Großer Beleg

Flexible Raytracerarchitekturen zur Berechnung komplexer optischer Phänomene

Oliver Franzke ©2003
Matrikelnummer: 2774039

Inhaltsverzeichnis

1 Einleitung 1
2 Entwurf einer flexiblen Raytracerarchitektur 2
2.1 Das Prinzip des Raytracings 2
2.1.1 Schnittbestimmung 2
2.1.2 Sichtbarkeitsbestimmung 4
2.1.3 Shading 5
2.1.4 Zusammenfassung und Ausblick 9
2.2 Die Analyse der Anforderungen 10
2.2.1 Die Anforderungen an den Raytracer 10
2.2.2 Abgrenzung von einem professionellen Renderer 10
2.2.3 Einige Anmerkungen zu den verwendeten Konventionen 11
2.3 Die Architektur 12
2.3.1 Repräsentation der Szene 12
2.3.2 Daten des Schnitts 16
2.3.3 Berechnung der Beleuchtung 16
2.3.4 Materialien und Texturen 17
2.3.5 Hilfsobjekte 21
2.3.6 Bildsynthese 23
2.3.7 Einige Anmerkungen zur Implementierung 24
2.3.8 Zusammenfassung und Ausblick 25
2.3.9 Ergebnisbilder 25
3 Fallstudie: 3D Studio Max® 28
3.1 Einführung 28
3.2 Die Architektur 28
3.2.1 Abhängigkeiten und Animierbarkeit 28
3.2.2 Repräsentation der Szene 29
3.2.3 Daten des Schnitts 32
3.2.4 Lichtobjekte und Beleuchtung eines Punktes 33
3.2.5 Materialien und Texturen 35
3.2.6 Rendering 37
3.2.7 Serialisierung 38
3.2.8 Import und Export von Szeneninformationen 39
3.2.9 Implementierung eines Exporters 39
3.2.10 Zusammenfassung und Ausblick 40

Danksagung 41
Schlusswort 41
Weiterführende Literatur 42
Quellennachweis 43

Abbildungsverzeichnis

Abbildung 2. 1 – Der Strahl 2
Abbildung 2. 2 – Die Ebene 2
Abbildung 2. 3 – Das Prinzip einer Fotokamera 4
Abbildung 2. 4 – Die virtuelle Kamera 4
Abbildung 2. 5 – Sichtbarkeitsbestimmung (Beispiel) 5
Abbildung 2. 6 – Das Gesetz von Lambert 6
Abbildung 2. 7 – Das Phongsche Beleuchtungsmodell 6
Abbildung 2. 8 – Schattentest durch Raytracing 7
Abbildung 2. 9 – Reflektion 7
Abbildung 2. 10 – Refraktion 8
Abbildung 2. 11 – Das Geometriemodul 12
Abbildung 2. 12 – Vertex - Normalen 13
Abbildung 2. 13 – Das Polygonmesh 14
Abbildung 2. 14 – Das 3dEntity 14
Abbildung 2. 15 – Szene und Szeneneinträge 15
Abbildung 2. 16 – SampleInfo – Struktur 16
Abbildung 2. 17 – Das Lichtmodul 17
Abbildung 2. 18 – Das Materialmodul 18
Abbildung 2. 19 – Material und MapHolder 19
Abbildung 2. 20 - Materialassoziation 20
Abbildung 2. 21 – Das Bildmodul 21
Abbildung 2. 22 – Laden von Szenen 22
Abbildung 2. 23 - Dateimanager 22
Abbildung 2. 24 - Raytracer 24
Abbildung 2. 25 – Geometrische Primitive 26
Abbildung 2. 26 - Burg 26
Abbildung 2. 27 - Reflektionen 27
Abbildung 2. 28 – Schachbrett 27

Abbildung 3. 1 – Abhängigkeiten und Animierbarkeit 29
Abbildung 3. 2 – INodes und Objects 30
Abbildung 3. 3 - Mesh 31
Abbildung 3. 4 – ShadeContext 32
Abbildung 3. 5 – Das Lichtmodul 34
Abbildung 3. 6 – Das Materialmodul 36
Abbildung 3. 7 – Serialisierung 39

Ich bin von der Wissenschaft tief beeindruckt.
Ohne sie gäbe es nicht all diese wunderbaren Dinge,
mit denen wir uns heute herumschlagen dürfen.
Sidney Harris

1 Einleitung

Die Bedeutung von computergenerierten Bildern und Animationen ist in den letzen Jahren explosionsartig gestiegen. Diese Entwicklung hängt von mehreren Faktoren ab. Erst einmal ist die ständig größer werdende Rechenleistung der Computer bei gleichzeitig sinkenden Preisen zu bemerken. Brauchte man früher noch einen extrem teuren Supercomputer oder eine ebenso kostspielige Renderfarm, so können heute auf herkömmlichen PCs nahezu photorealistische Bilder in nur wenigen Minuten, sogar von Amateuren, erzeugt werden.
Einhergehend mit dieser technischen Entwicklung, stieg auch das Interesse der Medien an der dreidimensionalen Technik, da die Kosten- und Zeitansprüche sanken.
Heutzutage wird nahezu überall bei Film und Fernsehen eine digitale Bearbeitung der Daten durchgeführt und nicht nur das, viele Einstellungen werden von vornherein im Computer produziert.
Diese Möglichkeiten stehen dabei nicht nur großen Hollywoodproduktionen zur Verfügung. Selbst die Werbeindustrie und Fernsehsender nutzen sie bereits ausgiebig. Ob nun virtuelle Außerirdische in der Werbung oder der Vorspann einer täglichen Nachrichtensendung, überall werden die computergenerierten Bilder verwendet.
In großen Film- und Kinoproduktionen werden fantastische Welten gezeigt, die nur schwer und teilweise überhaupt nicht durch herkömmliche Techniken erstellt werden können. Mittlerweile ist diese Entwicklung so weit fortgeschritten, dass ganze Fernsehserien ausschließlich mit dem Computer hergestellt werden.
Das Prinzip dahinter ist aber stets dasselbe. In einem Modellierungsprogramm werden die dreidimensionalen Szenen erstellt. Danach wird diese Beschreibung der Objekte durch eine weitere Software in ein Bild übersetzt. Dieser Vorgang wird auch „rendern“ (auf Deutsch: übersetzen) genannt. Eine gängige Methode dies zu tun, ist das so genannte „Raytracing“, also die Strahlenverfolgung. Hierbei werden virtuelle Lichtstrahlen in die Szene geschossen und dann bei ihrer Interaktion mit den Objekten beobachtet, wodurch der Farbwert eines Pixels berechnet wird. Erfolgt dies für jeden Pixel, entsteht ein computergeneriertes Bild.
Professionelle Komplettpakete, die sowohl das Modellieren als auch das Rendern ermöglichen, sind zum Beispiel 3D Studio Max® von Discreet, Lightwave® von NewTek und Maya® von Alias|Wavefront. Häufig wird jedoch ein Werkzeug nicht für alle Schritte des Prozesses verwendet, sondern für jede Phase ein anderes. So ist es durchaus üblich, dass eine Szene zwar mit Maya® modelliert, dann aber mit einer ganz anderen Software in ein Bild transformiert wird, zum Beispiel mit Renderman® von Pixar. Die Gründe dafür sind, dass der integrierte Renderer von Maya® bestimmte Effekte nicht unterstützt, die aber benötigt werden oder dass er einfach zu langsam oder zu unstabil für die Produktion ist.
So lässt sich seit dem Jahr 2000 beobachten, dass sich die Programme immer mehr auf eine spezielle Phase konzentrieren. So gibt es heute eine Vielzahl von professionellen Renderern, zum Beispiel Mental Ray® von mental images, Brazil® von Splutterfish oder Final Render® von cebas, um nur einige zu nennen.
Durch das wachsende Interesse der Medien an dieser Technologie wurde das Augenmerk vieler Wissenschaftler auf die 3D - Grafik gelenkt. Durch ihre Forschung wird sich wohl auch in Zukunft die Qualität und Geschwindigkeit der Ergebnisse weiter erhöhen.
Der Unterschied zwischen realen und künstlichen Bildern wird immer geringer und in vielen Fällen ist er überhaupt nicht mehr auszumachen!

Es ist nicht genug zu wissen,
man muß auch anwenden;
es ist nicht genug zu wollen,
man muß auch tun.
Johann Wolfgang von Goethe

2 Entwurf einer flexiblen Raytracerarchitektur

2.1 Das Prinzip des Raytracings

Es soll nicht Ziel dieses Belegs sein, das Raytracing in allen Details zu erklären.
In diesem Unterkapitel wird deshalb lediglich auf die für den konkreten Raytracer wichtigen Aspekte des Prinzips der Strahlenverfolgung eingegangen.
Weiterführende Informationen sind in der im Anhang aufgeführten Literatur zu finden.
Leser, die bereits Kenntnisse über das Prinzip des Raytracings besitzen, können also bedenkenlos dieses Unterkapitel überspringen.
In den folgenden Abschnitten wird ein dreidimensionaler affiner Raum vorausgesetzt.

2.1.1 Schnittbestimmung

Alle Punkte auf einem Strahl können folgendermaßen beschrieben werden:

$$\mathbf{P}(t) = \mathbf{P}_0 + t*(\mathbf{P}_1 - \mathbf{P}_0) \quad \text{für} \quad t = [0,1] \qquad \mathbf{(2.1)}$$

$\mathbf{P}_0$ wird auch Ursprung und $\mathbf{P}_1$ als Ziel des Strahls bezeichnet. Durch den Parameter t wird eine lineare Interpolation zwischen diesen beiden Punkten durchgeführt. Im folgenden ist mit $\vec{\mathbf{a}}$ der Vektor von $\mathbf{P}_0$ zu $\mathbf{P}_1$ gemeint, dieser wird auch Richtungsvektor des Strahls genannt. Er ist in der Regel unnormalisiert!

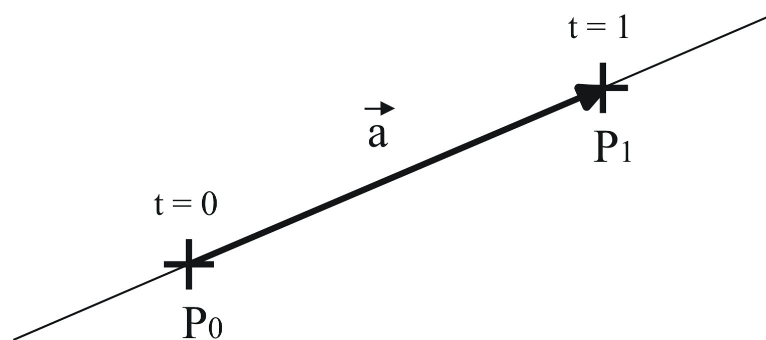

Abbildung 2. 1 – Der Strahl

Die Szene selbst besteht aus geometrischen Objekten, die sich durch einen mathematischen Ausdruck darstellen lassen. Beim Raytracing werden die voluminösen Körper durch ihre Oberfläche beschrieben. Diese Darstellungsform wird auch als b-rep (Abkürzung für **B**oundary **Rep**resentation) bezeichnet.
Das einfachste Beispiel ist sicherlich eine Ebene. Sie kann durch einen Punkt **U** im Raum und einen Vektor $\vec{\mathbf{n}}$, der auch Normale genannt wird, beschrieben werden. Die Normale steht stets senkrecht auf der Fläche.

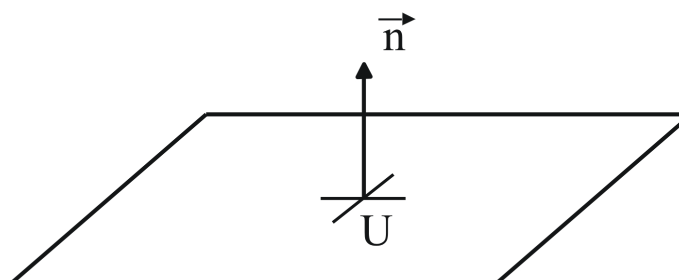

Abbildung 2. 2 – Die Ebene

Entgegen der Darstellung in Abbildung 2.2 besitzt die Ebene eine unendliche Ausdehnung. Mathematisch lässt sie sich folgendermaßen erfassen:

$$\vec{\mathbf{n}} \circ \vec{\mathbf{x}} + d = 0 \qquad \textbf{(2.2)}$$

wobei $d = -\vec{\mathbf{n}} \circ \vec{\mathbf{u}}$ gilt und $\vec{\mathbf{u}}$ der Ortsvektor des Punktes **U** darstellt.
Alle Punkte **X** des Raums, die die Gleichung (2.2) erfüllen, bilden die Ebene, wobei mit $\vec{\mathbf{x}}$ der Ortsvektor des Punktes **X** gemeint ist.

