindeutige Nummer (0–65535) zur Identifikation der Lichtquelle.

AreaLightSource handle shader parameterlist (*Quick RenderMan*)

RiAtmosphere (char *name...):

Setzt als Attribut einen Atmosphären-Shaders, der für den Raum zwischen der Kamera und einer Oberfläche verwendet wird. *name* ist der (Datei-)Name des Shaders; die zu übergebenen Werte für Instanzvariablen stehen in der Parameterliste. Die Shader können, z.B. wie die Standard-Shader 'fog' und 'depth-cue', abhängig von der Entfernung der reflektierenden Oberfläche, eine Farbe in das in die Kamera eintreffende reflektierte Licht mischen. Der Atmosphären-Shader ist der einzige Volumen-Shader-Typ, für den Standard-Shader existieren. Die Renderer des NeXT können nur diese beiden Standard-Shader 'fog' und 'depth-cue' verwenden.

RIB BINDING

Atmosphere name parameterlist

RiAttribute (char *name...):

Belegung eines Attributes (*name*) mit den Werten aus der folgenden Parameterliste. Attribute werden innerhalb eines Welt-Blocks (**RiWorldBegin()**, **RiWorldEnd()**) in Attribut-Blöcken (**RiAttributeBegin()**, **RiAttributeEnd()**) lokal auf dem Attribut-Stack gespeichert und können beliebig oft verändert werden. Alle Attribute zusammengenommen bilden den Grafikzustand, den sog. Graphics-State. Beispielsweise gelten die aktuelle Transformation, die Oberflächenfarbe und die eingesetzten Shader als Attribute. Im Gegensatz zu Attributen dürfen Optionen **RiOption()** nicht innerhalb des Welt-Blocks geändert werden, sie gelten global, können aber außerhalb eines Welt-Blocks überschrieben werden. Standard-Attribute (und Optionen) können auch direkt durch Interface-Aufrufe geändert werden. Die Parameterliste ist abhängig von dem verwendeten Attribut. Die Attributnamen korrellieren zu den entsprechenden **Ri**-Interface-Aufrufen. Sie werden aber klein und ohne voranstehendes **Ri** geschrieben. Implementationen können die Menge der behandelten Attribute einschränken und nicht behandelte Attribute ignorieren oder Nichtstandard-Attribute verwenden (Escape-Funktion). Da diese Nichtstandard-Attribute, wie der Name schon sagt, nicht in die Interface-Definition einbezogen sind, ist **RiAttribute()** ein Mechanismus um sie relativ konform zu ändern. Hängt das Aussehen eines Bildes sehr stark von diesen Attributen ab, ist eine Portabilität der einzelnen Grafik natürlich nicht mehr gegeben.

RIB BINDING

Attribute name parameterlist

RiAttributeBegin ():

Start eines Attribut-Blocks. Alle nach **RiAttributeBegin()** gemachten Attributänderungen bleiben bis zum korrespondierenden **RiAttributEnd()** oder einer erneuten Änderung gültig. Sie werden auf dem Attribut-Stack gespeichert. Attribut-Blöcke können ineinander verschachtelt werden. Wird im *Quick RenderMan* **RiAttributBegin()** außerhalb eines Welt-Blocks verwendet, werden zusätzlich zu den Attributen auch die Optionen auf dem Attribut-Stack abgelegt.

RIB BINDING

AttributeBegin –

RiAttributeEnd ():

Ende eines Attribut-Blocks, stellt den vor dem korrespondierenden **RiAttributeBegin()** gültigen Attributzustand wieder her. In der vom *Quick RenderMan* verwendeten Interface-Spezifikation [PixQRM] werden auch alle im Attribut-Block erzeugten Handles ungültig.

RIB BINDING

AttributeEnd –

RiBasis (RtBasis ubasis, RtInt ustep, RtBasis vbasis, RtInt vstep):

Setzt die Basismatrizen und die Knotenschrittweiten für (bikubische) B-Spline-Oberflächen getrennt für die parametrischen u- und v-Richtungen. *ubasis* und *vbasis* sind Zeiger auf die Matrizen (z.B. die vorgefertigten **Ri...BASIS** Konstanten). *ustep* und *vstep* enthalten die Inkremetierungen für die Knotennummern in den beiden parametrischen Richtungen (für die vorgefertigten Basismatrizen sind das die entsprechenden **RI_...STEP** Konstanten) zum Anschluß an den nächsten nächsten 4×4 Patch.

RIB BINDING

Basis uname ustep vname vstep

Basis uname ustep vbasis vstep

Basis ubasis ustep vname vstep

Basis ubasis ustep vbasis vstep

RiBegin (RtToken name):

Start eines RenderMan Programm-Blocks. *name* enthält, falls es die Implementierung des Interfaces erlaubt, die Bezeichnung der zu verwendenden Rendering-Methode (z.B. Scanline, ZBuffer, Radiosity oder Raytracing, RIB-Code-Ausgabe). **RI_NULL** selektiert den jeweiligen Standard-Renderer. Erlaubt eine Implementierung nur einen Renderer, braucht (bei K&R-C) kein Parameter angegeben werden. Die Hauptblöcke sind nicht ineinander schachtelbar, wohl aber anreihbar. Das Ende eines Programm-Blocks wird durch **RiEnd()** gegeben. Renderer können nach dem **RiBegin()** ihre Arbeit aufnehmen oder alles zwischenspeichern und erst nach dem **RiEnd()** mit dem Rendern beginnen.

Der *Quick RenderMan* unterstützt wegen der Kontexte eine abweichende Definition von **RiBegin()**: **RtToken RiBegin(RtString handle,...)**. Eine Kontextumschaltung wird unter Verwendung des gelieferten Tokens mit **RiContext()** möglich. *handle* ist der Name für den Kontext des Programm-Blocks. In der Parameterliste können drei Token-Parameter Paare verwendet werden: `"renderer"` wählt zwischen den beiden möglichen Renderern `"draft"` (default, interaktiver Renderer) und `"archive"` (Rib-Archivierung), `"filepath"` dient dazu den Namen der Ausgabedatei (default ist `"ri.rib"` für den `"archive"` Renderer zu bestimmen, `"format"` gibt das RIB-Format `"asciifile"` (default) oder `"binaryfile"` an. Anders als in der Standard-Beschreibung kann **RiBegin()** zu jeder Zeit aufgerufen werden. Die Erzeugung neuer Kontexte ist unabhängig vom Grafik-Zustand des Renderers. Nach dem Aufruf der Funktion ist der neue Kontext

gültig. Kann ein neuer Kontext erzeugt werden, wird das Token des Kontextes geliefert, ansonsten liefert die Funktion **RI_NULL**.

RiBound (RtBound bound):

Setzen des Bounding-Box-Attributes (konvexe Hülle in Form eines Quaders). Diese gilt innerhalb des aktuellen Attribut-Blocks als Hülle für alle folgenden Objekte. Sie wird als Array von sechs Koordinaten: $[x_{min}\, x_{max}\, y_{min}\, y_{max}\, z_{min}\, z_{max}]$ im Objektkoordinatensystem gegeben und kann dem Renderer, z.B. bei der Berechnung des Detaillierungsgrades als Hilfestellung dienen. Außerhalb der Hülle liegende Oberflächen können u.U. geklippt werden.

RIB BINDING

Bound xmin xmax ymin ymax zmin zmax

Bound [xmin xmax ymin ymax zmin zmax]

RiCircle (RtFloat radius, RtFloat thetamax...):

Erweiterung von *Quick RenderMan* zum Zeichnen eines linienförmigen Kreisbogens ([PixQRM]).

$$\theta = u \cdot \theta_{max}, 0 \leq u \leq 1$$
$$x = radius \cdot \cos(\theta)$$
$$y = radius \cdot \sin(\theta)$$
$$z = 0.0$$

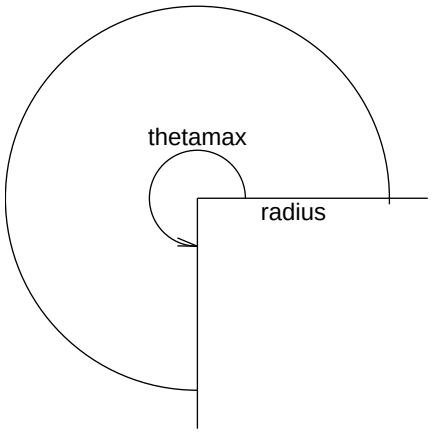

Abbildung A.1: Der Kreis

In der Parameterliste können alle für Linien gültigen Parameter stehen. Varying Parameter sind zweielementige Arrays, es wird vom Start- zum Endpunkt des Kreisbogens interpoliert, z.B. die Linienfarbe (rgb) von blau nach gelb:

```
RtFloat csary[] = {
    0.0, 0.0, 1.0,
    1.0, 1.0, 0.0
};
RiCircle(2.0, 180.0, RI_CS, csary, RI_NULL)
```

RIB BINDING

Circle radius thetamax parameterlist

RiClipping (RtFloat near, RtFloat far):

Setzen der vorderen und der hinteren Klippebene im Kamerakoordinatensystem. *near* und *far* entsprechen den Enfernungen (z-Koordinaten) der vorderen und hinteren Grenze des abzubildenden Raums. Die Werte von *near* und *far* müssen in dem Bereich $[\mathbf{RI\_EPSILON}, \mathbf{RI\_INFINITY}]$ liegen und $near \leq far$ muß gelten. **RI_EPSILON** und **RI_INFINITY** sind gleichzeitig die Vorbelegungen.

RIB BINDING

Clipping near far

RiColor (RtColor Cs):

Ersetzt das Attribut Oberflächenfarbe und kann von einem Shader benutzt werden. Das Attribut kann auch lokal für eine Definition einer Oberfläche durch den varying **RI_CS** Parameter gesetzt werden.

RIB BINDING

Color $\mathrm{c}_0\ \mathrm{c}_1\ \mathrm{c}_2$

Color $[\mathrm{c}_0\ \mathrm{c}_1\ \mathrm{c}_2]$

RiColorSamples (RtInt N, RtFloat nRGB$[\mathrm{N}]\,[3]$**, RtFloat RGBn**$[3]\,[\mathrm{N}]$**):**

Standardmäßig benutzt RenderMan einen dreidimensionalen Vektor **RtColor** zur Farbrepräsentierung (rgb, entsprechend den roten, grünen und blauen Farbanteilen bei additiver Farbmischung). Sollte es nötig werden, andere Werte zur Darstellung eines Farbwertes zu benutzen, kann die **RiColorSamples()** Routine aufgerufen werden. *N* enthält die Dimension des neuen Farbraums. Die folgenden beiden Parameter sollen die Matrizen für die Konvertierung von dem neuen N-dimensionalen Farbraum in den RGB-Raum (*nRGB*) und umgekehrt (*RGBn*) enthalten. Die Konvertierung geschieht durch Matrizenmultiplikation.

RIB BINDING

ColorSamples nRGB RGBn

Anm.: ‘nRGB’ und ‘RGBn’ sind Matrizen (in $[,]$ geschrieben)

RiConcatTransform (RtMatrix transform):

Konkateniert die 4×4 Transformationsmatrix *transform* mit der aktuellen CTM: $CTM = transform \times CTM$. Die Transformation wird so vor der der aktuellen CTM ausgeführt.

RIB BINDING

ConcatTransform transform

RiCone (RtFloat height, RtFloat radius, RtFloat thetamax...):

Plaziert eine Kegeloberfläche.

height gibt die Entfernung von der Basis zur Spitze, *radius* den Radius der Basis und *thetamax* einen mathematisch positiven Drehwinkel (um die Z-Achse) für die Erzeugung des Mantels an. Die Punkte der Oberfläche hängen wie folgt von den parametrischen (u, v) Koordinaten ab:

$$\theta = u \cdot \theta_{max}$$
$$x = radius \cdot (1 - v) \cdot cos(\theta)$$
$$y = radius \cdot (1 - v) \cdot sin(\theta)$$
$$z = v \cdot height$$

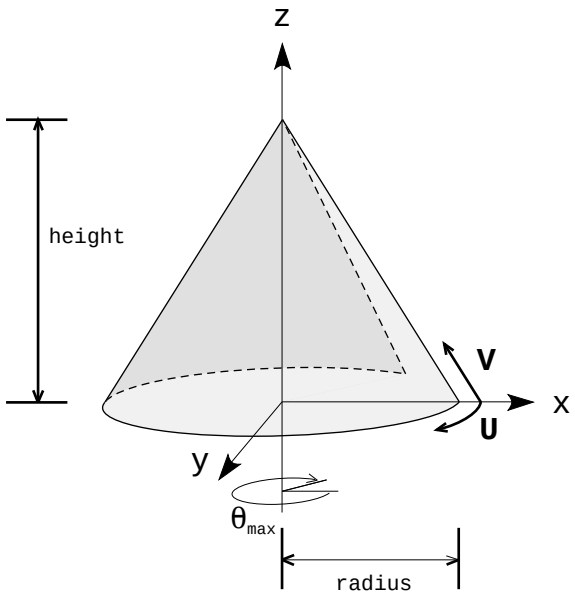

Abbildung A.2: Der Kegel, aus [PixSpec]

Als Parameter können Opazität (**RI_OS**) und Farbe (**RI_CS**) dienen.

RIB BINDING

Cone height radius thetamax parameterlist

Cone [height radius thetamax] parameterlist

RtToken RiContext (RtToken context, RtToken reserved):

Erweiterung von *Quick RenderMan* zur Kontextumschaltung. Macht einen durch **RiBegin()** erzeugten, noch gültigen Kontext *context* zum aktuellen. *reserved* steht für eine leere Parameterliste, muß also immer **RI_NULL** sein. Der Rückgabewert der Funktion ist der vorher eingesetzte aktuelle Kontext oder, falls nicht existent, **RI_NULL**.

RiCoordinateSystem (RtToken space):

Markiert das aktuelle Koordinatensystem (gegeben durch die aktuelle Transformationsmatrix) mit einem Namen *space*. Zwei verschiedene Koordinatensysteme dürfen nicht mit dem selben Namen markiert werden. Später können mittels **RiTransformPoints()** Punkte zwischen benannten Koordinatensystemen transformiert werden, um z.B. die relative Plazierung von Objekten untereinander zu erlauben. In den Erweiterungen von *Quick RenderMan* steht wegen dem zu 3.1 verschiedenen Scope-Mechanismus *space* normalerweise für das Token eines Handles, das zuvor außerhalb des Attribut-Blocks mit **RiCreateHandle()** erzeugt wurde. Im momentanen Implementierungsstand von *Quick RenderMan* sind noch keine benutzerdefinierten Koordinatensysteme zugelassen.

RIB BINDING

CoordinateSystem space

RiCreateHandle (char *handle, RtToken type):

Erweiterung von *Quick RenderMan* zur Erzeugung von Handles (s. [PixQRM]). Mit Hilfe dieser Funktion können zu einem vorgezogenen Zeitpunkt Handles für eine Datenstruktur oder externe Ressource, die in einem inneren Attribut-Block definiert werden, erzeugt werden. Es kann anschließend auch nach der Definition der Ressource im äußeren Attribut-Block auf das Handle zugegriffen werden. In der Version 3.1 des Interfaces ließ der Scopemechanismus diesen Zugriff auch ohne die Verwendung eines **RiCreateHandle()** zu. Die folgenden Interface-Funktionen, die Handles erzeugen, können vordefinierte Handles verwenden:

- **RiAreaLightSource()**, identisch zu **RiLightSource()**, erzeugt ein Handle für die Ressource `"lightsource"`
- **RiCoordinateSystem()**, erzeugt ein `"coordinatesystem"`-Handle, momentan können keine benutzerdefinierten Koordinatensysteme verwendet werden
- **RiLightSource()**, erzeugt ein `"lightsource"`-Handle
- **RiMacroBegin()**, erzeugt ein `"macro"`-Handle
- **RiObjectBegin()**, erzeugt ein `"object"`-Handle, momentan entspricht dieses Handle einem `"macro"`-Handle
- **RiResource()** kann dazu verwendet werden, um vier verschiedene Handle-Typen zu erzeugen: `"image"`, `"macro"`, `"shader"` und `"texture"`.

Im Parameter *typ* muß der Typ des Handles angegeben werden:

`"archive"`: RIB-Archive Ressource

`"coordinatesystem"`: Application-definiertes Koordinatensystem

`"image"`: Hardware Frame Buffer oder eine ‘Image Data File Resource’

`"lightsource"`: Lichtquelle oder eine ‘Area Lightsource’

`"macro"`: RenderMan Komando-Makro

`"object"`: Zusammengesetztes Objekt

`"shader"`: Programmierte ‘Shader Resource’ (`.slo`-Datei)

`"texture"`: ‘Texture Map Resource’

name ist der Name der Ressource oder ein Token eines schon erzeugten Handles. Die Rückgabe der Funktion ist das Token eines neuerzeugten vordefinierten Handles, ein Token eines schon existierenden Handles oder im Fehlerfall **RI_NULL**.

RIB BINDING

CreateHandle handle type

RiCropWindow (RtFloat left, RtFloat right, RtFloat top, RtFloat bottom):

Mit dieser Funktion kann der Ausschnitt aus dem aktuellem Bild, der gerendert werden soll, im normierten Bildkoordinatensystem (nicht die Pixelkoordinaten) festgelegt werden. X läuft von links nach rechts, Y von oben nach unten. Die Standard-Belegung ist (0.0, 1.0, 0.0, 1.0) und entspricht dem gesamten Bild (s. **RiScreenWindow()**). Durch ein Verändern der Werte wird das Seitenverhältnis der Pixel nicht verändert. Es können durch geeignete Parameterwahl Teile eines Bildes gerendert und später ‘auf Stoß’ zusammengefügt werden. Der Renderer muß sicherstellen, daß die Bilder pixelgenau aneinander passen, wenn in unterschiedlichen Teilbildern die gleichen Randkoordinaten verwendet werden.

RIB BINDING

CropWindow xmin xmax ymin ymax

CropWindow [xmin xmax ymin ymax]

RiCurve (RtToken type, RtInt nvertices, RtToken wrap...):

Erweiterung von *Quick RenderMan* zur Darstellung von Kurven. Der Parameter *type* gibt den Typ der Interpolierung der Kurve an, entweder `"linear"` (Polygon) oder `"cubic"` (kubischer Spline). In *nvertices* wird die Anzahl der Kontrollpunkte der stückweise definierten Kurve[2] übergeben. Mit *wrap* kann bestimmt werden ob die Kurve geschloßen wird (periodisch verläuft, `"periodic"`) oder nicht (`"nonperiodic"`). Die Parameterliste muß mindestens eine Positionsangabe: (`"P"`,`"Pw"`, oder `"Pz"`) beinhalten. `"cubic"` Kurven verwenden die Basismatrizen und die *ustep* und *vstep* Schrittgrößen der mit **RiBasis()** gemachten Angaben.

[2] Der Endpunkt eines Kurvensegments ist der Startpunkt des folgenden.

RIB BINDING

Curve type wrap parameterlist

RiCylinder (RtFloat radius, RtFloat zmin, RtFloat zmax, RtFloat thetamax...):

Plazieren einer Zylinderoberfläche mit dem Radius *radius*. *zmin* und *zmax* geben die Ausdehnung in der Höhe (z-Achse) an, *thetamax* ist der (mathematisch positive) Drehwinkel mit dem der Mantel geschlossen wird. **RI_OS** und **RI_CS** können als Parameter verwendet werden. Die Mantelpunkte können aus den parametrischen Koordinaten gewonnen werden:

$$\theta = u \cdot \theta_{max}$$
$$x = radius \cdot \cos(\theta)$$
$$y = radius \cdot \sin(\theta)$$
$$z = zmin + v \cdot (zmax - zmin)$$

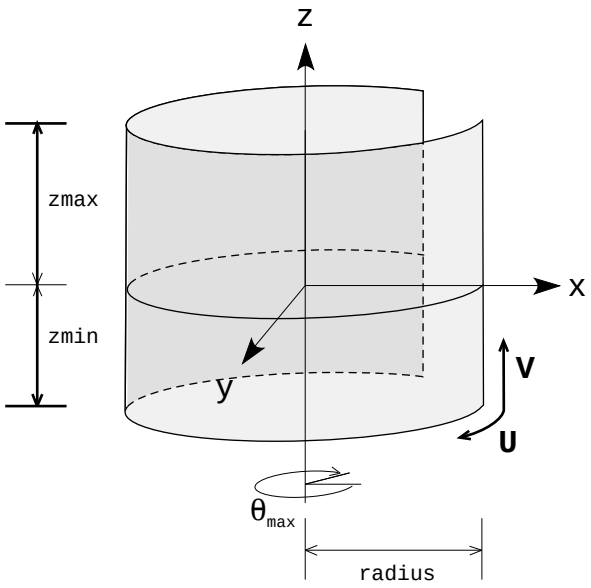

Abbildung A.3: Der Zylinder, aus [PixSpec]

RIB BINDING

Cylinder radius zmin zmax thetmax parameterlist

Cylinder [radius zmin zmax thetmax] parameterlist

RtToken RiDeclare (char *name, char *declaration):

Deklariert im aktuellen Attribut-Block Name und Typ einer Variable für die Parameterliste und die Verwendung in Shadern. Shader können auf die in den Parameterlisten für eine Oberfläche angegebenen benutzerdefinierten Variablen durch die Instanzvariablen (Shaderparameter) zugreifen. *name* ist der Bezeichner und *declaration* der Typ. Als Typ sind **float**, **integer**, **point**, **color** und **string** erlaubt. Sie können durch die Speicherklasse `varying` (Array von Werten, die den jeweiligen Punktknoten zugeordnent werden, die Werte werden beim Schattieren über die dazwischenliegenden Oberflächenteile interpoliert) oder durch die Standard-Speicherklasse `uniform` (bleiben während des gesamten Schattierungsvorgangs über eine Oberfläche konstant) gegeben werden, z.B '`varying int`', '`int`', '`varying point`' oder '`uniform color`'. Nach dem Typ kann noch eine Feldgröße *n* in eckigen Klammern folgen, z.B.: `varying float[4]`. Ein Wert besteht in diesem Fall aus einer Aneinanderreihung von *n* einzelnen Werten. Die Syntax der Deklaration ist:

```
[class] type ['['n']']
```

Rückgabewert der Funktion ist das Token der Variable oder **RI_NULL** im Fehlerfall.

Neben den durch den Benutzer definierten Parametern existieren schon eine Reihe von vordefinierten Parametern, z.B. ist die Normale der Oberfläche '`N`' als '`varying point`' gegeben, d.h. zu jedem Knoten eines Polygon-Netzes soll eine Normale existieren. Namen von schon deklarierten Variablen dürfen nicht mehr verwendet werden (s.a. die **RI_...** Token).

Information	Name	Klasse	Typ	Anzahl	Token
Position	`"P"`	varying	**RtPoint**	1	**RI_P**
	`"Pz"`	varying	**RtFloat**	1	**RI_PZ**
	`"Pw"`	varying	**RtFloat**	4	**RI_PW**
Normale	`"N"`	varying	**RtPoint**	1	**RI_N**
	`"Np"`	uniform	**RtPoint**	1	**RI_NP**
Farbe	`"Cs"`	varying	**RtColor**	1	**RI_CS**
Opazität	`"Os"`	varying	**RtColor**	1	**RI_OS**
Textur	`"s"`	varying	**RtFloat**	1	**RI_S**
Koord.	`"t"`	varying	**RtFloat**	1	**RI_T**
	`"st"`	varying	**RtFloat**	2	**RI_ST**

In den Parameterlisten wird hinter dem Token nur ein Zeiger auf einzelne Werte (bei uniform, entspr. einem einelementigen Array) bzw. Arrays (bei varying, Anzahl der Spalten entsprechend des Elementtyps, z.B. 3 Floats bei Punkten, Anzahl der Zeilen = Anzahl der Knoten einer Oberfläche) von Werten angegeben. In den Arrays stehen wie in C die Spalten nebeneinander.

Quick RenderMan erweitert die Syntax der Namen, die in unterschiedlichen Tabellen unterschiedlicher Tabellen-Namespaces gehalten werden können:

```
[[namespace:]table:]var
```

Es existieren zwölf Tabellen-Namespaces, die den sechs Shader-Typen und den sechs Typen der implementierungsabhängigen Interface-Erweiterungen entsprechen:

- `"light"`
- `"surface"`
- `"volume"`
- `"imager"`
- `"displacement"`
- `"transformation"`
- `"attribute"`
- `"option"`
- `"geometricapproximation"`
- `"geometry"`
- `"hider"`
- `"resource"`

Die Angabe der Namespaces ist, falls die Tabelle eindeutig angegeben werden kann, optional. Wird kein Tabellenname angegeben, wird eine neue Variable in einer globalen Tabelle gespeichert.

RIB BINDING

Declare name declaration

RiDeformation (char *name...):

Setzen des aktuellen Deformations-Shaders (Transformations-Shader) mit dem Namen *name*. Er ermöglicht, unterschiedliche Transformationen an den Oberfläche durchzuführen. Die Parameterliste wird für die Belegung der Instanzvariablen verwendet. *Quick RenderMan* unterstützt keine Deformations-Shader, es wird bei der Verwendung eines solchen Shaders immer die Einheitsmatrix an die aktuelle CTM konkateniert; der Shader wird ignoriert.

RIB BINDING

Deformation name parameterlist

RiDepthOfField (RtFloat fstop, RtFloat focallength, RtFloat focaldistance):

Option zur Einstellung der Tiefenschärfe der Kamera. Der Fokus liegt in der Entfernung *focaldistance* von der Linse. Objekte, die mehr als *fstop* Einheiten vom Fokus entfernt sind, erscheinen je nach Brennweite *focallength* verschwommen. Die Einheit von *focallength* ist die gleiche wie die von *fstop* und entspricht der Einheit der Entfernung von der

Kamera entlang der z-Achse. Der Linsendurchmesser ergibt sich aus $focallength/fstop$. Für die Standard-Einstellung *fstop* gleich **RI_INFINITY**, wird eine Nadellochkamera emuliert und die Oberflächen überall scharf dargestellt. Die Tiefenschärfe-Option ist also ausgeschaltet. Der Wert für *focallength* sollte > 0 und < 1. Für Werte < 0 wird die 'Depth Of Field' Option abgeschaltet. Werte ≥ 1 liefern keine sinnvollen Ausgaben mehr und die Rechenzeiten werden unerträglich.

Ist die Tiefenschärfe eingestellt, wird ein Punkt im Objektbereich nicht als Punkt im Bildbereich sondern eher als Kreis (circle of confusion) mit einem Durchmesser *C* dargestellt [PixSpec]:

$$C = \frac{focallength}{fstop} \cdot \frac{focaldistance \cdot focallength}{focaldistance - focallength} \cdot \left| \frac{1}{depth} - \frac{1}{focaldistance} \right|$$

RIB BINDING

DepthOfField fstop focallength focaldistance

DepthOfField –

Der zweite Befehl spezifiziert eine Nadellochkamera.

RiDetail (RtBound bound):

Hülle eines Objekts in Objektkoordinaten, die zur Berechnung des Detaillierungsgrades, die Pixelfläche, die der projizierte Quader auf dem Bildschirm einnimmt, herangezogen wird. Die Hüllenangabe ist ein Attribut und gilt für alle im gleichen Attribut-Block folgenden Primitive.

RIB BINDING

Detail minx maxx miny maxy minz maxz

Detail [minx maxx miny maxy minz maxz]

RiDetailRange (RtFloat offlow, RtFloat onlow, RtFloat onhigh, RtFloat offhigh):

Durch die Parameter werden vier Werte des Detaillierungsgrad gegeben, die die Kurve der 'relativen Wichtigkeit' ('nie vorhanden':0 bis 'immer vorhanden':1) eines Objektmodells bestimmen. Die Kurve steigt linear im Bereich *offlow* und *onlow* von 0 bis 1, bleibt zwischen *onlow* und *onhigh* konstant auf 1 und fällt von *onhigh* bis *offhigh* linear zurück auf 0. Für verschiedene Detaillierungsgrade können so die Wichtigkeiten eines Modelles definiert werden. In den Fuzzy-Bereichen (die Kurven überschneiden sich) kann der Renderer auch zwischen den Modellen interpolieren. Werte von 0 bis **RI_INFINITY** sind erlaubt. $onhigh_i = offlow_{i+1}$ und $offhigh_i = onlow_{i+1}$ muß nicht unbedingt gelten; die Kurven geben keine Auswahlwahrscheinlichkeiten an. Da ein bestimmtes Modell im Bereich zwischen $onlow_i$ und $onhigh_i$ immer sichtbar ist, dürfen sich die Kurven in diesem Bereich nicht überschneiden, auch sollte für jeden Detaillierungsgrad (0 - **RI_INFINITY**) mindestens ein Modell sichtbar sein. **RiDetailRange()** ist ein Attribut. Alle bis zum nächsten **RiDetailRange()** oder Blockende folgenden Interface-Aufrufe zählen zum Modell.

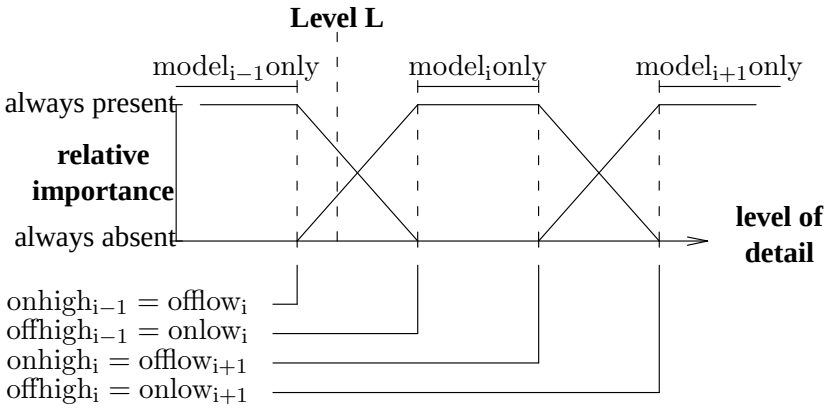

Abbildung A.4: Detaillierungsgrad, aus [Upstill89]

Renderer, die den Detaillierungsgrad nicht einbeziehen, stellen immer das Modell mit dem höchsten $onhigh$ Wert, d.h. das komplexeste, dar. Benutzt der Renderer eine Interpolierung kann **RiDetailRange()** z.B. auch zur Darstellung für den Morphing Spezialeffekt verwendet werden. Wenn in aufeinanderfolgenden Frames der **RiRelativeDetail()** mit der Framenummer skaliert wird, können beim Rendern unterschiedliche Modelle aus der Beschreibung verwendet und interpoliert werden. Die Frames können anschließend zu einer Animation zusammengestellt werden.

RIB BINDING

DetailRange offlow onlow onhigh offhigh

DetailRange [offlow onlow onhigh offhigh]

RiDisk (RtFloat height, RtFloat radius, RtFloat thetamax...):

Erzeugen eines Diskus-Objekts (deformierter, oben und unten geschlossener Zylinder mit der Höhe Null) mit Radius *radius* und math. positiven Schließwinkel *thetamax*. Die Oberflächenpunkte können durch die folgenden Gleichungen aus den parametrischen Koordinaten gewonnen werden:

$$\theta = u \cdot \theta_{max}$$
$$x = (1 - v) \cdot radius \cdot \cos(\theta)$$
$$y = (1 - v) \cdot radius \cdot \sin(\theta)$$
$$z = height$$

RIB BINDING

Disk height radius thetamax parameterlist

Disk [height radius thetamax] parameterlist

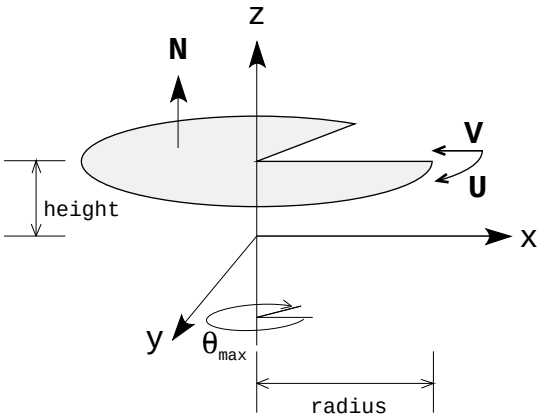

Abbildung A.5: Die Scheibe, aus [PixSpec]

RiDisplacement (char *name...):

Setzt den 'Displacement'-Shader *name* als Attribut und übergibt ihm die Parameter aus der Liste als Belegung für die Instanzvariablen. 'Displacement'-Shader können die Normale und die Höhen von Punkten einer Oberfläche vor dem Schattieren verändern. *Quick RenderMan* implementiert keine 'Displacement'-Shader. Als Name kann das (vordefinierte) Token eines Shaders verwendet werden (s. **RiResource()**, **RiCreateHandle()**). 'bumpy' ist der einzige Standard 'Displacement'-Shader.

RIB BINDING

Displacement name parameterlist

RiDisplay (char *name, RtToken type, RtToken mode...):

Setzt das Anzeigegerät, z.B. Bildschirm (bzw. das Frambuffer-Device, falls der Parameter *type* den Wert **RI_FRAMEBUFFER** hat) oder eine Datei (*type* ist **RI_FILE**). *name* ist der Bezeichner des logischen Geräts. *Quick RenderMan* verwendet Handles die durch **RiResource()** spezifiziert wurden. *mode* gibt mit Kombinationen aus den Strings `"rgbz"`, `"a"` (Alpha) oder `"z"` (Höhen) die Art des Bitmaps an. Der Parameter wird vom *Quick RenderMan* ignoriert, er setzt immer `"rgba"`, mit dem festen Wert 1.0 für `"a"`, als Ausgabeformat ein. In der Parameterliste können gerätespezifische Optionen übergeben werden.

RIB BINDING

Display name type mode parameterlist

RiEnd ():

Ende eines RenderMan Programm-Blocks, der mit **RiBegin()** eingeleitet wurde. Nach dem Beenden dieser Funktion sind alle Bilder des Programm-Blocks fertig gerendert.

Wird mit Kontexten gearbeitet, ist nach einem **RiEnd()** kein Kontext mehr gültig. Ein noch existierender Kontext muß explizit mit **RiContext()** eingesetzt werden.

RiErrorHandler (RtFunc handler):

Fügt die Funktion, auf die *handler* zeigt, als Fehlerbehandlungsroutine ein. Sie wird bei jedem Fehler, der beim Rendern auftritt, aufgerufen. Es stehen schon drei Handler zur Fehlerbehandlung zur Verfügung. Zum einem **RiErrorPrint()**, gleichzeitig der Default-Handler[3], welcher den Fehler auf die Standard-Ausgabe schreibt, **RiErrorAbort()** schreibt Fehlermeldung und beendet das Programm falls der Fehler ernst genug ist (severity **RIE_ABORT**) und **RiErrorIgnore()**, welcher den Fehler einfach ignoriert. Der *Quick RenderMan* besitzt zusätzlich den **RiErrorDefault()** Handler, der **RiErrorAbort()** entspricht, das Programm aber ab der **RIE_SEVERE** Fehlerstufe abbricht. Soll ein eigener Handler installiert werden, muß dieser wie folgt deklariert sein:

```
void MyHandler(RtInt code, RtInt severity, char *message)
```

Für den *Quick RenderMan*:

```
void MyHandler(RtInt code, RtInt severity, char *message,
               RtToken routine, RtToken context)
```

die Parameter besitzen bei einem Aufruf folgende Belegungen:

code: Der Fehlercode, symbolische **RIE_...** Konstanten sind in der Datei `ri.h` definiert

severity: Fehlerstufe, symbolische **RIE_...** Konstanten sind in `ri.h` definiert

message: Text, der den Fehler beschreibt

routine: Beschreibt das Unterprogramm vom RenderMan in dem der Fehler aufgetreten ist.

context: Kontext, in dem der Fehler aufgetreten ist.

Die Fehlerbehandlungsroutine kann z.B. Fehlercode und -stufe in globalen Variablen speicheren um dem aktuellen Rendering-Prozeß (an markanten Stellen müssen die Variablen dann abgefragt werden) zu überlassen, was passieren soll und/oder die Standard-Handler aufrufen.

RIB BINDING

ErrorHandler ”ignore”

ErrorHandler ”print”

ErrorHandler ”abort”

ErrorHandler ”default” (nur *Quick RenderMan*)

[3] Beim *Quick RenderMan* ist es der Handler **RiErrorDefault()**.

RiExposure (RtFloat gain, RtFloat gamma):

Korrektur der Farbkomponenten eines Pixels. Je nach Ausgabegerät und Rezeptor (z.B. menschl. Auge, Film) kann der Farbwert eines Pixels unterschiedlich wahrgenommen werden. Um diesen Effekt auszugleichen, existiert diese Funktion. Die Intensität (I) eines Pixels, das auf den Phosphor eines handelsüblichen CRT-Monitors erscheint, ist nicht-linear von der einwirkenden Spannung (V) abhängig, c.a.: $I = V^{\gamma}$. Die Konstante γ ist von Monitortyp zu Monitortyp verschieden. Mit der Funktion **RiExposure()** kann nun die Intensität und der nicht-linar wahrgenommene Kontrast zwischen Pixeln an Ausgabeeinheit und Rezeptor angepaßt werden:

$color_{output} = (color_{input} \times \ gain)^{1/\gamma}$

$gain$ und γ sind standardmäßig mit 1.0 vorbelegt, haben also keinen Einfluß.

Exakte Farbreproduktion ist, wenn überhaupt, nur sehr schwer zu erreichen ([FolVDFH90] Chromatic Color, [Rogers85], [RogE90] Maureen C. Stone *Color Printing for Computer Graphics*, [Lind89]), vor allem jedoch nicht nur durch die Modifikation der Intensität. Aufgrund dessen können hierfür spezielle Shader (sog. 'Imager'-Shader) eingesetzt werden, die nach der Berechnung eines Pixels mit dessen Farbwert aufgerufen werden und diesen nach einem durch den Benutzer bestimmten Algorithmus verändern können.

RIB BINDING

Exposure gain gamma

RiExterior (char *name...):

Setzen des 'Exterior'-Shaders, des Shaders, der das Volumen zwischen Objekt und Lichtquelle darstellt. Das Licht, das auf eine Oberfläche trifft, kann so vor der Berechnung der Farbe der Oberfläche verändert werden. Neben diesem Shader existieren noch zwei andere Volumen-Shader: der 'Interior'-Shader für das Licht, das durch einen Körper strahlt und der Atmosphären-Shader für eine Veränderung des Lichts in dem Volumen zwischen Objekt und Kamera (z.B. für atmosphärische Effekte wie Nebel). Volumen-Shader sind Bestandteil des aktuellen Kontextes. Es kann jeweils nur ein Volumen-Shader pro Oberfläche (Volumen) gesetzt werden, wohl aber unterschiedliche Shader für unterschiedliche Oberflächen. Im Gegensatz zu dem Oberflächen-Shader braucht kein Volumen-Shader gesetzt sein. Volumen-Shader modellieren die Lichtveränderung auf dem Weg durch ein Volumen, während Oberflächen-Shader die Reflexions- und Refraktionseigenschaften einer Oberfläche beschreiben. Da mit dem RenderMan Interface keine Körper sondern nur Oberflächen beschrieben werden können, werden die 'Interior'-Shader an die entsprechenden Oberflächen gebunden.

Volumen-Shader bekommen allgemein Intensität und Farbe des Lichts, sowie die Richtung aus der es in das Volumen einfällt und liefern Intensität und Farbe des ausfallenden Lichtes.

Bis auf das Einbeziehen von Atmosphäre (Atmosphäre-Shader) besitzen Volumen-Shader in der Erzeugung realistisch wirkender Computergrafiken noch wenig Bedeutung. Einsatz-

gebiet sind z.B. Aufbereitungen von Kernspinresonanz- und Computer-Tomographien in der Medizin ([DreCH88], [Meinzer93]).

RIB BINDING

Exterior name parameterlist

RiFormat (RtInt xresolution, RtInt yresolution, RtFloat pixelaspectratio):

Mit dieser Funktion kann das Format des Ausgabe-Pixel-Rechtecks (s.a. 2.5) bestimmt werden. Anzahl der horizontalen (*xresulution*) und vertikalen (*yresulution*) Pixel und deren Seitenverhältnis x/y *pixelaspectratio* (normalerweise 1). Die Optionen die hier gesetzt werden sind gerätespezifisch und sollten nur wenn unbedingt nötig geändert werden. Die Standard-Belegungen sind auf das aktuelle Ausgabegerät abgestimmt.

RIB BINDING

Format xresolution yresolution pixelaspectratio

RiFrameAspectRatio (RtFloat frameaspectratio):

Der *frameaspectratio* ist das Verhältnis von Breite zur Höhe des Ausgabebildes. Die Ausgabe wird mit der linken oberen Ecke ausgerichtet. Wird die Prozedur nicht aufgerufen wird dem 'Aspectratio' der Defaultwert, der aus der Pixelauflösung und dem Pixel-Aspectratio gewonnen wird zugewiesen: $(xresolution \cdot pixelaspectratio)/yresolution$

RIB BINDING

FrameAspectRatio frameaspectratio

RiFrameBegin (RtInt number):

Start eines 'Frame-Blocks' für Animationssequenzen. **RiFrameEnd()** beendet einen solchen Block. Frames dürfen nicht ineinander geschachtelt werden. Ein Programm-Block darf mehrere Frames beinhalten, Frames mehrere Weltblöcke (z.B. zur Generierung von Shadow und 'Environment Maps'). Als *number* wird die Platznummer in der sequentiellen Abfolge übergeben. Alle Optionen sind innerhalb eines Frames lokal.

RIB BINDING

FrameBegin –

FrameBegin int

RiFrameEnd ():

Das Ende eines Frame-Blocks, der mit **RiBeginFrame()** eingeleitet wurde. Die Optionen und Attribute, die innerhalb des Blocks verändert wurden, werden auf ihre zuvor gültigen Belegungen zurückgesetzt. Lichtquellen und Objekte, die im Block definiert wurden, werden ungültig.

RIB BINDING

FrameEnd –

RiGeneralPolygon (RtInt nloops, RtInt nverts[]...):

RiGeneralPolygon() erlaubt die Spezifikation eines allgemeinen Polygons, d.h. planar, konkav und mit Löchern. Die Funktion **RiPolygon()** kann dazu verwendet werden, ein konvexes Polygon effizienter zu rendern. Das erste Polygon gibt die Flächenumrandung wieder. Alle folgenden Polygone dienen zur Angabe von Löchern, die aus dem ersten Polygon 'herausgestanzt' werden sollen. Als Parameter *nloops* wird die Anzahl der Polygone übergeben. *nverts* ist ein Feld der Größe *nloops*, dessen Elemente die Anzahl der Punkte, aus dem ein Polygon besteht, beinhalten. Als Parameter folgen Knotenliste **RI_P** und optional die Normalenendpunkte **RI_N**. Die Normale berechnet sich in diesem Fall aus $Normalenendpunkt - Knotenpunkt$. Die Normalen können zur Schattierung herangezogen werden. Die Anzahl der Knoten ergibt sich aus: $\sum_{i=0}^{nloops-1} nverts[i]$. Falls eine Oberfläche aus mehreren Polygonen modelliert werden soll (Polyhedren), müssen mit mehreren **RiGeneralPolygon()** Aufrufen die Endpunkte und Normalen von Kanten mehrmals (je nachdem wieviel Kanten in einem Knoten zusammenstoßen) angegeben werden. Um Speicher und Rechenzeit zu sparen, können solche Oberflächen durch gesonderte Interface-Routinen (**RiPointsPolygons()**, **RiGeneralPointsPolygons()**) spezifiziert werden. Wieder andere Routinen erlauben das Interpolieren und Approximieren von Kurven und gebogenen Oberflächen durch B-Splines (**RiPatch()**, **RiPatchMesh()**, **RiNuPatch()**).

RIB BINDING

GeneralPolygon nvertices parameterlist

RiGeometricApproximation (RtToken type, RtFloat value):

Funktion der 3.1 Version. Je nach Entwicklungsprozeß einer Grafik ist es nötig, diese grob und schnell oder qualitativ hochwertig, dafür aber langsam zu rendern. Mit Hilfe der **RiGeometricApproximation()** Routine kann dem Renderer mitgeteilt werden durch welche, einfacher zu rendernde Oberflächen ein Oberflächenprimitiv approximiert werden kann. Voreingestellt ist der Approximationstyp *type* **RI_FLATNESS** mit *value* 0.5. *value* gibt hierbei die maximale Abweichung einer Polygon-Approximation von einer gebogenen Originaloberfläche in Pixeln an. Eine Polygon-Oberfläche wird mit dieser Einstellung maximal bis zu einem halben Pixel von einer gebogenen Originaloberfläche abweichen. Die Anzahl der Polygone, in die eine Oberfläche aufgeteilt wird, ist auf diese Weise an ihre Krümmung angepaßt. Flachere Oberflächen können durch weniger Polygone approximiert werden als stärker gebogene. Je nach Implementierung können noch andere Approximationstypen existieren, die es dem Anwender ermöglichen, Einfluß auf die Qualität zu nehmen.

RIB BINDING

GeometricApproximation "flatness" value

GeometricApproximation type value (nur 3.1)

RiGeometricApproximation (RtToken type...):

Der *Quick RenderMan* beinhaltet nur noch aus Kompatibilitätsgründen den **RI_FLATNESS** Approximationstyp. Es werden stattdessen die Typen `"tesselation"` und

die noch nicht implementierte `"deviation"` verwendet. `"tesselation"` benötigt den `uniform float[2]` Parameter `"parametric"`. In ihm wird die polygonale Aufteilung einer Oberfläche in den beiden parametrischen Richtungen definiert. `"deviation"` verwendet den `uniform float` Parameter `"raster"`, der ähnlich wie **RI_FLATNESS** den erlaubten Abstand der approximierten Oberfläche vom Original in Pixeln angibt.

RIB BINDING

GeometricApproximation type parameterlist

RiGeometricRepresentation (RtToken type):

Für *type* sind bei der NeXT Implementierung folgende Werte möglich:

'points': Die Oberflächen werden nur als Stützpunkte repräsentiert

'lines': Die Oberflächen werden als Liniengitter repräsentiert

'primitive': Die Oberflächen werden als (gekrümmte) Flächen repräsentiert

Der Renderer versucht den angegebenen Repräsentationstyp immer mit der ihm möglichen höchsten Stufe zu erreichen.

RIB BINDING

GeometricRepresentation type

RiGeometry (RtToken type...):

Definiert ein implementationspezifisches Primitiv. *Quick RenderMan* unterstützt das Primitiv `"teapot"` mit der leeren Parameterliste **RI_NULL**. Die Parameterliste hängt vom Primitiv und der Implementation ab.

RIB BINDING

Geometry type parameterlist

RiHider (RtToken type...):

Option für die eine Auswahl des Algorithmus, der für die Hidden-Surface Eliminierung verwendet werden soll. Die Interface-Beschreibung schreibt nur drei Standard-Typen vor: **RI_HIDDEN**, **RI_PAINT** und **RI_NULL**. Je nach Implementierung können weitere Typen unterstützt werden, die zusätzlich noch mit Parametern aus der Parameterliste versorgt werden können. **RI_HIDDEN** ist der Default-Typ und besagt, daß der Renderer seinen Standard-Algorithmus zur Eliminierung verdeckter Flächen verwenden soll. **RI_PAINT** besagt, daß alle Oberflächen in der Reihenfolge ihrer Erzeugung dargestellt werden sollen. Später definierte können eher definierte Flächen überdecken. **RI_NULL** besagt, daß überhaupt keine Ausgabe stattfinden soll. Letztere Option kann zur Fehlersuche im RenderMan Programm verwendet werden. *Quick RenderMan* unterstützt zusätzlich die beiden Hider `"sketch"` und `"pick"`. `"sketch"` arbeit ähnlich wie **RI_HIDDEN** (`"hidden"`), statt auf Pixelebene aber auf Polygonebene. Er ist deshalb weniger genauer

aber schneller als `"hidden"`. `"pick"` erlaubt das ‘Picken’ von einzelnen Objekten mit einer minimalen z-Koordinate in dem aktuellen Crop-Bereich, es wird keine Grafik ausgegeben.

RIB BINDING

Hider type parameterlist

RiHyperboloid (RtPoint point1, RtPoint point2, RtFloat thetamax...):

Erzeugung eines Hyperboloids als Drehköper. *point1* und *point2* geben eine Linie an, die mit dem Winkel *thetamax* um die z-Achse des aktuellen Koordinatensystems gedreht werden. Auf diese Weise kann man Teile des Mantels eines Hyperboloids und seine Grenzfälle Zylinder, Kegel und Diskus erzeugen. Es können die **RI_OS** und **RI_CS** Parameter verwendet werden. Der Hyperboloid ist durch die folgenden Gleichungen definiert:

$$\theta = u \cdot \theta_{max}$$
$$x_r = (1 - v)x_1 + v \cdot x_2$$
$$y_r = (1 - v)y_1 + v \cdot y_2$$
$$z_r = (1 - v)z_1 + v \cdot z_2$$
$$x = x_r \cdot \cos(\theta) - y_r \cdot \sin(\theta)$$
$$y = x_r \cdot \sin(\theta) - y_r \cdot \cos(\theta)$$
$$z = z_r$$

Die Linienendpunkte seien hierzu:

$$point1 = (x_1, y_1, y_2), point2 = (x_2, y_2, z_2)$$

RIB BINDING

Hyperboloid $\mathrm{x_1\ y_1\ z_1\ x_2\ y_2\ z_2}$ thetamax parameterlist

Hyperboloid $[\mathrm{x_1\ y_1\ z_1\ x_2\ y_2\ z_2\ thetamax}]$ parameterlist

RiIdentity ():

Löscht die aktuelle Transformation im aktuellen Transformations- oder Attribut-Block und stellt so die Weltkoordinaten als Objektkoordinaten wieder her.

RIB BINDING

Identity –

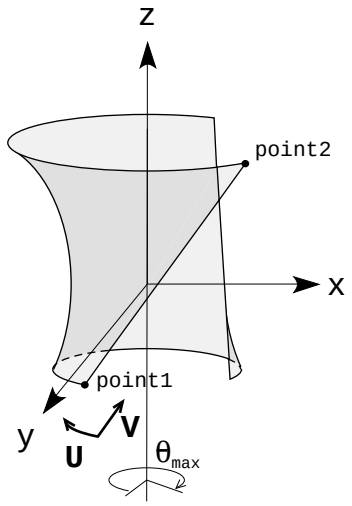

Abbildung A.6: Der Hyperboloid, aus [PixSpec]

RiIlluminate (RtLightHandle light, RtBoolean onoff):

Dient dazu, Lichtquellen ein- oder auszuschalten. Die Funktion existiert, weil Lichtquellen, sobald sie erzeugt wurden, alle folgenden Objekte beleuchten. Der Status einer Lichtquelle ist Teil des Attribut-Blocks. Das **RtLightHandle** *light* wird bei der Erzeugung einer Lichtquelle mit **RiLightSource()** oder **RiAreaLightSource()** geliefert. Ist *onoff* mit dem Wert **RI_FALSE** belegt, wird die Lichtquelle ausgeschaltet, mit dem Wert **RI_TRUE** wird sie eingeschaltet. Die Handels werden nach dem Ende des sie umgebenden Welt- oder Frame-Blocks ungültig (beim *Quick RenderMan* schon nach dem Attribut-Block, indem die Lichtquelle definiert wurde).

RIB BINDING

Illuminate lightsourcenumber onoff

RiImager (char *name...):

Setzt den ‘Imager’-Shader als Option. Der ‘Imager’-Shader kann dazu verwendet werden, Farbtransformationen an schon berechneten Bildpunkten nach der Einfärbung aber vor der Quantisierung, der Abbildung von Bildpunkten auf die Pixel eines Ausgabegeräts, durchzuführen. Imager erlauben eine allgemeinere Farbreproduktion (siehe auch [Lind89]) als die durch **RiExposure()**. Der Shader-Typ wird vom *Quick RenderMan* nicht unterstützt.

RIB BINDING

Imager name parameterlist

RiInterior (char *name...):

Setzt den Volumen-Shader für das Innere von Körpern als Attribut. Der Volumen-Shader wird immer dann aufgerufen, wenn Licht auf dem Weg zur Kamera durch einen Körper (bzw. die Oberfläche, die ihn repräsentiert) fällt (s.a. **RiExterior()**). Der Shader verändert die Farbe des Lichts. Der Shader-Typ wird vom *Quick RenderMan* nicht unterstützt.

RIB BINDING

Interior name parameterlist

RtLightHandle RiLightSource (char *name...):

Um eine Lichtquelle in eine Szene zu plazieren, wird ein 'Lightsource'-Shader verwendet. Es können mit Hilfe dieser Shader theoretisch beliebig viele Lichtquellen in eine Szene eingeführt werden. Die Anzahl der tatsächlich möglichen Lichtquellen kann aber durch eine Implementierung begrenzt sein. Im Gegensatz zu anderen Shadern bleiben Lichtquellen ab ihrer Einfügung im gesamten Programm-Block erhalten und können nicht durch andere Lichtquellen ersetzt werden (trifft nicht auf den *Quick RenderMan* zu). Sie werden alle in einer Liste aufbewahrt. Es ist so möglich, mehrere Lichtquellen zu plazieren. Mit **RiIlluminate()** können die Lichtquellen einzeln ein- oder ausgeschaltet werden. Die Lichtquelle selbst ist (da sie ein Shader ist) nicht sichtbar. Mit einer anderen Interface-Funktion ist es möglich, einer Lichtquelle die Geometrie einer Oberfläche zu geben (s. **RiAreaLightSource()**).

Lichtquellen existieren im Gegensatz zu den anderen Shadern nach Beendigung des Attribut-Blocks (gilt nicht für *Quick RenderMan*), in dem sie definiert wurden weiter, nicht aber ihr Zustand. Durch die Defaultannahme, daß alle nicht definierten Lichtquellen ausgeschaltet sind, werden die in einem Attribut-Block definierten Lichtquellen nach dem Ende des Blocks abgeschaltet. Sie können aber jederzeit wieder eingeschaltet werden. Nach ihrer Definition sind Lichtquellen standardmäßig eingeschaltet. Die Shaderinstanzen und Handels werden erst nach dem Ende des sie umschließenden Frame- oder Welt-Blocks undefiniert. Lichtquellen, die außerhalb des Welt-Blocks (bzw. Frame-Blocks) definiert wurden, bleiben auch nach dessen Ende bestehen. Für den *Quick RenderMan* müssen zusätzlich vordefinierte Handles oder Ressource-Handles verwendet werden (s. **RiCreateHandle()**, **RiResource()**)

Aufgabe des Lichtquellen-Shaders ist es, Intensität und Farbe des Lichtes zu berechnen, daß von einer Lichtquelle auf einen Punkt einer Oberfläche fällt. Es gibt in der Interface-Beschreibung schon vier vordefinierte Lichtquellen-Shader: Ambiente, parallelstrahlige und punktförmige Lichtquellen sowie Scheinwerfer.

`"ambientlight"`: Das ambiente Licht bescheint alle Oberflächen unabhängig von ihrer Position und Ausrichtung mit gleicher Intensität und Farbe. Als Parameter können 'intensity' **RtFloat** und 'lightcolor' **RtColor** dienen. Falls nicht anders angegeben, besitzt die Intensität den Wert 1 und die Farbe ist weiß (alle Komponenten des Farbvektors sind 1).

Ambientes Licht entspricht annäherungsweise einem sehr stark gestreuten (diffusen) Licht. Durch das Einfügen einer schwachen ambienten Lichtquelle wird vermieden, daß Teile eines Körpers überhaupt nicht beleuchtet werden.

`"distantlight"`: Die Lichtstrahlen einer sehr weit entfernten Lichtquelle (etwa der Sonne) treffen nahezu parallel aus einer bestimmten Richtung ein. Neben der Intensität und Farbe (wie beim ambienten Licht) kann deshalb eine Richtung der Lichtstrahlen durch die Punkte **RtPoint** der Parameter 'from' (Default ist (0,0,0) im Welt-Koordinatensystem) und 'to' (Default ist (0,0,1) im Welt-Koordinatensystem) angegeben werden (Richtung: to–from). 'from' ist nicht die Position der Lichtquelle sondern dient nur dazu, die Strahlenrichtung festzulegen.

Die Helligkeit eines durch eine solche Lichtquelle beschienenen Punktes einer diffusen Oberfläche ist abhängig von dem Winkel zwischen der Normalen in diesem Punkt und dem Lichtstrahl. Ist die Oberfläche in diesem Punkt der Lichtquelle direkt zugewandt, besitzt er maximale Helligkeit, ist er der Lichtquelle abgewandt, wird er überhaupt nicht beleuchtet.

`"point"`: Eine punktförmige Lichtquelle zeichnet sich dadurch aus, daß das von ihr in alle Richtungen gleichmäßig ausgesandte Licht quadratisch zur Entfernung von der Lichtquelle abnimmt. Die Lichtstrahlen besitzen nicht alle die gleiche Richtung. Sie zeigen auf das Zentrum der Lichtquelle, den Punkt *from* (Default ist (0,0,0) im Kamerakoordinatensystem). Neben der Position können noch die Intensität und die Farbe des Lichts geändert werden. Da das Licht gerichtet ist, ist die Helligkeit des bestrahlten Körpers wie bei einer Lichtquelle mit parallelen Strahlen abhängig von seiner Ausrichtung zur Lichtquelle.

`"spotlight"`: Die komplexeste Standard-Lichtquelle stellt der Scheinwerfer dar. Seine Lichtstrahlen werden von der Lichtquelle kegelförmig ausgestrahlt. Die Position der Lichtquelle (*from*) und die Strahlenrichtung ($to - from$, A sei der zugehörige Einheitsvektor) werden benutzt.

Der Intensitätsabfall des Lichts wird durch zwei sich überlagernde Funktionen als Faktor aus dem Bereich $[0, 1]$ bestimmt. In eine (**cosine falloff**) der Funktionen geht der Kosinus des Winkels zwischen dem Beleuchtungsvektor (**L**, Einheitsvektor von der Scheinwerferposition zum Oberflächenpunkt im Koordinatensystem der Lichtquelle) und der Strahlenrichtung A ein. *beamdistribution* beeinflußt den Abfall des Faktors innerhalb der beleuchteten Hemisphäre zum Rand hin: $(L \cdot A)^{beamdistribution}$. Am Rand liefert die Funktion entsprechend des Kosinus 0 und in Strahlenrichtung 1. In der unbeleuchteten Hemisphäre wird immer ein Wert von 0 angenommen.

Eine andere Funktion simuliert die 'Klappenöffnung' des Scheinwerfers. Ein Faktor fällt sanft (Hermiteinterpolation) zwischen einem Winkel *conedeltaangle* und dem Kegelrand *coneangle* ab, ist 1 bis zum Winkel *conedeltaangle* und 0 bei einem Winkel größer als *coneangle*. Die Ergebnisse beider Funktionen mit der Intensität untereinander multipliziert ergeben die aktuelle einfallende Intensität des Lichts für einen Punkt auf einer Oberfläche.

RIB BINDING

LightSource name parameterlist

RiLine (RtInt nvertices...):

Erweiterung von *Quick RenderMan* zur Darstellung von stückweise linearen Linienzügen mit *nvertices* Knoten. Bei $nvertices = 1$ wird ein Punkt (Pixel) gezeichnet. Die Parameter müssen mindestens eine Positionsangabe (Typ: **RI_P** oder **RI_PW**) mit *nvertices* Knoten beinhalten.

RIB BINDING

Line parameterlist

RiMakeBump (char *imagefile, char *bumpfile, RtToken swrap, RtToken twrap, RtFloatFunc filterfunc, RtFloat swidth, RtFloat twidth...):

Diese Funktion dient dazu, eine Bitmapdatei (*imagefile*) in eine vom RenderMan Interface lesbare 'Bump Map'-Datei (*bumpfile*) umzuwandeln. Das Format der Dateien ist durch die Implementation des Interfaces festgelegt. Die Bitmap-Datei sollte nur einen Kanal für eine durch 'Bump Map' realisierte Höhenverschiebung enthalten. Durch Heranziehen von den Höhen nebenliegender Pixel wird eine Verschiebung des Normalenvektors berechnet. Die Pixelauflösungen von Bitmap-Datei und 'Bump Map'-Datei bleiben zwar gleich, die parametrischen Koordinaten (texture space) des 'Bump Maps' laufen aber im Bereich von $[0, 1]$ für die s (horizontale) und die t (vertikale) Richtung. Als Pixelfilter *filterfunc* kann eine der vordefinierten Funktionen **RiGausianFilter()**, **RiCatmullRomFilter()**, **RiSincFilter()**, **RiTriangleFilter()** oder **RiBoxFilter()** eingesetzt werden, genausogut kann aber auch eine eigene Funktion, die einen entsprechenden **RiFloat**-Wert liefert verwendet werden. **swidth** und *twidth* (normalerweise 1 oder 2) geben an, wieviele Pixel aus der Umgebung aus dem s/t Raum von der Filterfunktion zur Berechnung des aktuellen Werts herangezogen werden. *swarp* und *twarp* bestimmt, was beim 'Mappen' mit Werten außerhalb der $[0, 1]$ Grenzen geschieht. **RI_PERIODIC** wiederholt das Muster mit der Periode 1, **RI_CLAMP** verwendet dem der Grenze nächsten Wert. **RI_BLACK** liefert den Wert 0 für alle Koordinaten außerhalb der Grenzen. In der Parameterliste können implementationsabhängige Angaben (z.B. das verwendete Dateiformat) gemacht werden. Die Parameter der **RiMakeTexture()** Funktion (für das 'Texture Mapping') haben eine analoge Bedeutung. Der *Quick RenderMan* implementiert diese Funktion, wie alle Texturfunktionen, nicht.

RIB BINDING

MakeBump picturename texturename swrap twrap filter swidth twidth parameterlist

Mögliche Filter sind: `"box"`, `"triangle"`, `"catmull-rom"`, `"b-spline"`, `"gaussian"` und `"sinc"`

RiMakeCubeFaceEnvironment (char *px, char *py, char *nx, char *ny, char *pz, char *nz, char *reflfile, RtFloat fov, RtFunc filterfunc, RtFloat swidth, RtFloat twidth...):

Diese Funktion wird wie **RiMakeLongEnvironment()** im Zusammenhang mit der Darstellung von spiegelnden Oberflächen verwendet. Mit dem RenderMan Interface werden

Spiegelungen durch ‘Texture Maps’ (‘Environment Mapping’) realisiert. Um Verzerrungen bei der Abbildung zu vermeiden, werden zwei verschiedene ‘Mapping’-Funktionen verwendet. Eine für gebogene und eine für plane Oberflächen. Diese Funktion wird für gebogene Oberflächen verwendet. Das ‘Texture Map’ entspricht den sechs Seiten eines Würfels mit der Kamera im Zentrum. Dazu werden von der Mitte des Objekts, auf das das Spiegelbild projiziert werden soll, die sechs Seiten gerendert. Die so erhaltenen Bilddateien werden mittels **RiMakeCubeFaceEnvironment()** als ‘Texture Map’ gespeichert und anschließend durch einen entsprechenden Shader auf das Objekt abgebildet. Die ‘Texture Maps’ werden spiegelverkehrt gerendert (linkshändige Koordinatensysteme rechtshändig und umgekehrt).

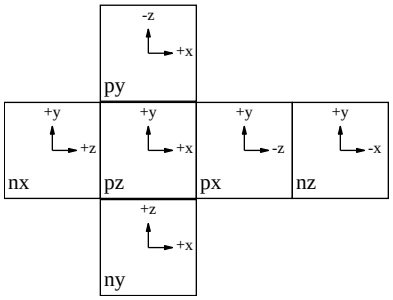

Abbildung A.7: Orientierungen der Texturen für **RiMakeCubeFaceEnvironment()**, aus [Upstill89]

Der Parameter *fov* gibt das Gesichtsfeld in die horizontalen und vertikalen Richtungen an. Bei einem Wert von 90 passen die Oberflächen des Würfels genau aneinander, kleine Überlappungen (zwei bis drei Grad) können das Resultat verbessern ([Upstill89]).

Die Parameter *filterfunc*, *swidth* und *twidth* entsprechen denen von **RiMakeBump()**.

reflfile ist ein Zeiger auf den Namen der Ausgabedatei in die die sechs Bilder, die den Oberflächen des Würfels entsprechen, geschrieben werden. Die restlichen sechs Parameter geben die Namen der Bilddateien der sechs Bilder an, aus denen *relfile* erstellt wird. Das Format der Dateien ist implementationsabhängig (beim NeXT 24 Bit TIFF-Dateien).

Die Parameterliste kann implementationsabhängige Parameter, z.B. eine Kennung für das Dateiformat der Eingabedatei, enthalten. Für *Quick RenderMan* ist diese Funktion nicht definiert.

Mit ‘Environment Maps’ können Körper, die sich selbst (oder Teile von sich selbst) spiegeln nicht behandelt werden.

RIB BINDING

MakeCubeFaceEnvironment px nx py ny pz nz texturename fov filter swidth twidth parameterlist

Mögliche Filter sind: `"box"`, `"triangle"`, `"catmull-rom"`, `"b-spline"`, `"gaussian"` und `"sinc"`

RiMakeLatLongEnvironment (char *imagefile, char *reflfile, RtFunc filterfunc, RtFloat swidth, RtFloat twidth...):

imagefile ist der Name der Eingabedatei (beim NeXT eine 24 Bit Farb TIFF-Datei), *reflfile* der Name der Ausgabedatei, des 'Environment Maps'. Die Formate sind implementationsabhängig. Die restlichen Parameter entsprechen denen von **RiMakeCubeFaceEnvironment()**. Auch diese Funktion kann für den *Quick RenderMan* nicht verwendet werden.

In der **environment()** Funktion der Shading Language kann auf diese 'Environment Map' mit Hilfe der Längen- und Breiten-Information der sphärischen Projektion zugegriffen werden. Die Länge (*latitude*) läuft in linkshändigen Koordinatensystemen von links 0 Grad nach rechts 360 Grad, in rechtshändigen entsprechend von rechts 0 Grad nach links 360 Grad. Die Breite (*longitude*) läuft immer von -90 Grad im Süden nach 90 Grad im Norden. Um den Richtungsvektor (*dx,dy,dz*) zu erhalten, für den **environment()** eine Farbe liefert, ist folgende Umrechnung nötig:

$$dx = \cos(longitude) \cdot \cos(latitude)$$
$$dy = \sin(longitude) \cdot \cos(latitude)$$
$$dz = \sin(latitude)$$

RIB BINDING

MakeLatLongEnvironment picturename texturename filter swidth twidth parameterlist

Mögliche Filter sind: `"box"`, `"triangle"`, `"catmull-rom"`, `"b-spline"`, `"gaussian"` und `"sinc"`

RiMakeShadow (char *picfile, char *shadowfile...):

Wie Spiegelungen werden auch Schatten durch 'Texture Maps' verwirklicht. Ein 'Shadow Map' entspricht dem Z-Buffer, der von einer Lichtquelle als Projektionszentrum angelegt werden kann. *picfile* ist ein Zeiger auf den Namen einer Datei, die den Inhalt eines solchen Z-Buffers enthält. Aus dieser Datei wird das 'Shadow Map' *shadowfile* erzeugt, das von einem Lichtquellen-Shader verwendet werden kann. Besitzt ein Punkt einer Oberfläche, von der Lichtquelle aus gesehen, einen kleineren oder gleichen Wert des ensprechenden Bereichs aus dem 'Shadow Map', beleuchtet die Lichtquelle diesen Punkt. Prinzipbedingte Schwierigkeiten entstehen durch durchscheinende Körper, nicht-punktförmige Lichtquellen (Schattenrand) und Lichtreflexionen. Die Funktion ist im *Quick RenderMan* nicht implementiert.

RIB BINDING

MakeShadow picturename texturename parameterlist

RiMakeTexture (char *imagefile, char *texturefile, RtToken swrap, RtToken twrap, RtFloatFunc filterfunc, RtFloat swidth, RtFloat twidth...):

Diese Funktion wandelt Bitmap-Dateien in das Format von 'Texture Map'-dateien um. Ein entsprechender Oberflächen-Shader kann dann das Bild des 'Texture Maps' auf eine Oberfläche projizieren. Die Parameter haben die gleiche Bedeutung wie die von **MakeBump()**. Beim NeXT kann als Eingabedatei eine 24 Bit tiefe TIFF-Datei verwendet werden. Die Funktion ist im *Quick RenderMan* nicht implementiert.

RIB BINDING

MakeTexture picturename texturename swrap twrap filter swidth twidth parameterlist

Mögliche Filter sind: `"box"`, `"triangle"`, `"catmull-rom"`, `"b-spline"`, `"gaussian"` und `"sinc"`

RtToken RiMacroBegin (RiString name...):

Erweiterung von *Quick RenderMan*, ersetzt Objektblöcke (s. Einbinden von RIB-Makros).

RIB BINDING

MacroBegin name parameterlist

RiMacroEnd ():

Erweiterung von *Quick RenderMan*, schließt einen Makro-Block.