Die Basisoperation des Raytracings ist es nun alle Schnittpunkte eines Strahls mit einem Objekt zu finden. Im Falle der Ebene kann der Strahl unendlich viele, genau einen oder gar keinen Schnittpunkt besitzen.
Zuerst sollte festgestellt werden, ob der Strahl parallel zur Ebene verläuft. Dies kann schnell durch das Skalarprodukt der Normale der Ebene mit dem Richtungsvektor des Strahls überprüft werden. Wenn $\vec{\mathbf{n}} \circ \vec{\mathbf{a}} = 0$ gilt, so verläuft der Strahl parallel zur Ebene. Um zu berechnen ob der Strahl nun in der Ebene liegt und damit unendlich viele Schnittpunkte besitzt, gilt es festzustellen ob ein beliebiger Punkt des Strahls in der Ebene liegt. Dazu kann man zum Beispiel den Ortsvektor des Ursprungs $\mathbf{P_0}$ in Gleichung (2.2) einsetzen. Ist die Gleichung erfüllt so liegt der Strahl in der Ebene. Ist sie nicht erfüllt so verfehlt er sie und es existiert kein Schnittpunkt mit der Ebene.
Häufig wird jedoch $\vec{\mathbf{n}} \circ \vec{\mathbf{a}} \neq 0$ gelten und somit existiert höchstens genau ein Punkt, der sowohl auf dem Strahl als auch auf der Ebene liegt. Um die Koordinaten von eben diesen Schnittpunkt zu berechnen, setzt man die (vektorisierte) Gleichung (2.1) in die Ebenenbeschreibung (2.2) ein:

$$\vec{\mathbf{n}} \circ \left(\vec{\mathbf{P}}_0 + t * \vec{\mathbf{a}}\right) + d = 0$$

Durch Umstellung erhält man:

$$\vec{\mathbf{n}} \circ \vec{\mathbf{P}}_0 + \vec{\mathbf{n}} \circ \left(t * \vec{\mathbf{a}}\right) + d = 0$$

$$t * \vec{\mathbf{n}} \circ \vec{\mathbf{a}} = -\vec{\mathbf{n}} \circ \vec{\mathbf{P}}_0 - d$$

$$\mathrm{t} = -\frac{\vec{\mathbf{n}} \circ \vec{\mathbf{P}}_0 + \mathrm{d}}{\vec{\mathbf{n}} \circ \vec{\mathbf{a}}} \qquad \textbf{(2.3)}$$

Gilt $0 \leq t \leq 1$ so ist ein Schnittpunkt des Strahls mit der Ebene gefunden. Die Koordinaten können dann berechnet werden, indem man *t* in Gleichung (2.1) einsetzt. Ist die genannte Bedingung für *t* nicht erfüllt, wird die Ebene vom Strahl verfehlt.

Eine ähnliche Berechnung kann nun für beliebige andere Grundkörper durchgeführt werden. Es folgt eine kleine Auswahl von Objekten mit ihren mathematischen Beschreibungen:

Kugel
$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2$$

Zylinder (offener Zylinder)
$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

Ellipsoid

$$\frac{(x-x_c)^2}{\alpha^2}+\frac{(y-y_c)^2}{\beta^2}+\frac{(z-z_c)^2}{\gamma^2}=1$$

Wobei mit x_c, y_c und z_c jeweils die Mittelpunktkoordinaten der Körper bezeichnet sind. Es ist leicht zu sehen, dass bei den drei genannten Objekten entweder kein, genau ein oder genau zwei Schnittpunkte mit einem Strahl existieren können. Bei dem Zylinder besteht zudem die Möglichkeit, dass sich unendlich viele Punkte finden lassen, die sowohl auf dem Strahl als auch auf dem Zylinder liegen.

2.1.2 Sichtbarkeitsbestimmung

Doch wie kann man mit der im Abschnitt 2.1.1 beschriebenen relativ einfachen Operation ein Bild erzeugen? Um dies zu verstehen sollte man sich zuerst die Aufnahme von Fotografien vor Augen führen. Durch eine Linse wird das Abbild der Umwelt auf einen lichtempfindlichen Film projiziert.

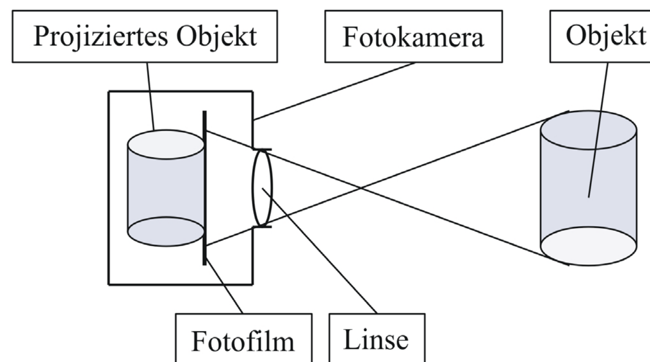

Abbildung 2. 3 – Das Prinzip einer Fotokamera

Genau diese Methode versucht man nun im Computer nachzuahmen. Der Fotofilm wird dabei durch eine so genannte „View Plane" ersetzt. Außerdem wird das Projektionszentrum in die Kameraposition verschoben.

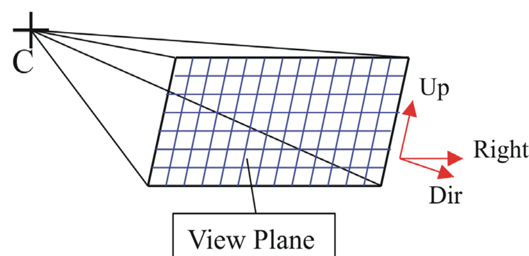

Abbildung 2. 4 – Die virtuelle Kamera

Mit **C** ist die Position der Kamera gemeint. Die Vektoren Up, Right und Dir stellen das Koordinatensystem der virtuellen Kamera dar. Wie man in der Abbildung außerdem erkennen kann ist die View Plane gerastert. Man kann sich das folgendermaßen vorstellen: Man transformiert das Bild, welches erzeugt werden soll, einfach an die Stelle der View Plane. Um nun den Farbwert eines Pixels zu bestimmen, legt man einen Strahl, mit dem Ursprung bei **C**, durch das Zentrum des zu berechnenden Pixels. Nun schneidet

man ihn mit allen in der Szene vorhandenen Objekten. Der naheste Schnittpunkt ist der sichtbare Oberflächenpunkt in diesem Pixel. Führt man diese Berechnung für jeden Pixel aus, so erhält man am Ende ein computergeneriertes Bild.
Diese Art von Sichtbarkeitsbestimmung wird auch Bildraumalgorithmus genannt.

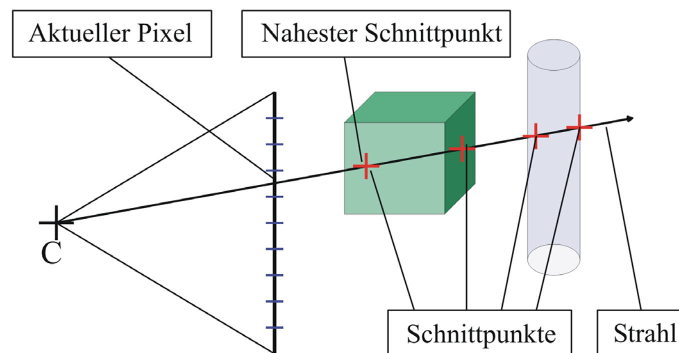

Abbildung 2. 5 – Sichtbarkeitsbestimmung (Beispiel)

2.1.3 Shading

Im vorherigen Abschnitt wurde verdeutlicht, wie man mit Hilfe des Prinzips der Strahlenverfolgung für jeden Pixel den von der Kamera aus sichtbaren Oberflächenpunkt berechnen kann. In diesem Abschnitt soll nun dargestellt werden, wie durch ein Beleuchtungsmodell der Farbwert des jeweiligen Pixels errechnet wird. Das hier beschriebene Modell ist ein erweitertes lokales Beleuchtungsmodell. Der Vorgang der Bestimmung des Farbwerts wird auch Shading (auf Deutsch: Schattierung) genannt.
In einem lokalen Beleuchtungsmodell hängt der Farbwert an einem Punkt im Raum von folgenden Parametern ab:

- Position
- Orientierung (Normale) und
- Material des Oberflächenpunktes
- vorhandene Lichtquellen

In dem hier präsentierten Modell werden außerdem alle anderen Objekte der Szene, zumindest teilweise, in die Berechnung mit einbezogen.
Um eine gewisse Grundhelligkeit in der Szene zu erzeugen, wird zunächst ein so genanntes Ambientes Licht hinzugefügt. Die Intensität **I** (Helligkeit) an einem Oberflächenpunkt wird dann folgendermaßen berechnet:

$$I = k_a * I_a , \qquad \textbf{(2.4)}$$

wobei k_a ein materialabhängiger Reflexionskoeffizient und I_a die Intensität der ambienten Lichtquelle darstellt. Es ist leicht zu sehen, dass mit dieser Berechnungsvorschrift keine Farbvariation über die Oberfläche stattfindet. Dafür sollen die nicht ambienten Lichtquellen sorgen. Der Einfachheit halber wird hier nur auf eine Punktlichtquelle eingegangen.
Um die Helligkeit des Punktlichts an einem Oberflächenpunkt zu berechnen, bedient man sich dem Gesetz von Lambert für diffuse Reflektoren, wonach die Intensität des von einem diffusen Material reflektierten Lichts von dem Winkel zwischen der Normale der Oberfläche und der Richtung, aus der das Licht eingestrahlt wird, abhängt. Genauer gesagt hängt die Intensität vom Kosinus des genannten Winkels ab.

Auch empirisch wird dies schnell klar, da eine Oberfläche um so heller erscheint, je kleiner dieser Winkel wird. Befindet sich die Lichtquelle hinter der Oberfläche, so werden an dem Punkt auch keine Photonen von ihr empfangen.

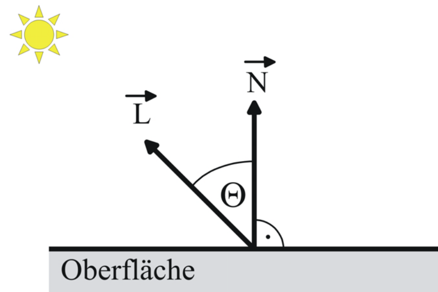

Abbildung 2. 6 – Das Gesetz von Lambert

Bei der Berechnung der Intensität wird (2.4) um einen diffusen Term erweitert:

$$I = k_a * I_a + k_d * I_{Lq} * \max\left[0, \left(\vec{\mathbf{N}} \circ \vec{\mathbf{L}}\right)\right], \qquad \textbf{(2.5)}$$

wobei mit k_d ein Reflektionsfaktor und mit I_{Lq} die Intensität der Lichtquelle gemeint ist. Da die Vektoren $\vec{\mathbf{N}}$ und $\vec{\mathbf{L}}$ normalisiert sind, kann der Kosinus zwischen ihnen durch das Skalarprodukt berechnet werden.
Mit Hilfe des Modells von Phong ist es möglich die Helligkeitsbestimmung (2.5) um ein Glanzlicht zu erweitern. Dabei wird der Kosinus des Winkels zwischen dem an der Normale $\vec{\mathbf{N}}$ reflektierten Lichtvektor $\vec{\mathbf{R}}$ und dem Sichtvektor $\vec{\mathbf{V}}$ betrachtet.

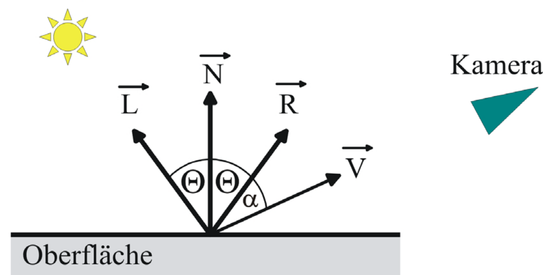

Abbildung 2. 7 – Das Phongsche Beleuchtungsmodell

Die Berechnung der Intensität sieht nun folgendermaßen aus:

$$I = k_a * I_a + I_{Lq} * \left(k_d * \max\left[0, \left(\vec{\mathbf{N}} \circ \vec{\mathbf{L}}\right)\right] + k_s * \left(\vec{\mathbf{V}} \circ \vec{\mathbf{R}}\right)^n\right), \qquad \textbf{(2.6)}$$

wobei k_s ein materialabhängiger (spekularer) Reflektionskoeffizient ist. Mit n kann die Schärfe des Glanzlichts reguliert werden. Je größer der Wert für n ist, desto kleiner wird das Highlight.
Bis zu der Formel (2.6) wurde ein vollständig lokales Beleuchtungsmodell beschrieben. Nun wird die Formulierung um einige Aspekte eines globalen Modell erweitert. Zuerst einmal kann man die Schattenberechnung mit einbeziehen:

$$I = k_a * I_a + S * I_{Lq} * \left(k_d * \max\left[0, \left(\vec{\mathbf{N}} \circ \vec{\mathbf{L}}\right)\right] + k_s * \left(\vec{\mathbf{V}} \circ \vec{\mathbf{R}}\right)^n\right), \qquad \textbf{(2.7)}$$

wobei S genau dann den Wert Eins annimmt, wenn der betrachtete Oberflächenpunkt nicht im Schatten eines anderen Objekts liegt.
Wird das Licht blockiert, d.h. der Punkt liegt im Schatten, so nimmt S den Wert Null an. Dadurch wird der Pixel also nicht durch diese Lichtquelle erhellt.
Doch wie bestimmt man nun die Entscheidungsvariable S?
Durch das Prinzip der Strahlenverfolgung lässt sich auch diese Frage leicht beantworten. Um den Wert zu berechnen schießt man also einen Strahl von der Position der Lichtquelle aus zu dem betrachteten Oberflächenpunkt. Lässt sich ein Schnittpunkt auf dem Strahl finden, so liegt der betrachtete Punkt im Schatten und man setzt S gleich Null, andernfalls wird S auf Eins gesetzt.

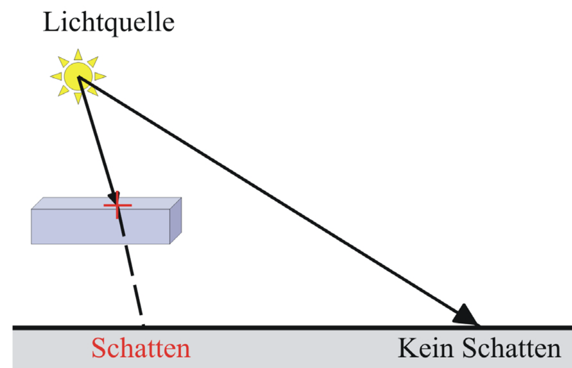

Abbildung 2. 8 – Schattentest durch Raytracing

Mit der in (2.7) dargestellten Form können rauhe Materialien beschrieben werden. Doch wie können spekulare Oberflächen modelliert werden, wie einen Spiegel oder Glas? Das Glanzlicht, welches durch das PHONGsche Modell erzeugt wird, reicht zur Simulation solcher Effekte nicht aus. Spiegelungen und Transmissionen sollten zusätzlich in das Shading mit einbezogen werden.
Und tatsächlich ist es sehr einfach möglich, durch das Raytracing solche Effekte zu modellieren. Betrachten wir zunächst die Reflektion. Um sie zu berechnen braucht man eigentlich nichts weiter tun als den Sichtvektor $\vec{\mathbf{V}}$ an der Oberflächennormale $\vec{\mathbf{N}}$ zu spiegeln und dann einen weiteren Strahl vom betrachteten Punkt aus in die reflektierte Richtung $\vec{\mathbf{V}}'$ zu verfolgen.