RIB BINDING

MacroEnd –

RiMacroInstance (RtToken macro...): Erweiterung von *Quick RenderMan*, instanziert einen Makro-Block.

RIB BINDING

MacroInstance macro parameterlist

RiMatte (RtBoolean onoff):

Attribut, alle folgenden Oberflächen werden als Platzhalter behandelt (falls $onoff = \mathbf{RI\_TRUE}$). Sie werden nicht schattiert, verdecken aber die dahinterliegenden Objekte. In der Ausgabe erscheinen sie transparent.

RIB BINDING

Matte onoff

RiMotionBegin (RtInt N, RtFloat time1, ... timeN):

Einleitung von der Erzeugung von 'Motion Blur', d.h. Unschärfe aufgrund von Bewegung. Ein solcher Block muß mit **RiMotionEnd()** geklammert werden. Die Werte *time1* bis *timeN* bilden eine aufsteigende Folge von Zeitwerten, an denen ein Interface-Befehl,

Mögliche Befehle in ‘Motion Blur’-Blocks

Transformations	Geometry	shading
RiTransform	**RiBound**	**RiColor**
RiConcatTransform	**RiDetail**	**RiOpacity**
RiPerspective	**RiPolygon**	**RiLightSource**
RiTranslate	**RiGeneralPolygon**	**RiAreaLightSource**
RiRotate	**RiPointsPolygon**	**RiSurface**
RiScale	**RiPointsGeneralPolygon**	**RiInterior**
RiSkew	**RiPath**	**RiExterior**
	RiPatchMesh	**RiAtmosphere**
RiProjection	**RiNuPatch**	
RiDisplacement	**RiSphere**	
RiDeformation	**RiCone**	
	RiCylinder	
	RiHyperboloid	
	RiParaboloid	
	RiDisk	
	RiTorus	

Entnommen aus: [PixSpec]

der innerhalb des Blocks steht *N* mal aufgerufen wird, maximal aber so lange die Kamera 'belichtet'. Das Bild wird schließlich aus den Funktionsaufrufen interpoliert. 'Motion Blur' findet vor allem Anwendung in der Animation. Werden gestochen scharfe Bilder animiert, läuft die Animationssequenz Gefahr, stroboskopähnlich zu wirken. Bei einzelnen Bildern bewirkt 'Motion Blur' eine Dynamik, die anders nicht zu erreichen ist. Das menschliche Auge ist gewohnt, bewegte Bilder unscharf wahrzunehmen. Durch die Verwendung von Motinblur-Blocks ist es möglich, die Verzerrung nur auf das Objekt, bzw. den Befehl, der selbst an der Bewegung teilhat einzusetzen. Werden mehrere Objekte und Befehle für den Bewegungseffekt verwendet, muß für jedes Element ein eigener Block eingeführt werden. Es dürfen nicht alle Interface-Aufrufe in einem 'Motion Blur'-Block verwendet werden.

Die Zeitwerte stehen mit dem Wert, der mit der Option **RiShutter()** (der Öffnungszeit der Kamera) gesetzt wurde in Zusammenhang. Eine Dimensionierung der Werte ist nicht vorgegeben. 'Motion Blur' ist für den *Quick RenderMan* nicht implementiert.

RIB BINDING

MotionBegin $[\mathrm{t}_0\ \mathrm{t}_1\ \ldots\ \mathrm{t}_{n-1}]$

RiMotionEnd ():

Ende des 'Motion Blur'-Blocks.

RIB BINDING

MotionEnd –

RiNuPatch (RtInt nu, RtInt uorder, RtFloat uknot[], RtFloat umin, RtFloat umax, RtInt nv, RtInt vorder, RtFloat vknot[], RtFloat vmin, RtFloat vmax...):

Dient zur Angabe eines NURB-Patches (s.a. **RiPatch()**). Bei NURBs kann, im Gegensatz zu normalen B-Splines, die Knotenverteilung im parametrischen Raum direkt beeinflußt und der Grad der Interpolation festgelegt werden. NURBS basieren immer auf B-Splines; das Basismatrix-Attribut (**RiBasis()**) wird nicht verwendet. Mit Hilfe von Trimkurven (s. **RiTrimCurve()**) können Teile des Patches 'herausgestanzt' werden.

Die u-Parameter haben folgende Bedeutung (v analog):

nu: Anzahl der Kontrollpunkte in u-Richtung. Es muß gelten $nu \geq uorder$, ist $nu < uorder$ wird der Wert von *uorder* entsprechend heruntergesetzt.

uorder: Grad des interpolierenden Polynoms + 1.

uknot[]: Knotenvektor mit $nu + uorder$ Werten, $uknot[k] \leq uknot[k+1]$

umin, umax: Grenzen des ausgewerteten parametrischen Raums in u-Richtung, es muß gelten:

$$uknot_{uorder-1} \leq umin < umax \leq uknot_{nu}$$

In der Parameterliste müssen mindestens die $(nu \cdot nv)$ Kontrollpunkte stehen: **RI_P** bei nicht-rationalen **RI_PW** bei rationalen nicht-uniformen B-Splines. Ist die Anzahl der Kontrollpunkte größer als der Grad des Polynoms, wird die Oberfläche stückweise interpoliert.

RIB BINDING

NuPatch nu uorder uknot umin umax nv vorder vknot vmin vmax parameterlist

RiNuCurve (RtInt nvertices, RtInt order, RtFloat knot[],
RtFloat min, RtFloat max...):

NURB-Raumkurven sind eine Erweiterung vom *Quick RenderMan* (s. [PixQRM]). Die Parameter entsprechen denen von **RiNuPatch()** für eine parametrische Richtung (u).

RIB BINDING

NuCurve order knot min max parameterlist

RtObjectHandle RiObjectBegin ():

Durch den Definitions-Block zwischen **RiObjectBegin()** und **RiObjectEnd()** können Interface-Aufrufe zu einem Funktions-Block, der z.B. die Definition immer wiederkehrender Bildteile enthält, zusammengefaßt werden. Die Ausführung der Befehle wird bis zu dem mehrfach möglichen Aufruf von **RiObjectInstance()**, mit dem entsprechenden Handle als Parameter, verzögert. Das Verwenden von Definitionsblöcken kann zu Laufzeitvorteilen gegenüber normalen C-Funktionen führen, da wiederholtes Aufrufen von gleichen Befehlsfolgen vermieden wird. Nicht alle Renderer unterstützen diese Funktion. Die Implementation auf dem NeXT verwendet an Stelle der Objekt-Blöcke Makros. **RiObjectBegin()** baut nur eine interne Repräsentation auf. Zusätzlich liefert die Funktion abweichend von der Interface-Beschreibung als Wert **RiToken** anstatt **RiObjectHandle()**.

RIB BINDING

ObjectBegin handlenumber

RiObjectEnd ():

Ende eines Definitions-Blocks.

RIB BINDING

ObjectEnd –

RiObjectInstance (RtObjectHandle handle):

Ausführen der Interface-Routinen aus einem Definitions-Block der zu *handle* gehört.

RIB BINDING

ObjectInstance handlenumber

RiOpacity (RtColor Os):

Setzen des Attributs der Undurchsichtigkeit (Opazität) der Komponenten der Oberflächenfarbe. Der Wert 0 für alle Farbkomponenten entspricht einer vollständig transparenten Oberfläche, der voreingestellte Wert 1 einer vollständig opaken.

RIB BINDING

Opacity $c_0\ c_1\ c_2$

Opacity $[c_0\ c_1\ c_2]$

RiOption (char *name...):

Besetzt eine (Nichtstandard-)Option *name* durch die Werte der Parameterliste. Im Gegensatz zu den Attributen müssen alle Optionen vor dem Start des Welt-Blocks (**RiWorldBegin()**) gesetzt sein und gelten global während des Renderns. Optionen sind z.B. die Kameraposition und -Richtung, die Öffnungszeit der Linse und die Tiefenschärfe. Es können auch implementationsabhängige Optionen existieren.

RIB BINDING

Option name parameterlist

RiOrientation (RtToken orientation):

Setzt als Option die Orientierung des Koordinatensystems neu. Voreingestellt ist ein linkshändiges Koordinatensystem **RI_LH**. Ein rechtshändiges Koordinatensystem kann durch den aktuellen Parameter **RI_RH** eingestellt werden. Mit der Orientierung des Koordinatensystems ändert sich entsprechend auch der Drehsinn. Die Funktion ist für den *Quick RenderMan* nicht implementiert.

RIB BINDING

Orientation orientation

RiParaboloid (RtFloat rmax, RtFloat zmin, RtFloat zmax, RtFloat thetamax...):

Erzeugt einen Teil eines Mantels eines Parabolids als Drehkörper. Die z-Achse dient als Drehachse, die Drehung beträgt *thetamax* Grad. *rmax* ist der maximale Radius, d.h. der Radius an der Z-Koordinate *zmax*, der 'obersten' Position des Paraboloids. Unten ist der Paraboloid durch *zmin* beschränkt. Die Spitze, immer an der Position (0,0,0), kann so gekappt werden. *zmin* und *zmax* müssen immer positiv sein. Die folgenden Gleichungen definieren die Oberfläche:

$$\theta = u \cdot \theta_{max}$$
$$z = zmin + v \cdot (zmax - zmin)$$
$$r = rmax \cdot \sqrt{z/zmax}$$
$$x = r \cdot \cos(\theta)$$
$$y = r \cdot \sin(\theta)$$

RIB BINDING

Paraboloid rmax zmin zmax thetamax parameterlist

Paraboloid [rmax zmin zmax thetamax] parameterlist

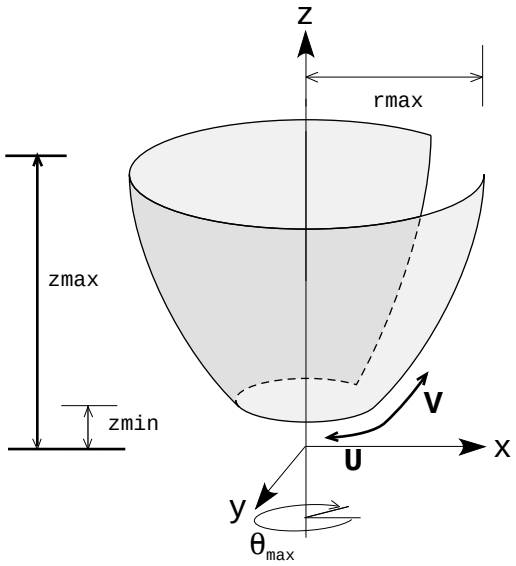

Abbildung A.8: Das Paraboloid, aus [PixSpec]

RiPatch (RtToken type...):

Definition eines einzelnen Patches, einer einzelnen parametrischen Oberfläche, die durch vier Ecken, ähnlich wie ein im Raum schwebender elastischer Lappen, begrenzt ist. Die parametrischen Koordinaten *u* (horizontal) und *v* (vertikal) laufen in den Grenzen $[0, 1]$. Es können zwei Typen von Patches verwendet werden, bilineare **RI_BILINEAR** oder biquadratische **RI_BICUBIC** (bi...wegen der zwei parametrischen Richtungen *u* und *v*). Für einen bilinearen Patch wird die Oberfläche zwischen den vier in der Parameterliste angegebenen Eckpunkten im parametrischen Raum linear interpoliert. Quadratische Patches kommen nicht mit den vier Eckpunkten aus, da sie die Oberfläche durch eine bikubische Splineoberfläche (s. **RiBase()**) interpolieren oder approximieren. In der Parameterliste wird in diesem Fall eine geometrische 4×4 Matrix von 16 Kontrollpunkten übergeben.

RIB BINDING

Patch type parameterlist

RiPatchMesh (RtToken type, RtInt nu, RtToken uwrap, RtInt nv, RtToken vwrap...):

Mit dieser Funktion kann dem Interface mitgeteilt werden, wie ein größeres Gitternetz in einzelne bilineare und bikubische Patches aufgeteilt werden soll. *nu* und *nv* geben die Anzahl der Punkte des Gitters in den jeweiligen param. Richtungen an (mind. je 2 für bilineare und 4 für bikubische). Insges. müssen also $nu \times nv$ Kontrollpunkte in der Parameterliste aufgeführt werden, zuerst die in *u*-Richtung. *uwrap* und *vwrap* können die

Werte **RI_PERIODIC** oder **RI_NONPERODIC** annehmen. Sind Netze periodisch wird bei der Interpolierung das Ende in der jeweiligen Richtung zur Erzeugung von geschlossenen Oberflächen wieder mit dem Anfang zu einem Patch zusammengeführt. Die Anzahl der Knoten muß entsprechend stimmen. Das 'Weiterrücken' des Patch-Fensters über das Gitter geschieht anhand der Angaben, die mit **RiBasis()** gemacht wurden und sind von der Basismatrix des verwendeten Splinetyps abhängig.

RIB BINDING

PatchMesh type nu uwrap nv vwrap parameterlist

RiPerspective (RtFloat fov):

Option, die das Gesichtsfeld der Kamera bestimmt. *fov* ist der Winkel an der Spitze des Frustums. Der Interface-Aufruf ist redundant zu dem *focallength* Parameter von **RiDepthOfField()**.

RIB BINDING

Perspective fov

RiPixelFilter (RtFloatFunc function, RtFloat xwidth, RtFloat ywidth):

Erlaubt das Installieren einer Filterfunktion für die Berechnung der endgültigen Bildpunkte aus den Pixeln. Durch Filtern kann das Aliasing, vor allem die Treppchenbildung an Oberflächenrändern, vermieden werden. Das RenderMan Interface bietet eine Reihe von Standard-Filtern, die bereits bei **RiMakeBump()** aufgeführt wurden. *xwidth* und *ywidth* geben dem Filter einen Anhalt, wieviele Pixel aus der näheren Umgebung des aktuellen Ausgabebereichs in die Berechnung des aktuellen Bildpunkts einbezogen werden können (Breite der Filterfunktion). 1.0 entspricht der Gitterzelle, die ein Bildpunkt einnimmt.

RIB BINDING

PixelFilter type xwidth ywidth

type ist ein Filter aus: `"box"`, `"triangle"`, `"catmull-rom"`, `"gaussian"` und `"sinc"`

RiPixelSamples (RtFloat xsamples, RtFloat ysamples):

Option, die die Anzahl der Pixel angibt, in die ein Bildpunkt aufgeteilt wird (Sample-Rate). Eingestellt sind je zwei Pixel pro x- und y-Richtung eines Bildpunktes. Für jeden endgültigen Bildpunkt werden also vier Helligkeitswerte berechnet.

RIB BINDING

PixelSamples xsamples ysamples

RiPixelVariance (RtFloat variation):

Das RenderMan Interface erlaubt aus Gründen der Effektivität eine adaptive Sample-Rate (Pixel/Bildpunkt: s.a. **RiPixelSamples()**). Die Helligkeitsabweichung der Pixel in einem

Bildpunkt vom Mittelwert (*variation*) kann vor der Quantisierung statistisch berechnet werden und normalerweise durch die Erhöhung der Sample-Rate verkleinert werden. Der Parameter *variation* gibt eine obere Schranke der zulässigen Varianz (threshold) an. Wird die Schranke überschritten, wird die Sample-Rate automatisch angepaßt. Normal wird die Varianz durch Werte zwischen 0 und 1 ausgedrückt (die Quantisierungsstufen sind maschinenabhängig, die Helligkeiten liegen zwischen $[0, 1]$). Allgemein läßt sich sagen: Je niedriger *variation* desto höher die Bildqualität. Ist die Varianz hoch, kann sie durch mehr Pixel pro Bildpunkt und mehr Rechenaufwand verringert werden. Durch Dithering können Artifakte der Quantisierung (Falschkonturen) vermindert werden (*ditheramplitude* in **RiQuantize()**).

RIB BINDING

PixelVariance variation

RiPointsGeneralPolygons (RtInt npolys, RtInt nloops[], RtInt nverts[], RtInt verts[]...):

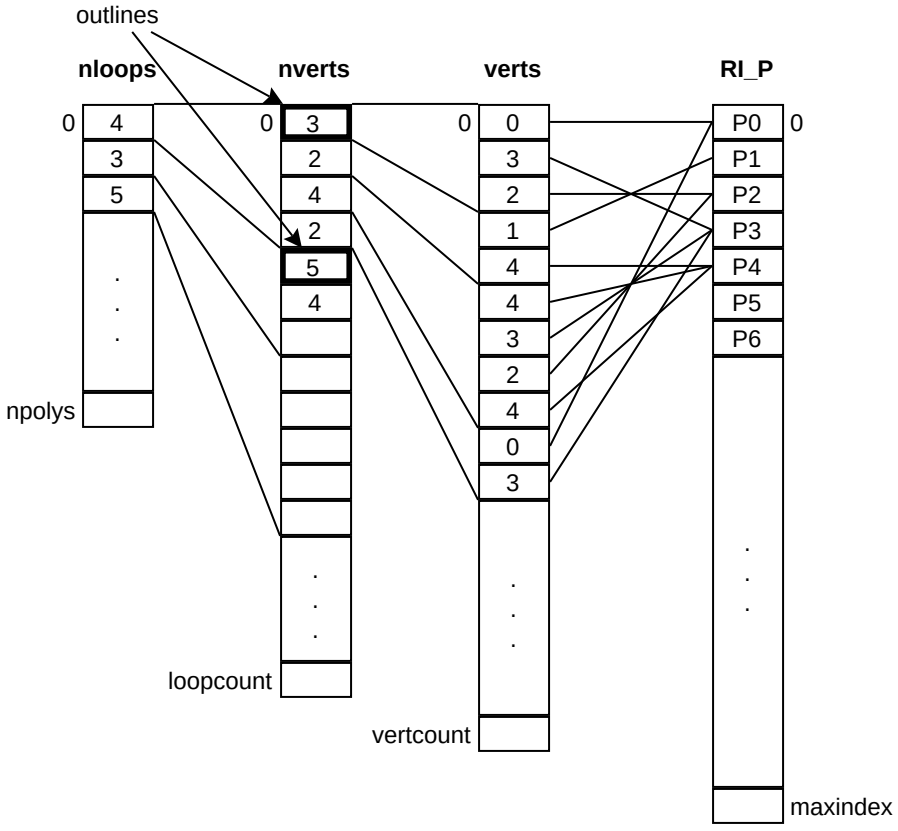

Abbildung A.9: Parameter der **RiPointsGeneralPolygons()** Funktion

Durch diese Funktion ist es möglich, ein Netz von allgemeinen Polygonzügen (planar, konkav mit Löchern) zu spezifizieren. Der Parameter *npolys* gibt die Gesamtzahl der Polygone an. In $nloops[0..npolys-1]$ stehen die Anzahlen der Polygonschleifen pro Polygonzug (1. Schleife eines Polygons ist das Outline, alle weiteren Schleifen sind Löcher). Die Anzahl der Elemente im *nverts* Feld ist

$$loopcount = \sum_{p=0}^{npolys-1} nloops[p]$$

und

$$vertcount = \sum_{l=0}^{loopcount-1} nverts[l]$$

ist die Anzahl der Elemente von *verts*. Die Elemente aus *verts* beinhalten die Indizees aus der Parameter-Wert-Liste, der Knotenliste **RI_P**, der Normalen **RI_N** oder anderen `varying` Variablen. Mindestens eine Positionsangabe eines Knotens muß vorhanden sein.

RIB BINDING

PointsGeneralPolygons nloops nvertices vertices parameterlist

RiPointsLines (RtInt nlines, RtInt nvertices[], Rtint vertices[]...):

Polygonzüge, eine Erweiterung von *Quick RenderMan*. *nlines* ist die Anzahl der Polygone, *nvertices* die Anzahl der Knoten pro Polzgonzug, *vertices* sind die Indizees in die `varying` Variablen der Parameterliste, die mindestens eine Positionsangabe pro Knoten (z.B. **RI_P**) beinhalten muß.

RIB BINDING

PointsLines nvertices vertices parameterlist

RiPointsPolygons (RtInt npolys, RtInt nvertices[], RtInt vertices[]...):

Dient zur Angabe eines Polygonnetzes (planar, konvex, ohne Löcher). Der Aufruf entspricht dem von **RiPointsGeneralPolygons()** mit jeweils $nloops[i] = 1$.

RIB BINDING

PointsPolygons nvertices vertices parameterlist

RiPolygon (RtInt nvertices...):

Spezifizierung eines einzelnen planaren und konvexen Polygonzuges (ohne Löcher) durch eine Knotenliste mit *nvertices* Punkten (s.a. **RiGeneralPolygon()**).

RIB BINDING

Polygon parameterlist

RiProcedural (RtPointer data, RtBound bound, RtFunc refineproc, RtFunc freeproc):

Diese Funktion bietet eine Möglichkeit des prozeduralen Modellierens mit dem RenderMan Interface. Eine Funktion

void refineproc(RtPointer data, RtFloat levelofdetail)

wird zum Zeitpunkt des Renderns mit dem aktuellen Detaillierungsgrad (Fläche von Bildpunkten, die das Objekt ausfüllt) und dem Zeiger auf einen Daten-Block *data* als Parameter aufgerufen. **refineproc()** führt direkt oder indirekt eine beliebige Anzahl von Interface-Aufrufen aus.

void freeproc(RtPointer data)

wird nach dem Rendern aufgerufen und dazu benutzt, den Daten-Block *data* wieder freizugeben. Die Boundingbox *bound*, in Objektkoordinaten angegeben, hilft dem Interface bei der Berechnung des Detaillierungsgrads. *data* sollte nicht verändert werden, da ein Modell u.U. mehrmals benutzt werden kann. Der Speicher darf nur in freeproc() freigegeben werden.

RiProjection (char *name...):

Erlaubt die Definition der 3D-2D Projektionsmethode (als Option) durch ihren Namen: **RI_PERSPECTIVE**, **RI_ORTHOGRAPHIC**, **RI_NULL**. **RI_NULL** setzt die Identitätsmatrix als Projektionsmatrix ein. Mit **Ri_Perspective()** und **Ri_Transform()** kann der Benutzer dann seine eigene Projektionsmatrix bilden. Neben diesen vordefinierten Projektionen können noch implementationsabhängige Projektionen existieren oder Deformations-Shader eingesetzt werden. Werden spezielle Projektionen nicht unterstützt, wird die orthographische Projektion als Default verwendet. In der Parameterliste können Steuerparameter (z.B: **RI_FOV** Gesichtsfeld bei der perspektivischen Projektion) übergeben werden.

RIB BINDING

Projection ”perspective”parameterlist

Projection ”orthographic”

Projection name parameterlist

RiQuantize (RtToken type, RtInt one, RtInt min, RtInt max, RtFloat ditheramplitude):

Erlaubt eine Steuerung der Quantisierung (Umwandlung der kontinuierlichen Helligkeitswerte der Pixel in diskrete Helligkeitsstufen der Bildpunkte) und des Ditherings (Quantisierungsrauschen, Verauschen der Helligkeit zur Vermeidung von Falschkonturen, die durch das menschliche Auge, durch den Mach-Band-Effekt, noch zusätzlich

verstärkt werden können). Die Funktion kann unabhängig für Farbwerte mit Opazität ($type = \mathbf{RI\_RGBA}$) und Tiefenwerte ($type = \mathbf{RI\_Z}$) aufgerufen werden.

Der Parameter *one* bestimmt die Quantisierung durch die Abbildung von Fließkomma- in Fixkommazahlen. Der Wert 0 stellt die Quantisierung ab, es werden in diesem Fall die originalen, evtl. geditherten, Werte, die im Intervall $[0, 1]$ liegen, geliefert.

ditheramplitude ist ein Faktor mit dem eine Zufallszahl aus $[-1, 1]$ skaliert wird, bevor sie auf den noch ungerundeten Fließkommawert addiert wird.

Nach dem Runden des so erhaltenen Ergebnisses, wird der Zwischenwert auf das Intervall $[min,\ max]$ geklippt.

$$value = \mathrm{round}(one \cdot value + ditheramplitude \cdot \mathrm{random}());$$

$$value = \mathrm{clamp}(value, min, max);$$

Voreingestellt ist eine Quantisierung auf ein 8-Bit-Display (**one** = 255) mit einer Ditheramplitude von 0.5. Tiefeninformationen werden per Default nicht gedithert ($ditheramplitude = 0$).

RIB BINDING

Quantize type one min max ditheramplitude

RiReadArchive (RtToken resource, RtFunc callback...):

Erweiterung von *Quick RenderMan*, s. Einbinden von RIB-Makros.

RIB BINDING

ReadArchive resource parameterlist

RiRelativeDetail (RtFloat relativedetail):

Option, die einen Faktor definiert, der mit dem Detaillierungsgrad (s.a. **RiDetail()**, **RiDetailRange()**) multipliziert wird (Standard-Belegung = 1). Falls die Modelle entsprechend definiert wurden, kann mit einem Wert > 1 die Qualität eines Bildes heraufgesetzt und durch einen Wert < 1 vermindert werden.

RIB BINDING

RelativeDetail relativedetail

RtToken RiResource (RtString handle, RtToken type...):

Erweiterung von *Quick RenderMan* für das Einlesen von externen Ressourcen (s. [PixQRM]). Bilder, Shader, Texturen und RIB-Entities werden als Ressourcen bezeichnet. Sie können entweder in einer Datei oder einem Speicher-Block stehen. In dem Parameter *handle* wird der Name der Ressource bestimmt. Ressourcen sind Attribute und deshalb lokal zum aktuellen Attribut-Block. *type* gibt den Typ der Ressource an:

`"archive"`: RIB-Archiv

`"image"`: ‘hardware frame buffer’ oder ‘image data file’

`"shader"`: Programmierter Shader (`.slo`-Datei)

`"texture"`: ‘Texture Map’

Jeder Ressource-Typ besitzt einen korrespondierenden Handle-Typ (siehe **RiCreateHandle()**). Ein `"image"` kann durch einen entsprechenden Aufruf von **RiDisplay()** erzeugt werden.

Die Parameter werden entsprechend des Speichermediums der Ressource übergeben:

`"address"`: **RtPointer**, Zeiger auf den Speicher-Block, in dem die Ressource steht[4]. Als Token kann auch **RI_ADDRESS** verwendet werden:

`"windowid"`: **RtInt**, bestimmt ein offenes Fenster

`"filepointer"`: Zeiger auf eine geöffnete Datei (**FILE ***)

`"filepath"`: Zeiger auf den Pfadnamen einer Datei (**char ***)

Die Rückgabe der Funktion ist das Token **RtToken** der Funktion oder **RI_NULL** im Fehlerfall. Das Token kann in den Funktionen, die eine Ressource benötigen (z.B. **RiSurface()** und **RiDisplay()**), verwendet werden.

RIB BINDING

Resource handle type parameterlist

RiReverseOrientation ():

Dreht die Orientierung des Koordinatensystems um (von rechtshändig nach linkshändig und umgekehrt).

RIB BINDING

ReverseOrientation –

RiRotate (RtFloat angle, RtFloat dx, RtFloat dy, RtFloat dz):

Rotiert das aktuelle Objektkoordinatensystem math. positiv mit *angle* Grad um die Achse, die durch den Vektor (dx, dy, dz) gegeben ist.

RIB BINDING

Rotate angle dx dy dz

RiScale (RtFloat sx, RtFloat sy, RtFloat sz):

Skaliert das Objektkoordinatensystem um die Faktoren *sx*, *sy* und *sz*.

RIB BINDING

Scale sx sy sz

[4] In der Parameterliste wird immer die Adresse eines Werts übergeben!

RiScreenWindow (RtFloat left, RtFloat right, RtFloat bottom, RtFloat top):

Rechteckausschnitt in (genormten) Bildkoordinaten (s.a. 2.5), der exakt auf das Ausgabe-Bitmap abgebildet wird. Das Interface wählt, falls diese Funktion nicht aufgerufen wird, die Koordinaten so, daß in der Ausgabe keine Verzerrungen auftreten. Ist der *frameaspectratio* (s. **RiFrameAspectRatio()**) größer oder gleich eins haben die Koordinaten folgende Werte:

$$(-frameaspectratio, frameaspectratio, -1, 1)$$

Ist der *frameaspectratio* kleiner oder gleich eins, sind die Koordinaten wie folgt belegt:

$$(-1, 1, -1/frameaspectratio, 1/frameaspectratio)$$

RIB BINDING

ScreenWindow left right bottom top

ScreenWindow [left right bottom top]

RiShadingInterpolation (RtToken type):

Der Parameter *type* bestimmt, ob interpoliert (**RI_SMOOTH**, Gouraud Shading) oder konstant schattiert (**RI_CONSTANT**, Fazetten) werden soll. Der Befehl wirkt sich insbesondere bei Polygonen aus.

RIB BINDING

ShadingInterpolation "constant"

ShadingInterpolation "smooth"

RiShadingRate (RtFloat size):

Gibt die Pixelfläche an, die in die Schattierung einbezogen wird. $size = 1$ entspricht dem Phong Shading, für jedes Pixel wird der Shader aufgerufen. Je größer *size* gewählt wird, desto gröber aber schneller wird die Interpolation. Bei einem Wert von **RI_INFINITY** wird der Shader nur für jede Ecke eines Polygons, in daß eine Oberfläche aufgeteilt werden kann, genau einmal aufgerufen. Die Art der Interpolation kann durch **RiShadingInterpolation()** beeinflußt werden.

RIB BINDING

ShadingRate size

RiShutter (RtFloat openTime, RtFloat closeTime):

Option, die die Steuerung der Öffnungszeit der Kamera ermöglicht. *opentime* und *closetime* müssen mit den Zeiten in den 'Motion Blur'-Blocks korrespondieren. Sind beide Werte gleich, wird keine Bewegungsverzerrung dargestellt und das Modell ausgewählt, daß in die entsprechende Zeit fällt. $openTime < closeTime$ muß gelten.

RIB BINDING

Shutter openTime closeTime

RiSides (RInt nsides):

Option, die angibt, ob die Oberflächen der Vorder- und Rückseiten eines Objekts gerendert werden ($nsides = 2$) oder nur die Vorderseiten ($nsides = 1$). Falls alle Rückseiten verdeckt sind, bzw. die Normalen aller sichtbaren Oberflächen zur Kamera weisen (ist der Fall, wenn alle Oberflächen geschlossen und undurchsichtig sind), kann mit $nsides = 1$ eine Geschwindigkeitssteigerung erreicht werden, ohne daß Oberflächen in der Ausgabe fehlen.

RIB BINDING

Sides nsides

RiSkew (RtFloat angle, RtFloat dx1, RtFloat dy1, RtFloat dz1, RtFloat dx2, RtFloat dy2, RtFloat dz2):

Dehnt den Achsenvektor $(dx1, dy1, dz1)$ um *angle* Grad parallel zum Achsenvektor $(dx2, dy2, dz2)$.

RIB BINDING

Skew $\mathrm{angle\ dx_1\ dy_1\ dz_1\ dx_2\ dy_2\ dz_2}$

Skew $[\mathrm{angle\ dx_1\ dy_1\ dz_1\ dx_2\ dy_2\ dz_2}]$

RiSolidBegin (RtToken type):

Start eines CSG-Blocks, der durch **RiSolidEnd()** abgeschlossen wird. Die Blöcke dürfen ineinander verschachtelt werden. Zwischen einer Folge von Blöcken dürfen keine weiteren Interface-Aufrufe gemacht werden. Die Verknüpfung der 'Körper' innerhalb eines Solid-Blocks wird durch *type* angegeben: **RI_PRIMITIVE**, **RI_INTERSECTION**, **RI_UNION** oder **RI_DIFFERENCE**. Der innerste Block muß immer einen geschlossenen **RI_PRIMITIVE** Körper definieren. Falls ein Renderer, wie der *Quick RenderMan*, die Solid-Block-Option nicht unterstützt, werden die CSG-Funktionen ignoriert und wie gewohnt nur Oberflächen dargestellt.

RIB BINDING

SolidBegin type

Als 'type' kann einer der folgenden vier Strings verwendet werden: `"primitive"`, `"intersection"`, `"union"` und `"difference"`.

RiSolidEnd ():

Ende eines CSG-Blocks.

RIB BINDING

SolidEnd –

RiSphere (RtFloat radius, RtFloat zmin, RtFloat zmax, RtFloat thetamax...):

Definiert einen Kugelmantel mit dem Radius *radius*. Der Mantel wird durch einen Drehkörper (Halbkreis mit einer Drehung um *thetamax* um die z-Achse) gegeben. Er kann oben und unten durch geeignete *zmin* und *zmax* Werte gekappt werden. Als Parameter[5] können wie in den anderen Primary-Funktionen Angaben über Farbe **RI_CS** und Opazität **RI_OS** des Körpers gemacht werden. Die folgenden Gleichungen erzeugen die Kugel in Abhängigkeit der parameterischen Koordinaten *u* und *v*:

$$radius > 0, zmax \geq zmin, zmax \geq -radius, zmin \leq radius$$

$$\phi_{min} = \begin{cases} \arcsin\left(\frac{zmin}{radius}\right) & \text{wenn } zmin > -radius \\ -90.0 & \text{wenn } zmin \leq -radius \end{cases}$$

$$\phi_{max} = \begin{cases} \arcsin\left(\frac{zmax}{radius}\right) & \text{wenn } zmax < radius \\ 90.0 & \text{wenn } zmax \geq radius \end{cases}$$

$$\phi = \phi_{min} + v \cdot (\phi_{max} - \phi_{min})$$

$$\theta = u \cdot \theta_{max}$$

$$x = radius \cdot \cos(\theta) \cdot \cos(\phi)$$

$$y = radius \cdot \sin(\theta) \cdot \cos(\phi)$$

$$z = radius \cdot \sin(\phi)$$

RIB BINDING

Sphere radius zmin zmax thetamax parameterlist

Sphere [radius zmin zmax thetamax] parameterlist

RiSurface (char *name...):

Setzen des aktuellen Oberflächen-Shaders innerhalb eines Attribut-Blocks. Mit den Parametern werden die Instanzvariablen des Shaders belegt. Die Renderer auf dem NeXT unterstützen folgende Standard Oberflächen-Shader:

- `"constant"`
- `"matte"` (nicht zu verwechseln mit **RiMatte()**)
- `"metal"`
- `"shinymetal"` (beim *Quick RenderMan* wie ”metal”, wegen fehlender ‘Texture Mapping Capability’),
- `"plastic"`
- `"paintedplastic"` (wie `"plastic"`, wegen fehlender ‘Texture Maps’).