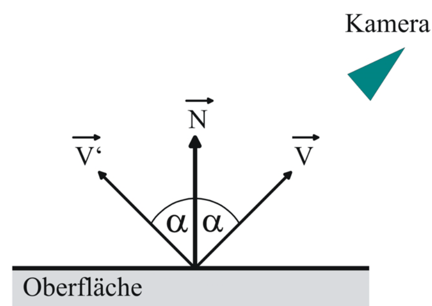

Abbildung 2. 9 – Reflektion

Formel (2.7) mit der zusätzlichen Berechnung der Reflektion sieht folgendermaßen aus:

$$I = k_a * I_a + S * I_{Lq} * \left(k_d * \max\left[0, \left(\vec{\mathbf{N}} \circ \vec{\mathbf{L}}\right)\right] + k_s * \left(\vec{\mathbf{V}} \circ \vec{\mathbf{R}}\right)^n \right) + k_R * I_R, \qquad \textbf{(2.8)}$$

wobei I_R die Helligkeit der Reflektion und k_R ein Intensitätsfaktor darstellt. Besitzt der Strahl in die reflektierte Richtung $\vec{\mathbf{V}}'$ keinen Schnittpunkt mit einem Objekt der Szene, so ist I_R gleich Null. Lässt sich anderseits eine Oberfläche finden, die den Strahl schneidet, so muss I_R wieder mit Formel (2.8) berechnet werden. Hier liegt natürlich ein gewisses Problem, da es unter bestimmten Bedingungen, z.B. in einem vollverspiegeltem Raum, zu einer unendlichen Rekursion kommen kann. Relativ schnell löst man dies, in dem man die rekursive Berechnung bei einer gewissen maximalen Tiefe abbricht und I_R dann einfach gleich Null setzt.
Der wichtigste Fakt der Reflektion wurde jedoch noch nicht erwähnt, nämlich wie eigentlich der Sichtvektor $\vec{\mathbf{V}}$ an der Normale $\vec{\mathbf{N}}$ gespiegelt wird. Dies kann durch einfache Vektorrechnung durchgeführt werden:

$$\vec{\mathbf{V}}' = 2 * \vec{\mathbf{N}} * \left(\vec{\mathbf{V}} \circ \vec{\mathbf{N}}\right) - \vec{\mathbf{V}}$$

Die Berechungsvorschrift (2.8) mit einer Transmission zu erweitern sollte nun kein großes Problem darstellen, da prinzipiell genau wie bei der Reflektion vorgegangen werden kann:

$$I = k_a * I_a + S * I_{Lq} * \left(k_d * \max\left[0, \left(\vec{\mathbf{N}} \circ \vec{\mathbf{L}}\right)\right] + k_s * \left(\vec{\mathbf{V}} \circ \vec{\mathbf{R}}\right)^n \right) + k_R * I_R + k_T * I_T, \qquad \textbf{(2.9)}$$

wobei mit I_T die Helligkeit der Transmission und mit k_T der Intensitätsfaktor der Transmission gemeint ist. Die Berechnung des Transmissionsvektors ist allerdings aufwändiger als bei der Reflektion, denn es muss das SNELLsche Gesetz beachtet werden. Es beschreibt die Abhängigkeiten, die bei einem optischen Übergang von einem Material zu einem anderen betrachtet werden müssen. Dies bezeichnet man auch als Refraktion.

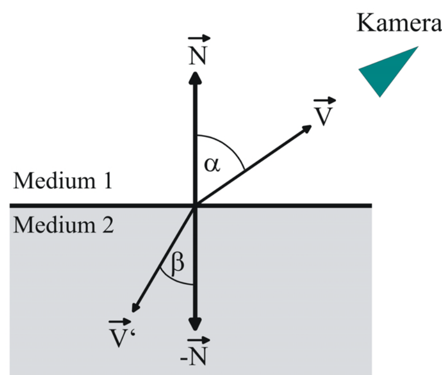

Abbildung 2. 10 – Refraktion

Mathematisch lässt sich das SNELLsche Gesetz durch den folgenden Ausdruck beschreiben:

$$\eta_{it} = \frac{\eta_i}{\eta_t} = \frac{\sin\beta}{\sin\alpha},$$

wobei η_i der Brechungsindex von Medium 1 und η_t der von Medium 2 ist. η_{it} wird auch als relativer Brechungsindex bezeichnet. Durch einige Umstellungen lässt sich der refraktierte Vektor auch direkt berechnen:

$$\vec{\mathbf{V}}' = \vec{\mathbf{N}} * \left(\eta_{it} * \left[\vec{\mathbf{N}} \circ \vec{\mathbf{V}}\right] - \sqrt{1 - \eta_{it}^2 * \left(1 - \left[\vec{\mathbf{N}} \circ \vec{\mathbf{V}}\right]\right)}\right) - \eta_{it} * \vec{\mathbf{V}}$$

2.1.4 Zusammenfassung und Ausblick

In diesem Unterkapitel wurde gezeigt, wie prinzipiell computergenerierte Bilder erzeugt werden können. Dabei wird die Welt durch mathematische Terme beschrieben und Licht als Strahl aufgefasst.
Wichtig ist jedoch, dass alle Objekte der Szene eine sehr viel größere Ausdehnung besitzen als die Wellenlänge des Lichts. Wäre dies nicht der Fall, so müssten zusätzliche Effekte, wie Beugung oder Interferenz, mit in die Berechnung einbezogen werden.
Zum Beispiel könnte das Doppelspaltexperiment mit dem oben beschriebenen Modell nicht realistisch nachgebildet werden.
Außerdem besitzen die Photonen in dieser virtuellen Welt, eine unendlich große Geschwindigkeit. Denn wird irgendwo in der Szene eine Lichtquelle „eingeschaltet“, dann ist der Effekt sofort zu sehen, d.h. das Licht benötigt keine Zeit um sich von einem Punkt in der Szene zu einem anderen zu bewegen. Die scheint im ersten Moment eine gewagte Voraussetzung zu sein, so wissen wir doch, dass die Lichtgeschwindigkeit niemals überschritten werden kann.
Unsere Szenen werden aber in der Regel eine sehr kleine Ausdehnung besitzen, so dass die Zeit, in der die Photonen von der Lichtquelle zu einem Oberflächenpunkt unterwegs sind, sowieso fast Null sein wird. Letztendlich würde der Aufwand einer physikalisch korrekten Berechnung, den Nutzen nicht rechtfertigen.
Wenn sehr große Szenen simuliert werden sollen, kann es aber durchaus nötig sein, dass auch die Ausbreitungsgeschwindigkeit mit einbezogen wird.
Weiterhin wurde beschrieben, wie der Farbwert eines Pixels im Ergebnisbild bestimmt werden kann. Mit dem erläuterten Beleuchtungsmodell lassen sich lokale Beleuchtungseffekte und perfekt spiegelnde oder refraktierende Lichtinteraktionen beschreiben.
Mit Hilfe des Raytracings lässt sich ebenfalls ein vollständiges globales Beleuchtungsmodell implementieren. Dieses so genannte Monte – Carlo - Raytracing simuliert dann zusätzlich nicht perfekt spiegelnde Reflektionen und Refraktionen. Außerdem ist es sogar möglich, dass diffuse Lichtinteraktionen realistisch berechnet werden können. Diese erweiterte Methode benötigt jedoch sehr viel mehr Rechenzeit.
Die Rechenzeit spielt ganz allgemein in der Praxis eine große Rolle. Deswegen werden zusätzliche räumliche Datenstrukturen verwendet, um die Strahlenanfrage zu beschleunigen. Die Idee hinter diesem Prinzip ist, von vorneherein viele Objekte erst gar nicht mit dem spezifizierten Strahl zu schneiden, weil er sich überhaupt nicht in der Nähe dieser Objekte befindet. Das bedeutet zwar eine gewisse Vorverarbeitung der Szene, die aber durch sehr viel höhere Rendergeschwindigkeit wieder wettgemacht wird.
Häufig wird das Raytracing sogar mit Methoden der Echtzeit – 3D Grafik kombiniert. Zum Beispiel kann der Z – Buffer – Algorithmus einer modernen Grafikkarte, die erste Strahlenanfrage sehr beschleunigen. Dann wird durch den Pixel kein Strahl gelegt und mit der Szene geschnitten, sondern anhand des Z – Buffers kann entschieden werden, welches geometrische Objekt in dem aktuellen Pixel zu sehen ist.
Diese Methoden setzen jedoch meist voraus, dass die gesamte Szene durch Dreiecksnetze beschrieben wird.
Nichtsdestotrotz ist es möglich durch den beschriebenen Ansatz eine photorealistische Darstellung von vielen Szenen zu erreichen.

2.2 Die Analyse der Anforderungen

In diesem Unterkapitel soll es darum gehen, die dem Großen Beleg zu Grunde liegende Aufgabe zu erläutern. Außerdem gilt es die zu erreichenden Anforderungen von denen eines professionellen Programms abzugrenzen.

2.2.1 Die Anforderungen an den Raytracer

Das Ziel dieser Arbeit ist es, die Architektur eines Raytracers zu entwickeln, der zu Lehrzwecken das Prinzip eines Renderers erklären soll. Es soll also das Zusammenspiel und die grundsätzliche Wirkungsweise der einzelnen Komponenten aufgezeigt werden. So weit es geht wird der Entwurf des Systems so flexibel wie möglich gestaltet, jedoch geht die Übersichtlichkeit und Verständlichkeit stets vor.
Auch werden nicht alle möglichen Optimierungen der Algorithmen besprochen, dazu wird es im Anhang weiterführende Literaturhinweise geben.
Der Leser soll durch diesen Beleg befähigt werden selbst einen Raytracer zu implementieren und dann Experimente und Erweiterungen an den einzelnen Komponenten durchzuführen. Obwohl die Codeausschnitte in C++ verfasst sind, sollte es keine Probleme geben das System in einer beliebigen anderen Sprache umzusetzen.

Die Anforderungen an den Raytracer sind:

- eine übersichtliche Architektur, die das Prinzip eines Renderers verständlich macht
- weitestgehend Flexibilität zu wahren
- Unterstützung mehrerer geometrischer Primitivkörper

2.2.2 Abgrenzung von einem professionellen Renderer

Typische Eigenschaften eines professionellen Renderers sind:

- maximale Qualität der erzeugten Bilder
- hohe Geschwindigkeit
- Flexibilität
- numerische Stabilität
- Unterstützung von nur wenigen Primitiven, wie Dreiecken und Volumenkörpern
- Mehrprozessorunterstützung
- verteiltes Rendern, z.B. in einer Renderfarm
- Unabhängigkeit von der Systemtechnik, wie Betriebssystemen und Prozessoren

Im Vergleich zu den im Abschnitt 2.2.1 genannten Anforderungen fallen große Unterschiede auf. Geschwindigkeit und Qualität nehmen in der Praxis einen sehr hohen Stellenwert ein, um diese Eigenschaften soll es in der zu entwickelnden Architektur nicht primär gehen, da sie häufig lediglich Spezialisierungen allgemeiner Prinzipien darstellen. In einem professionellen Renderer werden meist nur Dreiecke und Volumina als Szenenobjekte unterstützt. Eigenschaften die indirekt „nur“ der erhöhten Geschwindigkeit dienen, wie die Mehrprozessorunterstützung und das verteilte Rendern, bedeuten in Wirklichkeit extrem viel Planungs- und Programmieraufwand. Diese Themen sind ohne Frage sehr interessant, würden aber den Rahmen dieses Belegs bei weiten sprengen. Das Selbe gilt für die numerische Stabilität und die Unabhängigkeit von der Systemtechnik. Hiermit beschäftigen sich ganze Heerscharen von Wissenschaftlern.
Nun kann man sich die Frage stellen, ob der generelle Aufbau eines solchen Programms trotz der großen Unterschiede der Anforderungen noch realistisch erklärt werden kann. Dem ist natürlich so, denn obwohl die professionellen Renderer andere Ziele verfolgen, sind die grundlegenden Prinzipien der Bilderzeugung doch die Selben.

Als Beispiel soll hier der Unterschied bei den geometrischen Primitiven dienen. Obwohl die Strahlenverfolgung an sich eine Vielzahl von geometrischen Elementen als Szenenobjekte ermöglicht, so werden in der Praxis häufig lediglich Dreiecke verwendet.
Das Dreieck hat den Vorteil, dass es das kleinste Element ist, welches eine Fläche repräsentieren kann, wobei mit „klein" hier die Anzahl der Eckpunkte gemeint ist. Außerdem ist es möglich jede Oberfläche eines Körpers im Raum durch ein Dreiecksnetz zu beschreiben. Weiterhin kann man sich bei der Optimierung der Algorithmen auf Dreiecke beschränken, was neben weniger Programmieraufwand meistens auch eine höhere Geschwindigkeit zur Folge hat. Auf Grund dieser Eigenschaften beschränken sich professionelle Renderer auf Dreiecksnetze.
In Modellierungsprogrammen hingegen können häufig auch andere Primitive verwendet werden, wie zum Beispiel Freiformflächen und Quadriks. Diese werden dann je nach Qualitätsanspruch vor der Bilderzeugung in Dreiecksnetze transformiert.
Da das Prinzip der Bilderzeugung mit Hilfe von Strahlenverfolgung durch die Verwendung von unterschiedlichen geometrischen Basisobjekten deutlicher zum Vorschein kommt, werden diese von der zu entwickelnden Architektur unterstützt. Außerdem lassen sich mit wenig Aufwand komplexere Objekte erzeugen, weil diese nicht erst trianguliert werden müssen.