[5] `varying` Variablen beinhalten vier Werte für die Punkte an den parametrischen Koordinaten: (umin, vmin), (umax, vmin), (umin, vmax), (umax, vmax)

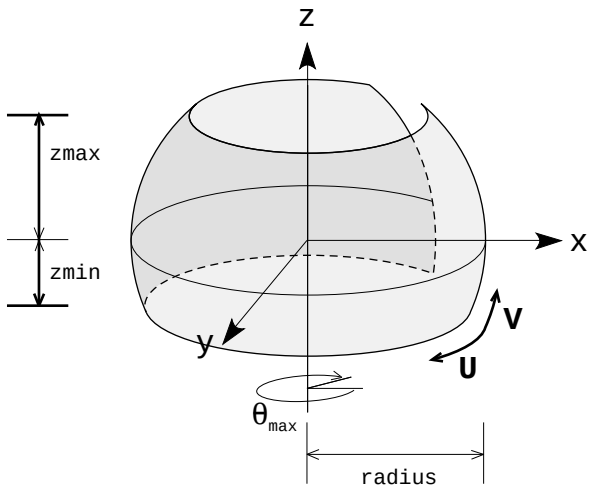

Abbildung A.10: Die Kugel, aus [PixSpec]

Zusätzlich werden zwei implementations-abhängige Oberflächen-Shader angeboten: `"defaultsurface"` und `"show_nxnynz"`. Programmierbare Shader sind für den *Quick RenderMan* noch nicht implementiert.

RIB BINDING

Surface name parameterlist

RiSynchronize (RtToken mode)

Erweiterung von *Quick RenderMan* zur Synchronisierung parallel laufender Renderprozesse. Der Modus *mode* steuert im aktuellen Kontext, wie schnell die Kontrolle an eine ausführende Applikation zurückgegeben wird. Der Synchronisierungsmodus wird entweder dauerhaft gesetzt (die ersten drei Modi) oder nur momentan beeinflußt (die letzten drei Modi).

`"synchronus"`: Wartet bis jeweils alle gemachten Interface-Aufrufe beendet wurden, ändert den aktuellen Synchronisierungsmodus des aktuellen Kontexts.

`"unbuffert"`: Setzt den aktuellen Modus, jeder Interface-Aufruf wird direkt an den Renderer übertragen.

`"asynchronus"`: Setzt den aktuellen Modus, die Interface-Aufrufe können vor der Übertragung an einen Renderer gepuffert werden.

`"flush"`: Überträgt alle gepufferten RenderMan Interface-Aufrufe und kehrt sofort danach zurück. Der aktuelle Modus wird nicht geändert.

`"wait"`: Überträgt alle gepufferten RenderMan Interface-Aufrufe und kehrt erst zurück, wenn alle Berechnungen beendet wurden. Der aktuelle Modus wird nicht geändert.

`"abort"`: Abbrechen des Renderns des aktuellen Bilds. Der Welt-Block wird geschlossen. Der Grafikzustand wird entsprechend restauriert. Der aktuelle Modus wird nicht geändert. Wird **RiSynchronize()** mit diesem Modus außerhalb des Welt-Blocks aufgerufen, geschieht nichts.

RIB BINDING

Synchronize mode

RiTextureCoordinates (RtFloat s1, RtFloat t1, RtFloat s2, RtFloat t2, RtFloat s3, RtFloat t3, RtFloat s4, RtFloat t4):

Erlaubt eine Skalierung einer Textur in *s* und *t* Richtung des Texturkoordinatensystems. (*s1*,*t1*) – (*s4*,*t4*) geben die Projektion des Einheitsquadrates im Parameterraum (0,0)–(1,1) auf die Texturkoordinaten an. Bei Werten größer 1 entscheiden die Angaben, die bei der Texturerzeugung mit **RiMakeTexture()** (*swrap*, *twrap*) gemacht wurden, wie die Bereiche außerhalb des Parameterraums behandelt werden: Keine Textur oder die Textur periodisch immer wieder von vorn.

RIB BINDING

TextureCoordinates $\mathrm{s}_1\ \mathrm{t}_1\ \mathrm{s}_2\ \mathrm{t}_2\ \mathrm{s}_3\ \mathrm{t}_3\ \mathrm{s}_4\ \mathrm{t}_4$

TextureCoordinates $[\mathrm{s}_1\ \mathrm{t}_1\ \mathrm{s}_2\ \mathrm{t}_2\ \mathrm{s}_3\ \mathrm{t}_3\ \mathrm{s}_4\ \mathrm{t}_4]$

RiTorus (RtFloat majorrad, RtFlot minorrad, RtFloat phimin, RtFloat phimax, RtFloat thetamax...):

Erzeugt einen Torusmantel als Drehkörper. *majorrad* ist der Kreisradius, *minorrad* der Radius des 'Schlauchs'. *phimin* und *phimax* sind die Drehwinkel innerhalb denen ein Meridian des Mantels erzeugt wird, *thetamax* ist der Winkel, in dem ein Breitenkreis des Torusmantel geschlossen werden soll. **RI_OS** und **RI_CS** Token-Array-Paare können in der Parameterliste auftauchen. Die folgenden Gleichungen bestimmen den Torus:

$$\theta = u \cdot \theta_{max}$$

$$\phi = phimin + v \cdot (phimax - phimin)$$

$$r = minorradius \cdot \cos(\phi)$$

$$z = minorradius \cdot \sin(\phi)$$

$$x = (majorradius + r) \cdot \cos(\theta)$$

$$y = (majorradius + r) \cdot \sin(\theta)$$

RIB BINDING

Torus rmajor rminor phimin phimax thetamax parameterlist

Torus [rmajor rminor phimin phimax thetamax] parameterlist

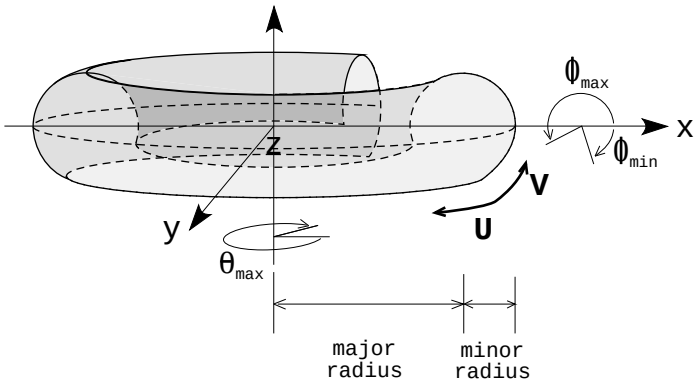

Abbildung A.11: Der Torus, aus [PixSpec]

RiTransform (RtMatrix transform):

Ersetzt die aktuelle Transformationsmatrix innerhalb eines Attribut-Blocks oder eines Transformations-Blocks.

RIB BINDING

Transform transform

RiTransformBegin ():

Beginn eines Transformations-Blocks. Die aktuelle Transformationsmatrix wird auf dem Attribut-Stack zwischengespeichert.

RIB BINDING

TransformBegin –

RiTransformEnd ():

Ende eines Transformations-Blocks.

RIB BINDING

TransformEnd –

RtPoint *RiTransformPoints (RtToken fromspace, RtToken tospace, RtInt npoints,
RtPoints point[]):

Transformiert *npoints* Punkte vom Koordinatensystem *fromspace* in das Koordinatensystem *tospace*. Beispielsweise vom Kamerakoordinatensystem **RI_CAMERA** in das Weltkoordinatensystem **RI_WORLD**.

RiTranslate (RtFloat dx, RtFloat dy, RtFloat dz):

Multipliziert eine Translationsmatrix links an die aktuelle Transformationsmatrix.

RIB BINDING

Translate dx dy dz

RiTrimCurve (RtInt nloops, RtInt ncurves[], RtInt order[], RtFloat knot[], RtFloat min[], RtFloat max[], RtInt n[], RtFloat u[], RtFloat v[], RtFloat w[]):

Trimmen einer NURB Oberfläche (s. **RiNuPatch()**) mit Hilfe von *nloops* explizit geschlossenen NURB-Loops, die im parametrischen Raum der zu trimmenden Oberfläche definiert werden. Die Parameter haben folgende Bedeutung:

nloops: Anzahl der NURB-Loops

ncurves[]: Anzahl der Kurven pro Loop, der Nte Loop ($0 \leq N < nloops$) besitzt ncurves$[N]$ Kurven, die Gesamtzahl der Kurven C ist

$$C = \sum_{N=0}^{nloops-1} ncurves[N]$$

order[]: Ordnung + 1 einer Kurve k, ($0 \leq k < C$)

n[]: Anzahl der Kontrollpunkte des NURBs k, ($n[k] \geq order[k]$)

min[], max[]: Parametrische Grenzen der Kurve k

knot[]: Knotenvektor des NURBs, pro Kurve k existieren $n[k] + order[k]$ Werte, mit $knot[k] \leq knot[k+1]$

u[], v[], w[]: Homogene Koordinaten aus dem parametrischen Raum der NURB-Oberfläche der Kontrollpunkte der NURB-Trimmkurve. Die Kurven eines Loops müssen aneinander anliegen. Wird die Reihenfolge so gewählt, daß sie im Uhrzeigersinn angeordnet sind, bleibt das Innere der Kurve von dem Patch erhalten, andernfalls das Äußere.

RIB BINDING

TrimCurve ncurves order knot min max n u v w

RIB BINDING

Version number

Anm.: Protokoll-Versionsnummer des RIB-Streams (momentan 3.03) wird nur in RIB-Dateien verwendet.

RiWorldBegin ():

Leitet den Welt-Block ein. Weltblöcke dürfen nicht geschachtelt werden, wohl aber eine Sequenz bilden[6]. Innerhalb eines Welt-Blocks können ausschließlich Attribute verändert und Objekte definiert werden.

[6] Mehrere Weltblöckke können z.B. dazu verwendet werden, Bilder für 'Environment Maps' zu erzeugen.

RIB BINDING

WorldBegin –

RiWorldEnd ():

Ende eines Welt-Blocks. Lichtquellen und Objekte, die im Block definiert wurden, werden ungültig.

RIB BINDING

RiWorldEnd –

A.3 Quick RenderMan Zusätze

Zusätzlich zu den hier aufgeführten Funktionen können für den *Quick RenderMan* einige zusätzliche Funktionen zum Picken in Abhängigkeit eines Z-Werts und zum Einlesen von RIB-Archiven verwendet werden. Die Spezifikationen sind in [PixQRM] zu finden, die Deklarationen in `qrman.h`. Das Picken von Objekten in einem Rechteck aus Bildschirmkoordinaten wurde im 3DKit durch die **N3DCamera**-Methode **selectShapesIn:** verwirklicht.

RtVoid QRMSetPickCallback (RtVoid (*func)(RtPickCallback *))

RtVoid QRMSetPickCallback (RtVoid (*func)(RtInt, RtInt *, RtFloat))[7]

Einsetzen einer Pick-Callback-Routine. Die Funktion wird von einem speziellen Pick-Hider (**RI_PICK** kann als Option mit **RiHider()** gesetzt werden) aufgerufen. Es werden in diesem Fall keine Ausgaben erzeugt. Das 'Picken' kann über den Attribut-Stack durch das Attribut **RI_PICKING** gesteuert werden. Mit dem Parameter **RI_PICKABLE** kann durch den folgenden Integer (es wird wie üblich einen Zeiger auf eine Integer-Variable übergeben) gesteuert werden, ob das Picken aktiviert (Wert: 1) oder deaktiviert (Wert: 0) sein soll. Das Picken ist standardmäßig eingeschaltet. Mit dem Parameter **RI_PICKTAG** kann ein Integer (wieder über ein Zeiger auf eine Variable) auf einer Tag-Liste abgelegt werden. Die Grösse der Tag-Liste entspricht immer der Tiefe der Schachtelung der Attribut-Blöcke (+1), es wird immer das letzte Element der Tag-Liste von dem **RI_PICKTAG** Parameter beeinflußt. Der Pick-Hider ruft bei aktiviertem Picken die Callback-Routine mit der aktuellen Listengröße, der Tag-Liste und dem kleinsten Z-Wert des gefundenen Grafik-Primitivs (z.B. **eines RiSphere()**) auf.

Die Typ **RtPickCallback** ist wie folgt deklariert:

```
typedef struct {
    RtInt   tagCount;    // number of elements in tagList
    RtInt   *tagList;    // list of picktags
    RtFloat z;
} RtPickCallback;
```

[7] Diese Deklaration ist nur in der Dokumentation, jedoch nicht in `qrman.h` aufgeführt.

RtVoid QRMGetRibHandlers (RtRIBHandlers *handlerPointer)

RtVoid QRMSetRibHandlers (RtRIBHandlers *handlerPointer)

Wenn der *Quick RenderMan* eine RIB-Ressource liest (**RiReadArchive()**), 'parst' er die Befehle und ruft eine spezielle Handler-Routine für jeden dieser Befehle auf. Eine solche Routine besitzt die gleichen Parameter wie die entsprechende **Ri...()** Interface-Funktion, bzw. **Ri...V()** Interface-Funktion falls die Anzahl der Parameter variabel ist. Mit der Funktion **QRMGetRibHandlers()** kann ein Verbund mit den Funktions-Zeigern auf diese Handler-Routinen gefüllt werden. Einzelne Zeiger können mit Zeigern auf eigene Routinen überschrieben und mit **QRMSetRibHandlers()** in das System eingesetzt werden. Die längliche Deklaration von **RtRIBHandlers** kann in `qrman.h` nachgesehen werden. Hier ein Ausschnitt:

```
typedef struct RIBHandlers {
    ...
    RtVoid (*SolidEnd)(void);
    RtVoid (*SphereV)(RtFloat, RtFloat, RtFloat,
        RtFloat, RtInt, RtToken [], RtPointer []);
    ...
    RtVoid (*Translate)(RtFloat, RtFloat, RtFloat);
    ...
} RtRIBHandlers;
```

Die folgende Funktion ist nur als Deklaration in der Header-Datei `qrman.h` zu finden. Die Dokumentation fehlt deshalb.

RtVoid QRMSetInfoCallback (RtVoid (*func)(RtInfoCallback *))

Die Deklaration des Typs **RtInfoCallback** lautet:

```
typedef struct {
    RtFloat bound[6];    // rendered camera space bounds
} RtInfoCallback;
```

A.4 Die Shading Language

A.4.1 Shader und Funktionen

Die Shading Language erlaubt das Schreiben von Schattierungsalgorithmen in einer C-ähnlichen Syntax. Zur Unterstützung der Shader können zusätzliche Funktionen geschrieben werden. Es stehen für diese Aufgabe spezielle Datentypen, globale Steuervariablen, Operatoren und fest eingebaute Funktionen zur Verfügung. Die Instanzvariablen der Schader werden als Parameterliste im Kopf des Shaders definiert und können (ähnlich der C++ Schreibweise [Strou91] für die Standard-Belegung von Parametern) mit 'Defaults' belegt werden. Anders als C-Parameter,

werden die Belegungen dieser Instanzvariablen der Shaderinstanz bei seiner Erzeugung zugewiesen und nur die Werte der `varying` Variablen pro Aufruf beim Shading gesetzt. Das Einsetzen eines Shaders als Attribut entspricht in etwa einem Konstruktoraufruf in objektorientierten Sprachen. Bei allen Parametern handelt es sich im Gegensatz zu denen in C um Referenzparameter. Die Namen der Instanzvariablen sind zwar frei wählbar, dennoch sollten, wo möglich, Standard-Namen verwendet werden: *Kd*, *Ks* und *Ka* für die diffusen, spekularen und ambienten Koeffizienten, *from* für die Position einer Lichtquelle u.s.w. Shaderinstanzen können durch die Interface-Aufrufe: **RiAreaLightSource()**, **RiAtmosphere()**, **RiDeformation()** (manchmal auch als **RiTransformation()** zu finden, der Shader wird auch Transformations-Shader genannt), **RiDisplacement()**, **RiExterior()**, **RiImager()**, **RiInterior()**, **RiLightSource()** und **RiSurface()** in dem aktuellen Attribut-Block erzeugt werden. Es hilft, sich die Shader-Typen als Spezialisierungen eines abstrakten Shaders vorzustellen. Es kann mit Ausnahme der Lichtquellen-Shader nur eine Instanz pro Shader-Typ als Attribut gültig sein. Die Parameter eines Shaders können dem Interface durch einen Aufruf von **RiDeclare()** bekannt gemacht werden. Shader werden ausschließlich vom System aufgerufen und zwar pro sichtbaren Oberflächenpunkt einmal. Funktionen dürfen aus Shadern und anderen Funktionen heraus aufgerufen werden. Rekursion, gleich ob direkt oder indirekt, ist nicht erlaubt. Im Unterschied zu Funktionen können Shader keinen Rückgabewert liefern. Sie legen ihre Berechnung ausschließlich in den dafür vorgesehenen *output* Steuervariablen ab. Als Ausgangspunkt für ihre Berechnungen können neben den Instanzvariablen auch *input* Steuervariablen dienen. Beispielsweise wird die Opazität einer Oberfläche einem Oberflächen-Shader in der *input* Variablen *Os* übergeben. Der Shader gibt dem Renderer die aktuelle, berechnete Opazität in der *output* Variablen *Oi* zurück. RenderMan unterscheidet für die Variablen zwei Speicherklassen: `varying` und `uniform`. In der ersten Speicherklasse liegen Variablen, die sich über die Oberfläche ändern (C-Arrays, pro Knoten einer Oberfläche ein Element, dazwischenliegende Werte werden interpoliert), in der zweiten die, die über die Oberfläche gleich bleiben (einelementige Arrays). Typisches Beispiel für eine `varying` Variable ist die Oberflächennormale *N*. Sie kann für jeden Gitterknoten vom Benutzer angegeben werden. Zwischen den Gitterpunkten wird ihr Wert interpoliert. Die Kameraposition *E* hingegen ist eine typische `uniform` Variable; es gibt nur genau eine Kameraposition. Es gibt auch Variablen, auf die beide Speicherklassen zutreffen können. Ein Beispiel hierfür ist die Oberflächenfarbe *Cs*, sie ist entweder uniform auf dem ganzen Körper oder es wird zwischen Farben, die einem bestimmten Gitterknoten zugeordnet sind, interpoliert. Die Möglichkeit die Steuervariablen auszulesen oder ihnen einen Wert zuzuweisen ist abhängig vom verwendeten Shader-Typ, Lichtfarbe *Cl* und Richtung *L* dürfen sinnvollerweise nur in den **solar()**, **illuminate()** Iteratoren (s. weiter unten) der ‘Lightsource’-Shader und dem **illuminance()**-Iterator des Oberflächen-Shaders verwendet werden. Manche Steuervariablen haben, wenn sie in einem ‘Lightsource’-Shader verwendet werden nur eine Bedeutung, wenn dieser als ‘Area Lightsource’-Shader eingesetzt wurde.

Je nach Shader-Typ werden unterschiedliche Ausgabevariablen verändert. Aufgabe des Oberflächen-Shaders ist u.a., Opazität und Farbe eines Oberflächenpunkts in den Variablen *Oi* und *Ci* bereit zu stellen. Verändert ein Oberflächen-Shader die Opazität nicht, ist es notwendig, daß er explizit den Wert der Ausgabevariablen *Oi* auf den Wert der Eingabevariablen *Os* setzt, damit der Punkt den Wert des entsprechenden Attributs zugewiesen bekommt. Ein ‘Displacement’-Shader kann die Farben der Oberfläche nicht verändern, da er mit einem normalen Oberflächen-Shader kombiniert werden kann und letzteren nicht stören soll. Der ‘Displacement’-Shader kann

dafür beispielsweise die Normale *N* der Oberfläche neu setzen oder einen Punkt konkret in der Höhe verschieben. Ausgangspunkt für das Verändern der Normale ist die geometrische Normale *Ng*. Der Shader kann die neue Normale nach einem beliebigen Algorithmus berechnen oder durch ein 'Bump Map' aus einer Datei bestimmen.

[Upstill89] und [PixSpec] beinhalten eine ausführliche Beschreibung der Shading Language und Beispiele der Verwendung.

A.4.2 Syntax

Die Syntax entspricht weitgehend der von C. Allerdings wurden einige Sprachkonstrukte weggelassen und andere hinzugefügt. Es dürfen keine Typen definiert werden, pro Datei darf nur ein Shader und beliebig viele Funktionen definiert werden. Shader besitzen im Gegensatz zu Funktionen keinen expliziten Return-Wert.

Zu dem einzigen skalaren Basistyp **float** (Fließkomma) existieren noch drei zusammengesetzte Typen. Der Datentyp **point** wird für 3D-Punkte und 3D-Vektoren verwendet. **color** ist der Typ einer Farbe mit standardmäßig 3 **float**-Komponenten (voreingestellt ist der 'rgb' Farbraum). **string** ist der Typ einer Zeichenkette (C-Schreibweise), der im Prinzip nur für Dateinamen, Objektbenennungen und Handles verwendet wird. Alle diese Typen dürfen auch als Return-Typ einer Funktion verwendet werden. Die in der Sprache zur Verfügung stehenden Operatoren sind mit den verschiedenen Typen überladen. Eine automatische Typkonvertierung kann bei der Zuweisung geschehen. Funktionen dürfen nicht überladen werden.

Funktionen werden wie in C (vor der ANSI Norm) definiert. Dem Namen des Rückgabe-Typs folgt der Funktionsname und die Parameterliste. Shader hingegen werden durch ein Schlüsselwort eingeleitet, das den Shader-Typ angibt: **light**, **displacement**, **surface**, **volume**, **transformation** (entspr. **RiDeformation()**) oder **imager**. Es folgt der Name des Shaders und in Klammern die Liste der Instanzvariablen mit den Defaultbelegungen in einer C++-ähnlichen (aber nicht gleichen) Schreibweise. Die aus C bekannte **switch** Anweisung fehlt, da der elementare Skalar ein Float-Wert ist, für den diese Anweisung nicht definiert ist. Eine 'do-while' Schleife ist nicht in der Sprachdefinition enthalten. Wie bei C-Compilern üblich, läuft auch beim Shader-Compiler vor der eigentlichen Übersetzung ein Präprozessor über den Programmtext. Der Präprozessor entspricht in seiner Verwendung dem des C-Präprozessors `cpp`. Entsprechend können Kommentare und #-Anweisungen verwendet werden. Die erste beschriebene Zeile eines Shaders muß einen (sinnvollen) Kommentar enthalten. Die Syntax der Sprache kann aus der folgenden EBNF entnommen werden (`definition` ist die Startvariable, optionale Teile stehen in eckigen Klammern, Wiederholung ist durch geschwungene Klammern gekennzeichnet, Alternativen sind durch einen senkrechten Strich getrennt, Terminale stehen in einfachen Häckchen).

```
definition ::= function | shader | {function} shader {function};

function ::= [typename] identifier '(' [formdefs] ')' procblock;
shader   ::= shadertype [shaderspace]
             identifier '(' [formdefs] ')' procblock;

typename   ::= 'float' | 'string' | pspace | cspace;

shadertype ::= 'light'| 'displacement'| 'surface'|
```

```
              'volume'| 'transformation'| 'imager';

shaderspace ::= pspace | cspace | pspace cspace | cspace pspace;

cspace     ::= 'color' [cspacename];
cspacename ::= 'rgb' | 'hsv' | 'xyz' | 'XYZ' | 'xyY' |
               'YIQ' | stringconstant;

pspace     ::= 'point' [pspacename];
pspacename ::= 'current' | 'camera' | 'world'  | 'shader' |
               'object'  | 'screen' | 'raster' | stringconstant;

identifier ::= letter {letter|digit};
letter     ::= 'a'|...|'z'|'A'|...|'Z'|'_';
digit      ::= '0'|...|'9';

procblock ::= '{' [defs] {statement} '}';

defs     ::= def ';' | defs def ';';
formdefs ::= def | formdefs ';' def;
def      ::= [storageclass] typename identifier ['=' expr]]
             {',' identifier ['=' expr]};

storageclass ::= 'varying'| 'uniform';

statement ::= ( 'break' [integer] | 'continue' [integer] |
                'return' expr |
                funccall |
                asgnexpr |
                compound |
                block
               ) ';' ;

compound ::= 'if' '(' relation ')' statement ['else' statement] |
           'while' '(' relation ')' statement |
           'for' '(' expr ',' relation ',' expr ')' statement |
           'illuminance' '(' expr [ ',' expr ',' expr] [',' expr] ')'
               statement |
           'illuminate' '(' expr [ ',' expr ',' expr] ')' statement |
           'solar' '(' [expr ',' expr] ')' statement;

block ::= '{' statement {statement} '}';

funccall ::= identifier '('[expr {',' expr}] ')';

relation ::= '(' relation ')' | expr relop expr |
             relation logop relation | '!'relation;

expr ::= primary | expr binop expr | '-'expression |
         relation '?' expr ':' expr;

primary ::= float | texture | identifier | stringconstant |
            [cast] funccall | asgnexpr |
            '(' expr ')' | '(' expr ',' expr ',' expr ')';
```

```
cast     ::= 'float' | 'point' | 'color' | 'string';

asgnexpr ::= identifier asgnop expr;

binop    ::= '.' |
             '/' | '*' |
             '^' |
             '+' | '-';
relop    ::= '>' | '>=' | '<=' | '<' |
             '==' | '!=';
logop    ::= '&&' |
             '||';
asgnop   ::= '=' | '+=' | '-=' | '*=' | '/=';

texture          ::= textureType '(' texture_filename [channel]
                     {',' expr} ')';
texture_type     ::= 'texture' | 'environment' | 'bump' | 'shadow';
texture_filename ::= stringconstant | identifier;
channel          ::= '[' integer ']';

float   ::= (digit{digit}['.'{digit}]|'.'digit{digit})
            [(e|E)[-|+]digit{digit}];
integer ::= digit{digit};
```

Wie aus der Syntax ersichtlich ist, werden zwei neue Operatoren eingeführt: '`.`' und '`^`', das Skalar- und das Kreuzprodukt. Die Operatoren sind normalerweise nur auf Vektoren definiert. Die Shading Language überläd diese Operationen so, daß Floats zu Punkten gewandelt werden (der Y- und Z-Koordinate wird 0 zugewiesen). Farben werden bezüglich der beiden Operatoren wie die entsprechenden Vektoren mit der größten Dimension behandelt. ϕ sei der Winkel zwischen den beiden Vektoren A und B.

$$A.B = |A| \cdot |B| \cdot \cos(\phi)$$
$$|A \hat{} B| = |A| \cdot |B| \cdot \sin(\phi)$$

Die anderen Operatoren (+, $-$, u.s.w.) sind entsprechend mit **point** und **color** überladen. Die Operatorpräzedenzen können in [Upstill89], 'Chapter 14, Section Syntax' nachgeschlagen werden; sie entsprechen denen in der Programmiersprache C. Für Stringkonstanten `stringconst` gelten die in C üblichen Konventionen. In [Upstill89], 'Chapter 14, Section Global Variables' kann die Verwendbarkeit der Steuervariablen gefunden werden.

Neben den beiden Operatoren werden das **illuminance()** in Surface-Shadern, **illuminate()** und als Spezialfall des letzteren das **solar()** Konstrukt in Light-Shadern neu eingeführt. Bei ihnen handelt es sich im Prinzip um Iteratoren, die es ermöglichen, die auf einer Oberfläche eintreffenden und von einer Lichtquelle ausgehenden Lichtstrahlen zu behandeln. Die Iteratoren dürfen nicht verschachtelt werden, **illuminate()** und **solar()** dürfen nicht gleichzeitig verwendet werden.

Das illuminance() Konstrukt

Das Konstrukt kontrolliert die Integrierung des eintreffenden Lichts. Es kann in Oberflächen-Shadern verwendet werden. In **illuminance()** sind die beiden zusätzlichen Variablen *Cl* (Lichtfarbe) und L (Vektor zur Lichtquelle, nicht unbedingt normalisiert) definiert. Auf sie kann lesend zugegriffen werden. Der Iterator wird für jede in Frage kommende Lichtquelle genau einmal durchlaufen.

illuminance (position, axis, angle, [nSamples]) statement

Die Parameter definieren einen Kegel (durch die Achse durch sein Zentrum und den halben Öffnungswinkel in Radianten gegeben), der mit seiner Spitze auf den beleuchteten Punkt weist. Ist $angle = \pi$ entspricht die erste Form der zweiten, die gesamte Sphäre um den Punkt wird betrachtet. Der optionale Parameter *nSamples* gibt die Anzahl der Samples an, die untersucht werden um das Integral auszuwerten. Fehlt der Parameter wird die Auswertung so effizient wie möglich vorgenommen.

illuminance (position, [nSamples]) statement

Spezialfall des ersten Falls, $angle = \pi$.

Das illuminate() und das solar() Konstrukt

Sie sind die inversen Funktionen zu **illuminance()**: Sie werden in Lichtquellen-Shadern verwendet um die Ausleuchtung von gerichteten Lichtquellen festzulegen. In Shadern für ambientes Licht fehlen diese Konstrukte. In **illuminate()** und **solar()** kann auf die beiden Variablen *Cl* (Lichtfarbe) und L (Vektor zur beleuchteten Oberfläche) zugegriffen werden. Die Lichtfarbe soll nach Möglichkeit gesetzt werden. Der Iterator wird für jede der beleuchteten Oberflächen durchlaufen.

Innerhalb von **illuminate()** ist die Länge von L gleich der Entfernung zur beleuchteten Oberfläche. Das **solar()** Konstrukt dient Lichtquellen über der gesamten Sphäre eines Objekts.

illuminate (position, axis, angle) statement

Das Licht wird von *position* ausgehend innerhalb eines durch *axis* und *angle* gegebenen Kegels ($2 \cdot angle$ ist der Öffnungswinkel des Kegels) ausgestrahlt.

illuminate (position) statement

Das Licht wird von *position* ausgehend in alle Richtungen geworfen.

solar () statement

Die Lichtstrahlen werden aus allen Punkten im Unendlichen in alle Richtungen (Sphäre um den beleuchteten Punkt, Beispiel: Sternenlicht) ausgestrahlt.

solar (axis, angle) statement

Die Lichtstrahlen werden innerhalb des durch *axis* und *angle* gegebenen Kegels (Apex am beleuchteten Punkt) ausgestrahlt. Ist der $angle = 0$ kann ein 'Distant Lightsource'-Shader definiert werden — alle Strahlen sind parallel.

A.4.3 Shading-Steuer-Variablen

Die folgende Tabelle ([Upstill89], [PixSpec]) zählt die, in den Shadern verwendbaren, Variablen auf. In einigen Shadern haben die Variablen eine andere als die aufgeführte Bedeutung, so ist in Volumen-Shadern *Ci* nicht die Oberflächenfarbe sondern die Farbe des eintreffenden Lichts.

Typ	Name	Speicherklasse	Zweck
color	*Cs*	`varying/uniform`	Oberflächenfarbe
color	*Os*	`varying/uniform`	Oberflächenopazität
point	*P*	`varying`	Oberflächen-Position
point	*dPdu*	`varying`	Steigungsvektor in u-Richtung
point	*dPdv*	`varying`	Steigungsvektor in v-Richtung
point	*N*	`varying`	Shading-Oberflächen-Normale, vorbelegt mit *Ng*
point	*Ng*	`varying/uniform`	geometrische Oberflächen-Normale *dPdu^dPdv*
point	*Ps*	`varying`	Beleuchtete Position
float	*u*, *v*	`varying`	Parametrische Position
float	*du*, *dv*	`varying/uniform`	Änderung der u, v Koordinaten des Elements
float	*s*, *t*	`varying`	Texturkoordinaten
point	*L*	`varying/uniform`	Vektor von Oberfläche zu Lichtquelle
color	*Cl*	`varying/uniform`	Lichtfarbe
point	*I*	`varying`	Einfallender Vektor (von Kamera)
point	*E*	`uniform`	Kameraposition
point	*ncomps*	`uniform`	Anzahl der Farbkomponenten
point	*time*	`uniform`	Aktuelle Verschlußzeit
alpha	*ncomps*	`uniform`	Pixelopazität
color	*Ci*	`varying`	Oberflächenfarbe
color	*Oi*	`varying`	Oberflächenopazität

Die anschließend Tabelle (ebenfalls aus [Upstill89] und [PixSpec] zusammengefaßt) gibt die Zugriffsmöglichkeiten auf die verschiedenen Variablen in den verschiedenen Shadern wieder.

Typ	Name	Surf.	Light	Displ.	Trans.	Volume	Imager
color	*Cs*	R					
color	*Os*	R					
point	*P*	R	R	RW	RW	R	R
point	*dPdu*	R	R†	R			
point	*dPdv*	R	R†	R			
point	*N*	R	R†	RW	RW		
point	*Ng*	R	R†	R			
point	*Ps*		R				
float	*u*, *v*	R	R†	R			
float	*du*, *dv*	R	R†	R			
float	*s*, *t*	R	R†	R			
point	*L*	R*	R*				
color	*Cl*	R*	RW*				
point	*I*	R				R	
color	*Ci*	RW				RW	RW
color	*Oi*	RW				RW	RW
point	*E*	R	R			R	
point	*ncomps*	R	R	R		R	R
point	*time*	R	R	R	R	R	R
alpha	*alpha*						R

* in **solar()**, **illuminate()** oder **illuminance()**

† nur in ‘Area Lightsource’ verwenden

A.4.4 Funktionen der Shading Language

color ambient (point P)

Liefert die aktuelle ambiente Farbe am Punkt *P* innerhalb des Objektkoordinatensystems. Eine ambiente Lichtquelle kann mit der Interface-Funktion **RiAmbient()** gesetzt werden.

float area (point P)

Liefert die Fläche der differentiellen Oberfläche:

length((**Du**(*P*)**du*)^(**Dv**(*P*)**dv*))

point bump (string name[floatchannel], point norm, point dPds, point dPdt, *textcoords...)*

Funktion für das ‘Bump Mapping’. Die zweidimensionalen Texturkoordinaten *textcoords* sind entweder wegzulassen (es werden dann die Standard-Texturkoordinaten verwendet) oder können als

float s, t;
oder
float s1, t1, s2, t2, s3, t3, s4, s4;

angegeben werden. *name* ist der Name einer Textur, die mit **RiMakeBump()** erzeugt wurde. Der *channel* (mit den eckigen Klammern) ist optional. Er gibt den Kanal an, aus dem die Tiefenwerte der 'Texture Map' gelesen werden, sein Defaultwert ist 0. Die Werte von *N*, *dPds* und *dPdt* bilden ein lokales Linkssystem, sodaß die durch *dPds* und *dPdt* aufgespannte Ebene, am aktuellen Punkt die Tangentialebene zur Oberfläche bildet. Der rückgegebene Wert der Funktion kann zur Oberflächennormale addiert werden um ein Turbulenz zu erhalten:

```
N += bump("bumps", N, dPdu, dPdv);
```

Unterstützt ein System das 'Bump Mapping' nicht, wird die Funktion immer den Vektor $(0, 0, 0)$ (entspr. keiner Turbulenz) liefern.