2.2.3 Einige Anmerkungen zu den verwendeten Konventionen

Bei der beschriebenen Architektur handelt es sich um einen objektorientierten Ansatz. Dadurch werden alle Diagramme zur Veranschaulichung des Softwaredesigns in der Unified Modeling Language (UML) dargestellt.
Der Quellcode ist in C++ geschrieben und folgt der Ungarischen Notation.

2.3 Die Architektur

In den zwei vorherigen Unterkapiteln wurde sowohl der mathematische Apparat als auch die Anforderungen an die Architektur des Raytracers beschrieben. Jetzt soll die Frage beantwortet werden, wie beides in einem System vereint werden kann.
In der Literatur und im Internet findet man sehr viele Informationen über die mathematischen Grundlagen des Raytracings. Die Eigenschaften und Anforderungen an einen professionellen Renderer werden dort ebenfalls erläutert.
Interessanterweise lassen sich aber über die Designprinzipien eines solchen Programms kaum Fakten finden. Dies liegt wohl auch daran, dass man die Architektur eines Renderers nicht so einfach verallgemeinern kann, da eine solche Software extrem komplex ist.
In gewisser Weise soll mit diesem Beleg versucht werden diese Informationslücke zu schließen. Dabei vertritt der Autor dieses Belegs die Meinung, dass sich zumindest bestimmte Fakten, wie das Zusammenspiel der einzelnen Module, prinzipiell verallgemeinern lassen.

2.3.1 Repräsentation der Szene

Die Welt soll durch unterschiedliche geometrische Grundkörper beschrieben werden. An dieser Stelle steht auch schon die erste größere Designentscheidung an. Wie repräsentiert man ein Objekt so, dass die konkrete Implementierung weitestgehend unabhängig von seiner Abstraktion ist? Es sollte einfach möglich sein neue Körper hinzuzufügen, ohne dass große Änderungen am Bilderzeugungsalgorithmus durchgeführt werden müssen.
Um diese Anforderung zu erfüllen wird die abstrakte Klasse **IMesh** eingeführt. Die wichtigste Aufgabe dieser Klasse ist es, eine Abstraktion der Schnittoperation bereitzustellen. Es ist leicht zu sehen, dass der Schnittpunkt mit einer Ebene anders berechnet wird als zum Beispiel der einer Kugel. Das heißt, die Ebene legt fest, wie der Schnittpunkt mit einem Strahl konkret ermittelt wird. (siehe Abschnitt 2.1.1)
Damit die unterschiedlichen Primitive auseinander gehalten werden können, sollte **IMesh** außerdem eine Typinformation beinhalten.

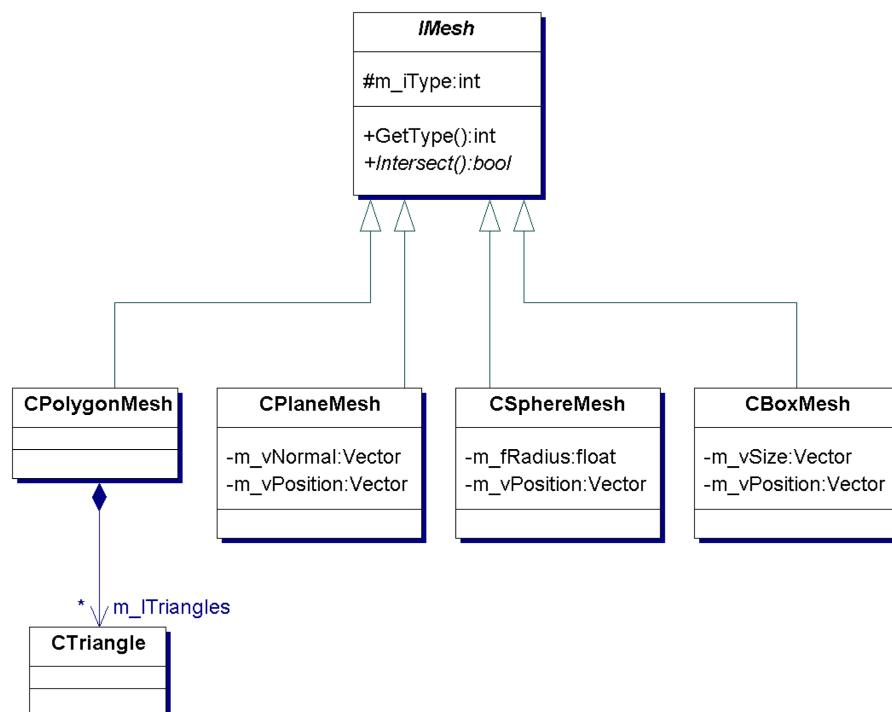

Abbildung 2. 11 – Das Geometriemodul

Wie in Abbildung 2.11 zu sehen ist, erfolgt die Abstraktion des Schnitts mit einem Strahl durch die Methode Intersect(). Jede der konkreten Klassen implementiert sie anders. Dass diese Abstraktion sinnvoll ist, kann auch daran erkannt werden, dass die verschiedenen Grundkörper jeweils andere Informationen für den Schnitt benötigen. Die Kugel benötigt zum Beispiel einen Radius und eine Position für die Schnittoperation. Eine Ebene setzt wiederum voraus, das eine Normale und eine Position vorhanden ist. Für das Dreiecks-netz wird nicht einmal ein Positionsvektor gefordert. Es lässt sich also keine einheitliche Datenmenge finden, die von allen Primitiven gebraucht wird.
Nun soll das **CPolygonMesh** noch etwas genauer betrachtet werden. In der obigen Abbildung beinhaltet diese Klasse lediglich eine Komposition von **CTriangle**s. Jetzt kann man sich die Frage stellen, wo die Position der Eckpunkte der Dreiecke gespeichert werden soll. Natürlich können sie direkt in der **CTriangle** – Klasse gehalten werden. Dieser Weg ist jedoch sehr speicheraufwändig, da ein und der selbe Eckpunkt häufig von mehreren Dreiecken geteilt wird. Deshalb wird eine weitere Klasse mit dem Namen **CVertex** in das System eingeführt. Sie hält die Position eines Polygoneckpunktes. In einem Dreieck muss nun lediglich eine Referenz auf jeden Eckpunkt vorhanden sein. Verschiedenen **CTriangle**s ist es somit möglich auf den Selben Vertex zu verweisen, ohne jedoch die Positionen redundant zu speichern.
Andererseits ist es aber nötig, dass die Normalen an den Polygoneckpunkten in der Klasse **CTriangle** verwaltet werden. Der Grund dafür liegt auf der Hand, wenn man sich einen polygonalen Würfel vorstellt. Obwohl ein Eckpunkt von bis zu sechs Dreiecken geteilt wird, existieren jedoch mindestens drei verschiedene Normalen an der Selben Stelle. Die gleiche Argumentation kann für die Texturkoordinaten durchgeführt werden, d.h. auch sie werden im Dreieck gespeichert und nicht im Eckpunkt.

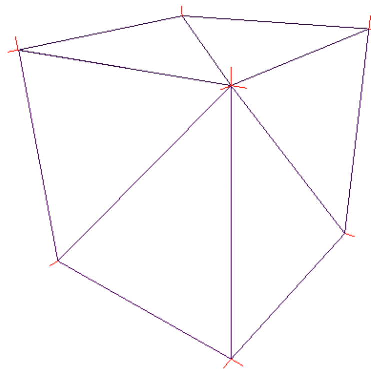

Abbildung 2. 12 – Vertex - Normalen

Zusätzlich zu den Normalen der Eckpunkte speichert das **CTriangle** noch eine eigene Normale. Diese steht aber, im Gegensatz zu den Vertex - Normalen, immer senkrecht auf dem Dreieck. Diese Normale wird für die Vorüberprüfung benötigt, ob der spezifizierte Strahl das Dreieck überhaupt schneiden kann. Zeigen nämlich der Richtungsvektor des Rays und die Normale des Dreiecks in die selbe Richtung, dann wird keine weitere Schnittüberprüfung benötigt, da der Betrachter quasi von hinten auf die Fläche schaut. An dieser Stelle sollte jedoch besondere Obacht gelten. Wenn nämlich ein Vertex verschoben wird, ohne dass die Normalen der Dreiecke neu berechnet werden, die an dem translierten Polygoneckpunkt „hängen“, kann es zu Fehlern beim Raytracing kommen. Eine Lösung dieses Problems wäre die Verwendung des Observer Entwurfsmusters. Noch einfacher ist es allerdings kurz vor der Bildsynthese alle Dreiecksnormalen neu zu berechnen.

Abbildung 2.13 zeigt das entsprechende Klassendiagramm.

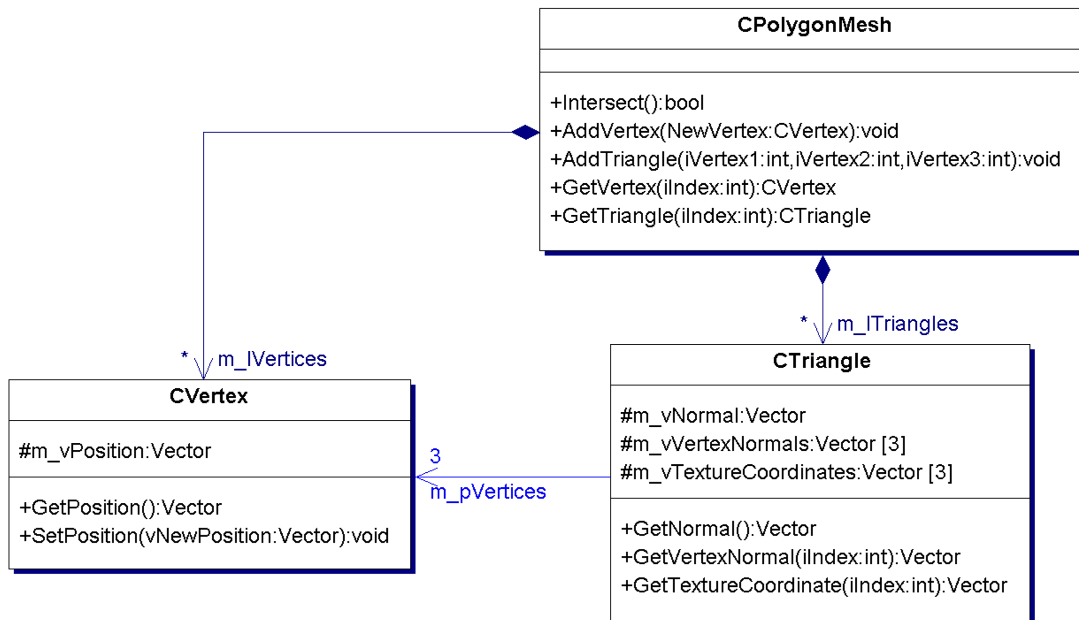

Abbildung 2. 13 – Das Polygonmesh

Natürlich reichen die mathematischen Grundkörper noch nicht aus um eine vollständige Szene zu beschreiben. Es werden zusätzlich Lichtquellen und eine Kamera benötigt. Diese Objekte besitzen alle eine Position und eine Orientierung im Raum. Das heißt es ist sinnvoll eine weitere Generalisierung, nämlich ein abstraktes **I3dEntity** einzufügen. Diese abstrakte Klasse ist unter anderem für die Verwaltung der Weltmatrix zuständig, welche sowohl Position als auch Orientierung des Objekts beinhaltet.
Da es in der Praxis nicht üblich ist **IMesh** selbst zu einer Spezialisierung von **I3dEntity** zu machen, wird dafür die Zwischenklasse **CGeometry** benötigt, die das **IMesh** hält. Dafür gibt es mehrere gute Gründe. Einerseits sollte zum Bespiel die Instanziierung von geometrischen Primitiven möglich sein, andererseits könnte **CGeometry** auch mehrere Objekte vom Typ **IMesh** besitzen und damit eine Level of Detail Darstellung zulassen.

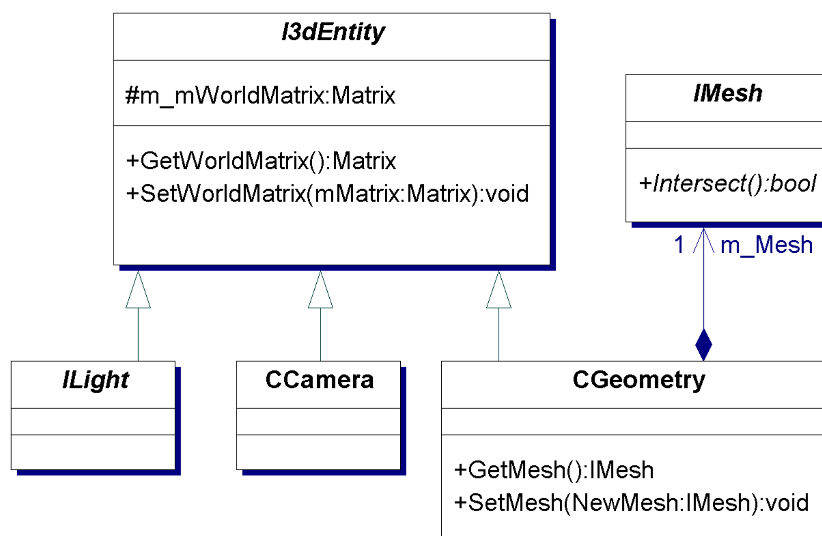

Abbildung 2. 14 – Das 3dEntity

Nun kann eine Klasse eingefügt werden, die alle Objekte der Szene speichert und verwaltet. Sie soll einfach **CScene** heißen. Die Welt besteht aber nicht nur aus dreidimensionalen Objekten. Materialien und Texturen usw. gehören ebenfalls zu ihr.
Da Materialien in der Regel keine Weltmatrix benötigen, muss eine weitere Generalisierung eingeführt werden, um Konsistenz zu wahren. Die abstrakte Klasse, die dies ermöglicht heißt **ISceneEntity**. Wie bei **IMesh** wird eine Information über den Typ des konkreten Szeneneintrags benötigt.
Diese Generalisierung kann außerdem nützliche Funktionen zur Verfügung stellen, wie zum Beispiel die Speicherung und Verwaltung der Namen der Einträge.
Durch **ISceneEntity** wird es möglich alle denkbaren Objekte einer Szene einheitlich zu organisieren. Alles was dafür getan werden muss, ist von der abstrakten Klasse **ISceneEntity** abzuleiten.
Durch diese Generalisierungen wird ein gutes Stück Flexibilität erreicht. Es ist nun sehr einfach ein neues Objekt hinzuzufügen, egal ob es Geometrie, eine Lichtquelle, ein Material oder etwas ganz anderes beschreibt.