In der Parameterliste können zusätzliche Angaben (wie im RIB-Code) gemacht werden (s. [PixSpec], 'Texture Mapping Functions'), die Angaben, die in den entsprechenden **RiMake...** Funktionen gemacht wurden, werden dadurch teilweise überdefiniert:

Name	Beschreibung
`"fidelity"`	Genauigkeit des berechneten Werts
`"samples"`	Die effektive Sample-Rate beim Filtern
`"swidth"`	Die Anzahl der in s-Richtung aus der Umgebung des betrachteten Punktes in den Filter einfließenden Punkte
`"twidth"`	Die Anzahl der in t-Richtung aus der Umgebung des betrachteten Punktes in den Filter einfließenden Punkte

point calculatenormal (point P)

Berechnet die Normale am Punkt *P* des Objektkoordinatensystems und liefert den entsprechenden Einheitsvektor:

Du(*P*)^**Dv**(*P*)

Die Funktion wird normalerweise nach einem 'Displacement' aufgerufen um die Normale *N* wieder herzustellen.

float comp (color c, float index)

Liefert die Komponente *index* von *c*.

float depth (point P)

Liefert die Tiefe des Punktes *P*, gegeben in Kamerakoordinaten. Die Tiefe liegt zwischen 0 auf der vorderen Clip-Ebene und 1 auf der hinteren.

color diffuse (point norm)

Liefert die diffuse Komponente des Beleuchtungsmodells.

float distance (**point p1, point p2)**

Berechnet die Entfernung von *p1* nach *p2*.

float environment (string name[floatchannel]**, texture coords...)**

color environment (string name[floatchannel]**, texture coords...)**

Liefert einen gefilterten Wert aus einer Environment-Textur. Die *texture coords* werden entweder als:

point R; /* Punkt auf der Oberfläche */

oder als

point R1, R2, R3, R4; /* Punkt, Richtungen im Raum (x,y,z Koordinatensystem) */

angegeben. Der Rückgabewert der Funktion hängt von dem Typ der Variablen ab, an die er zugewiesen wird. Der Startkanal (Default: 0) ist optional. Die Anzahl der ausgelesenen Kanäle hängt vom Rückgabetyp ab. Der Name der 'Envionment Map'-Datei ist der, der in **RiMake...Environment()** vergeben wurde. Die Parameter entsprechen denen von **bump()**.

point faceforward (point N, point I)

Dreht Vektor *N* so um, daß er in die entgegengesetzte Richtung des Vektors *I* zeigt.

fresnel (point I, point N, float eta, float Kr, float Kt, [point R, point T]**)**

Liefert den Reflexionskoeffizienten *Kr* und den Transmissionskoeffizienten *Kt* nach der Fresnel Formel. *I* ist der einfallende Vektor, *N* die Normale der Ebene, *eta* der Quotient aus den beiden Brechungsindizees des Volumens aus dem der Inzidenzvektor kommt und in das er hineinragt. Optional kann der reflektierte und der, nach dem Snelliusschen Brechungsgesetz **refract()** berechnete, transmittierte (gebrochene) Vektor geliefert werden.

float incident (string name, value)

Zugriff auf die Volumenvariable *name* eines Volumen-Shaders, der an die Oberfläche gebunden ist. Es wird das Volumen verwendet, aus dem der Inzidenzvektor *I* kommt. Existiert die Variable und ist vom richtigen Typ (entsprechend *value*: **float**, **color** oder **point**), liefert die Funktion ihren Wert in *value* und gibt 1 zurück; im Fehlerfall gibt sie 0 zurück.

float length (point V)

Liefert die Länge des Vektors *V*.

color mix (color color0, color color1, float a)

Liefert den interpolierten Farbwert:

`(1-`*value*`) * color0 + value * color1`

float normalize (point V)

Liefert die normalisierte Form (Länge = 1) des Vektors V.

float opposite (string name, value)

Zugriff auf die Volumenvariable *name*, aus dem Volumen-Shader, der an die Oberfläche gebunden ist. Es wird das Volumen verwendet, in das der Inzidenzvektor I zeigt. Existiert die Variable und ist vom richtigen Typ (entsprechend *value*: **float**, **color** oder **point**), liefert die Funktion ihren Wert in *value* und gibt 1 zurück; im Fehlerfall gibt sie 0 zurück.

color phong (point norm, point eye, float size)

Implementiert das (spekulare) Phong Beleuchtungsmodell.

printf (string format, val1, . . . , valN)

Schreibt auf die Standard-Ausgabe (wie **printf()** in C). Für die Formatangaben können `"%f"` (für **float**), `"%p"` (für **point**), `"%c"` (für **color**) und `"%s"` (für **string**) verwendet werden.

point reflect (point I, point N)

Liefert den Reflexionsvektor zu einer Normale N und einem Inzidenzvektor I

point refract (point I, point N, float eta)

Liefert den Inzidenzvektor I, nachdem er an einer Oberfläche mit der Normalen N gebrochen wurde. *eta* ist der Quotient aus den beiden Brechungsindizees des Volumens aus dem der Inzidenzvektor kommt und in das er hineinragt. Der Vektor wird nach dem Snelliusschen Brechungsgesetz (s.a. [Phys], Brechung und Dispersion) berechnet. Bei totaler Reflexion liefert die Funktion den Null-Vektor.

setcomp (color c, float index, float value)

Belegt die Komponente *index* $(1 - 3)$ einer Farbe c mit dem Wert *value*.

setxcomp (point P, float val)

Setzt die x-Koordinate eines Punktes P.

setycomp (point P, float val)

Setzt die y-Koordinate eines Punktes P.

setzcomp (point P, float val)

Setzt die z-Koordinate eines Punktes P.

float shadow (string name[float channel]**, texture coordinates. . .)**

Funktion für das ‘Shadow Mapping’. *name* ist der Name der Textur, die mit **RiMakeShadow()** aus einer ‘Depth Map’ erzeugt wurde. Die Angabe eines Startkanals *channel* ist optional (Default: 0). Es wird nur auf einen Kanal zugegriffen. Die Texturkoordinaten (*texture coordinates*) können entweder als

point P; /* Punkt auf der Oberfläche */

oder als

point P1, P2, P3, P4; /* Punkt, Richtungen im Raum (x,y,z) Koordinatensystem */

angegeben werden. Der Rückgabewert der Funktion ist 1, wenn ein Punkt vollkommen im Schatten liegt oder 0 wenn er vollständig beleuchtet wird. Unterstützt ein System das 'Shadow Mapping' nicht, liefert die Funktion immer 0.

color specular (point norm, point eye, float roughness)

Liefert die spekulare Komponente des Beleuchtungsmodells.

color trace (point location, point direction)

Liefert das an Punkt *location* aus der Richtung *direction* einfallende Licht. Wird *Ray Tracing* vom System nicht unterstützt, liefert diese Funktion immer die schwarze Farbe.

color texture (string name $[\mathbf{float\ channel}]$**, texture_coordinates...)**

Normales 'Texture Mapping', s. **bump()**. Die Textur muß mit **RiMakeTexture()** erzeugt worden sein.

point transform (string tospace, point P)

Transformiert einen Punkt *P* zwischen dem aktuellen `"current"` Koordinatensystem und *tospace*.

point transform (string fromspace, string tospace, point P)

Transformiert einen Punkt *P* zwischen den beiden benannten Koordinatensystem *fromspace* und *tospace*.

float xcomp (point P)

Liefert die x-Koordinate von *P*.

float ycomp (point P)

Liefert die y-Koordinate von *P*.

float zcomp (point P)

Liefert die z-Koordinate von *P*.

A.4.5 Die mathematischen Funktionen

Deriv (num, float denom)

Berechnet die Ableitung des ersten Arguments unter der Abhängigkeit des zweiten mit Hilfe der Kettenregel:

$$\mathbf{Deriv}(num, denom) = \frac{\mathbf{Du}(num)}{\mathbf{Du}(denom)} + \frac{\mathbf{Dv}(num)}{\mathbf{Dv}(denom)}$$

Mathematische Funktionen mit der Rückgabe ‘float’

Name	Funktion
sin(a)	Sinus (Parameter ist als Radiant gegeben)
asin(a)	Arcussinus
cos(a)	Cosinus
acos(a)	Arcuscosinus
tan(a)	Tangens
atan(a)	Arcustangens
atan(y, x)	Arcustangens(y/x), entspr. dem Winkel der Steigung
PI	Konstante $\pi = 3.14159\ldots$
radians(a)	Wandelt Radiant nach Grad
degrees(a)	Wandelt Grad nach Radiant
sqrt(x)	Wurzelfunktion
pow(x,y)	Potenzfunktion x^y
exp(x)	Exponentialfunktion e^x
log(x)	Natürlicher Logarithmus von x
mod(a, b)	a modulo b
abs(x)	Betrag von a
sign(x)	Signumfunktion für x: -1 (falls $x < 0$), sonst +1
min(a, b)	Minimum von a und b
max(a, b)	Maximum von a und b
clamp(a, min, max)	a in das Intervall $[a, b]$ klammern
ceil(x)	Kleinste ganze Zahl größer als x
floor(x)	Größte ganze Zahl kleiner als x
round(x)	Die zu x nächste ganze Zahl
step(min, max, val)	0 falls $val < min$, 1 falls $val \geq max$, sonst lineare Interpolation zwischen 0 und 1
smoothstep(min, max, val)	0 falls $val < min$, 1 falls $val \geq max$, sonst sanfte Hermite-Interpolation (S-Kurve) zwischen 0 und 1

Entnommen aus: [Upstill89]

Du ((float—color—point) p)

Liefert die Steigung in der parametrischen u-Richtung.

Dv ((float—color—point) p)

Liefert die Steigung in der parametrischen v-Richtung.

(float—color—point) noise (float val)

(float—color—point) noise (float u, float v)

(float—color—point) noise (point pt)

Liefert den zu einem Argument gehörenden Zufallswert. Der Rückgabewert wird entsprechend dem cast-Operator oder der Zuweisung geliefert.

(float—color—point) random ()

Liefert einen Zufallswert mit den Koeffizienten zwischen 0 und 1.

float spline (float value, float f1, float f2, . . . , float fn, float fn1)

color spline (float value, color f1, color f2, . . . , color fn, color fn1)

point spline (float value, point f1, point f2, . . . , point fn, point fn1)

Catmull-Rom Spline-Interpolierung, gibt den interpolierten Wert an der parametrischen Stelle *value* ($0 \leq value \leq 1$) zurück. Es müssen mindestens vier Kontrollpunkte vorhanden sein.

A.4.6 Der Shading Language Compiler: shader

Der Compiler kann von der Shell aus aufgerufen werden:

```
/usr/prman/shader [-Iincdir] [-o outputfile] [-s srcfile] [-v] [-q]
                  [-Uname] [-Dname] [-Dname=def] files...
```

Für Shaderquelldateien sollte das Anhängsel `.sl` gewählt werden. Der Compiler erzeugt Dateien mit der Endung `.slo`. `-v` ist die 'verbose' Option, die den Compiler dazu veranlaßt, ausführlichere Fehlermeldungen auszugeben. `-q` ist das Gegenteil davon, Meldungen werden vermieden. Da der Compiler vor der Übersetzung den C-Präprozessor `cpp` über die Quelldateien laufen läßt, sind auch die normalen Präprozessor-Optionen möglich: `-I` (Festlegen der Verzeichnisse in denen die `#include`-Dateien gesucht werden), `-U` (Definition eines Names rückgängig machen) und `-D` (Namen definieren). Die `-o` Option dient dazu, den Standard-Ausgabenamen zu überschreiben. `outputfile` kann entweder ein Dateiname oder einer der folgenden Spezialbezeichner sein:

- Ausgabe wird auf 'stdout' geschrieben

-src Der Ausgabedateiname wird aus dem der Eingabedatei gewonnen, indem der Verzeichnispfad entfernt wird und eine `.sl` Endung durch eine `.slo` Endung ersetzt wird.

-shader Standard-Einstellung: Ausgabedateiname wird durch Shadernamen und der Endung `.slo` gewonnen.

Die `-s` Option erlaubt die explizite Eingabe des Quelldateinamens zur Namesgewinnung der Ausgabedatei, wenn die Eingabe aus der Standard-Eingabe gelesen wird. Fehlen die Dateinamen `files` liest der Compiler von der Standard-Eingabe.

A.4.7 Ergänzungen zur Einbindung in C

In der Include-Datei '`ri/slo.h`' sind einige Deklarationen zu finden, die die Schnittstelle zwischen dem RenderMan C-Interface und dem Aufruf eines Shaders unterstützen. Auch 3DKit verwendet diese Deklarationen in dem **N3DShader** Objekt durch den eigenen Typ **SLOArgs**.

```
typedef enum {
    SLO_TYPE_UNKNOWN,
    SLO_TYPE_POINT,
    SLO_TYPE_COLOR,
    SLO_TYPE_SCALAR,
    SLO_TYPE_STRING,

    /* The following types are primarily used for shaders */
    SLO_TYPE_SURFACE,
    SLO_TYPE_LIGHT,
    SLO_TYPE_DISPLACEMENT,
    SLO_TYPE_VOLUME,
    SLO_TYPE_TRANSFORMATION,
    SLO_TYPE_IMAGER
} SLO_TYPE;

typedef enum {
    SLO_STOR_UNKNOWN,
    SLO_STOR_CONSTANT,
    SLO_STOR_VARIABLE,
    SLO_STOR_TEMPORARY,
    SLO_STOR_PARAMETER,
    SLO_STOR_GSTATE
} SLO_STORAGE;

typedef enum {
    SLO_DETAIL_UNKNOWN,
    SLO_DETAIL_VARYING,
    SLO_DETAIL_UNIFORM
} SLO_DETAIL;
```

```
typedef struct {
    float   xval;
    float   yval;
    float   zval;
} POINT;

typedef float SCALAR;

typedef struct slovissymdef {
    char *svd_name;
    SLO_TYPE svd_type;
    SLO_STORAGE svd_storage;
    SLO_DETAIL svd_detail;
    char *svd_spacename;
    union {
        POINT   *pointval;
        SCALAR  *scalarval;
        char    *stringval;
    } svd_default;
    union svd_defaultvalu {
        POINT   svd_pointval;
        SCALAR  svd_scalarval;
    } svd_defaultval;
    unsigned svd_valisvalid : 1;
} SLO_VISSYMDEF;

#define NULL_SLOVISSYMDEF ((SLO_VISSYMDEF *) 0)
```

B. Die 3DKit Objektstruktur

Die 3DKit Objektstruktur ist der Zugang von Objective C zum RenderMan Interface unter NeXTSTEP in Verbindung mit dem AppKit. Die Header-Dateien sind unter dem `3Dkit` Pfad zu finden. Neben den Deklarationen, die zu 3DKit gehören: `3Dkit/next3d.h` (kleines 'd'), `3Dkit/N3DCamera.h` u.s.w. — alle können mit `3Dkit/3Dkit.h` geladen werden — existiert zusätzlich die `ri/ri.h` Datei mit dem Standard C Interface-Deklarationen des RenderMans. Sie wird automatisch mit den 3DKit Header-Dateien eingebunden. Die Datei `ri/qrman.h` enthält spezielle Deklarationen für den *Quick RenderMan.* Deklarationen für die Shader-Parameter stehen in `ri/slo.h`.

Bis auf eine Ausnahme, dem Typ **SLOArgs**, beginnen alle Typen, Konstanten, globalen Variablen, Funktionen und Klassen mit '**N3D**'. Neben den 3DKit Deklarationen werden noch einige globale Deklarationen des NeXTs (u.a. der Typ **BOOL** mit seinen Werten **YES** und **NO** oder der Typ **NXColor**) verwendet.

B.1 Typen

Alle Typen, bis auf **SLOArgs**, sind in `3DKit/next3d.h` deklariert. **SLOArgs** ist in `3Dkit/3DShader.h` zu finden.

N3DProjectionType

Die unterschiedlichen Projektionstypen der **N3DCamera** sind im Aufzählungstyp **N3DProjektionType** zusammengefaßt: **N3D_Perspective**, **N3D_Orthographic**.

```
typedef enum {
    N3D_Perspective,
    N3D_Orthographic
} N3DProjectionType;
```

N3DLightType

Die Standardlichtquellen für **N3DLight** Instanzen sind im **N3DLightType** Aufzählungstyp deklariert: ambientes Licht **N3D_AmbientLight**, Punktlichtquelle **N3D_PointLight**, Lichtquelle mit parallelen Strahlen **N3D_DistantLight** und Scheinwerfer **N3D_SpotLight**.

```
typedef enum {
    N3D_AmbientLight,
```

```
    N3D_PointLight,
    N3D_DistantLight,
    N3D_SpotLight
} N3DLightType;
```

N3DAxis

Um Drehungen im dreidimensionalen Raum mit der Maus zu ermöglichen, stellt 3DKit die **N3DRotator** Klasse zur Verfügung. Die Elemente aus dem Aufzählungstyp **N3DAxis** können zur Aktivierung der Drehachsen verwendet werden: **N3D_AllAxes**, **N3D_XAxis**, **N3D_YAxis**, **N3D_ZAxis**, **N3D_XYAxis**, **N3D_XZAxis**, **N3D_YZAxis**.

```
typedef enum {
    N3D_AllAxes,
    N3D_XAxis,
    N3D_YAxis,
    N3D_ZAxis,
    N3D_XYAxes
    N3D_XZAxis,
    N3D_YZAxis
} N3DAxis;
```

N3DHider

Der Hidden-Surface Algorithmus kann mit Elementen aus **N3DHider** bestimmt werden: **N3D_HiddenRendering** (Verwendung des Z-Buffer Hidden-Surface-Algorithmus), **N3D_InOrderRendering** (Shapes in der Reihenfolge ihrer Erzeugung rendern), **N3D_NoRendering** (Shape-Hierarchie nicht ausgeben, es wird kein Bild produziert). Der ‘Hider’ wird durch die **setHider:** Methode des **N3DCamera**-Objekts gewählt.

```
typedef enum {
    N3D_HiddenRendering = 0,
    N3D_InOrderRendering,
    N3D_NoRendering
} N3DHider;
```

N3DShapeName

Die Struktur **N3DShapeName** kann ID und Namen einer Grafikfigur beinhalten.

```
typedef struct {
    char id[6];
    char *name;
} N3DShapeName;
```

N3DSurfaceType

N3DSurfaceType nimmt Einfluß auf die Qualität des Renderns. Es kann entweder in Form einer Punktwolke **N3DPointCloud**, als Gitternetz **N3D_WireFrame**, schattiertes Gitternetz **N3D_-ShadedWireFrame**, mit flachem Shading **N3D_FacetedSolids** oder interpolierendem Shading **N3D_SmoothShading** gerendert werden.

```
typedef enum {
    N3D_PointCloud = 0,
    N3D_WireFrame,
    N3D_ShadedWireFrame,
    N3D_FacetedSolids,
    N3D_SmoothSolids
} N3DSurfaceType;
```

SLOArgs

In einem Feld vom Typ **SLOArgs** kann Wert und Typ einer Shader-Instanzvariablen gespeichert werden. Die Deklaration von **SLO_VISSYMDEF** kann in `ri/slo.h` gefunden werden.

```
typedef struct {
    SLO_VISSYMDEF symb;
    union {
        float   fval;
        RtPoint pval;
        NXColor cval;
        char    *sval;
    } value;
} SLOArgs;
```

B.2 Konstanten

In `3Dkit/N3DShape` sind symbolische Konstanten definiert, die den Update-Status der aktuellen CTM (composite transformation matrix) eines Bildelements und ihrer Inversen repräsentieren: **N3D_BOTH_CLEAN**, **N3D_CTM_DIRTY**, **N3D_CTM_INVERSE_DIRTY**, **N3D_CTM_BOTH_DIRTY**.

B.3 Globale Variablen

In `3Dkit/next3d.h` sind die konstante homogene Einheitsmatrix ‘`const RtMatrix N3DIdentityMatrix`’:

```
{{1,0,0,0},
 {0,1,0,0},
 {0,0,1,0},
 {0,0,0,1}}
```

der Ursprung '`const RtPoint N3DOrigin`' $\{0, 0, 0\}$ und der Identifier des NeXT 'Pasteboards' für RIB-Daten '`NXAtom N3DRIBPboardType`' als '**extern**' Variablen deklariert.

B.4 Funktionen

3DKit definiert einige Hilfsfunktionen zur Unterstützung der Methoden. Einige von ihnen sind in `3Dkit/next3d.h` als `#define` Makros geschrieben; sie beginnen mit **N3D_**.

Zugriff auf Datenkomponenten

void N3D_XComp (RtFloat *theVector):

Liefert die X-Komponente eines Vektors oder Punkts *theVector*.

void N3D_YComp (RtFloat *theVector):

Liefert die Y-Komponente von *theVector*.

void N3D_ZComp (RtFloat *theVector):

Liefert die Z-Komponente von *theVector*.

void N3D_WComp (RtFloat *theVector):

Liefert die homogene Komponente von *theVector* (4. Komponente).

Daten konvertieren

void N3D_ConvertBoundToPoints (RtBound theBound, RtPoint *thePoints):

Schreibt den Ursprung ($\{x_{min},\ y_{min},\ z_{min}\}$) einer RenderMan Bounding Box nach *thePoints*$[0]$ und seine Ausdehnung ($\{x_{max},\ y_{max},\ z_{max}\}$) nach *thePoints*$[1]$.

void N3D_ConvertPointsToBound (RtPoint *thePoints, RtBound theBound):

Erzeugt aus zwei Punkten: Ursprung *thePoints*$[0]$ und Ausdehnung *thePoints*$[1]$ einer Bounding Box die RenderMan Notation:

$$\{x_{min},\ y_{min},\ z_{min},\ x_{max},\ y_{max},\ z_{max}\}$$

Daten kopieren

void N3D_CopyBound (RtBound sourceBound, RtBound destBound):

Kopiert *sourceBound* nach *destBound*.

void N3D_CopyMatrix (RtMatrix sourceMatrix, RtMatrix destMatrix):

Kopiert *sourceMatrix* nach *destMatrix*.

void N3D_CopyPoint (RtPoint sourcePoint, RtPoint destPoint):

Kopiert *sourcePoint* nach *destPoint*.

Schnitt von Gerade und Ebene
void N3DIntersectLinePlane (RtPoint *endPoints, RtPoint planeNormal, RtPoint planePoint, RtPoint *intersection):

Die Funktion versucht den Schnittpunkt der Geraden durch *endPoints*$[0]$ und *endPoints*$[1]$ und einer Ebene zu berechnen. Das Ergebnis wird in **intersection* geliefert. Ist die Gerade parallel zur Ebene, setzt die Funktion alle drei Komponenten von **intersection* auf **NaN**. Die Ebene ist durch *planePoint*, einen Punkt auf der Ebene und *planeNormal* einen Punkt im Raum derart gegeben, daß der Vektor $planeNormal - planePoint$ senkrecht auf der Ebene steht.

Matrizenmanipulation
void N3DMultiplyMatrix (RtMatrix preTransform, RtMatrix postTransform, RtMatrix resultTransform):

Matrizenmultiplikation: $resultTransform = preTransform \times postTransform$.

float N3DInvertMatrix (RtMatrix theTransform, RtMatrix theInverse):

Liefert die Determinante von *theTransform* als Rückgabe und die Inverse der Matrix in *theInverse*.

Transformationen
void N3DMult3DPoint (RtPoint thePoint, RtMatrix theTransform, RtPoint newPoint):

Multipliziert einen Punkt mit einer Matrix:

$$newPoint = thePoint \times theTransform$$

void N3DMult3DPoints (RtPoint *thePoints, int pointCount, RtMatrix theTransform, RtPoint *newPoints):

Multipliziert ein Array von Punkten mit einer Matrix:

$$newPoint_i = thePoint_i \times theTransform,\ 0 \leq i < pointCount$$

B.5 Objektklassen

B.5.1 N3DCamera

Erbt von: `View:Responder:Object`

Deklariert in: `3DKit/N3DCamera.h`

Dieses Objekt erlaubt die Darstellung 3-dimensionaler Szenen innerhalb eines Fensters, deren Ausgabe auf einen Drucker und bildet eine Schnittstelle zur Eventbehandlung vom AppKit, dem

objektorientierten Zugang zur Benutzeroberfläche. **N3DCamera** ist bezogen auf die Blockstruktur des RenderMan Interfaces für alle Blöcke außerhalb des Weltblocks zuständig. Der Aufruf von **RiWorldBegin()** und **RiWorldEnd()** selbst und das Setzen der Default-Attribute, wird ebenfalls vom Kameraobjekt ausgeführt. Für die verschiedenen Programmbereiche sind unterschiedliche Methoden zuständig. Die Kamera bestimmt den Renderer, der verwendet werden soll: *Quick RenderMan* oder `prman`. Mit Hilfe des letzteren kann eine photorealistische Ausgabe im EPS- oder TIFF-Format erzeugt werden.

Anders als bei Nicht-3D-Views dient die Methode **drawSelf::** der Kamera nicht zur Ausgabe von 2D PostScript Grafik. Sie steuert die 3D-Ausgabe einer **N3DShape**-Objekthierarchie mit dem RenderMan und die anschließenden PostScript Ausgaben. PostScript Ausgaben können in einer überdefinierten **drawPS::** Methode erzeugt werden.

Ein und dieselbe Objekthierarchie kann von mehreren Kameras mit unterschiedlichen Einstellungen gerendert werden. Das ‘World Shape’ kann schon in der Methode **initFrame:** gesetzt werden.

Die Standardposition des Augenpunkts der Kamera liegt an den Weltkoordinaten (0, 0, 0). Die Kamera ist auf den Punkt (0, 0, 1) ausgerichtet. 3DKit benutzt das linkshändige Koordinatensystem, das auch im RenderMan Interface standardmäßig eingestellt ist. Aus dem Vektor vom Augenpunkt zum Ausrichtungspunkt wird die u-Achse des Kamerakoordinatensystems gebildet. Die Kamera kann um dieses Achse rotiert werden (linke Hand Regel: positiver Winkel für eine Drehung im Uhrzeigersinn). Die Achsen des Kamerakoordinatensystems (s:horizontal, t:vertikal, u:Tiefe) bilden standardmäßig ein kartesisches System. Der Ursprung ist der Augenpunkt.

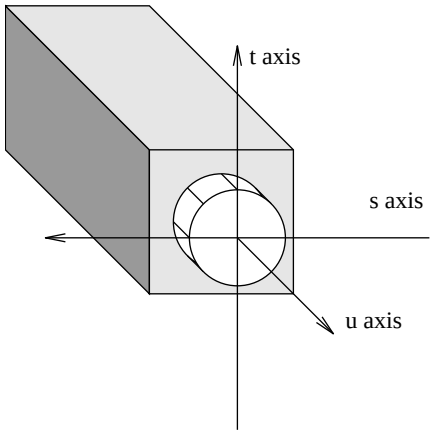

Abbildung B.1: Das Kamerakoordinatensystem

Die Kamera verwaltet eine Liste mit globalen Lichtquellen, die, falls sie nicht zur **N3DShape**-Hierarchie gehören, relativ zur Kamera plaziert werden können.

Die Kamera legt den Algorithmus zum Entfernen verdeckter Oberflächenteile fest (Hider). Je nach Oberflächentyp und gewünschter Geschwindigkeit kann ein Algorithmus gewählt werden:

Hider	Oberfläche	Geschwindigkeit
N3D_HiddenRendering	N3D_SmoothSolids, N3D_FacetedSolids	langsam
N3D_InOrderRendering	N3D_ShadedWireFrame, N3D_WireFrame, N3D_PointCloud	schneller
N3D_NoRendering	alle	am schnellsten

Instanzvariablen

unsigned int globalWindowNum: Default: 0, globale Fensternummer des Servers

RtToken windowResource: RenderMan Ressource Token des Kamera-Ausgabefensters

N3DProjectionType projectionType: Default: **N3D_Prespective**, aktueller Projektionstyp der Kamera

RtToken contextToken: RenderMan Kontext der Kamera

id worldShape: Default: Eine **N3DShape**-Instanz, Wurzel der zu rendernden Grafikobjekt-Hierarchie

List *lightList: Default: Leere Liste, Liste mit globalen **N3DLight**-Lichtquellen

id delegate: Default: **nil**, der 'Delegate' der Kamera — wird nach dem Fertigstellen eines photorealistischen Bildes benachrichtigt

NXColor backgroundColor: Default: **NX_COLORBLACK**, Fensterhintergrundfarbe (nur für den *Quick RenderMan* relevant)

N3DHider hider: Default: **N3D_HiddenRendering**, Typ des aktuellen Hidden-Surface-Algorithmus

struct _cameraFlags cameraFlags: Zustand der Kamera

```
struct cameraFlags {
    unsigned int degenerate:1;
    unsigned int windowChanged:1;
    unsigned int needsWindow:1;
    unsigned int basisChanged:1;
    unsigned int canRender:1;
    unsigned int usePreTM:1;
    unsigned int doFlush:1;
    unsigned int inWorldBlock:1;
    unsigned int drawBackground:1;
};
```

degenerate: **YES**, bei einer Projektionsfläche 0

windowChanged: **YES**, falls das Kamerafenster seit dem letzten Rendern verändert wurde

needsWindow: **YES**, falls die Kamera noch nicht in eine Fenster-View-Hierarchie integriert wurde

basisChanged: **YES**, falls sich die Kamerakoordinaten seit dem letzten Rendern verändert haben

canRender: **YES**, falls alle Ressourcen zum Rendern vorhanden sind.

usePreTM: Default **NO**, **YES** falls links and die Transformationsmatrix *transform* vor dem Rendern die *preTransform* Matrix multipliziert werden soll

doFlush: Default **YES**, **YES** falls mittels **flushRIB** auf das Beenden des interaktiven Renderns gewartet werden soll

inWorldBlock: **YES**, falls sich der Programmablauf innerhalb des Weltblocks befindet

drawBackground: Default **YES**, **YES** falls der Fensterhintergrund vor dem Rendern mit der Hintergrundfarbe gefüllt werden soll

struct _projectionRectangle projectionRectangle: Koordinaten der Projektionsfläche (das RenderMan Screen Window)

```
struct _projectionRectangle {
    float l, r, t, b;
};
```

Default: $\{-1.0, 1.0, 1.0, -1.0\}$

RtPoint eyePoint: Default: $\{0.0, 0.0, 0.0\}$ (**N3DOrigin**) Kameraposition, Augenpunkt

RtPoint viewPoint: Default: $\{0.0, 0.0, 1.0\}$ View-Punkt, auf den die Kamera ausgerichtet ist

float rollAngle: Default: 0.0, Drehwinkel der Kamera um ihre u-Achse

float fov: Default: 40.0, Öffnungswinkel der Kamera, Winkel des Apexs des Frustums

float pixelAspectRatio: Default: 1.0, Seitenverhältnis der Pixel

float nearPlane: default: 0.01, Enfernung der vorderen Clip-Ebene zum Augenpunkt

float farPlane: Default: 1000.0, Enfernung der hinteren Klippebene zum Augenpunkt

RtPoint sBasis: Default: $\{1.0, 0.0, 0.0\}$, Basis des Kamerakoordinatensystems (horizontal). Werden die Standardwerte der Basisvektoren belassen, entsprechen die Einheiten des Kamerakoordinatensystems denen im Weltkoordinatensystem.

RtPoint tBasis: Default: $\{0.0, 1.0, 0.0\}$, Basis des Kamerakoordinatensystems (vertikal)

RtPoint uBasis: Default: $\{0.0, 0.0, 1.0\}$, Basis des Kamerakoordinatensystems (Tiefe)

RtMatrix preTransform: Default: **nil**, Matrix für die Prämultiplikation

RtMatrix transform: Default: **N3DIdentityMatrix**, Matrix für die Welt-zur-Kamera Projektion

Methoden

Initialisierung und Freigabe

– init

Ruft `[self initFrame:NULL]` auf und liefert deren Rückgabe. **init:** braucht im Gegensatz zu **initFrame:** in abgeleiteten Objekten nicht überdefiniert werden.

– initFrame :(const NXRect *)fRect

Ruft die **initFrame:** Methode der Superklasse auf, setzt den Renderkontext (wird von **RiBegin()** geliefert) und initialisiert anschließend die anderen Instanzvariablen mit ihren Standardwerten. Die Methode liefert **self**.

initFrame: kann überdefiniert werden, um das Weltobjekt zu erzeugen, Kameraposition, globale Lichtquellen, Klippebenen und andere Optionen zu setzen. Mit Frame ist in diesem Zusammenhang der Rechteck-Rahmen der View gemeint und nicht der RenderMan Frame. Der Rückgabewert dieser Methode ist **self**.

– free

Freigabe von einer **N3DCamera** Instanz und deren zugehörenden Speicher außer der Shape-Hierarchie und der Liste mit den globalen Lichtquellen. Die Methode kann überdefiniert werden. **free** ruft am Ende die Methode [`super free`] und liefert auch deren Wert.

Alles zeichnen

– (BOOL)lockFocus

Den PostScript Fokus auf die View setzen. Anschließend folgt der Aufruf der **RiDisplay()** Interface-Routine, um für die 3D-Ausgabe das Fenster und den Frame-Buffer des Kameraobjekts festzulegen. Das Projektionsrechteck wird für die aktuelle Fenstergröße und die Kamera-Ausrichtung festgelegt. Die Methode liefert den Wert von `[super lockFocus]` und braucht nicht überdefiniert werden.

– unlockFocus

Beenden der 3D-Ausgabe und Freigeben des Fokus durch einen Aufruf von [`super unlockFocus`], deren Wert auch geliefert wird. Die Methode braucht nicht überdefiniert werden.

– drawSelf :(NXRect *)rects :(int)nRects

Steuerung der 3D und PostScript Grafikausgabe. Zuerst wird der Hintergrund ausgegeben (falls die Instanzvariable $doesDrawBackgroundColor == \mathbf{YES}$). Anschließend werden kameraspezifische 3D-Ausgaben und das Weltobjekt durch einen Aufruf der Methode **render** geklippt ausgegeben. Zum Schluß wird **drawPS::** für 2D PostScript Ausgaben aufgerufen.

Kameraspezifische 3D-Ausgaben können durch das Überdefinieren der **renderSelf:** Methode, PostScript Ausgaben durch das Überdefinieren von **drawPS::** gemacht werden.

Im Gegensatz zu anderen View-Objekten braucht **drawSelf::**, die im Kameraobjekt die Ausgabe steuert, selbst aber keine Ausgaben macht, nicht überdefiniert werden.

Zeichnen

– render

Steuerung der 3D-Ausgabe eines RenderMan Frames. Die Methode öffnet einen Frame durch den Aufruf der Interface-Routine **RiFrameBegin**(*frameNumber*) und einen Weltblock durch den Aufruf der Methode **worldBegin:**. Danach werden kameraspezifische 3D Ausgaben (**renderSelf::**) gemacht, die globalen Lichtquellen gerendert und die Objekthierarchie (Weltobjekt und Nachfolger) durch einen Aufruf der **render** Methode des Weltobjekts ausgegeben. Nach der Ausgabe wird der Weltblock durch **worldEnd:** und der Frame durch **RiFrameEnd()** geschlossen. Falls vor dem Aufruf von **render** die Methode **setFlushRIB:** mit dem Argument **YES** aufgerufen wurde (ist der Standard), wird nach der 3D-Ausgabe mittels **flushRIB** auf das Beenden des Renderns gewartet; anschließende PostScript Ausgaben werden so nicht gestört. Die Methode liefert **self**.

– renderSelf :(RtToken)Context

Abstrakte Methode, die überdefiniert werden kann, um kameraspezifisches Rendern am Anfang des Weltblocks auszuführen. Es können z.B. Standard-Attribute gesetzt werden. Die Methode liefert immer **self**.

– setFlushRIB :(BOOL)flag

Wird die Methode mit dem Wert **YES** für *flag* aufgerufen, sendet **render** nach der eigentlichen 3D-Ausgabe noch ein **flushRIB**, um auf das Beenden des Renderns zu warten, damit folgende Fensterausgaben nicht gestört werden. Die Methode liefert **self**.

– (BOOL)doesFlushRIB

Liefert **YES**, wenn am Ende der **render** Methode auf das Ende des Renderns gewartet wird. Default ist **YES**.

– flushRIB

Wartet auf das Beenden der *Quick RenderMan*-3D-Ausgabe des Kamerakontextes durch ein ‘flushen’ der RIB-Pipeline. Die Methode liefert **self**. Sie entspricht der **NXPing()** Funktion von NeXTSTEP.

PostScript Ausgabe

– drawPS :(NXRect *)rects :(int)nRects

Diese abstrakte Methode kann überdefiniert werden, um nach der 3D-Ausgabe im Kamerafenster PostScript Ausgaben zu machen. Es können Kamerarotation, -position und -richtung oder Rechteckhüllen von selektierten Objekten mit Kontrollpunkten angezeigt werden. Die Methode entspricht der **drawSelf::** Methode in Nicht-3DKit Views. Der Rückgabewert ist immer **self**.

Hintergrundfarbe

– setBackgroundColor :(NXColor)c

Setzen der Fensterhintergrundfarbe; liefert **self**.

– (NXColor)backgroundColor

Liefert die aktuelle Fensterhintergrundfarbe.

– setDrawBackgroundColor :(BOOL)flag

Ein **flag** mit dem Wert **YES** bewirkt, daß der Fensterhintergrund vor der eigentlichen Grafikausgabe mit der Hintergrundfarbe gefüllt wird; liefert **self**.

– (BOOL)doesDrawBackgroundColor

Liefert **YES**, wenn der Hintergrund des Fensters vor der eigentlichen Grafikausgabe mit der Hintergrundfarbe gefüllt wird.

PostScript Transformationsverwaltung

– setFrame :(const NXRect *)fRect

– moveTo :(NXCoord)x :(NXCoord)y

– moveBy :(NXCoord)deltaX :(NXCoord)deltaY

– sizeTo :(NXCoord)width :(NXCoord)height

– sizeBy :(NXCoord)deltaWidth :(NXCoord)deltaHeight

Überdefinierte Methoden, um das Verhältnis von den 2D PostScript Koordinaten im Inneren des Fensters (der Superview) zum 3D RenderMan Koordinatensystem zu berechnen. Die Methoden rufen die gleichnamigen Methoden von **super** und teilen dem Interface mittels **RiDisplay()** die richtigen Daten für das Projektionsrechteck mit. Die Methoden liefern **self**.

– rotateTo :(NXCoord)angle

– rotateBy :(NXCoord)deltaAngle

Überschrieben, um zu verhindern, daß das PostScript Koordinatensystem gedreht wird. Liefern **self**.

Grafikobjekt Hierarchie

– setWorldShape :a3DObject

Ersetzt durch ein Einsetzen eines Weltobjekts die Objekthierarchie. Der Oberflächentyp und der Hiddensurface-Algorithmus müssen explizit durch einen Aufruf von **setSurfaceTypeForAll:chooseHider:** gesetzt werden. Die Funktion liefert das alte Weltobjekt.

– worldShape

Liefert die Wurzel der zu rendernden Objekthierarchie.

Globale Beleuchtung
– addLight :aLight

Das von **N3DLight** abgeleitete *aLight* wird in die Liste der globalen Lichtquellen der Kamera eingetragen. Globale Lichtquellen werden immer als erste Objekte gerendert und können so potentiell die gesamte Szene beleuchten. *aLight* kann, braucht aber nicht Bestandteil der Objekthierarchie sein. Der Instanz *aLight* wird durch die `[aLight setGlobal:YES]` Message mitgeteilt, daß sie nun global ist. Die Methode liefert **self**.

– removeLight :aLight

Fügt *aLight* aus der Lichtliste aus und sendet eine `[aLight setGlobal:NO]` Message.

– lightList

Die Methode liefert die Lichtliste, die **N3DLight** Instanzen enthält. Die Liste kann dazu verwendet werden, Operationen auf allen Listenelementen zu starten: Z.B. können alle Lichtquellen durch das Senden einer **switchLight:** Methode an- oder ausgeschaltet werden.

Picken
– selectShapesIn :(const NXRect *)selectionRect

Liefert eine Liste aller 3D-Objekte, die innerhalb der 2D PostScript Koordinaten *selectionRect* sichtbar und selektierbar ($[aN3DShape\ isSelectable] == YES$) sind. Die Kamera verwaltet nur eine Liste. Nach einem erneuten Aufruf dieser Methode wird sie überschrieben. Die Liste darf nicht freigegeben werden.

Projektionsrechteck
– setProjectionRectangle :(float)left :(float)right :(float)top :(float)bottom

Setzt das Projektions-Rechteck der Kamera in Interface-Koordinaten. Die Werte werden als Parameter für den Aufruf der Interface-Funktion **RiScreenWindow()** verwendet. Nach einem Aufruf dieser Methode passen sich die Screen Window Koordinaten nicht mehr an ein verändertes Fenster an! Die Defaultkoordinaten des Rechtecks können durch die folgende Methode eines abgeleiteten Kameraobjekts bestimmt werden:

```
- resetProjectionRectangle
{
    NXRect r;
    float aRatio, defaultRect[4];

    [self getFrame:&r]; // PostScript Koordinaten der View holen

    // Laengste Seite hat immer die
    // Screen Window Koordinaten: -1 - 1
```

```
    if ( r.size.width > r.size.height ) {
        defaultRect[0] = -1;
        defaultRect[1] = 1;
        aRatio = r.size.height / r.size.width;
        defaultRect[2] = aRatio;
        defaultRect[3] = -aRatio;
    } else {
        defaultRect[2] = 1;
        defaultRect[3] = -1;
        aRatio = r.size.width / r.size.height;
        defaultRect[0] = -aRatio;
        defaultRect[1] = aRatio;
    }

    [self setProjectionRectangle:defaultRect[0] :defaultRect[1]
        :defaultRect[2] :defaultRect[3]];

    return self;
} // resetProjectionRectangle
```

Die Methode liefert **self**.

– getProjectionRectangle :(float *)left :(float *)right :(float)top :(float *)bottom

Liefert die Ausmaße des Projektionsrechtecks über die Referenz-Parameter, **self** ist der Rückgabewert.

Projektionstyp

– setProjection :(N3DProjectionType) aProjectionType

Setzt den Projektionstyp der Kamera: **N3D_Perspective** oder **N3D_Orthographic**; liefert **self**.

– (N3DProjectionType)projectionType

Liefert den aktuellen Projektionstyp der Kamera.

Prätransformationsmatrix

– setPreTransformMatrix :(RtMatrix)newPreTM

Setzt eine 3D-Transformationsmatrix (Prätransformationsmatrix), die auf die Kamera vor der eigenen Transformationsmatrix angewendet wird.

– getPreTransformMatrix :(RtMatrix)preTM

Liefert die Prätransformationsmatrix in *preTM* und **self** als Rückgabewert.

– setUsePreTransformMatrix :(BOOL)flag

Mit dem Wert **YES** für *flag* wird die Verwendung der Prätransformationsmatrix eingeschaltet.

– (BOOL)usesPreTransformMatrix

Liefert **YES** wenn die Prätransformationsmatrix der Kamera verwendet werden soll.

Ausrichtung der Kamera

– setEyeAt :(RtPoint)fromPoint toward:(RtPoint)toPoint roll:(float) aRollAngle

Festlegen von Kameraposition *fromPoint* und View-Punkt *toPoint* in Weltkoordinaten. Der Drehwinkel der Kamera *aRollAngle* wird in der Standardtransformationsmatrix der Kamera abgelegt und angewendet, um die Kamera nach ihrer Positionierung um *aRollAngle* Grad um den Augenvektor zu drehen.

– getEyeAt :(RtPoint *)anEyePoint toward:(RtPoint *)aViewPoint roll:(float *) aRollAngle

Liefert **self** als Rückgabewert, sowie Kamera-Position, View-Punkt ($aViewPoint - anEyePoint$ ist der Ausrichtungsvektor) und Drehwinkel über die Referenz-Parameter.

– moveEyeBy :(float)ds :(float)dt :(float)du

Verschiebt die Kamera in ihrem eigenen Koordinatensystem relativ zu ihrer Position um den Vektor (ds, dt, du); liefert **self**.

– rotateEyeBy :(float)dElev :(float)dAzim about:(RtPoint)pivotPt

Dreht die Kamera *dElev* (Elevation) Grad vertikal und *dAzim* (Azimuth) Grad horizontal um den Welt-Punkt *pivotPt*.

Clip-Ebenen

– setClipPlanesNear :(float)aNearPlane far:(float)aFarPlane

Setzt die vordere (*aNearPlane*) und hintere (*aFarPlane*) Klippebene des Frustums in Kamerakoordinaten. Es muß gelten:

$$\mathbf{RI\_EPSILON} \leq aNearPlane < aFarPlane \leq \mathbf{RI\_INFINITY}$$

Die Werte werden bei Grenzüberschreitung auf das Minimum, bzw. Maximum geklammert; liefert **self**.

– getClipPlanesNear :(float *)aNearPlane far:(float *)aFarPlane

Liefert **self** als Returnwert. Die Entfernungen der vorderen und hinteren Klippebene werden in den Referenzparametern zurückgegeben.

Abbildungsbereich

– setFieldOfViewByAngle :(float)aFieldOfView

Setzt den Öffnungswinkel (Apex des Frustums) der Kamera auf *aFieldOfView* Grad. Der Wert sollte zwischen 0 und 180 Grad liegen, 40 ist Default. Ein Verkleinern des Wertes entspricht einem Hineinzoomen in das Weltobjekt (Tele), ein Vergrößern einem Herauszoomen (Weitwinkel). Die Methode liefert **self**.

– setFieldOfViewByFocalLength :(float)aFocalLength

Setzt den Öffnungswinkel der Kamera über die Brennweite. Die Brennweite entspricht dem Abstand zwischen einem gedachten Brennpunkt und der Projektionsfläche (als gedachte Linse). Der Öffnungswinkel des Frustums kann dann entsprechend berechnet werden:

$$fov = 2 \cdot \arctan\left(\frac{Abstandder Projektionsflaeche/2}{aFocalLength}\right)$$

– (float)fieldOfView

Liefert den Öffnungswinkel der Kamera.

Pixel-Aspect-Ratio

– setPixelAspectRatio :(float)pixAspect

Setzen des Pixel-Aspect-Ratios: Verhältnis der physikalischen Breite zur Höhe eines Pixels. Der Aspect-Ratio hat bei breiten Pixeln Werte größer als Eins und bei schmalen Pixeln Werte kleiner als Eins. Der Aspect-Ratio entspricht Breite/Höhe der physikalischen Pixel. Ein Aufruf der Methode zeigte nur Wirkung, wenn das Projektionsrechteck der Kamera auf (0, 0, 0, 0) gesetzt wurde. Die Methode liefert **self**.

– (float)pixelAspectRatio

Liefert den Pixel-Aspect-Ratio, Standardwert ist 1.0.

Koordinatenkonvertierung

– convertPoints :(NXPoint *)points count:(int)npts fromSpace:aShape

Konvertiert *npts* 3D-Punkte *points* vom Objektkoordinatensystem des Objekts *aShape* in das 2-D PostScript-Koordinatensystem des Empfängers (Fenster der Kamera). Die Ergebnisse werden in die ersten beiden Komponenten des *point*-Arrays geschrieben (x, y). Rückgabewert ist **self**.

– convertPoints :(NXPoint *)mCoords count:(int)npts toWorld:(RtPoint *)wCoords

Konvertiert *npts* 2D-Punkte *mCoords* in $2 \cdot npts$ 3D-Weltkoordinaten. Der erste Punkt eines *wCoord* Paares entspricht den Koordinaten auf der vorderen Clip-Ebene, der zweite dem entsprechenden Punkt auf der hinteren Clip-Ebene. Das *wCoords* Array muß groß genug sein, um $2 \cdot npts$ Punkte vom Typ **RtPoint** aufnehmen zu können.

Rückgabewert der Methode ist **self**.

Crop Windows

– (int)numCropWindows

Diese Methode wird vor dem photorealistischen Rendern aufgerufen. Der gelieferte Wert entspricht der Anzahl der ‘Streifen’ in die ein Bild zerlegt wird, bevor es auf mehreren

Hosts gerendert werden kann. Der Wert entspricht gleichzeitig der Anzahl der beteiligten Hosts.

– cropInRects :(NXRect *)rects nRects:(int)n

In *rects* werden die Koordinaten von *n* Rechtecken geliefert, **self** dient als Rückgabewert. 3DKit teilt ein Bild, wenn es auf mehreren Hosts photorealistisch gerendert werden soll, in *n* (`[self numCropWindows]`) gleichhohe Streifen. Soll eine andere Aufteilung durchgeführt werden, muß diese Methode (evtl. auch *numCropWindows*) überdefiniert werden. Die Methode kann z.B. so umgeschrieben werden, daß nicht das Fenster, sondern die Projektion der Bounding Box des Weltobjekts zur Aufteilung herangezogen wird. Befindet sich das Weltobjekt in der Mitte des Bildschirms und bleiben die Ränder frei, kann es passieren, daß bei einer Aufteilung auf mehr als zwei Hosts nur die Rechner etwas zu tun bekommen, die den mittleren Teil rendern. Die Rechner, die den Rand bekommen, haben hingegen nichts zu tun. Das führt zu einer längeren Wartezeit als nötig.

RenderMan Frame-Nummer
– (int)frameNumber

Liefert immer 1. Der Wert der Methode dient der **RiFrame()** Funktion als Eingabe. **N3DMovieCamera** definiert diese Funktion über, um mehrere Frames rendern zu können.

Drucken
– (BOOL)canPrintRIB

Liefert **YES**, wenn die Ressourcen zum Rendern vorhanden sind.

RIB-Ausgabe
– copyRIBCode :(NXStream *)stream

Die Szenenbeschreibung wird als RIB-Code auf einen Stream *stream* geschrieben. Ziel kann z.B. eine `.rib`-Datei oder ein ‘Memory-Stream’ für ein ‘Pasteboard’ sein. Die Funktion liefert immer **self**.

Attribute für Weltblock
– worldBegin :(RtToken)context

Zweck dieser Methode ist der Aufruf der Interface-Routine **RiWorldBegin()**. Die Methode kann überdefiniert werden, um vor und direkt nach dem Aufruf dieser Interface-Routine weitere Interface-Routinen aufzurufen. Es können hier Makrodefinitionen (**RiMakroBegin()**, **RiMakroEnd()**) ausgeben werden. Die RenderMan Optionen müssen in dieser Methode vor dem Aufruf der **super**-Methode gesetzt werden. Die Methode liefert **self**.

– worldEnd :(RtToken)context

Aufruf der **RiWorldEnd()** Interface-Routine. Die Methode kann überdefinert werden, um irgendwelche Aufräumarbeiten nach dem Rendern zu starten.

Der ‘Delegate’

– setDelegate :theDelegate

Setzt das ‘Delegate’-Objekt der Kamera. Der ‘Delegate’ muß die Methode **camera:didRenderStream:tag:frameNumber:** (s.u. Methoden des ‘Delegates’ der **N3DCamera**) implementieren. Er wird nach dem photorealistischen Rendern durch diese Methode benachrichtigt. **setDelegate:** liefert **self**.

– delegate

Liefert den ‘Delegate’ der Kamera.

Hider

– setHider :(N3DHider)hider

Setzt den Typ des Hidden-Surface-Algorithmus des Renderers. Die Werte **N3D_HiddenRendering**, **N3D_InOrderRendering** oder **N3D_NoRendering** sind für *hider* gültig; liefert **self**.

– (N3DHider)hider

Liefert den Typ des aktuellen Hidden-Surface-Algorithmus (*Hider*).

– setSurfaceTypeForAll :(N3DSurfaceType)surface chooseHider:(BOOL)flag

Funktion zum Setzen des Oberflächentyps der Grafikobjekt-Hierarchie. Hat *flag* den Wert **YES** wird ein der Oberfläche entsprechend günstiger Hider gesetzt:

surface	hider
N3D_PointCloud	N3D_InOrderRendering
N3D_WireFrame	N3D_InOrderRendering
N3D_ShadedWireFrame	N3D_InOrderRendering
N3D_FacetedSolids	N3D_HiddenRendering
N3D_SmoothSolids	N3D_HiddenRendering

Die Methode liefert **self**.

Photorealistisches Rendern

– (int)renderAsEPS

Startet das photorealistische Rendern als neuen Prozeß. Die Methode liefert einen eindeutigen Integer, mit dem der ‘Delegate’ der Kamera den Renderprozeß identifizieren kann. Der ‘Delegate’ bekommt zusätzlich zu dieser Nummer den Stream mit dem fertigen Bild im EPS-Format und die zugehörende Frame-Nummer. Vor dem Start des photorealistischen Renderns wird ein **N3DRenderPanel** angezeigt, in dem der Benutzer die Hosts auf denen gerendert werden soll und die DPI-Auflösung einstellen kann. Wird im PostScript Format gerendert, wird der Hintergrund der Grafik immer Schwarz (außer bei **doesDrawbackgroundColor** == **NO**) ausgegeben, die PostScript Ausgaben aus **drawPS::** sind eingeschlossen.

– (int)renderAsTIFF

Wie oben, nur wird im TIFF-Format gerendert, PostScript Ausgaben werden nicht einkopiert.

Archivieren

– read :(NXTypedStream *)stream

Liest eine **N3DCamera** Instanz von dem *stream*; liefert **self**.

– write :(NXTypedStream *)stream

Schreibt die empfangende **N3DCamera** auf den *stream*; liefert **self**.

– awake

Wird nach dem Lesen des Archivs aufgerufen, um weitere Initialisierungen zu ermöglichen.

Methoden des 'Delegates' der N3DCamera

– camera :theCamera didRenderStream:(NXStream *)imageStream tag:(int)theJob frameNumber:(int)currentFrame

Diese Methode des 'Delegates' wird von 3DKit nach dem Beenden des photorealistischen Renderns aufgerufen. Das Rendern wurde entweder durch den Aufruf der **renderAsEPS** oder **renderAsTIFF** Methode gestartet. Durch die eindeutige Jobnummer, die diese beiden Methoden liefern und die der Delegate-Methode in *theJob* übergeben wird, kann der Typ der Daten auf dem *imageStream* leicht festgestellt werden. In *currentFrame* wird die RenderMan Frame-Nummer des Bildes übergeben. Die Delegate-Methode kann die Bilddaten in eine Datei schreiben oder in eine **NXImage** Struktur umwandeln und anschließend weiterverarbeiten lassen. Die Methode sollte **self** liefern. Wird die Applikation, die das Rendern gestartet hat vor der Fertigstellung der Bilder abgebrochen, wird diese Methode nicht aufgerufen und die Bilddaten gehen verloren. Zwischendateien bleiben in `/private/spool/render/...` stehen und müssen von Hand gelöscht werden.

B.5.2 N3DContextManager

Erbt von: `Object`

Deklariert in: `3DKit/N3DContextManager.h`

Der Kontextmanager behandelt die Verbindung vom interaktiven RenderMan zur Applikation. Er stellt eine Verbindung zum Renderer her, wenn eine Instanz des **N3DCamera** Objekts neu gezeichnet werden muß und gibt die Verbindung nach dem Rendern wieder frei.

Die Behandlung der Kontexte wird dem Anwendungsprogrammierer normalerweise von 3DKit abgenommen. Jede Applikation instanziert genau einen Kontextmanager und zu jedem Zeitpunkt ist höchstens ein Kontext aktiv.

Der Kontextmanager erzeugt durch den Aufruf der Interface-Routine **RiBegin()** Kontexte, schaltet zwischen gültigen Kontexten mittels der Interface-Routine **RiContext()** um und zerstört Kontexte durch den Aufruf von **RiEnd()**. Die Kontexte werden in einer Hash-Tabelle verwaltet. Neue Kontexte können beispielsweise erzeugt werden, um RIB-Dateien zu schreiben.

Instanzvariablen

RtToken mainContext: Hauptkontext. Das ist der Kontext der zuerst gerenderten **N3DCamera** Instanz. Der Hauptkontext kann immer durch [`[N3DContextManager new] mainContext`] abgefragt werden. Eine Managerinstanz wird durch die Klassenmethode **new** nur dann erzeugt, wenn sie noch nicht existiert. Existiert noch kein *mainContext*, wird er für den interaktiven Renderer erzeugt und selektiert.

id contextTable: Hash-Tabelle aller Kontexte

RtToken currentContext: Momentan selektierter Kontext

Methoden

Initialisierung und Freigabe

+ new

Der Methodenaufruf erzeugt, falls nötig, einen **N3DContextManager** für eine Applikation und liefert diesen als Rückgabewert.

– free

Zerstört alle noch vorhandenen Renderkontexte und löscht danach die Instanz; liefert **self**.

Hauptkontext

– (RtToken)mainContext

Liefert das Token des Hauptkontexts der Applikation. Falls dieses noch nicht existert, erzeugt der Kontextmanager ein entsprechendes Token und macht den Kontext zum aktuellen Kontext.

Erzeugung von anderen Kontexten

– (RtToken)createContext :(const char *)contextName

Erzeugt einen neuen Renderkontext für den interaktiven RenderMan und macht ihn zum aktuellen Kontext. Kann kein neuer Kontext erzeugt werden, liefert die Methode **RI_NULL** und beläßt den alten aktiven Kontext. Die Methode ruft einfach **CreateContext:withRenderer:** mit dem **withRenderer**-Argument **RI_DRAFT** auf. Der Kontext wird in der Kontexttabelle vom interaktiven Renderer unter dem Namen von **contextName** eingetragen. Ein Kontext wird erzeugt, indem **RiBegin()** mit den entsprechenden

Parametern aufgerufen wird. Das von dieser Interface-Routine gelieferte Token wird anschließend mit *contextName* als Schlüssel in der Hash-Tabelle des Kontextmanagers eingetragen und als aktueller Kontext geführt. Auch der *Quick RenderMan* (das Frontend) verwaltet den Kontext als gültigen Kontext.

– (RtToken)createContext :(const char *)contextName withRenderer:(RtToken)renderer

Erzeugt einen neuen Kontext mit dem Namen *contextName* für den Renderer *renderer*. Momentan werden folgende Renderer unterstüzt:

Typ	Funktion
RI_DRAFT	Darstellung der Szene auf dem Bildschirm
RI_ARCHIVE	Ausgabe der Szene als RIB-Code

Der Kontext für den **RI_ARCHIVE** Renderer kann auch mit der spezielleren Methode **createContext:toFile:** erzeugt werden. Die Methode **createContext:withRenderer:** erzeugt standardmäßig eine Datei `ri.rib` für die Ausgabe von **RI_ARCHIVE**. Der Kontext wird, falls er erzeugt werden konnte, zum aktuellen Kontext gemacht und dient als Rückgabewert. Konnte kein Kontext erzeugt werden, liefert die Funktion **RI_NULL**.

– (RtToken)createContext :(const char *)contextName toFile:(const char *)ribFileName

Erzeugen eines Kontextes zur Ausgabe der Szene als RIB-Datei. Konnte der Kontext erzeugt werden, wird er zum aktuellen Kontext. Alle folgenden Ausgaben werden dann auf die RIB-Datei geschrieben. Konnte der Kontext nicht erzeugt werden, liefert die Methode **RI_NULL**.

Ist *ContextName* **NULL**, wird ein eindeutiger Kontextname erzeugt. *ribFileName* sollte der vollständige Pfadname der zu erzeugenden RIB-Datei sein. Ist er **NULL** wird der RIB-Code in die Datei '`ri.rib`' in das aktuelle Verzeichnis geschrieben.

– (RtToken)createContext :(const char *)name toStream:(NXStream *)stream

In NeXTSTEP Release 3 ist diese Methode noch nicht implementiert und liefert deshalb immer **NULL**. Später soll sie wohl entsprechend **createContext:toFile:** RIB-Code auf einen Stream schreiben, der dann z.B. direkt für das 'Pasteboard' verwendet werden kann.

Verwaltung des aktuellen Kontexts

– (RtContext)setCurrentContext :(RtToken)aContext

Macht *aContext* (falls gültig) zum aktuellen Kontext. Alle folgenden Interface-Aufrufe werden danach von *Quick RenderMan* in diesem Kontext gerendert. Geliefert wird das Token des vorher aktiven Kontextes. Der Kontext wird intern durch einen Aufruf der Interface-Routine **RiContext()** umgeschaltet.

– (RtContext)setCurrentContextByName :(const char *)ContextName

Sucht in der Hash-Tabelle nach einem gültigen Kontext mit dem Namen *contextName*. Falls ein solcher gefunden werden konnte, wird er zum aktuellen Kontext. Die Funktion liefert den vorher selektierten Kontext.

– (RtContext)currentContext

Liefert das Token des aktuellen Kontexts.

Zerstören eines Kontexts

– (void)destroyContext :(RtToken)aContext

Zerstören des Kontextes *aContext*. Um einen Kontext zu zerstören, wird dieser zum aktuellen Kontext gemacht, anschließend wird die Interface-Routine **RiEnd()** aufgerufen, die das Rendern unter dem aktuellen Kontext beendet. Anschließend wird der Kontext aus der Hash-Tabelle des Managers entfernt. Zum Schluß wird der zuvor gültige aktuelle Kontext restauriert, bzw. falls dieser *aContext* war, auf **NULL** gesetzt.

– (void)destroyContextByName :(const char *)contextName

Sucht den an *contextName* gebundenen gültigen Kontext und verfährt dann wie **destroyContext:**.

Archivierung

– read :(NXTypedStream *)stream

Dearchivierung des Empfängers von einem typisierten Stream; liefert **self**.

– write :(NXTypedStream *)stream

Archivierung des Empfängers auf einen typisierten Stream; liefert **self**.

– awake

Initialisiert eine neu dearchivierte Instanz eines Kontextmanagers; liefert **self**.

B.5.3 N3DLight

Erbt von: `N3DShape:Object`

Deklariert in: `3DKit/N3DLight.h`

Dieses Objekt deckt die Standard-Shader der **RiLightSource()** Interface-Funktion ab. Es können die verschiedenen Lichtquellen, die vom Interface vorgegeben werden, in die Objekthierarchie eingegliedert werden und/oder als globale Lichtquellen verwendet werden. Es existieren folgende Lichttypen (s.a. **RiLightSource()**):

3DKit Typ	RenderMan Typ
N3D_AmbientLight	”ambientlight”
N3D_PointLight	”pointlight”
N3D_DistantLight	”distantlight”
N3D_SpotLight	”spotlight”

Die verschiedenen Methoden entsprechen den Parametern, die in der **RiLightSource()** Funktion verwendet werden und dienen zum Ein-/Ausschalten der Lichtquelle. Wird das Objekt gerendert, wird ein entsprechender **RiLightSource()** Aufruf abgesetzt. Lokale Lichtquellen beleuchten alle Objekte, die im selben Ast der Shape-Hierarchie später als die Lichtquelle folgen. Globale Lichtquellen beleuchten die gesamte Szene. Lichtquellen, die sich nicht in der Objekthierarchie befinden, können relativ zum Ursprung der Weltkoordinaten plaziert werden. Die Methode **addLight:** der **N3DCamera** kann dazu verwendet werden, die Lichtquelle in die Lichtliste einer Kamera einzufügen.

Instanzvariablen

RtToken lightHandle: RenderMan Token, daß **RiLightSource()** liefert

N3DLightType type: Lichttyp, Default: **N3D_AmbientLight**

RtPoint from: Position der Lichtquelle, relativ zum Ursprung des aktuellen Koordinatensystems (bei **N3D_AmbientLight** nicht verwendet), Default: (0, 0, 0)

RtPoint to: Punkt, in dessen Richtung das Licht strahlt (bei **N3D_AmbientLight** und **N3D_PointLight** nicht verwendet), Default: (0, 0, 1)

NXColor color: Lichtfarbe, Default: **NX_COLORWHITE**

RtFloat intensity: Intensität der Lichtquelle, Default: 1.0 (volle Intensität)

RtFloat coneangle: Ausstrahlungsbereich eines Scheinwerfers **N3D_SpotLight**, Default: 30 Grad

RtFloat conedelta: Winkel, ab dem die Strahlungsintensität des Scheinwerfers abgeblendet wird, Default: 5 Grad

RtFloat beamdistribution: Ablendrate des Scheinwerfers, Default: 2.0

lightFlags:
```
struct {
        unsigned int global : 1;
        unsigned int on : 1;
    }
```

global: **YES**, falls sich die Lichtquelle global ist, sich also in der Liste der globalen Lichtquellen einer **N3DCamera** Instanz befindet, Default: **NO**

on: **YES**, falls die Lichtquelle eingeschaltet ist. Lichtquellen sind vor dem Rendern ausgeschaltet und bleiben nach dem Rendern bis zum expliziten Abschalten eingeschaltet.

Methoden

Initialisierung

– init

Initialisiert eine **N3DLight**-Instanz mit den oben angegebenen Standardwerten; liefert **self**.

Beleuchtungstyp

– setType :(N3DLightType)aType

Setzt den Typ der Lichtquelle; liefert **self**.

– (N3DLightType)type

Liefert den Typ der Lichtquelle.

– makeAmbientWithIntensity :(RtFloat)i

Setzt als Lichtquellentyp ein ambientes Licht **N3D_AmbientLight** mit der Intensität *i*; liefert **self**.

– makePointFrom :(RtPoint)pf intensity:(RtFloat)i

Setzt eine Punktlichtquelle **N3D_PointLight** als Lichttyp an die Position *pf* mit der Intensität *i*; liefert **self**.

– makeDistantFrom :(RtPoint)pf to:(RtPoint)pt intensity:(RtFloat)i

Setzt als Typ eine Lichtquelle mit parallelen Strahlen **N3D_DistantLight** mit der Richtung von *pf* nach *pt* und der Intensität *i*; liefert **self**.

– makeSpotFrom :(RtPoint)pf to:(RtPoint)pt coneAngle:(RtFloat)ca coneDelta:(RtFloat)cd beamDistribution:(RtFloat)bd intensity:(RtFloat)i

Setzt als Lichttyp einen Scheinwerfer **N3D_SpotLight** ein. Er wird an der Stelle *pf* positioniert und strahlt zum Punkt *pt*. Der Ausstrahlwinkel beträgt *ca* Grad, ab dem Winkel *cd* wird mit einer Rate *bd* abgeblendet (s.a. **RiLightSource()**). Die Intensität bekommt den Wert von *i* zugewiesen; liefert **self**.

Behandlung der Beleuchtungsparameter

– setFrom :(RtPoint)from

Setzt die Position (*from*) der Lichtquelle (wird von **N3D_AmbientLight** nicht beachtet); liefert **self**.

– setFrom :(RtPoint)from to:(RtPoint)to

Setzt Position (*from*) und Punkt (*to*) in dessen Richtung die Lichtquelle scheint. Die Werte werden nur von den Lichtquellen beachtet (fließen in **RiLightSource()** ein), die diese Parameter benötigen; liefert **self**.

– getFrom :(RtPoint *)from to:(RtPoint *)to

Liefert die Position der Lichtquelle im Parameter *from* und den Punkt, auf den die Lichtquelle ausgerichtet ist, im Parameter *to*. Als Rückgabewert dient **self**.

– setConeAngle :(RtFloat)ca coneDelta:(RtFloat)cd beamDistribution:(RtFloat)bd

Setzt die zusätzlichen Parameter, die der Scheinwerfer **N3D_SpotLight** verwendet (s.a. **makeSpotFrom::::::**); liefert **self**.

– getConeAngle :(RtFloat *)ca coneDelta:(RtFloat *)cd beamDistribution:(RtFloat *)bd

Liefert die zusätzlichen Parameter, die der Scheinwerfer **N3D_SpotLight** verwendet, in den entsprechenden Parametern (s.a. **makeSpotFrom::::::**). Rückgabewert ist **self**.

– setIntensity :(RtFloat)i

Setzt die Intensität der Lichtquelle. Der Wert *i* soll wenn möglich im Interval von $[0, 1]$ liegen; liefert **self**.

– (RtFloat)intensity

Liefert die Intensität des Lichts.

Rendern

– renderSelf :(N3DCamera *)camera

Ist die Lichtquelle global, tut diese Methode nichts. Lokale Lichtquellen in der Objekthierarchie werden durch den Aufruf der RenderMan Routine **RiLightSource()** gerendert. Diese Routine kann für eigene Lichtquellen überdefiniert werden:

```
-renderSelf:(N3DCamera)camera {

    // Globale Lichtquellen nicht rendern
    if ( [self isGlobal] ) return self;

    // Interface-Aufrufe vor dem Rendern der Lichtquelle
    [super renderSelf:camera]; // Rendern der Lichtquelle
    // Interface-Aufrufe nach dem Rendern der Lichtquelle

    return self;
}
```

Liefert **self**.

– renderGlobal :(N3DCamera *)camera

Rendern einer globalen Lichtquellen. Die globalen Lichtquellen aus der Liste der Kamera werden vor dem Weltobjekt gerendert und beleuchten auf diese Weise die gesamte Szene. Die Methode kann nach folgendem Schema überdefiniert werden.

```
-renderGlobal:(N3DCamera)camera {

    // Interface-Aufrufe vor dem Rendern der Lichtquelle
    [super renderGlobal:camera]; // Rendern der Lichtquelle
    // Interface-Aufrufe nach dem Rendern der Lichtquelle

    return self;
}
```

Liefert **self**.