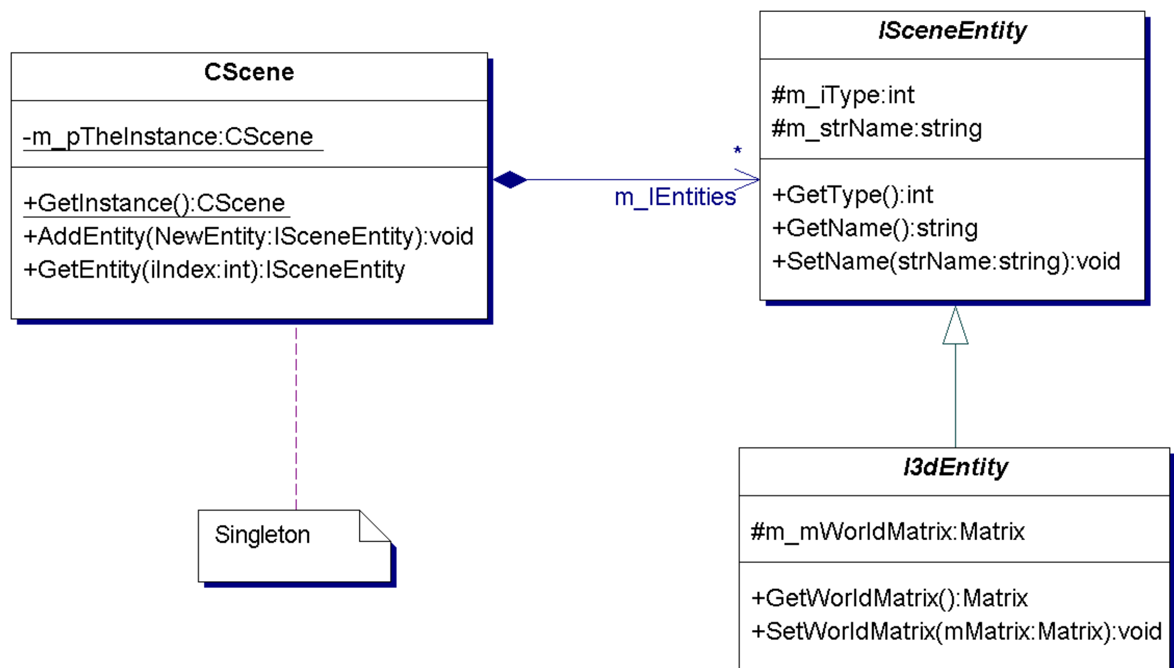

Abbildung 2. 15 – Szene und Szeneneinträge

Da häufig nur auf eine Art von Eintrag zugegriffen wird, kann es nötig werden, dass die konkreten Szeneneinträge zusätzlich jeweils in einer speziellen Liste gehalten werden müssen. Als Beispiel sollen hier die Lichtquellen genannt werden. Auf sie muss während des Shadings relativ häufig sehr schnell zugegriffen werden. Es wäre nicht sinnvoll jedes mal bei einer solchen Anfrage die gesamte Liste der Szeneneinträge nach Lichtquellen zu durchforsten. Deshalb werden die Lichter in einer zusätzlichen Liste in der Szene gespeichert. Dies bedeutet natürlich Redundanz, die aber durch eine höheren Rendergeschwindigkeit wieder wettgemacht wird.
An einem solchen Beispiel sieht man, mit welchen Problemen und Lösungsansätzen sich der Entwickler eines professionellen Renderers beschäftigen muss.

2.3.2 Daten des Schnitts

Dem einen oder anderen Leser ist eventuell aufgefallen, dass für die Intersect – Routine des Interfaces **IMesh** noch kein Rückgabewert angegeben wurde. Im ersten Moment neigt man dabei zu einer nicht vollständigen Lösung, nämlich einfach den Ortsvektor des Schnittpunktes zurückzugeben. Nur der Schnittpunkt reicht aber als Information nicht aus. Weiterhin sind Werte wie die Normale, die Texturkoordinate, das Material und die Distanz zum Ursprung des Strahls essentiell. Ohne Informationen über diese Eigenschaften kommt man bei der Berechnung des Farbwerts an dem Schnittpunkt sicherlich nicht sehr weit. Es wird also eine Datenstruktur gebraucht, die alle geforderten Informationen hält. Diese Struktur wird bei der Bildberechnung von der Intersect – Methode des **IMesh** gefüllt.

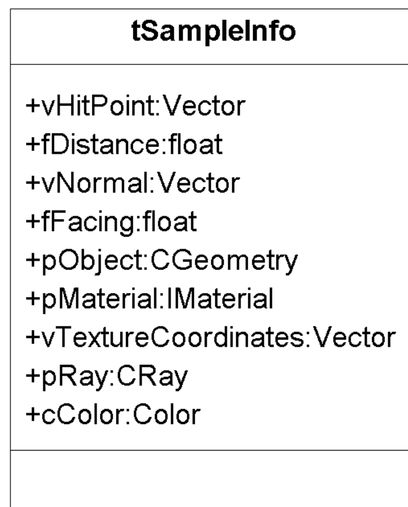

Abbildung 2. 16 – SampleInfo – Struktur

2.3.3 Berechnung der Beleuchtung

Nun sind alle Voraussetzungen vorhanden, um den Einfluss der Lichtquellen auf den aktuellen Schnittpunkt zu berechnen. Dem kritischen Leser ist wahrscheinlich aufgefallen, dass in der Abbildung 2.14 die Lichtquelle lediglich als abstrakte Klasse **ILight** dargestellt wurde. Dass dies Sinn macht, soll nun erläutert werden.
Im Abschnitt 2.1.3 wurde das Shading beschrieben. Die Berechnung des Farbwerts an einem Punkt im Raum mündete letztendlich in der Formel (2.9). Dabei wurde der Einfluss der Lichtquelle direkt in die genannte Berechnungsvorschrift integriert. Man könnte also meinen, dass die Ermittlung der Helligkeit gut im Material aufgehoben ist. Dem ist jedoch nicht so! Denn was passiert, wenn sich der Entwickler dazu entschließt neben einer Punktlichtquelle noch ein Scheinwerfer zu implementieren? Es ist leicht zu sehen, dass die Bestimmung der Helligkeit eines Punktlichts anders erfolgt als die eines Spotlights. Es sollte also eine Abstraktion von dieser Berechnung geben. Das Interface **ILight** wird somit um eine Routine Illuminate() erweitert. In der konkreten Implementierung wird dann entschieden, wie genau die Helligkeit an dem spezifizierten Ort berechnet wird.

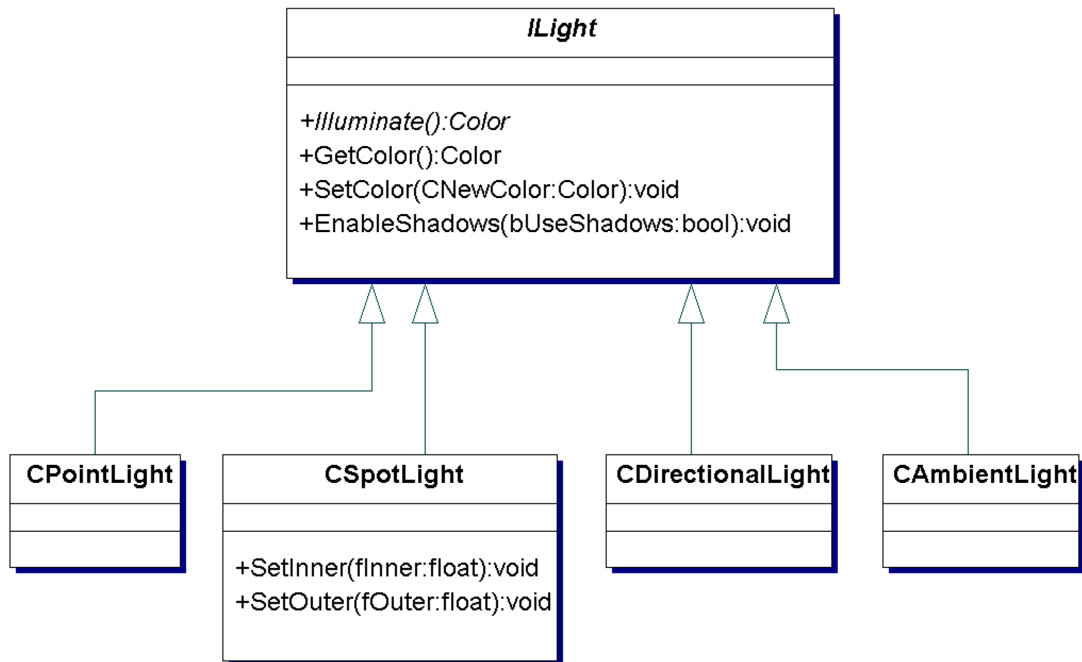

Abbildung 2. 17 – Das Lichtmodul

Mit den Methoden SetColor() und EnableShadows() soll angedeutet werden, dass die abstrakte Klasse **ILight** gemeinsame Eigenschaften verwaltet, die von alle Lichtquellen benötigt werden. Jede konkrete Spezialisierung kann zusätzlich eigene Werte besitzen, wie exemplarisch durch das **CSpotLight** dargestellt ist.
Die Routine Illuminate() besitzt jedoch keine Parameter, die den zu beleuchtenden Punkt näher spezifizieren. Vielmehr kann sich die konkrete Lichtquelle die aktuelle **tSampleInfo** - Struktur besorgen, wenn sie Informationen über den Punkt benötigt. Das ambiente Licht zum Beispiel braucht nämlich keine Werte des Schnittpunkts, um die Helligkeit zu bestimmen. Sie ist überall im Raum gleich.
Durch wiederholtes Anwenden des Strategy – Entwurfsmusters wird also erneut große Flexibilität erreicht. Mit der obigen Struktur treten keine Probleme auf, wenn eine weitere Art von Lichtquelle integriert werden soll. Es muss lediglich von **ILight** abgeleitet und die Illuminate – Methode anders implementiert werden.

2.3.4 Materialien und Texturen

Auch bei der Repräsentation der Charakteristik einer Oberfläche, kann wieder ernorme Flexibilität erzeugt werden. Dabei ist das Vorgehen ähnlich dem beim Geometrie- und Lichtmodul. Das heißt durch geschickte Trennung von Abstraktion und Implementierung sind spätere Erweiterungen bzw. Änderungen sehr leicht durchführbar.
Zunächst soll jedoch die Lücke in der Abbildung 2.15 geschlossen werden, wo noch keine Materialien und Texturen dargestellt wurden.
Wie in der Abbildung 2.18 erkenntlich ist, wurde **ISceneEntity** um die abstrakten Klassen **IMaterial** und **IMap** erweitert.

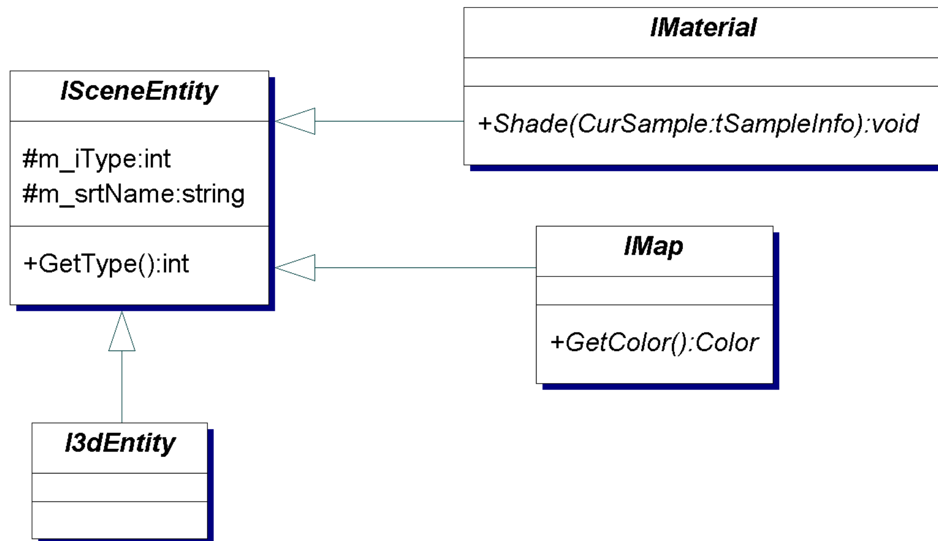

Abbildung 2. 18 – Das Materialmodul

Zunächst soll die Rolle der Map erläutert werden, da diese dann bei der genaueren Betrachtung des Materials benötigt wird. Die abstrakte Klasse **IMap** besitzt lediglich die Methode GetColor(). Das Berechnen eines Farbwerts ist ja auch die wesentlichste Aufgabe, die eine Textur zu erledigen hat. Wie sie das jedoch konkret ausführt, hängt sehr stark von der Art der Map ab. Während eine Bitmaptextur anhand der Texturkoordinaten einen Farbwert aus einer Bilddatei liest, wird ein prozedurales Schachbrettmuster wahrscheinlich die Position des betrachteten Punktes verwenden um zu entscheiden, ob der Farbwert nun Schwarz oder Weiß ist. Dieses kleine Beispiel zeigt schon, wie breit das Spektrum an dieser Stelle sein kann. Der Fantasie sind keine Grenzen gesetzt.
Der eine oder andere Leser mag sich nun vielleicht fragen, wo denn die GetColor() – Methode eigentlich die Texturkoordinaten herbekommt. Dies wird später noch näher erläutert. An dieser Stelle sei jedoch soviel gesagt, dass die aktuellen Werte der Schnittinformationsstruktur von einer zentralen Stelle abgerufen werden können.
Die abstrakte Klasse **IMaterial** besitzt ebenfalls nur eine definierte Methode, nämlich Shade(). Der Name der Routine verrät bereits ihre Funktion, sie führt die Farbberechnung aus. Wieder einmal legt erst die konkrete Unterklasse die explizite Implementierung fest. So kann sichergestellt werden, dass sowohl ein Lambert- als auch ein Phongmaterial integriert werden kann.
Tatsächlich ist die Darstellung der Materialien ein sehr entscheidender Knackpunkt des Raytracers. Werden an dieser Stelle zu viele Festlegungen gemacht, so sinkt die Flexibilität rapide ab und das soll ja gerade vermieden werden. Spätere Erweiterungen sind dann erschwert und enden häufig mit einem wilden Hacking. Andererseits kann eine gut definierte Schnittstelle die Arbeit wesentlich vereinfachen. Es sollte stets der Grundsatz gelten: Weniger ist mehr.
Wie in der Abbildung 2.18 ersichtlich ist, bekommt die Methode Shade() eine **tSampleInfo** – Struktur übergeben, d.h. der Punkt, für den ein Farbwert bestimmt werden soll, wird eindeutig festgelegt. Die Routine besitzt jedoch keine Parameter, die über die Lichtsituation an dem spezifizierten Schnittpunkt Aufschluss geben. Sie kann sich diese Werte aber besorgen, wenn sie benötigt werden sollten. Dem kritischen Leser fällt sicherlich weiterhin auf, dass die Methode keinen Rückgabewert besitzt, d.h. sie gibt den berechneten Farbwert nicht zurück. Der Grund dafür ist, dass die Shade - Routine den cColor – Wert der Sampleinformation füllt.