Verwaltung von globalen Lichtquellen

– setGlobal :(BOOL)flag

Diese Methode wird von der Kamera aufgerufen, wenn das Licht in die Liste der globalen Lichtquellen ein- ($flag == \mathbf{YES}$) oder ausgefügt wird. Die Methode wird nie direkt aufgerufen, kann aber überdefiniert werden, um auf das Ein- bzw. Ausfügen aus der Liste zu reagieren; liefert **self**.

– (BOOL)isGlobal

Liefert den Wert des *lightFlags.global* Flags. Ist der Wert **YES**, ist die Lichtquelle in der Liste der globalen Lichtquellen einer Kamera aufzufinden.

Ein-/Ausschalten

– switchLight :(BOOL)onOff

Schaltet das Licht ein oder aus; liefert **self**.

Farben

– setColor :(NXColor)c

Setzt die Farbe der Lichtquelle. Der Alphakanal der Farbe wird ignoriert; liefert **self**.

– (NXColor)color

Liefert die aktuelle Farbe der Lichtquelle. Der Alphakanal ist ohne Bedeutung.

Archivierung

– read :(NXTypedStream *)stream

Schreibt die Empfängerinstanz auf den *stream*; liefert **self**.

– write :(NXTypedStream *)stream

Liest die Empfängerinstanz vom *stream*; liefert **self**.

– awake

Wird nach dem Dearchivieren aufgerufen, um weitere Initialisierungen zu machen; liefert **self**.

B.5.4 N3DMovieCamera

Erbt von: `N3DCamera:View:Responder:Object`

Deklariert in: `3DKit/N3DCamera.h`

Zur Erstellung von Animationen wurde die Klasse **N3DMovieCamera** abgeleitet. Sequenzen können sowohl für den interaktiven als auch für den photorealistischen Renderer erzeugt werden. Beim Rendern können Kamera und Grafikobjekte modifiziert werden. Durch ein Überdefinieren der **renderSelf:** Methode können vor dem Renderen eines Frames in Abhängigkeit von der Frame-Nummer (*frameNumber*) die Kamera positioniert und das Weltobjekt entsprechend gesetzt werden. Mit der Methode **displayMovie** kann die Animation auf dem Bildschirm abgespielt werden. Die Animationen enstehen durch das nacheinander Darstellen von Rahmenblöcken (**RiBeginFrame()**, **RiEndFrame()**). Wie bei der **N3DCamera** kann durch den Aufruf von **renderAsEPS** und **renderAsTIFF** photorealistisch gerendert werden. Auch bei der **N3DMovieCamera** muß ein 'Delegate' die fertig gerenderte Grafik empfangen. Wird auf mehreren Hosts gerendert, bekommt jede Host nach Möglichkeit einen kompletten Rahmen zugewiesen.

Instanzvariablen

int frameNumber: Aktuelle Rahmennummer des Films, Default: 0

int startFrame: Nummer des Startrahmens des Films, Default: 0

int endFrame: Nummer des Endrahmens des Films, Default: 0

int frameIncrement: Erhöhung der Rahmennummern, Default: 1

Methoden

Initialisierung
– initFrame :(const NXRect *)fRect

Setzt die Defaultwerte der Instanzvariablen; liefert **self**.

RenderMan Ausgabe einer Seite
– render

Rendert entweder den aktuellen Frame, oder wenn gedruckt wird, die Frames, die den im Druckdialog eingestellten Seiten entsprechen; liefert **self**.

Interaktive Anzeige
– displayMovie

Zeigt die Animation auf dem Bildschirm an. Es wird der interaktive RenderMan verwendet. Die Frames werden nacheinander, von *startFrame* bis *endFrame* dargestellt. Die Rahmennummer wird jeweils um *frameIncrement* erhöht; liefert **self**.

Rahmennummern

– setFrameNumber :(int)aFrameNumber

Setzt die aktuelle Rahmennummer; liefert **self**.

– (int)frameNumber

Liefert die aktuelle Rahmennummer.

– setStartFrame :(int)start endFrame:(int)end incrementFramesBy:(int)skip

Setzt Start- und Endrahmennummer und die Rahmenerhöhung; liefert **self**.

– (int)startFrame

Liefert die Nummer des Rahmens, der in einer Animation als erstes dargestellt wird.

– (int)endFrame

Liefert die Nummer des Rahmens, der in einer Animation als letztes dargestellt wird.

– (int)frameIncrement

Liefert die Rahmenerhöhung.

Crop Windows

– (int)numCropWindows

Liefert 1, falls mehrere Frames ausgegeben werden sollen. Es wird in diesem Fall jeweils ein Frame kompett auf einer Host gerendert.

– cropInRects :(NXRect *)theRects nRects:(int)rectCount

Liefert **self** als Rückgabeparameter. Werden mehrere Frames gerendert, wird das umgebende Rechteck der Kamera in dem Referenzparameter *theRects* geliefert. *rectCount* wird nicht beachtet. Die Methode wurde überdefiniert, um vor dem Rendern auf mehreren Host das Aufteilen eines Rahmens in Streifen zu verhindern. Es wird immer ein Rahmen komplett auf einer Host gerendert. Wird nur ein Frame gerendert, entspricht die Ausführung der Methode der von **N3DCamera**.

Seitenverwaltung

– (BOOL)getRect :(NXRect *)theRect forPage:(int)thePage

Liefert **YES**, wenn für *thePage* ein Rahmen gerendert werden kann. In *theRects* wird in diesem Fall das Rechteck der Kamera zurückgegeben. Bei *frameIncrement* != 1 werden nicht alle Rahmen aus dem Bereich $>$ *startFrame* und $\leq$ *endFrame* gerendert.

– (BOOL)knowsPagesFirst :(int *)first last:(int *)last

Liefert **YES**. Die erste und letzte Nummer der Rahmen, die in einer Animation gerendert werden, werden in den Parametern zurückgegeben.

Photorealisches Rendern

– (int)renderAsEPS

In **N3DCamera** definiert; liefert **self**.

– (int)renderAsTIFF

In **N3DCamera** definiert; liefert **self**.

– (int)renderMovieAsEPSToDirectory:(char *) withSequenceName:(char *)

Noch nicht implementiert.

– (int)renderMovieAsTiffToDirectory:(char *) withSequenceName:(char *)

Noch nicht implementiert.

Archivierung

– read :(NXTypedStream *)stream

Liest den Empfänger von dem *stream*; liefert **self**.

– write :(NXTypedStream *)stream

Schreibt den Empfänger auf den *stream*; liefert **self**.

– awake

Initialisierungen nach dem Dearchivieren; liefert **self**.

Methoden des 'Delegates'

– camera :theCamera didRenderStream:(NXStream *)imageStream tag:(int)theJob frameNumber:(int)currentFrame

Siehe Beschreibung in **N3DCamera**; liefert **self**.

B.5.5 N3DRenderPanel

Erbt von: `Panel:Window:Responder:Object`

Deklariert in: `3DKit/N3DRenderPanel.h`

Dieses Panel wird angezeigt, bevor das photorealistische Rendern gestartet wird. Es erlaubt einem Benutzer interaktiv die Hosts auszuwählen, auf denen gerechnet werden soll. Zusätzlich kann die DPI-Auflösung der Ausgabe angegeben werden. Jede Applikation darf nur eine Instanz dieses Panels besitzen. Normalerweise wird das Panel nicht direkt von der Applikation behandelt. Es kann aber sein, daß sie eine 'Accessory-View' einfügt oder andere Anpassungen am Panel vornimmt.

Instanzvariablen

id browser: **NXBrowser** Instanz, die die Hosts auflistet

id nametext: Textfeld, das eine selektierte Host anzeigt

id notetext: Textfeld, das Bemerkungen über die Host anzeigt

id resolution: Textfeld zur Anzeige und Eingabe der Auflösung

char **hostnames: Pointer auf ein Feld von Namen der selektierten Hosts

id accessoryView: Optionale View, die von der Applikation in das Panel eingefügt werden kann

Methoden

Klasseninitialisierung
+ initialize

Erzeugt (wenn nötig) und initialisiert die einzige Instanz des Panels einer Applikation. Die Belegungen der Instanzvariablen werden aus der 'Defaults'-Datenbasis gelesen. Diese Methode darf nie direkt aufgerufen werden. Die Applikation erledigt den Aufruf, wenn zum ersten mal photorealistisch gerendert werden soll.

+ new

Erzeugt (wenn nötig) die einzige Instanz des Render-Panels einer Applikation. Diese Methode kann überdefiniert werden, wenn eine 'Accessory-View' zugefügt werden soll. Sie wird automatisch aufgerufen, wenn die Applikation das photorealistische Rendern anstößt.

Accessory-View
– setAccessoryView :aView

Setzt *aView* als *accessoryView* ein. Eine Applikation kann eine solche View bereitstellen, um zusätzliche Informationen anzuzeigen oder zu erfragen; liefert **self**.

– accessoryView

Liefert die optionale 'Accessory-View'.

Modalität
– (int)runModal

Startet den modalen Dialog mit dem Benutzer. Bevor das Panel angezeigt wird, wird der 'Browser' mit der Liste der Namen von den zum Rendern bereitstehenden Hosts gefüllt, die zuletzt selektierte Host wird wieder selektiert und eine Auflösung von 72 DPI wird eingestellt. Die Methode liefert 1, wenn der Benutzer den **Render**-Button aktiviert und 0, wenn der **Cancel**-Button aktiviert wird.

DPI-Auflösung
– (int)resolution

Liefert die vom Benutzer eingestellte DPI-Auflösung der Grafik.

Host Verwaltung
– (int)numSelectedHosts

Liefert die Anzahl der vom Benutzer selektierten Hosts.

– (char **)hostNames

Liefert die Liste mit den Namen der selektierten Hosts.

Browser-Delegate-Methode

– (int)browser :hostBrowser fillMatrix:mat inColumn:(int)col

Diese Methode wird vom 'Browser' *hostBrowser* aufgerufen, wenn er gefüllt, bzw. erneuert werden will. Die Methode füllt die Matrix *mat* mit den Namen der Hosts und liefert die Anzahl der Eintragungen. *col* wird nicht beachtet, da der *hostBrowser* nur eine Spalte besitzt. Die Hosts, die für ein Rendern in Frage kommen, können vom Systemadministrator mit der Applikation **RenderManager** eingestellt werden.

B.5.6 N3DRIBImageRep

Erbt von: `NXImageRep:Object`

Deklariert in: `3DKit/N3DRIBImageRep.h`

Objekt, mit dem RIB-Dateien gerendert werden können. Die RIB-Dateien müssen konform zu den RIB-Konventionen sein (s.a. RIB Konventionen, [PixSpec]). Der RIB-Code muß mit folgender Zeile beginnen:

```
##RenderMan RIB-Structure 1.0
```

Die Dateinamen müssen auf `.rib` enden. Die Dateien können auch vom 'Pasteboard' (Inhalt: **NX_RIBPasteboardType** Daten) oder von einem Stream geladen werden.

N3DRIBImageRep wird indirekt über eine **NXImage** Klasse verwendet. Dient die Datei dem interaktiven RenderMan als Eingabe, bestimmen der verwendete Hider und der Oberflächentyp die Qualität der Grafik. Der photorealistische RenderMan verwendet die in der RIB-Datei gemachten Angaben. Die Größe der Ausgabe hängt von dem **Format**-Statement der RIB-Datei ab.

Instanzvariablen

N3DHider hider: Der Hider für das interaktive Rendern,
Default: **N3D_HiddenRendering**

N3DSurfaceType surface: Oberflächentyp für das interaktive Rendern,
Default: **N3D_SmoothSolids**

NXColor backgroundColor: Hintergrundfarbe für das interaktive Rendern,
Default: **NX_COLORBLACK**

Methoden

Klassenmethoden

+ (const char * const *)imageUnfilteredFileTypes

Liefert ein **NULL** terminiertes String-Array. Das einzige Stringelement des Arrays ist `"rib"`. Die Methode wird von **NXImageRep** aufgerufen, um festzustellen, welche Dateitypen (erkennbar durch die Endungen) die **N3DImageRep** Klasse behandeln kann.

+ (const NXAtom *)imageUnfilteredPasteboardTypes

Liefert **NX_RIBPasteboardType**. Die Methode wird von **NXImageRep** verwendet, um herauszufinden, welche Datentypen die **N3DImageRep** Unterklasse behandeln kann.

+ (BOOL)canLoadFromStream :(NXStream *)ribstream

Testet ob auf dem *ribstream* RIB-Daten anliegen und liefert im positiven Fall **YES**, ansonsten **NO**. **NXImage** verwendet diese Methode, um zu testen, ob auf dem Stream RIB-Daten anliegen.

Deklarierung und Freigabe

– init

Generiert eine Fehlermeldung. Zur Initialisierung muß eine der beiden anderen Initialisiermethoden verwendet werden; liefert **self**.

– initFromFile :(const char *)ribFile

Initialisiert den frisch allokierten Empfänger mit der RIB-Repräsentation aus der Datei *ribFile*. Schlägt die Initialisierung fehl, liefert diese Methode **nil**, sonst **self**. Beim Initialisieren wird nur der Dateityp festgestellt und, wenn möglich, der **Format**-Befehl ausgewertet.

– initFromStream :(NXStream *)ribStream

Wie **initFromFile:**, nur daß von einem Stream gelesen wird.

– free

Deallokiert **N3DRIBImageRep**; liefert **nil**.

Zeichnen

– (BOOL)drawAt :(const NXPoint *)point

Gibt das Bild an den aktuellen Koordinaten *point* des aktuellen Ausgabegeräts aus. Die Methode ruft **drawIn:** mit den auf den Ursprung *point* gesetzten Koordinaten der Bounding Box auf. Liefert **YES** bei erfolgreichem Rendern, sonst **NO**.

– (BOOL)drawIn :(const NXRect *)rect

Gibt die Grafik innerhalb des Rechtecks *rect* in dem aktuellen Koordinatensystem des aktuellen Ausgabegerätes aus. Liefert **YES** bei erfolgreichem Rendern, sonst **NO**.

– (BOOL)draw

Gibt das Bild an den aktuellen Koordinaten (0, 0) des aktuellen Ausgabegeräts aus. Die Methode ruft **drawIn:** mit den Koordinaten der Bounding Box auf. Liefert **YES** bei erfolgreichen Rendern, sonst **NO**.

Größe

– getBoundingBox :(NXRect *)r

Liefert über die Refernz *r* das umgebende Rechteck der Grafik als Bildschirm-Koordinaten. Es werden die Angaben aus der **Format**-Anweisung der RIB-Datei verwendet. Der Ursprung des Rechtecks ist immer (0, 0). Besitzt die Datei diese Anweisung nicht, wird die Standardgröße (256 Breite $\times$ 192 Höhe) verwendet. Als Rückgabewert dient **self**.

– getSize :(NXSize *)theSize

Liefert die durch **getBoundingBox:** erhaltene Größe in der Referenz *theSize*. Als Rückgabewert dient **self**.

Hintergrundfarbe

– setBackgroundColor :(NXColor)col

Setzt die Fensterhintergrundfarbe *col* für das interaktive Rendern; liefert **self**.

– (NXColor)backgroundColor

Liefert die aktuell eingestellte Hintergrundfarbe.

Hider

– setHider :(N3DHider)aHider

Setzt *hider* als Typ für den Hidden-Surface-Algorithmus ein; liefert **self**.

– (N3DHider)hider

Liefert den Typ des aktuellen Hidden-Surface-Algorithmus.

Oberflächentyp

– setSurfaceType :(N3DSurfaceType)surfaceType

Setzt einen Oberflächentyp *surfaceType* für das interaktive Rendern; liefert **self**.

– (N3DSurfaceType)surfaceType

Liefert den aktuell eingestellten Oberflächentyp.

Archivierung

– read :(NXTypedStream *)stream

Liest eine **N3DRIBImageRep** von einem typisierten Stream; liefert **self**.

– write :(NXTypedStream *)stream

Schreibt eine **N3DRIBImageRep** auf einen typisierten Stream; liefert **self**.

B.5.7 N3DRotator

Erbt von: `Object`

Deklariert in: `3DKit/N3DRotator.h`

Ein **N3DRotator** Objekt kann dazu verwendet werden, eine 2D Mausbewegung in eine 3D Drehung umzusetzen. Zur Verwirklichung wird das Modell eines Trackballs verwendet. Wird mit der Maus innerhalb eines Kontrollkreises gefahren, wird eine Drehung um die X- bzw. Y-Achse angenommen. Ein Fahren mit der Maus außerhalb des Kontrollkreises wird als Drehung um die Z-Achse verstanden. Die folgende Skizze soll die Lage des gedachten Trackballs innerhalb eines rechteckigen Bildschirmausschnitts verdeutlichen. Das Rotator-Objekt besitzt keine Funktionen zur Darstellung.

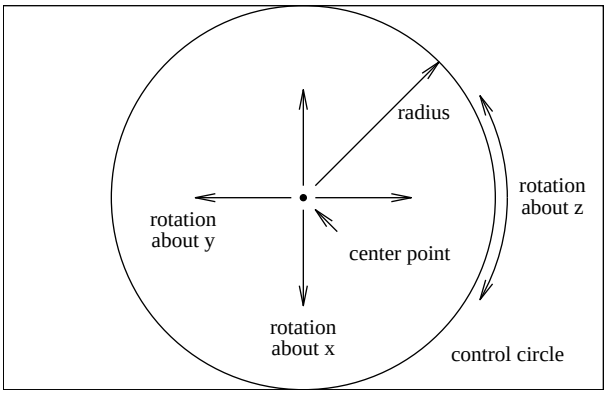

Abbildung B.2: **N3DRotator**, aus [NeXTDoc]

Position und Radius des Trackballs können gesetzt werden. Die Defaultposition füllt die Rechteckhülle des Kamerabildbereichs möglichst gut aus: Zentrum des Kreises ist in der Mitte des Rechtecks, der Radius entspricht der Hälfte der kürzeren Seite. Bei einem Verändern des Kamerafensters wird der Rotator entsprechend angepaßt. Die Drehachsen können durch die Methode **setRotationAxis:** auch einzeln aktiviert und deaktiviert werden.

Durch die Methode **trackMouse:to:rotationMatrix:andInverse:** kann eine lineare Mausbewegung in die entsprechende Drehmatrix und ihre Inverse umgesetzt werden. Die Matrizen können danach entsprechend auf Kamera oder Grafikobjekte angewendet werden.

Instanzvariablen

id camera: Die zum Rotator gehörende Kamerainstanz

NXRect bounds: Rechteckhülle (in 2D PostScript Koordinaten), in der die Cursorbewegung Einfluß nimmt

NXPoint center: 2D Koordinate des Zentrums des Kontrollkreises

float radius: Radius des Kontrollkreises

N3DAxis rotationAxis: Achsen, auf die eine Rotation angewendet werden kann, Default: **N3D_AllAxes** (es kann um alle Achsen gedreht werden)

Methoden

Initialisierung

– init

Initialisieren der Empfängerinstanz; liefert **self**.

– initWithCamera :aCamera

Eigentlicher Konstruktor, initialisiert die Instanz und stellt die Verbindung zur Kamera *aCamera* durch einen Aufruf von **setCamera:** her; liefert **self**.

Parameter setzen

– setCamera :aCamera

Verbindet den Rotator mit der Kamera *aCamera*. Das Zentrum des Kontrollkreises wird in das Zentrum der Rechteckhülle der Kamera gesetzt. Der Radius bekommt als Wert die Hälfte der Länge der kürzeren Seite zugewiesen; liefert **self**.

– setCenter :(const NXPoint *)c andRadius:(float)r

Setzt das Zentrum *c* und den Radius *r* des Kontrollkreises; liefert **self**.

Rotationsachsen

– setRotationAxis :(N3DAxis)axis

Ein Aufruf dieser Funktion dient dazu, die Freiheitsgrade der Rotation zu setzen. Durch den Parameter *axis* werden die Achsen, um die gedreht werden kann, bestimmt. Der gesetzte Wert hat Einfluß auf die Methode **trackMouseFrom:to:rotationMatrix:and-Inverse:**, die eine der Drehung entsprechende Matrix und ihre Inverse liefert. Der Typ **N3DAxis** ist in der Datei `3DKit/next3d.h` deklariert. Die Methode gibt **self** zurück.

– (N3DAxis)rotationAxis

Liefert die aktuell gesetzten Drehachsen.

Mausverfolgung

– trackMouseFrom :(const NXPoint *)firstMouse to:(const NXPoint *)lastMouse rotationMatrix:(RtMatrix)matrix andInverse:(RtMatrix)inverseMatrix

Kernmethode des Objekts. Die Aufgabe der Methode ist es, eine Mausbewegung in eine entsprechende 3D-Rotationsmatrix und ihre Inverse umzusetzen. Die Freiheitsgrade der Rotation sind durch das **rotationAxis** Feld bestimmt. Die Matrix wird in *matrix*, ihre Inverse in *inverseMatrix* zurückgegeben. Der Methoden-Rückgabewert ist **self**. Die beiden Parameter *firstMouse* und *lastMouse* enthalten die Koordinaten der linearen Mausbewegung, die in eine Drehung umgesetzt werden sollen. Die Methode wird meistens in der **mouseDown:** Methode der Kamera verwendet.

Da die Methode die Matrizen nicht auf bereits existierende Transformationen anwendet, muß *matrix* und *inverseMatrix* durch ein entsprechendes Programmstück weiterverwendet werden. Die *matrix* kann als Kameradrehung verwendet werden (z.B. mit **SetEyeAt:toward:roll:**), sie kann durch Linksmultiplikation an eine Transformationsmatrix eines **N3DShape** Objekts für eine Drehung in dessen Objektkoordinatensystem oder durch Rechtsmultiplikation an die Transformationsmatrix für eine Drehung innerhalb des Koordinatensystems des Vorgängers in der Objekthierarchie verwendet werden.

Archivierung

– read :(NXTypedStream *)stream

Dearchiviert einen Rotator von dem typisierten *stream*.

– write :(NXTypedStream *)stream

Archivieren der Instanz auf den typisierten *stream*.

B.5.8 N3DShader

Erbt von: `Object`

Deklariert in: `3DKit/N3DShader.h`

Ein **N3DShader** Objekt wird als Repräsentation eines Shaders in der 3DKit Objekthierarchie verwendet. Ein Shader kann in der Shading Language geschrieben und übersetzt worden sein oder als Standard-Shader zur Verfügung stehen. Das **N3DShader** Objekt ist eine Schnittstelle zu einer Shader-Funktion. Mit Hilfe des Objekts ist es möglich, Wert und Typ der Shader-Parameter abzufragen und deren Werte zu setzen. Mit **resetShaderArg:** können Parameter auf ihre Defaultwerte zurückgesetzt werden. Wird eine Shader-Instanz an ein **N3DShape** gebunden (**initWithShader:**, **setShader:**), wird der Shader in der Attribut-Umgebung des Objekts durch den Aufruf der **set:** Methode des Shaders instanziert und gilt danach auch in den hierarchisch

folgenden Objekten als Default-Shader. Für jedes Shape kann eine Shader-Instanz pro Shader-Typ gesetzt werden. Das RenderMan Interface erlaubt sechs verschiedene Shader-Typen: ‘surface’, ‘displacement’, ‘light’, ‘imager’, ‘volume’ und ‘transformation’. Von diesen wird nur eine begrenzte Anzahl von den auf dem NeXT implementierten Renderen unterstützt. Der interaktive Renderer unterstützt aus Geschwindigkeitsgründen nur die Oberflächen-Shader: ‘constant’, ‘matte’, ‘metal’, ‘plastic’ und ‘none’ und die Atmosphären-Shader: ‘depthcue’ und ‘fog’. Der photorealistische Renderer verwendet keine ‘Imager’- und ‘Transformation’-Shader. Von den ‘Volume’-Shadern werden nur die Atmosphären-Shader unterstützt.

Die Shader-Funktionen werden in dem Pfad:

```
~/Library/Shaders:/NextLibrary/Shaders:/LocalLibrary/Shaders
```

gesucht. Der photorealistische Renderer suchte in der momentanen Implementierung nicht in `~/Library/Shaders`. Ein entsprechender Eintrag in der ‘rendermn.ini’-Datei im Home-Verzeichnis zeigte auch keine Wirkung. Wird als Shader-Name der komplette Pfad angegeben, findet der photorealistische Renderer die Datei. Die RIB-Datei ist dann aber nicht mehr ohne weiteres auf andere Systeme (in denen die Shader in anderen Verzeichnissen stehen können) übertragbar.

Shader-Objektdateien (`.slo`-Dateien) können mit dem Shader-Compiler `shade` aus einer Shader-Quelldatei (hat die Standard-Endung `.sl`) erzeugt werden. Sie enthalten jeweils eine einzige Shader-Funktion und/oder mehrere Hilfsfunktionen. Jede Hilfsfunktion wird in eine eigene `.slo`-Datei (ihr Name entspricht dem Funktionsnamen) compiliert. Auf diese Weise können auch Shader, die nicht die Definition einer verwendeten Hilfsfunktion enthalten, auf eine solche zugreifen.

Instanzvariablen

NXColor color: Shader-Farbe (Default: weiß)

float transparency: Transparentheit des Shaders (Default: Opak, hat den Wert 0)

const char *shader: Name der Shader-Funktion (Default: **NULL**)

SLO_TYPE shaderType: Shader-Typ

int shaderArgCount: Anzahl der Shader-Parameter

SLOArgs *shaderArgs: Parameterwerte

NXZone *zone: Speicherzone, in die die Daten des Objekts gelegt wurden

Methoden

Initialisierung und Freigabe

– init

Initialisieren einer neuen Shader-Instanz, setzt die Instanzvariablen auf ihre Defaultwerte und liefert **self**.

– initWithShader :(const char *)aShader

Initialisiert den Empfänger und ruft die Methode **setShader:** mit *aShader* (Name der Shader-Funktion) auf, um die Shader-Funktion zu setzen; liefert **self**.

– free

Freigeben einer Shader-Instanz und ihrer Daten; liefert **nil**.

Shading Language Objektdatei
– setShader :(const char *)aShader

Setzt die Shader-Funktion, belegt Parameternamen, -typen und -werte. *aShader* ist der Name einer Shader-Objektdatei (ohne das Suffix '`.slo`'); liefert **self**.

– (const char *)shader

Liefert den Namen der Shader-Funktion, die mit der Instanz assoziert ist.

Shader-Farbe
– setColor :(NXColor) aColor

Setzt die Farbe des Shaders. Ist *useColor* gesetzt, wird beim Rendern die Interface-Funktion **RiColor()** mit dem entsprechenden Farbparameter ausgegeben. Die Funktion beeinflußt keine Parameter/Wert-Einträge der Shader-Funktion. Die Methode liefert **self**.

– (NXColor)color

Liefert die Shader-Farbe.

– setUseColor :(BOOL)flag

Wird die Methode mit dem Wert **YES** für den Parameter *flag* aufgerufen, wird die Shader-Farbe (siehe **setColor:**) durch einen Aufruf von **RiColor()** beim Rendern verwendet. Normalerweise wird die Farbe nur für Oberflächen-Shader verwendet. Da hierarchisch untergeordnete Shapes zwar ihre Farbe ändern, den Shader aber beibehalten können, ist die Verwendung der Oberflächen-Farbe und -Transparenz in den **N3DShader** Instanzen fraglich. Die Methode liefert **self**.

– (BOOL)doesUseColor

Liefert **YES** wenn die Shader-Farbe verwendet wird.

Shader-Transparenz
– setTransparency :(float)alphaValue

Setzt die Transparentheit des Shaders auf den Wert von *alphaValue* ($0 \leq alphaValue \leq 1$, 0 ist opak, 1 vollständig transparent); liefert **self**.

– (float)transparency

Liefert die aktuelle Transparenz der Oberfläche.

Behandlung der Shader-Parameter

– (int)shaderArgCount

Liefert die Anzahl der Shader-Parameter.

– (const char *)shaderArgNameAt :(int)i

Liefert den Namen des $i$ten Shader-Parameters, $0 \leq i < shaderArgCount$.

– (SLO_TYPE)shaderArgType :(const char *)aName

Liefert den Typ des Shader-Parameters mit dem Namen *aName*.

– (BOOL)isShaderArg :(const char *)aName

Liefert **YES** wenn *aName* der Name eines Shader-Parameters ist.

Mit den folgenden Funktionen können die Werte von Shader-Parametern *aName* gesetzt (**setShaderArg::**) und gelesen (**getShaderArg::**) werden. Ist der Parameter nicht von dem Typ des Wertes, findet nach Möglichkeit eine Typkonvertierung statt. Alle Funktionen liefern als Rückgabewert **self**.

– setShaderArg: :(const char *)aName floatValue:(float)fv

Mögliche Konvertierung nach String, Punkt **RtPoint** (belegt alle Koordinaten mit dem Wert *fv*) oder Farbe **NXColor** (durch die Verwendung von **NXConvertGreyToColor()**).

– setShaderArg :(const char *)aName stringValue:(const char *)sv

Mögliche Konvertierung nach **float**.

– setShaderArg :(const char *)aName pointValue:(RtPoint)pv

Mögliche Konvertierung nach **float** (Verwendung der X-Komponente), **NXColor** (Punkt wird als rgb-Triplett interpretiert und mit der Funktion **NXConvertRGBToColor()** in eine **NXColor** umgeformt) oder einen String, der die x, z und z Komponenten des Werts *pv* enthält.

– setShaderArg :(const char *)aName colorValue:(NXColor)cv

Mögliche Konvertierung nach **float** (verwendet den Wert, den **NXConvertColorToGrey()** liefert), **RtPoint** (die r, g, b Komponenten der Farbe werden in die x, y, z Komponenten des Punktes geschrieben) oder einen String, der die r, g und b Komponenten des Werts *pv* enthält.

– getShaderArg: :(const char *)aName floatValue:(float *)fv

Mögliche Konvertierung von **RtPoint**, **NXColor** und String.

– getShaderArg :(const char *)aName stringValue:(const char **)sv

Mögliche Konvertierung von **RtPoint**, **NXColor** und String.

– getShaderArg :(const char *)aName pointValue:(RtPoint *)pv

Mögliche Konvertierung von **float**, **NXColor** und String (über einen **float**).

– getShaderArg :(const char *)aName colorValue:(NXColor *)cv

Mögliche Konvertierung von **float**, **RtPoint** und String (über einen **float**).

– resetShaderArg :(const char *)aName

Setzt den Parameter *aName* auf seinen Defaultwert.

Shader-Typ
– (SLO_TYPE)shaderType

Liefert den Typ der Shader-Funktion. Ist das Objekt mit keiner Shader-Funktion verbunden, liefert die Funktion den Wert **SLO_TYPE_UNKNOWN**.

Anwendung der Shader-Funktion
– set

Setzen des Shaders. Die Methode wird von der **render:** Methode einer **N3DShape** Instanz für jeden der dort gesetzten Shader-Typen aufgerufen.

Die Methode setzt, falls **useColor YES** liefert, zuerst Farbe und Opazität mit den entsprechenden Interface-Routinen **RiColor()** und **RiOpacity()** und instanziert anschließend entsprechend des Shader-Typs einen Shader mit seinen Parametern durch den Aufruf der korrespondierenden Interface-Routine: **RiSurface()**, **RiAtmosphere()**. . . Die Methode liefert **self**.

Archivierung
– read :(NXTypedStream *)stream

Dearchivierung von dem typisierten *stream*; liefert **self**.

– write :(NXTypedStream *)stream

Archivierung auf den typisierten *stream*; liefert **self**.

B.5.9 N3DShape

Erbt von: `Object`

Deklariert in: `3DKit/N3DShape.h`

Die **N3DShape** Klasse dient zur Programmierung einer Objekthierarchie aus Attribut-Blöcken. Die **renderSelf:** Methode eines Objekts wird in **RiTransformBegin()** und **RiTransformEnd()** geschachtelt aufgerufen. In diesem Block werden anschließend auch die hierarchischen Nachfolger ausgegeben, sodaß die gesetzten Attribute auch für sie gelten. In einer überdefinierten **renderSelf:** können prinzipiell beliebig viele Attribute gesetzt, geometrische Primitive plaziert und andere Interface-Routinen aufgerufen werden. Die Instanzen der **N3DShape** Klasse werden untereinander hierarchisch verkettet. Sie werden von einer

N3DCamera-Instanz gerendert. Normalerweise werden nur in den Blattknoten Oberflächen erzeugt. Die Zwischenknoten sollten nur zur Gruppierung und zur Änderung von Attributen dienen. Die Render-Methoden werden ‘top down’ (descendant/ancestor-Relation) von links nach rechts (nextPeer/previousPeer-Relation) aufgerufen.

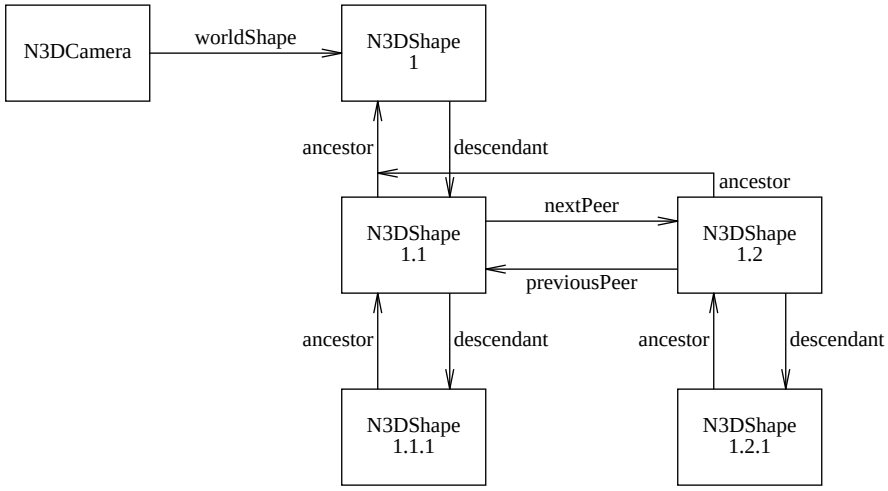

Abbildung B.3: **N3DShape**-Hierarchie, aus [NeXTDoc]

N3DShape bietet eine Anzahl von Methoden, um eine Hierarchie aufzubauen (. . . **link**. . .). Die Methode **getBoundigBox:** liefert, wenn die Hüllen korrekt gesetzt sind, die kleinste umgebende Hülle eines Objekts und seiner Nachkommen (Descendanten und deren Peers).

Es existieren auch Methoden, um Punkte zu und von einem hierarchisch übergeordneten Objektkoordinatensytemen in das eigene konvertieren zu können (**convertPoints:count::**). Auch können 3D Objektkoordinaten in ein 2D Kamerakoordinatensystem transformiert werden (**convertObjectPoints:count:toCamera:**). Die Objektkoordinaten-Konvertierungen können auch durch die Zugriffsfunktionen auf die drei Matrizen: *transform*, *compositeTransform* und *inverseCompositeTransform* durchgeführt werden.

Mit jeder **N3DShape**-Instanz kann je eine Instanz der sechs Shader-Typen verbunden sein (**setShader:**). Die Shader werden vor dem Aufruf der **renderSelf:** Methode von der steuernden **render:** Methode automatisch gesetzt (s.a. **N3DShader**). Das von **N3DShape** abgeleitete Objekt **N3DLight** stellt schon Repräsentationen der vier standardisierten Lichtquellen-Shader zur Verfügung. Der interaktive Renderer verwendet zur Oberflächendarstellung den aktuellen Oberflächentyp (*surfaceType*).

Es ist möglich, Objekte mit gleichen geometrischen Daten durch ein Delegate-Objekt, eine **N3DShape**-Instanz, die mit ihrer Methode **renderSelf:** die entsprechenden Interface-Routinen aufruft, darstellen zu lassen. **render:** ruft, ist ein ‘Delegate’ gesetzt, an Stelle der eigenen die **renderSelf:** Methode des ‘Delegates’ auf. Da **render:** die Koordinatentransformation, die durch die Methoden **setTransformMatrix:**, **scale:::**, **preTranslate:::** u.s.w. gesetzt wird, ausführt, kann vor der Ausgabe eines ‘Delegates’ eine beliebige affine Transformation stattfinden.

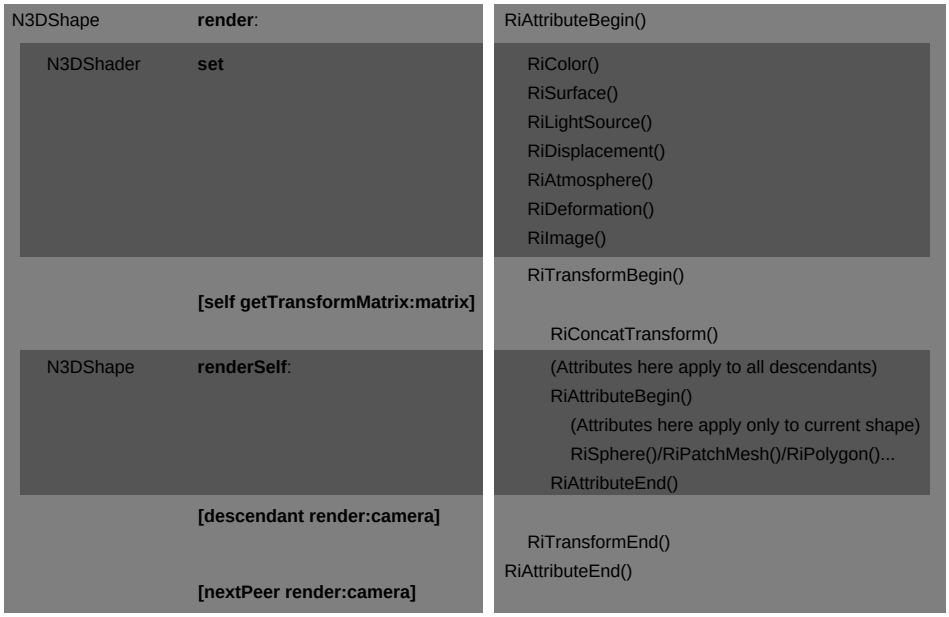

Abbildung B.4: Die Aufrufreihenfolge beim Rendern, aus [NeXTDoc]

Instanzvariablen

RtMatrix transform: Transformationsmatrix relativ zum direkten Vorgänger. Die Matrix wird vor dem Aufruf von **renderSelf:** mittels **RiConcatTransform()** mit der aktuellen Transformationsmatrix multipliziert. Die Matrix kann über die Methoden: **setTransformMatrix:** und **getTransformMatrix:** verwendet werden. Auch die Transformations-Methoden beeinflussen diese Matrix. Änderungen von *transform* beeinflussen auch die eigenen *compositeTransform* und *inverseCompositeTransform* und die der Nachfolger. Default: **N3DIdentityMatrix**

RtMatrix compositeTransform: Transformationsmatrix, mit der Punkte von einem Knoten auf oberer Hierarchie-Ebene (dem entferntesten Vorgänger, das 'World Shape') in das aktuelle Objektkoordinatensystem transformiert werden können. Mit der Methode **getCompositeTransform:relativeToAncestor:** kann auf die Matrix zugegriffen werden. Default: **N3DIdentityMatrix**

RtMatrix inverseCompositeTransform: Inverse von *compositeTransform*. Die Matrix wird zur Transformation vom aktuellen Objektkoordinatensystem in das Koordinatensystem des Vorgänger-Knotens auf oberster Hierarchie-Ebene verwendet. Zugriff auf die Matrix ist mit Hilfe der **getInverseCompositeTransform:relativeToAncestor:** Methode möglich. Default: **N3DIdentityMatrix**

RtBound boundingBox: Dreidimensionale Bounding Box des Objekts. Die Werte müssen

in den Blattknoten von Hand gesetzt werden. Die Dimension kann visuell überprüft werden, indem das Grafikobjekt als Quader gerendert wird: [`aCamera setDrawAsBox:YES`].

N3DShapeName *shapeName: Name und id einer Instanz.

N3DSurfaceType surfaceType: Oberflächentyp für den interaktiven Renderer, default: **N3D_WireFrame**

id surfaceShader: Oberflächen-Shader **N3DShader** für das photorealistische Rendern, default: **nil**

id displacementShader: 'Displacement'-Shader für das Rendern, default: **nil**

id lightShader: 'Lightsource'-Shader für das Rendern (im RenderMan Interface hingegen können mehrere dieser Shader gleichzeitig aktiv sein), default: **nil**

id imagerShader: 'Imager'-Shader für das Rendern, 'Imager' Shader (Farbtransformation) werden von den momentan für den NeXT erhältlichen Renderern noch nicht unterstützt, default: **nil**

id volumeShader: Volumen-Shader für das Rendern, momentan werden nur die Standard-Atmosphären-Shader unterstützt, default: **nil**

id transformationShader: Transformations-Shader für das Rendern werden momentan noch nicht unterstützt, default: **nil**

struct _shapeFlags shapeFlags: Verschiedene Zustände des Shapes

```
struct _shapeFlags {
  unsigned int selectable:1;
  unsigned int visible:1;
  unsigned int ancestorChanged:1;
  unsigned int compositeDirty:2;
  unsigned int drawAsBox:1;
  unsigned int isInstance:1;
  unsigned int hasShader:1;
}
```

selectable: **YES**, wenn das Shape selektiert werden kann, default: **NO**

visible: **YES**, wenn das Objekt *und* seine Nachfolger sichtbar sein sollen, default: **YES**.

ancestorChanged: Wird **YES**, wenn sich ein Vorgänger des Objekts geändert hat.

compositeDirty: Wird **YES**, wenn die *compositeTransform* Matrix und ihre Inverse neu berechnet werden müssen.

drawAsBox: Bei dem Wert **YES** wird das Grafikobjekt nur als Quader gerendert, default **NO**

isInstance: **YES**, wenn der ‘Delegate’ das Rendern durchführen soll.

hasShader: **YES**, wenn irgendwelche Shader mit dem Objekt verbunden sind.

id nextPeer: Nächster ‘Geschwisterknoten’, default: **nil**

id previousPeer: Vorheriger ‘Geschwisterknoten’, default: **nil**

id descendant: Hierarchischer Nachfolger, default: **nil**

id ancestor: Hierarchischer Vorgänger, default: **nil**

id renderDelegate: Optionaler ‘Delegate’, der gerendert werden kann, default: **nil**

Methoden

Initialisierung und Freigabe

– init

Initialisieren einer neuen Instanz. Setzt alle Instanzvariablen auf ihre Defaultwerte; liefert **self**.

– free

Gibt alle hierarchischen Nachfolger (‘descendent’ und ‘peers’ des ‘descendants’) durch ein [[`self descendant`] `freeAll`] frei, nicht aber die eigenen ‘peers’. Das Objekt wird aus der aktuellen ‘peer’-Liste herausgenommen und gelöscht. Bei Bedarf wird der nächste Geschwisterknoten der ‘descendant’ des direkten Vorgängers (s.a. **unlink**). Die Methode liefert **nil**.

– freeAll

Gibt die nächsten ‘peers’ und ihre ‘descendants’ rekursiv frei: [[`self nextPeer`] `freeAll`]. Anschließend werden das eigene Objekt und seine hierarchischen Nachfolger durch [`self free`] gelöscht. Die Methode liefert **nil**.

Rendern

– render :(N3DCamera *)camera

Steuert das Rendern des Objekts (siehe Abb. B.4) seines Descendanten und des nächsten Peers; liefert **self**.

Es wird ein Attribut-Block erzeugt. In diesem wird zuerst die Oberflächenfarbe (normalerweise an den Oberflächen-Shader gebunden) und alle vorhandenen Shader gesetzt. Anschließend wird ein Transformationsblock erzeugt (eigentlich überflüssig, weil die Transformationen auch durch den umgebenden Attribut-Block gesichert werden, der Transformationsblock direkt vor dem Attribut-Block endet und die Descendanten innerhalb des Transformationsblocks ausgegeben werden). In dem Transformationsblock wird die Transformationsmatrix des Objekts mit der RenderMan CTM verbunden, die

eigene Render-Methode **renderSelf:** aufgerufen und die Descendanten gerendert. Hinter dem äußeren Attribut-Block werden die folgenden Peers gerendert. Auf diese Weise wird die **N3DShape**-Objekthierarchie in eine entsprechend geschachtelte RenderMan Attribut-Blockstruktur umgewandelt. Die Methode kann überdefiniert werden, wenn Shape-Gruppen beispielsweise in Solid-Blöcken zusammengefaßt werden sollen.

– renderSelf :(N3DCamera *)camera

Abstrakte Methode, liefert **self**. Die Methode kann überdefiniert werden, um die Interface-Routinen RenderMans aufzurufen. Die geänderten Attribute gelten auch für die hierarchischen Nachfolger. Möchte man Attribute lokal ändern, kann man dieses durch einen weiteren **RiAttributeBegin()**/**RiAttributeEnd()** Block erreichen. Die Methode wird von **render:** aufgerufen, wenn kein 'Delegate' gesetzt ist und nicht als Quader gerendert werden soll. Es sollten in dieser Methode möglichst keine geometrischen Transformationen stattfinden, die die Bounding Box des Shapes unkontrolliert verändern, damit die Punktkonvertierungen richtig arbeiten. Für geometrische Transformationen können die entsprechenden Methoden des Objekts verwendet werden.

– renderSelfAsBox :(N3DCamera *)camera

Diese Methode wird von **render:** aufgerufen, wenn das *drawAsBox* Flag gesetzt ist. In diesem Fall wird nur ein Quader in den Ausmaßen der Bounding Box (bzw. der des 'Delegates') gerendert. Die Ausmaße der Bounding Box werden durch einen Aufruf von `[self getBoundingBox : ...]` erhalten. Sie entsprechen dem kleinsten Quader um das eigene Objekt und seiner Nachfolger. Die Methode liefert **self**.

Traversierung der Objekthierarchie

– nextPeer

Liefert den nächsten Geschwisterknoten oder **nil** am Ende der Liste.

– previousPeer

Liefert den vorigen Geschwisterknoten oder **nil** am Anfang der Liste.

– firstPeer

Liefert den ersten Geschwisterknoten in der Liste (entspricht, wenn möglich, dem Descendanten des direkten Vorgängers)

– lastPeer

Liefert den letzten Geschwisterknoten in der Liste

– descendant

Liefert den direkten hierarchischen Nachfolger oder **nil**, falls der Empfänger ein Blattknoten ist.

– lastDescendant

Liefert den letzten Descendanten in der *descendant*-Kette des Empfängers. Die Methode liefert **self**, falls der Empfänger schon ein Blattknoten ist.

– ancestor

Liefert den direkten hierarchischen Vorgänger des Empfängers oder **nil** wenn der Empfänger auf oberster Hierarchie-Ebene steht.

– firstAncestor

Liefert das Objekt auf oberster Hierarchie-Ebene.

– (BOOL)isWorld

Liefert **YES**, wenn der Empfängerknoten auf oberster Hierarchie-Ebene steht und keine 'linken' Geschwisterknoten mehr hat. Im anderen Fall liefert die Methode **NO**.

Verwaltung der Hierarchie

– linkPeer :aPeer

Fügt *aPeer* als nächsten Geschwisterknoten des Empfängers ein. *aPeer* darf beliebig viele Geschwisterknoten und Nachfolgerknoten besitzen. Der ehemalige 'rechte' Nachbar des Empfängers wird der neue 'rechte' Nachbar des am weitesten rechts stehenden Nachbarn von *aPeer*. Falls *aPeer* keine **N3DShape**-Instanz ist, bleibt ein Aufruf dieser Methode ohne Wirkung. Die Methode liefert **self**.

– linkDescendant :aDescendant

Fügt *aDescendant* als direkten Nachfolger des Empfängers ein. Der ehemalige direkte Nachfolger des Empfängers wird der neue direkte Nachfolger des letzten Objekts in der 'Descendant'-Kette von *aDescendant* — logischer wäre, wenn er am Ende der Peer-Kette eingefügt würde. Der Empfänger wird der neue 'Ancestor' von *aDescendant* und seiner Peers. Falls *aPeer* keine **N3DShape**-Instanz ist, bleibt ein Aufruf dieser Methode ohne Wirkung. Die Methode liefert **self**.

– linkAncestor :anAncestor

Setzt *anAncestor* als direkten Vorgänger des Empfängers und seiner Peers ein. Der Descendant von *anAncestor* wird nicht verändert. Die Methode liefert den ehemaligen direkten Vorgänger des Empfängers.

– unlink

Fügt den Empfänger (und damit auch seine Nachfolger) aus einer **N3DShape**-Hierarchie aus. War das Objekt der Descendant, wird sein ehemaliger rechter Nachbar der neue Descendant, bzw. der Descendant auf **nil** gesetzt, falls kein Nachbar vorhanden ist. Die Methode liefert **self**.

– group :toShape

Gruppiert den Empfänger als Nachfolger unter *toShape*, behält aber seine Position (sowie Größe und Skalierung) im Welt-Koordinatensystem durch Verändern der Transformationsmatrix des Empfängers bei. Die Methode kann dazu verwendet werden, durch

einen Benutzer selektierte Objekte durch Aufrufen dieser Methode mit dem gleichen *toShape* zu einer Gruppe zusammenzufassen. Hat *toShape* keinen Descendanten wird der Empfänger der neue Descendant, sonst wird der Empfänger durch einen Aufruf der Methode **linkPeer:** des Descendanten an diesen gebunden. Die Methode liefert **self**.

– ungroup

Löscht den Empfänger aus der Hierarchie. Der Descendant und seine Nachfolger behalten ihre Orientierung im Weltkoordinatensystem. Ihre Transformationsmatrizen werden entsprechend geändert.

Hat der Empfänger einen linken Peer und einen rechten Peer, werden diese untereinander verzeigert. Der Descendant des Empfängers wird anschließend mit **linkPeer:** rechts an den linken Peer gebunden. War der Empfänger der am weitesten links stehende Peer, wird sein Descendant der neue Descendant des Ancestors. Die Knoten unter dem Empfänger rutschen also in der Hierarchie eine Ebene höher. Die Verzeigerungen des Empfängers werden anschließend alle auf **nil** gesetzt (damit er auf einfache Weise durch **free** gelöscht werden kann). Die Methode liefert **self**.

Shader

– setShader :aShader

Setzt die **N3DShader** Instanz *aShader* entsprechend seines Typs als neuen Shader ein. Die Methode liefert die **id** auf den vorher gesetzten Shader des gleichen Typs.

– shaderType :(SLO_TYPE)type

Liefert die **id** des im Objekt gesetzten Shaders vom Typ *type* (**SLO_TYPE_**...) oder **nil**, falls kein solcher Shader gesetzt ist.

Oberfläche

– setSurfaceType :(N3DSurfaceType)surfaceType andDescendants:(BOOL)flag

Setzt den Oberflächentyp für den interaktiven Renderer. Hat *flag* den Wert **YES**, wird die Oberfläche auch für alle hierarchischen Nachfolger (Descendanten und deren Peers) verwendet. Die Kamera verwendet diese Methode in **setSurfaceTypeForAll:chooseHider:** für das Weltobjekt ($flag == \mathbf{YES}$). Die Methode liefert **self**.

– (N3DSurfaceType) surfaceType

Liefert den aktuell gesetzten Oberflächentyp.

Bounding Box

– getBoundingBox :(RtBound *)bBox

Liefert die Bounding Box über die Referenz *bBox*. Der Rückgabewert ist **self**. Die Bounding Box ist die Vereinigung der eigenen *boundingBox* Instanzvariablen und der der Nachfolger in den Objektkoordinaten des Empfängers. Die *boundingBox* wird nicht automatisch gesetzt und an Änderungen der geometrischen Primitive in **renderSelf:** angepaßt; der Programmierer muß hierfür selbst Sorge tragen.

– setDrawAsBox :(BOOL)flag

Wird diese Methode mit dem Wert **YES** für *flag* aufgerufen, wird anstelle von **renderSelf:** mit **renderAsBox:** gerendert. Ist der Wert **NO** wird **renderSelf:** aufgerufen.

Möchte man aus Effizienzgründen oder ähnlichen Überlegungen ein Objekt nur als Bounding Box rendern, kann man die Nachfolger des Objekts mit [[`self descendant`] `setVisible:NO`] unsichtbar machen[1]. Die Methode liefert **self**.

– (BOOL)doesDrawAsBox

Liefert **YES**, falls das Objekt nur als Box gerendert wird, sonst **NO**.

– getBounds :(NXRect *)sRect inCamera:camera

Liefert die Bounding Box über die Referenz *sRect* in 2D Kamerakoordinaten. Ist *camera* keine **N3DCamera** Instanz wird eine 'Exception' generiert. Als Rückgabewert dient **self**.

Punkte konvertieren

– convertObjectPoints :(RtPoint *)points count:(int)n toCamera:camera

Konvertiert *n* Punkte *points* vom 3D Objektkoordinatensystem des Empfängers in das 2D (PostScript) Koordinatensystem der Kamera *camera*. Die Punkte werden in den x, y Koordinaten des *points* Feldes zurückgegeben, die z Koordinaten sind ungültig. Rückgabewert ist **self**.

– convertPoints :(RtPoint *)points count:(int)n fromAncestor:(N3DShape *)aShape

Konvertiert *n* Punkte *points* vom Objektkoordinatensystem des Ancestors von *aShape* in das Koordinatensystem des Empfängers. Befindet sich *aShape* nicht hierarchisch über dem Empfänger oder ist **nil**, werden die Koordinaten vom Weltkoordinatensystem in das eigene konvertiert. Die Punkte werden im *points*-Array zurückgegeben. Rückgabewert ist **self**.

– convertPoints :(RtPoint *)points count:(int)n toAncestor:(N3DShape *)aShape

Konvertiert *n* Punkte *points* vom Koordinatensystem des Empfängers in das Koordinatensystem des Ancestors von *aShape*. Befindet sich *aShape* nicht hierarchisch über dem Empfänger oder ist **nil**, wird in das Weltkoordinatensystem konvertiert. Die Punkte werden im *points*-Array zurückgegeben. Rückgabewert ist **self**.

Selektierbarkeit

– setSelectable :(BOOL)flag

Setzt die Selektierbarkeit des Empfängers. Eine Selektierung geschieht durch die **selectShapesIn:** Methode einer **N3DCamera**. Rückgabe ist **self**.

[1] Die Bounding Box umfaßt auch alle Nachfolger eines Objekts

– (BOOL)isSelectable

Liefert **YES**, falls der Empfänger selektierbar ist, sonst **NO**.

Sichtbarkeit

– setVisible :(BOOL)flag

Setzt die Sichtbarkeit des Empfängers und aller seiner Nachkommen; liefert **self**.

– (BOOL)isVisible

Liefert **YES**, falls der Empfänger sichtbar ist, sonst **NO**.

Benennung

– setShapeName :(const char *)aName

Setzt als Namen des Empfängers *aName*; liefert **self**.

– (const char *)shapeName

Liefert den Namen des Empfängers.

Render-Delegate

– setRenderDelegate :anObject

Setzt *anObject* als ‘Delegate’ ein. Die Methode liefert den alten ‘Delegate’. Ist *anObjekt* keine **N3DShape**-Instanz tut diese Methode nichts und liefert **nil**.

– removeRenderDelegate

Löscht den ‘Delegate’ aus dem Objekt und liefert ihn als Rückgabewert der Methode.

– renderDelegate

Liefert den aktuellen ‘Delegate’ oder **nil**, falls keiner gesetzt ist.

Transformations Matrizen

– setTransformMatrix :(RtMatrix)tm

Ersetzt die Transformationsmatrix *transform* durch *tm*; liefert **self**.

– (RtMatrix)getTransformMatrix :(RtMatrix)theMatrix

Liefert über die Referenz *theMatrix* die Transformationsmatrix *transform* des Empfängers. Die Matrix dient zur Konvertierung von Punkten aus dem Koordinatensystem des Ancestors in das eigene Koordinatensystem. Die Methode wird von **render:** verwendet, um die Matrix mit der CTM zu verbinden. Die Methode kann überdefiniert werden, um die Matrix (und damit das Objektkoordinatensystem des Empfängers) vor dem Rendern zu verändern. Rückgabewert ist **self**.

– concatTransformMatrix :(RtMatrix)ctm premutiply:(BOOL)flag

Verbindet die Matrix *ctm* mit der aktuellen *transform* Matrix des Empfängers. Wenn der Wert des *flag*s **YES** ist, wird die Matrix links an die Transformationsmatrix multipliziert — das entspricht einer Anwendung dieser Matrix auf das *Objektkoordinatensystem* vor der Anwendung der aktuellen Transformationsmatrix. Ist der *flag* **NO**, wird die Matrix rechts an *transform* multipliziert, d.h. sie bezieht sich auf das *Koordinatensystem des Ancestors*. Die Methode liefert **self**.

– getCompositeTransformMatrix :(RtMatrix)ctm
relativeToAncestor:(N3DShape *)theAncestor

Liefert über die Referenz von *ctm* die Matrix, mit der Punkte vom Koordinatensystem von *theAncestor* in das Koordinatensystem des Empfängers transformiert werden können. Ist *theAncestor* **nil** oder kein hierarchischer Vorgänger des Empfängers, wird entsprechend die Matrix ab dem Weltobjekt verwendet. Rückgabewert ist **self**.

– getInverseCompositeTransformMatrix :(RtMatrix)ictm
relativeToAncestor:(N3DShape *)theAncestor

Liefert in der Referenz *ictm* die inverse Composite-Matrix, also die Matrix mit der Punkte aus dem Koordinatensystem des Empfängers in das Koordinatensystem von *theAncestor*, bzw. das Weltkoordinatensystem, falls *theAncestor* kein Vorgänger oder **nil** ist, transformiert werden können. Rückgabewert ist **self**.

Rotation, Skalierung, Verschiebung

– rotateAngle :(float)ang axis:(RtPoint)anAxis

Die Rotationsmatrix des Objektkoordinatensystems für Drehungen um *ang* Grad um die Achse vom Ursprung zum Punkt *anAxis* (im Koordinatensystem des Ancestors) wird rechts an die *transformation*-Matrix multipliziert und nimmt damit Einfluß auf das Koordinatensystem des Vorgängers (s.a. **concatTransformMatrix:premultiply:**); liefert **self**.

– preRotateAngle :(float)ang axis:(RtPoint)anAxis

Die Rotationsmatrix des Objektkoordinatensystems für Drehungen um *ang* Grad um die Achse vom Ursprung zum Punkt *anAxis* (beide im Objektkoordinatensystem) wird links an die *transformation*-Matrix multipliziert und nimmt damit Einfluß auf das eigene Koordinatensystem (s.a. **concatTransformMatrix::**); liefert **self**.

– scale :(float)sx :(float)sy :(float)sz

Skalierungsmatrix für die Skalierungsfaktoren *sx*, *sy*, *sz* in die Achsenrichtungen im Objektkoordinatensystem des Ancestors wird rechts an die Transformationsmatrix multipliziert; liefert **self**.

– preScale :(float)sx :(float)sy :(float)sz

Skalierungsmatrix für die Skalierungsfaktoren *sx*, *sy*, *sz* in die Achsenrichtungen im Objektkoordinatensystem des Empfängers wird links an die Transformationsmatrix multipliziert; liefert **self**.

– scaleUniformly :(float)s

Skaliert im Koordinatensystem des Ancestors mit **scale:::** gleichmässig in alle Achsenrichtungen um den Faktor *s*; liefert **self**.

– preScaleUniformly :(float)s

Skaliert im Objektkoordinatensystem mit **preScale:::** gleichmässig in alle Achsenrichtungen um den Faktor *s*; liefert **self**.

– translate :(float)tx :(float)ty :(float)tz

Translationsmatrix für die Strecken *tx*, *ty*, *tz* in die Achsenrichtungen im Koordinatensystem des Ancestors des Empfängers wird rechts an die Transformationsmatrix multipliziert; liefert **self**.

– preTranslate :(float)tx :(float)ty :(float)tz

Translationsmatrix für die Strecken *tx*, *ty*, *tz* in die Achsenrichtungen im Objektkoordinatensystem des Empfängers wird links an die Transformationsmatrix multipliziert; liefert **self**.

Archivierung

– read :(NXTypedStream *)stream

Liest den Empfänger von dem typisierten *stream*; liefert **self**.

– write :(NXTypedStream *)stream

Archiviert den Empfänger auf den typisierten *stream*; liefert **self**.

– awake

Die Methode wird direkt nach dem Dearchivieren aufgerufen, um zusätzliche Initialisierungen zu tätigen; liefert **self**.

Literaturverzeichnis

[Adams86] Adams, S.: *Meta Methods: The MVC Paradigm*, in HOOPLA: Hooray for Object Oriented Programming Languages, Everette, WA.: Object Oriented Programming for Smalltalk Application Developers Association, Vol. **1**(4) (Juli 1986)

[Adel94] Adelstein; Golden Richard III; Schweibert; Parent; Singhal: *A Distributed Graphics Library System*, Software — Practice and Experience, Vol. **24**(4) Seite 363–376 (April 1994)

[AdobeDPS] Adobe Systems Inc., *Programming the Display PostScript System with NeXTstep*, Addison-Wesley Publishing Company, Inc., Reading 1991

[AdobeType1] Adobe Systems Inc., *Adobe Type 1 Font Format*, 1990

[Adobe] Adobe Systems Inc., *PostScript Bd. 1–3*, Addison-Wesley (Deutschland) GmbH, Bonn 1988, 1990

[BBB87] Bartels; Beatty; Barskey: *An Introduction to Splines for use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann Publishers, Inc., Los Altos 1987

[CabMS87] Cabral; Max; Springmeyer: *Bidirectional Reflection Functions from Surface Bump Maps*, Computer Graphics (SIGGRAPH'87 Proceedings) **21**(4) Seite 273–281 (July 1987)

[ClaP91] Claußen; Pöpsel: *Der Realität auf der Spur. Radiosity*, c't (Juni 6/1991), Seite 204–214; (August 8/1991), Seite 196–208

[ClaP93] Claußen; Pöpsel: *Himmel und Hölle. Dreidimensionale Texturen und ihre Implementierung*, c't (Januar 1/1993), Seite 160–170

[CohG85] Cohen; Greenberg: *The Hemi-Cube A Radiosity Solution for Complex Environments*, Computer Graphics (SIGGRAPH'85 Proceedings) **19**(3) Seite 31–40 (July 1985)

[DreCH88] Drebin; Carpenter; Hanrahan: *Volume Rendering*, Computer Graphics (SIGGRAPH'88 Proceedings) **22**(4) Seite 65–74 (Aug. 1988)

[EngR91] Engeln-Müllges; Reutter: *Formelsammlung zur Numerischen Mathematik mit Turbo Pascal-Programmen*, 3. Aufl. BI-Wiss.-Verl., Mannheim 1991

[FolVDFH90] Foley; vanDam; Feiner; Hughes: *Computer Graphics*, Addison-Wesley, Reading 1990

[ForB88] Forsey; Bartels: *Hierarchical B-Spline Refinement*, Computer Graphics (SIGGRAPH'88 Proceedings) **22**(4) Seite 205–212 (Aug. 1988)

[Green91] Greenberg, Donald P.: *Imaginäre Bauten*, Spektrum der Wissenschaft (April 4/1991), Seite 104–110

[Haeb90] Haeberli: *Paint by Numbers: Abstract Image Representations*, Computer Graphics (SIGGRAPH'90 Proceedings) **24**(4) Seite 207–214 (Aug. 1990)

[HanL90] Hanrahan; Lawson: *A Language for Shading and Lighting Calculations*, Computer Graphics (SIGGRAPH'90 Proceedings) **24**(4) Seite 289–298 (Aug. 1990)

[HorP89] Hornung; Pöpsel: *3-D à la carte*, c't (April 4/1989); (Mai 5/1989); (Juli 7/1989); (August 8/1989); (Oktober 10/1989)

[HosL91] Hoschek; Lasser: *Grundlagen der Geometrischen Datenverarbeitung*, B. G. Teubner, Stuttgart 1989

[Kajiya86] Kajiya, James T.: *The Rendering Equation*, Computer Graphics (SIGGRAPH'86 Proceedings) **20**(4) Seite 143–149 (Aug. 1986)

[KayK86] Kay; Kajiya: *Ray Tracing Complex Scenes*, Computer Graphics (SIGGRAPH'86 Proceedings) **20**(4) Seite 269–278 (Aug. 1986)

[KerR91] Kerninghan; Ritchie: *Programmieren in C*, Carl Hanser Verlag, München

[Lind89] Lindbloom, B. J.: *Accurate Color Reproduction for Computer Graphics Applications*, Computer Graphics (SIGGRAPH'89 Proceedings) **23**(3) Seite 117–126 (July 1989)

[Mathe] Div. Autoren: *Handbuch der Mathematik*, Buch und Zeit Verlagsgesellschaft mbH, Köln (VEB Bibliographisches Institut Leipzig, 1986)

[MeiK91] Meinhardt; Klinger: *Schnecken- und Muschelschalen: Modellfall der Musterbildung*, Spektrum der Wissenschaft (August 8/1991), Seite 60–69

[Meinzer93] Meinzer, Hans-Peter: *Räumliche Bilder des Körperinneren*, Spektrum der Wissenschaft (Juli 7/1993), Seite 56–65

[Mitch94] Mitchell, William J.: *Digitale Photomanipulation*, Spektrum der Wissenschaft (April 4/1994), Seite 82–87

[NeXTDoc] NeXT Computer, Inc. *3.0, 3.1 Documentation* u.a. **17 – 3D Graphics Kit**, **Objective C**, 1992, 1993

[NewS86] Newman; Sproul: *Grundlagen der interaktiven Computergrafik*, McGraw-Hill Book Company GmbH, Hamburg 1986

[Peac85] Peachey, Darwyn R.: *Solid Texturing of Complex Scenes*, Computer Graphics (SIGGRAPH'85 Proceedings) **19**(3) Seite 279–286 (July 1985)

[PerH89] Perlin; Hoffert: *Hypertexture*, Computer Graphics (SIGGRAPH'89 Proceedings) **23**(3) Seite 253–263 (July 1989)

[Perlin85] Perlin, Ken: *An Image Synthesizer*, Computer Graphics (SIGGRAPH'85 Proceedings) **19**(3) Seite 287–296 (July 1985)

[Phys] Div. Autoren: *Metzler Physik*, J. B. Metzlersche Verlagsbuchhandlung, Stuttgart 1979

[PixQRM] Pixar: *Quick RenderMan Interface and Implemenation Specification*

[PixSpec] Pixar: *Pixar RenderMan Interface Specification, Version 3.1* Aktuelle Version des Interfaces ist 3.4. Eine überarbeitete und erweiterte Version 4.0 ist in Arbeit oder schon erhältlich.

[PreS85] Preparata; Shamos: *Computational Geometry, an Introduction* Springer-Verlag Inc., New York 1985

[QuiM91] Quien; Müller: *Computergraphik und gotische Architektur*, Spektrum der Wissenschaft (Dezember 12/1991), Seite 120–127, Seite 128–133

[ReeSC87] Reeves; Salsin; Cook: *Rendering Antialialised Shadows with Depth Maps*, Computer Graphics (SIGGRAPH'87 Proceedings) **21**(4) Seite 283–291 (July 1987)

[RogA90] Rogers; Adams: *Mathematical Elements for Computer Graphics*, McGraw-Hill, Inc., New York 1976, 1990

[RogE90] Rogers; Earnshaw: *Computer Graphics Techniques: Theory and Practice*, Springer-Verlag Inc. New York 1990

[Rogers85] Rogers, David F.: *Procedural Elements for Computer Graphics*, McGraw-Hill, Inc., New York 1976, 1985

[ShaC88] Shantz; Chang: *Rendering Trimmed NURBS with Adaptive Forward Differencing*, Computer Graphics (SIGGRAPH'88 Proceedings) **22**(4) Seite 189–198 (Aug. 1988)

[Smith87] Smith, Alvy Ray: *Planar 2-Pass Texture Mapping and Warping*, Computer Graphics (SIGGRAPH'87 Proceedings) **21**(4) Seite 263–272 (July 1987)

[Strou91] Stroustrup, Bjarne: *Die C++ Programmiersprache* 2. Aufl., Addison-Wesley (Deutschland) GmbH, Bonn 1992

[Turk91] Turk, Greg: *Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion*, Computer Graphics (SIGGRAPH'91 Proceedings) **25**(4) Seite 289–298 (Aug. 1991)

[Upstill89] Upstill, Steve: *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*, Addison-Wesley, Reading 1989

[Uter93a] Utermöhle, Michael: *Rendering im Netz*, iX (September 9/1993), Seite 50–60

[Uter93b] Utermöhle, Michael: *Grafik-Standard*, iX (September 9/1993), Seite 148–151

[WalCG87] Wallace; Cohen; Greenberg: *A Two-Pass Solution to the Rendering Equation: A Sythesis of Ray Tracing and Radiosity Methods*, Computer Graphics (SIGGRAPH'87 Proceedings) **21**(4) Seite 311–321 (July 1987)

[Will89] Willim, Bernd: *Leitfaden der Computer Grafik — Visuelle Informationsdarstellung mit dem Computer*, Drei-R-Verlag, Berlin 1989

[WitK91] Witkin; Kass: *Reaction-Diffusion Textures*, Computer Graphics (SIGGRAPH'91 Proceedings) **25**(4) Seite 299–308 (Aug. 1991)