Der folgende kurze Codeabschnitt soll dieses Prinzip verdeutlichen:

```
/*******************************************************************/
/*  Shade                                                          */
/*                                                                 */
/*  Berechnet den Farbwert an der Stelle des Samples.              */
/*******************************************************************/
void CLambertMaterial::Shade( tSampleInfo &Sample )
{
   // Die (einzige) Instanz der Szene besorgen
   CScene* pScene = CScene::GetInstance();
   // Den Anteil der Lichtquellen zum Farbwert hinzuaddieren
   for( int i=0; i<pScene->GetLightCount(); i++ )
   {
         ILight* pLight = pScene->GetLight( i );
         // Den reflektierten Lichtanteil berechnen
         Sample.cColor += this->GetDiffuseColor()*pLight->Illuminate();
   }
}
```

Nun soll die Verbindung der Materialien mit den Texturen erläutert werden. Die einfachste Weise Maps zu integrieren, ist sicherlich jedem Material ein Array von **IMap** – Objekten zuzuordnen. Dann müsste der Entwickler aber alle Methoden zur Verwaltung der Texturen jedes mal aufs neue implementieren, wenn er ein weiteres Material hinzufügt. Dieser Weg kann vereinfacht und verallgemeinert werden!
Es wird eine (abstrakte) Verwaltungsklasse benötigt, die genügend Funktionalität zur Verfügung stellt. Sie soll **IMapHolder** heißen.

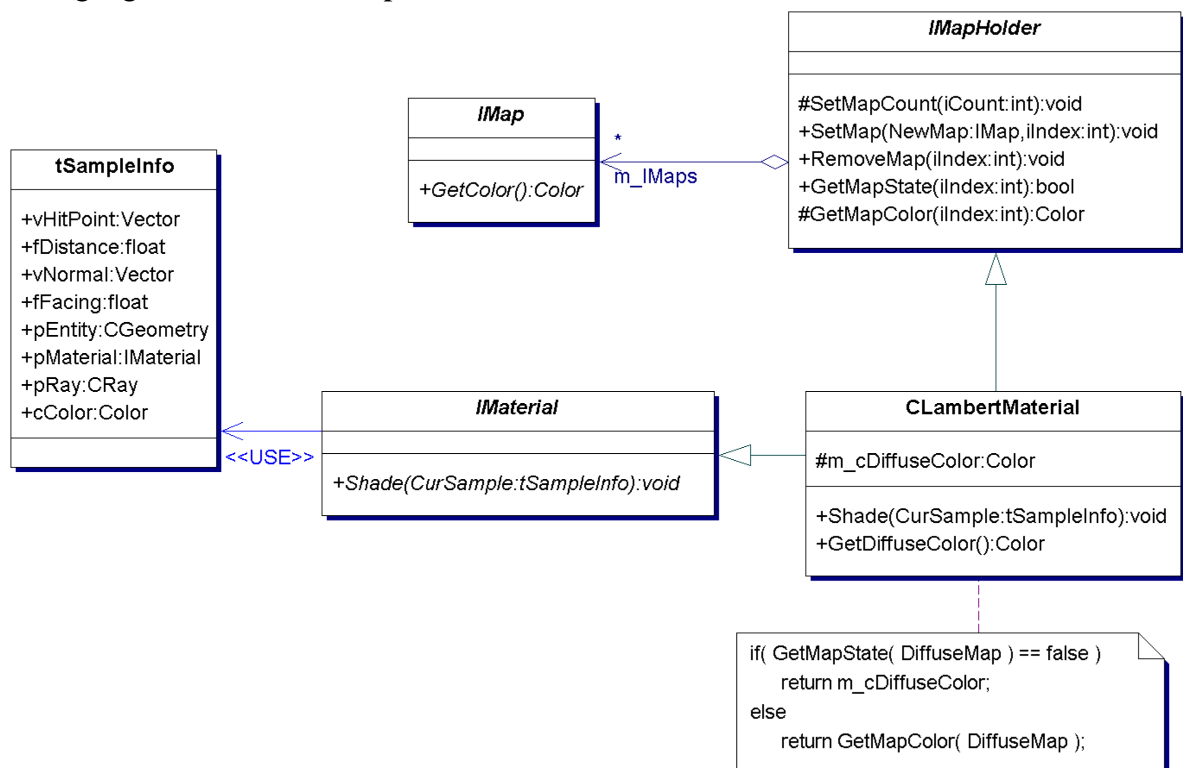

Abbildung 2. 19 – Material und MapHolder

Nur das Material selbst weiß wie viele Texturen es unterstützen möchte, deswegen ist die Methode SetMapCount() der **IMapHolder** - Klasse auch geschützt (protected). Letztendlich läuft das folgendermaßen ab: Im Konstruktor des (konkreten) Materials legt der Entwickler die Anzahl der Maps fest. Im obigen Beispiel könnte er sich vielleicht für drei

Texturen entscheiden, somit wäre ein Slot für eine ambiente, für eine diffuse und für die emittierende Texturmap vorhanden. Jetzt können explizite **IMap** – Objekte mit diesen Slots assoziiert werden. Wie in Abbildung 2.19 mit der (Implementierungs-) Notiz angedeutet wurde, gestaltet sich der Zugriff auf die Maps als transparent.
Die abstrakte Klasse **IMapHolder** hat aber noch weitere Vorteile. Erst einmal ist es nicht zwingend vorgeschrieben, dass jedes Material Texturen besitzt. Der Entwickler kann sich hier völlig frei entscheiden, ob Maps unterstützt werden. Sollte er sich dafür entscheiden, legt er zudem fest wie viele Texturslots konkret benötigt werden. Später kann das Material gegebenenfalls relativ einfach um einen zusätzlichen Slot erweitert werden. Außerdem ist die Benutzung der Funktionalität dieser Verwaltungsklasse natürlich nicht nur Materialien vorbehalten. Jedes andere Objekt kann durch Vererbung die Verwaltungsmethoden verwenden. Zum Beispiel kann eine konkrete **IMap** – Klasse selbst wieder eigene Maps besitzen. Eine Textur, die Farbwerte von anderen Maps kombiniert, kann hier als kleines Beispiel genannt werden.
Weiterhin ist eine Lichtquelle vorstellbar, die als Projektor agiert, d.h. sie projiziert eine Textur in eine bestimmte Richtung. Auch dabei braucht der Entwickler das Rad nicht neu zu erfinden, sondern kann die Verwaltungsklasse **IMapHolder** benutzen.
Durch die erreichte Flexibilität sind der Fantasie also abermals keine Grenzen gesetzt.
Zum Ende dieses Abschnitts muss aber noch ein wichtiger Aspekt erläutert werden. Bis jetzt wurde nämlich kein Wort darüber verloren, wie ein konkretes **IMesh** – Objekt mit einem Material assoziiert wird. Während zum Beispiel die Ebene und die Kugel sowieso nur ein Material benötigen, so kann es bei einem Polygonobjekt durchaus vorkommen, dass unterschiedliche Dreiecke auch verschiedene Materialien besitzen. Für dieses Problem existiert leider keine einfache und elegante Lösung. Eine Möglichkeit wäre aber, dem Interface **IMesh** ein Array mit Zeigern auf Materialien hinzuzufügen. Während nun die Ebene oder die Kugel immer das Material mit dem Index Null verwenden, müssten die Dreiecke um eine MaterialID erweitert werden. Dieser Wert stellt den Index für das (Material)Array dar.

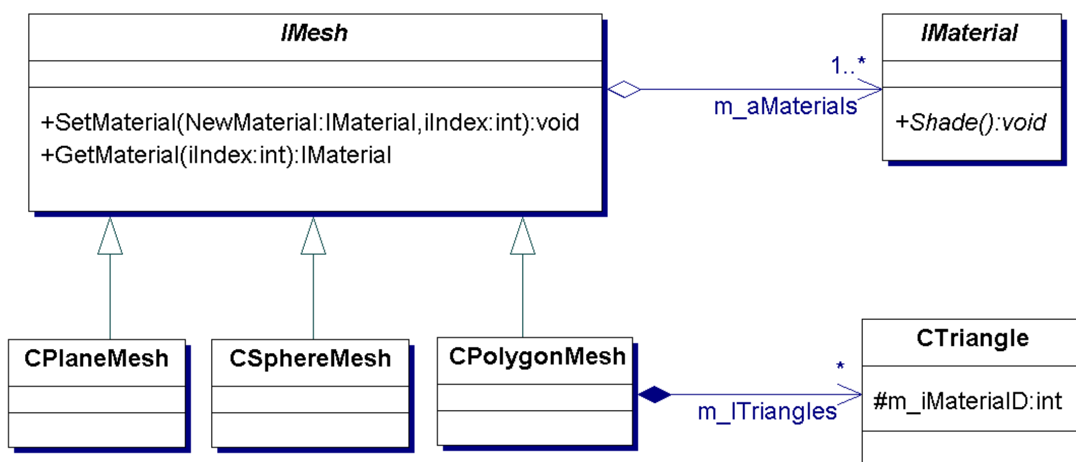

Abbildung 2. 20 - Materialassoziation

2.3.5 Hilfsobjekte

Bevor es um die eigentliche Bildsynthese geht, sollen noch einige Hilfsobjekte näher erklärt werden. Zunächst einmal werden natürlich mathematische Klassen wie **Vector**, **Matrix**, **Color** oder **CRay** für den Raytracer benötigt. Dabei kann einer der größten Vorteile einer Programmiersprache wie C++, nämlich das Überschreiben von Operatoren wie +, - oder *, voll und ganz ausgenutzt werden.
Die Klassen verhalten sich, wie man es erwartet. **Vector** besitzt zum Beispiel die Komponenten x, y und z. **Matrix** stellt eine 4x4 Matrix dar, **Color** ist ein RGB Farbvektor und **CRay** besitzt einen Ursprung und ein Ziel.
Viel interessanter sind aber Hilfsobjekte wie das **CImage**, welches eine Abstraktion eines zweidimensionalen Bildes darstellt. Letztendlich besitzt diese Klasse einen Puffer, der die Farbwerte der Pixel hält. Wie aber soll das Laden und Speichern eines Bildes flexibel integriert werden?
Sicherlich können die Methoden Load() und Save() direkt in **CImage** eingefügt werden. Dabei kommt es aber zu einem Problem. Erst einmal muss in den Routinen explizit festgelegt werden, welche Art von Bildformat unterstützt wird. Will der Entwickler später ein weiteres Format unterstützen, so muss er riesige if - Blöcke einfügen. Außerdem würde sich die Klasse **CImage** dadurch immer weiter aufblähen. Lade- und Speichermethoden haben häufig die Eigenschaft sehr komplex zu sein und damit viele Zeilen Code zu beanspruchen.
Viel besser ist es also, das Laden und Speichern von Bildern zu generalisieren. Dazu fügt man eine abstrakte Klasse **IImageFile** ein, die sämtliche Interaktionen mit Dateien übernimmt.

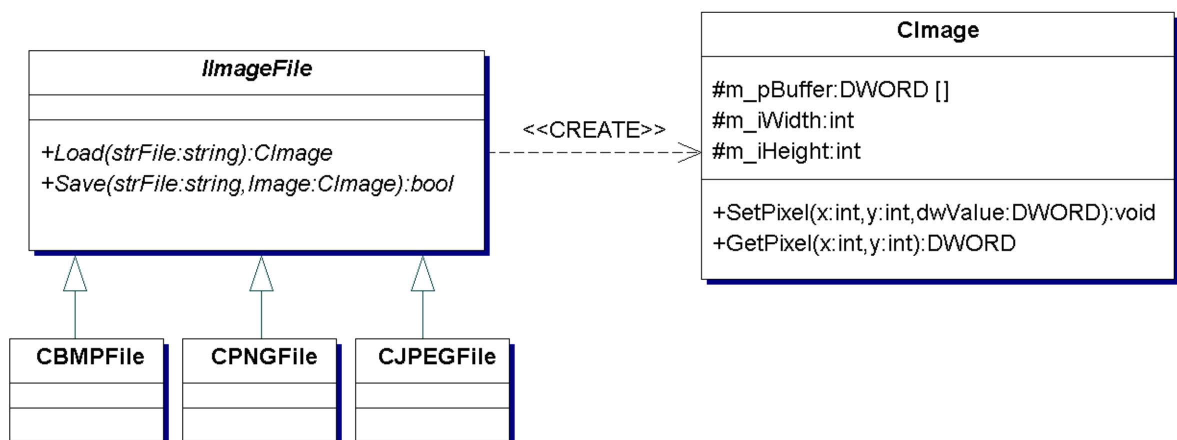

Abbildung 2. 21 – Das Bildmodul

Wie aus der Abbildung ersichtlich wird, kann mit der Methode Load() eine Bilddatei geladen werden. Diese Routine erzeugt dabei ein Objekt vom Typ **CImage**, in dessen Bildpuffer sich nun die Daten der Datei befinden. Das Speichern funktioniert sehr ähnlich. Der Methode Save() wird ein Bild übergeben, welches dann unter dem spezifizierten Dateinamen abgespeichert wird.
Der Vorteil dieser Struktur liegt auf der Hand. Die gesamte Funktionalität kann an verschiedenen Stellen wieder verwendet werden. Zum Beispiel kann sowohl der Raytracer ein **CImage -** Objekt besitzen, in das er die berechneten Farbwerte speichert, als auch eine Texturemap. Das Laden und Speichern ist dabei verallgemeinert.
Nun braucht man sicherlich nicht lange zu überlegen, wie das Laden von externen Szenen gestaltet werden kann. Auch dabei wird von dem eigentlichen Algorithmus abstrahiert und eine allgemeine Schnittstelle eingeführt. Diesmal heißt sie sinnvollerweise

ISceneFile. Die Methode Load() füllt dabei die Szene mit den in der Datei spezifizierten Objekten.

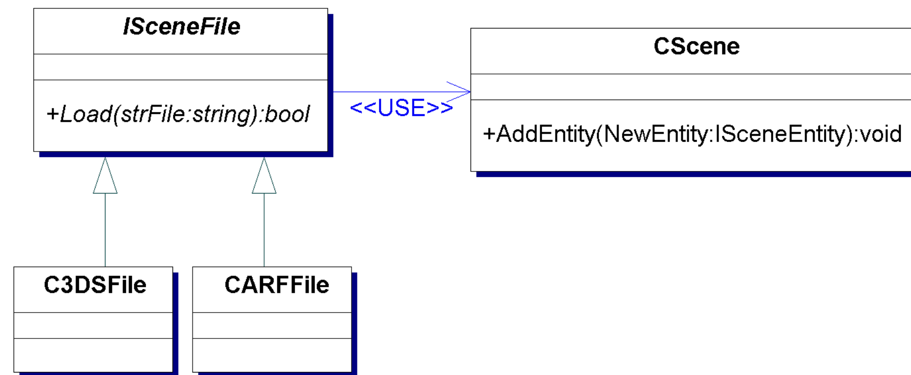

Abbildung 2. 22 – Laden von Szenen

Um dem Entwickler das Leben noch ein wenig einfacher zu machen und den Grad der Flexibilität weiter zu erhöhen, ist es außerdem sinnvoll eine Managerklasse zu entwerfen. Mit der Hilfe von dieser Klasse wird die Interaktion mit Dateien vollkommen unabhängig vom Dateityp. So ist eine Helferklasse mit dem Namen **CImageFileManager** denkbar, bei der alle **IImageFile**s registriert werden. Durch die Methoden ReadFile() und WriteFile() kann eine ganz allgemeine Anfrage an das System gerichtet werden. Anhand des spezifizierten Dateinamens entscheidet der Manager, welches **IImageFile** für die gewünschte Datei zuständig ist und ruft dann je nach Art der Anfrage entweder Load() oder Save() des konkreten **IImageFile**s auf. Durch diesen kleinen Trick wird es unglaublich einfach neue Typen von Dateien zu unterstützen. Sollte ein Entwickler also ein neues Bildformat unterstützen wollen, so braucht er lediglich das neue **IImageFile** – Objekt bei dem Manager zu registrieren und sofort können zum Beispiel Texturen von diesem neuen Typ geladen werden.
Ein ähnlicher Manager ist auch für die Szenendateien vorstellbar.

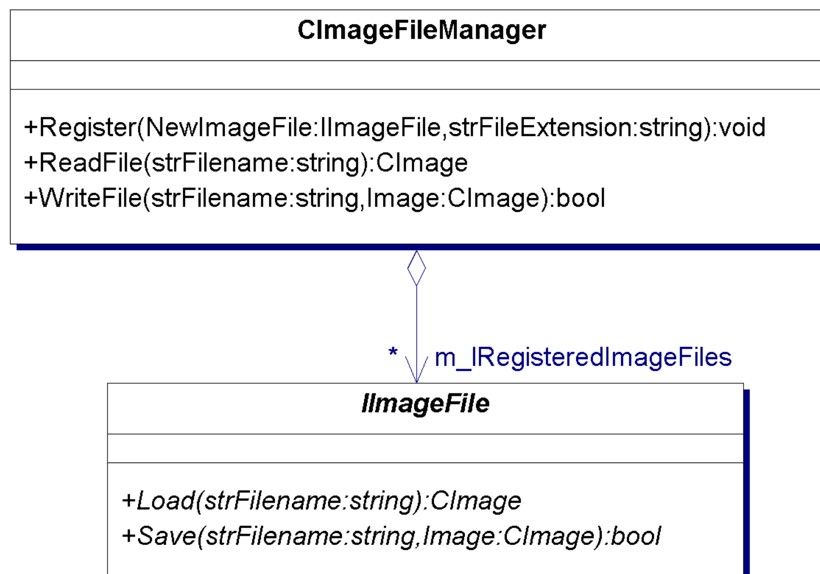

Abbildung 2. 23 - Dateimanager

2.3.6 Bildsynthese

Nun sind alle Zutaten vorhanden um die Bildsynthese durchzuführen. Aber auch an dieser Stelle kann für erhöhte Flexibilität gesorgt werden. Zunächst wird eine Applikationsklasse benötigt, die allgemeine Informationen wie die Auflösung des zu berechnenden Bildes speichert. Diese Klasse soll einfach **CRaytracer** heißen. Wie die Szene auch, ist sie ein Singleton, d.h. es existiert lediglich eine Instanz dieser Klasse. Natürlich könnte der Strahlenverfolgungsalgorithmus direkt im Raytracer untergebracht werden. Es ist jedoch sinnvoller den Algorithmus abzukoppeln, damit dieser später unabhängig vom Raytracer variiert werden kann. Zudem können so leichter räumliche Datenstrukturen implementiert und getestet werden, da nicht immer der alte Algorithmus überschrieben werden muss. Somit wird in das System eine Klasse mit dem Namen **CTracer** eingeführt, deren wichtigste Methode Trace() darstellt. Sie schneidet den spezifizierten Strahl mit der Szene. Bei einer Brute - Force Implementierung des Raytracings reicht Trace() die Anfrage einfach an alle Objekte vom Typ **IMesh** weiter. Diese füllen dann die Schnittinformationsstruktur. Das Ergebnis des Schnitts wird schließlich in der Klasse **CTracer** gespeichert. Nun können alle Objekte, die sich für das aktuelle Sample interessieren, auf die Informationen des Schnittpunktes zugreifen. Als Beispiel können hier Maps und Lichtquellen genannt werden. Diese bekamen ja standardmäßig keine **tSampleInfo** – Struktur mit als Parameter übergeben.
Die Methode Trace() gibt es in mehreren Ausführungen. Die einfache Variante ermittelt lediglich den nahesten Schnittpunkt, die erweiterte Version berechnet zusätzlich noch den Farbwert an der Position des Samples und gibt ihn an die aufrufende Funktion zurück. Die Klasse **CTracer** könnte außerdem um einen Raycache erweitert werden um zum Beispiel die Schattenberechnung zu beschleunigen. Dabei sollte der Entwickler jedoch entscheiden können, ob der Cache verwendet werden soll oder nicht.
Da das Raytracing ein rekursiver Algorithmus ist, muss auch darauf geachtet werden, dass Texturen, Lichtquellen und andere Objekte immer auf das aktuelle Sample Zugriff haben. Um diese Anforderung zu erfüllen wird ein Kellerspeicher benötigt, auf dessen Spitze sich der aktuell betrachtete Punkt in der Rekursion befindet. Durch die Operationen PushSampleInfo(), PopSampleInfo() und GetCurrentSampleInfo() kann der Stack kontrolliert werden. Sollte also zum Beispiel ein Material eine Reflektion berechnen wollen, so wird in der Shade – Methode die (erweiterte) Routine TraceColor() aufgerufen. Schneidet der Reflektionsstrahl ein anderes Objekt, so wird das Ergebnis automatisch auf dem Stack abgelegt, dann kann der Farbwert berechnet werden und schlussendlich wird das Sample wieder vom Kellerspeicher entfernt. Somit erfolgt die Verwendung des Stacks vollkommen transparent.
Bei der einfachen Version der Trace – Methode ist dies jedoch ein wenig anders. Das Ergebnis des Schnitts wird nicht automatisch auf dem Kellerspeicher abgelegt. Sollte der Entwickler jedoch der Meinung sein, dass die Informationen des Schnitts von anderen Objekten benötigt werden, so muss er sie manuell auf dem Stack ablegen und später wieder von ihm entfernen. Der Grund dafür ist, dass das System nicht wissen kann, ob und wenn ja, wie lange die Werte des Schnittpunktes benötigt werden.

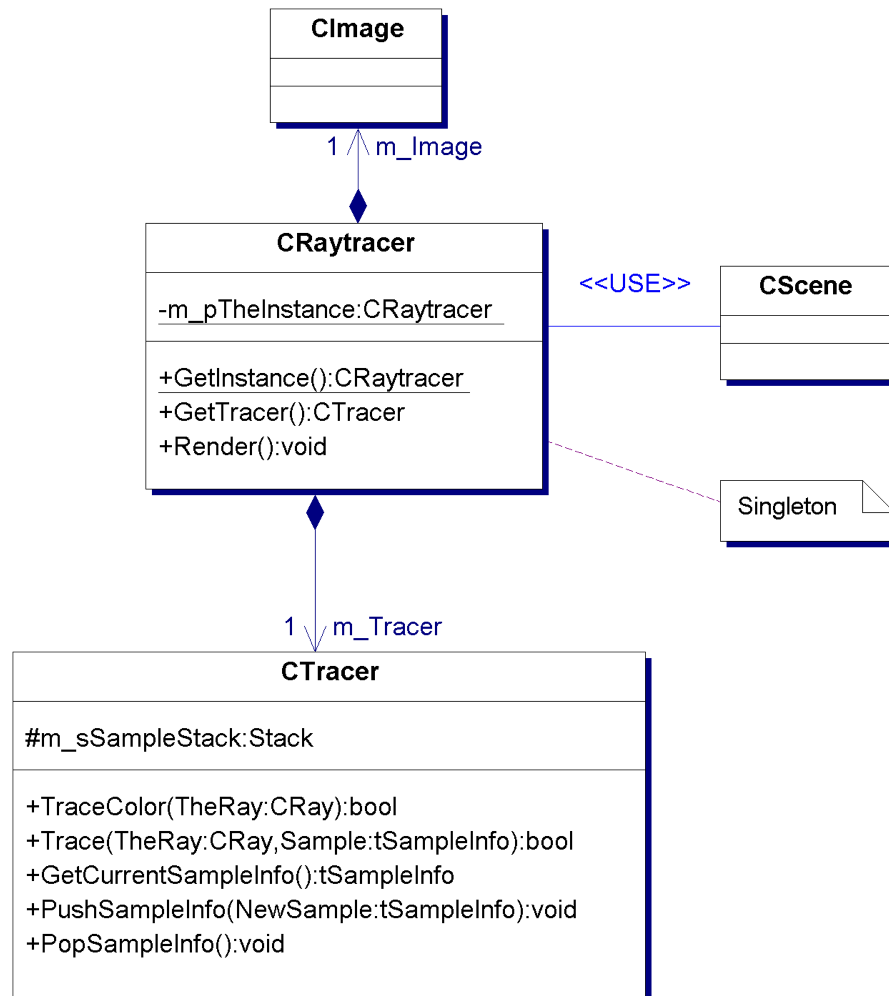

Abbildung 2. 24 - Raytracer

Zum Abschluss sollte noch ein Fakt erläutert werden. Die Szene besitzt genau eine Kamera, durch die sie betrachtet wird und genau eine ambiente Lichtquelle. Auf diese Objekte kann über spezielle Methoden der Szene zugegriffen werden.

2.3.7 Einige Anmerkungen zur Implementierung

Um die in den letzen Abschnitten erläuterten Prinzipien noch anschaulicher zu gestalten, wurde das oben beschriebene System vollständig implementiert und zusammen mit diesem Beleg veröffentlicht. Der (C++) Quelltext ist ausführlich kommentiert, somit sollten keine Verständnisprobleme auftreten.
Das oben beschriebene und das implementierte System sind weitestgehend identisch. Aus Gründen der Optimierung wurden jedoch kleinere Änderungen vorgenommen. So werden viele Parameter als Referenz oder als Zeiger anstatt als Wert übergeben. Außerdem befinden sich in den meisten implementierten Klassen mehr Methoden und/oder Attribute als in den Diagrammen dargestellt ist. Damit die UML Diagramme jedoch übersichtlich bleiben, wurde auf diese Implementierungsdetails in den Grafiken verzichtet.
Ein Leser, der sich für diese Details interessiert, sollte also ruhig einen Blick in die Implementierung werfen. Dort wird er dann mit allen Informationen versorgt.
Der Quelltext umfasst zirka 10000 Zeilen C++ - Code. (15000 Zeilen wenn Kommentarzeilen mitgezählt werden.)

2.3.8 Zusammenfassung und Ausblick

Im dritten Unterkapitel wurde Schritt für Schritt eine Softwarearchitektur für einen Raytracer entwickelt. Dabei wurde ganz besonders auf die Flexibilität des Entwurfs geachtet. Das beschriebene System ist sehr leicht erweiter- und änderbar.
Häufig wird dabei derselbe „Trick“ verwendet, nämlich die Trennung von der abstrakten Beschreibung und der konkreten Implementierung der Objekte. Dabei besitzt die abstrakte Oberklasse eine bestimmte Methode, die dann von den konkreten Unterklassen jeweils anders implementiert wird. Dadurch erhält jedes Objekt ein individuelles Verhalten. Tatsächlich ist es auch nur so effektiv möglich, ein breites Spektrum von verschiedenen Objekttypen zu unterstützen. Das schöne an diesem Prinzip ist die Allgemeingültigkeit. Ob nun verschiedene geometrische Objekte, Lichtquellen, Materialien oder Maps, durch eine flexible Schnittstelle kann eine sehr große Vielfalt von Szenenobjekten beschrieben werden.
In der Praxis wird dieses Prinzip noch sehr viel weiter ausgebaut. Bei fast allen professionellen Programmen ist es möglich Erweiterungsmodule zu schreiben. Dabei benötigt der Programmierer nicht einmal den Zugriff auf den gesamten Quelltext der Software, sondern lediglich auf eine (kleinere) Programmierschnittstelle. Die so genannten Plugins werden dann beim Start des Programms automatisch geladen und können somit ab diesem Zeitpunkt verwendet werden. Um eine solche Programmierschnittstelle wird es vor allem im Kapitel 3 gehen.
Außerdem wurde im letztem Unterkapitel erläutert, wie der Bilderzeugungsalgorithmus in unterschiedliche Module zerlegt werden kann, die dann während der Berechnung optimal zusammenarbeiten. Dabei konnte beobachtet werden, dass es nicht immer Sinn macht, die ursprüngliche mathematische Beschreibung als ein Stück zu übernehmen. Durch geschickte Trennung der einzelnen Teile wurde abermals erhöhte Flexibilität erreicht.
Weiterhin wurde auf Aspekte wie das Laden und Speichern von Bildern oder Szenendateien eingegangen. Dafür wurden spezielle Klassen in das System integriert, die für die Serialisierung der Daten zuständig sind. Durch eine Managerklasse konnte zum Schluss sogar ein völlig transparenter Zugriff auf die externen Dateien erreicht werden.
Hierbei ist anzumerken, dass dieses Vorgehen in der Praxis fast genauso verwendet wird, wie es oben beschrieben wurde.
Zum Abschluss des Unterkapitels wurde dann noch gezeigt, dass der eigentliche Strahlenverfolgungsalgorithmus ebenfalls generalisiert werden sollte. Dadurch wird es später sehr viel einfacher den Raytracer zu optimieren. Es könnten zum Beispiel räumliche Datenstrukturen eingeführt werden. Ein Voxelgitter oder eine Octtree – Struktur sind wohl die bekanntesten und am häufigsten verwendeten Optimierungsdatenstrukturen.

2.3.9 Ergebnisbilder

Im letztem Abschnitt werden einige Ergebnisbilder präsentiert, die mit dem implementierten Raytracer erstellt wurden. Es kann leider nur eine sehr kleine Auswahl gezeigt werden.

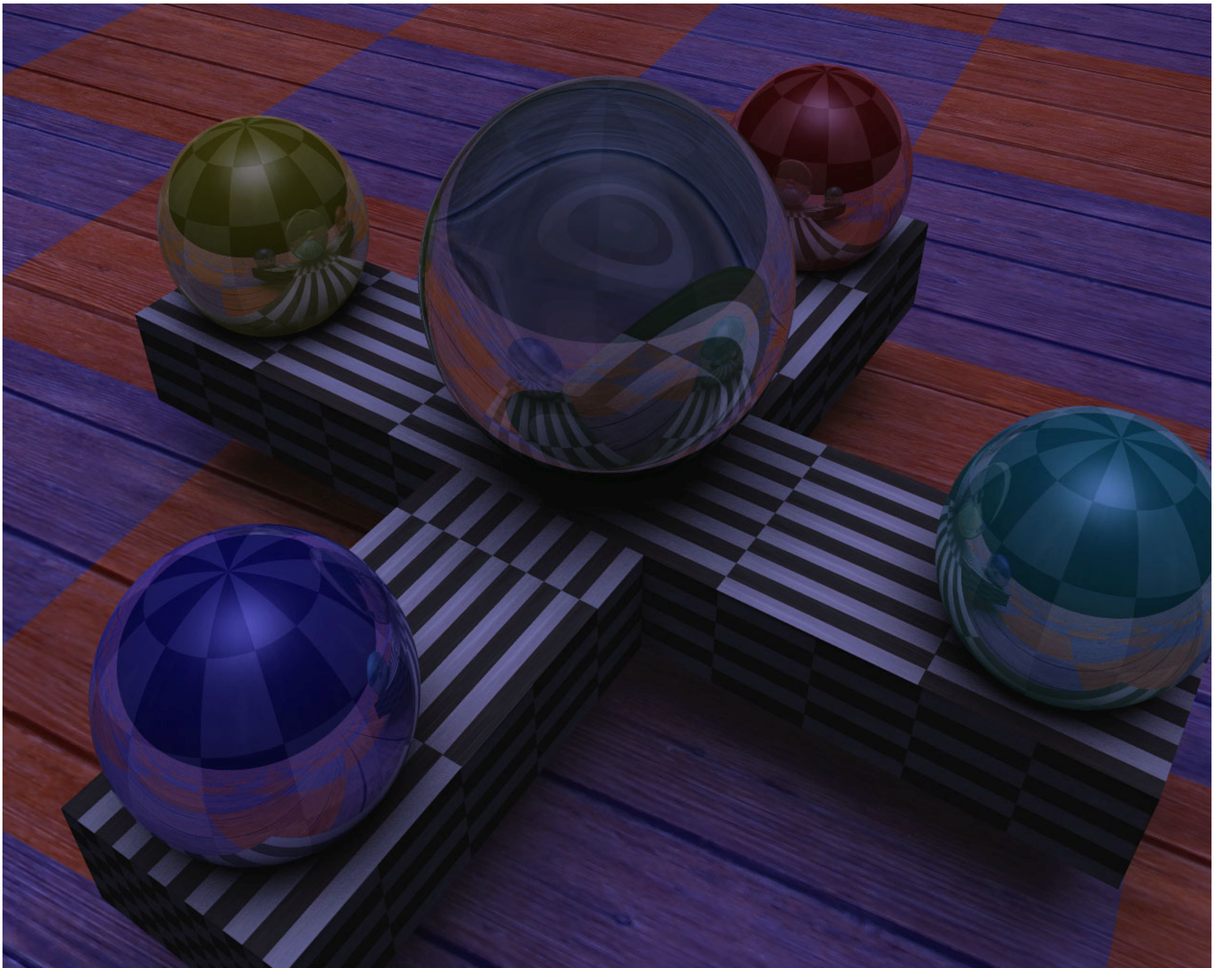

Abbildung 2. 25 – Geometrische Primitive

Abbildung 2. 26 - Burg

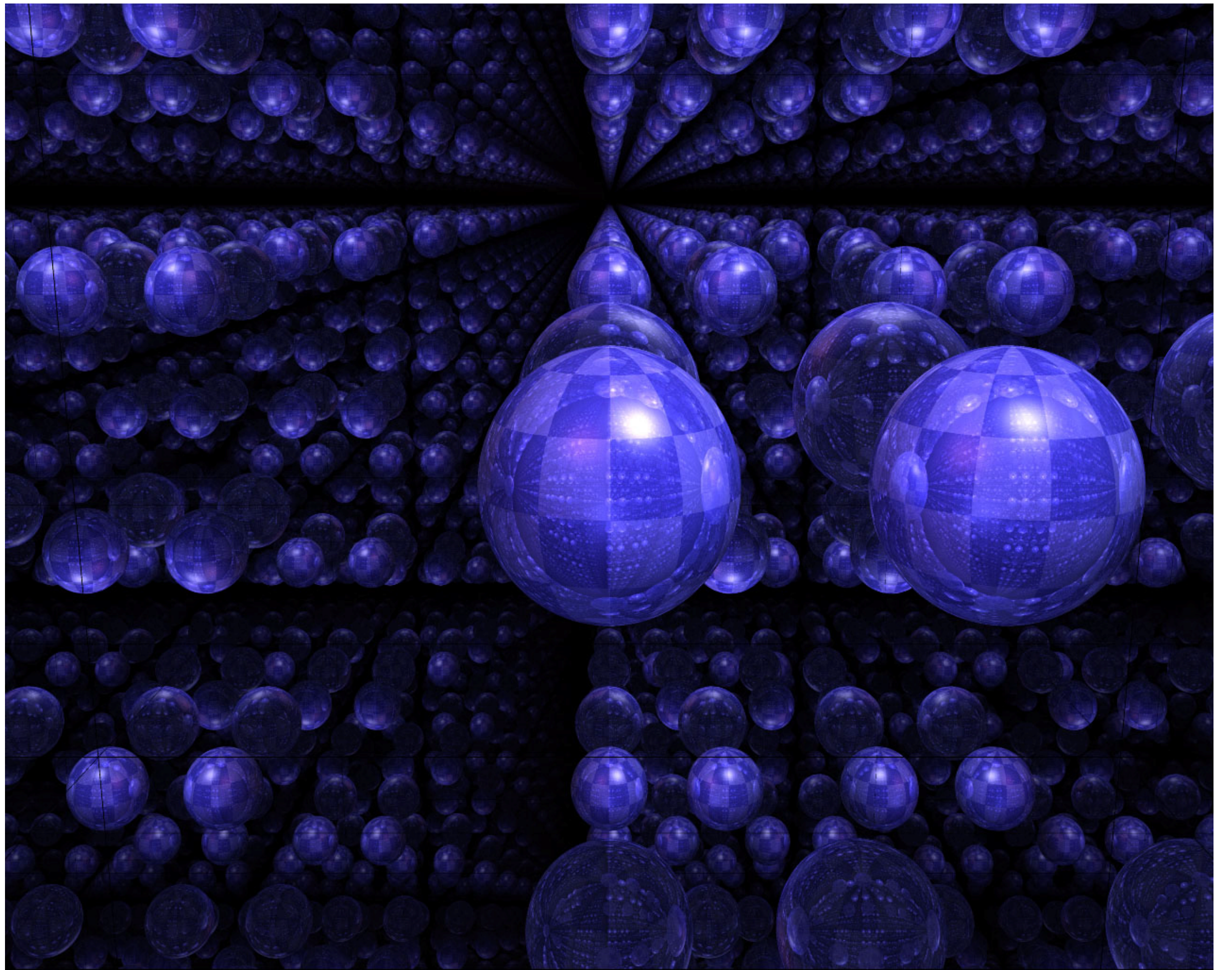

Abbildung 2. 27 - Reflektionen

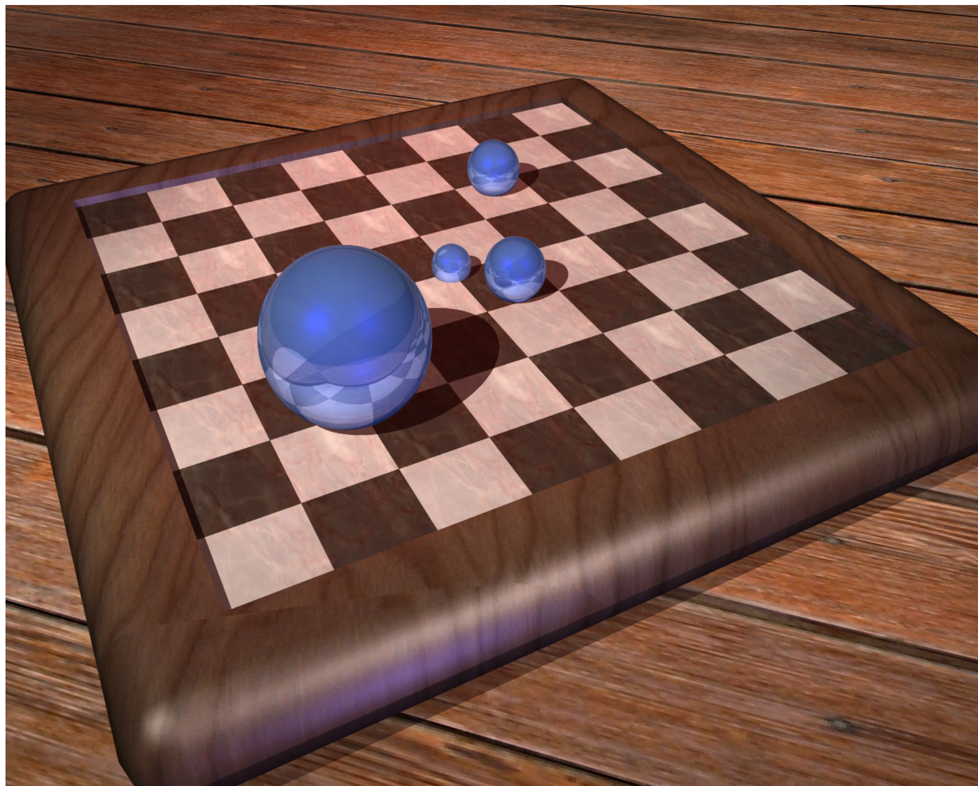

Abbildung 2. 28 – Schachbrett

Die Zeit wird kommen,
wo unsere Nachkommen sich wundern,
da wir so offenbare Dinge nicht gewußt haben.
Lucius Annaeus Seneca (4 v.Chr. - 65 n.Chr.)

3 Fallstudie: 3D Studio Max®

3.1 Einführung

3D Studio Max® ist eines der wichtigsten Modellierungs- und Renderprogramme, dass seit vielen Jahren auf dem internationalen Markt existiert. Ein Grund für den Erfolg und die große Beliebtheit ist sicherlich auch die Änderungsfreundlichkeit. Es kann durch so genannte Plugins erweitert werden. Das sind Zusatzmodule, die dynamisch während des Starts des Programms geladen werden. Um ein solches Plugin zu schreiben benötigt man lediglich die Programmierschnittstelle mit dem Namen „MaxSDK“, die bei jeder Version der Software standardmäßig mit beiliegt.
Im Internet sind aber bereits riesige Mengen von fertigen Plugins zu finden, die zumeist kostenlos zum Download bereitstehen. Das Spektrum der existierenden Plugins ist riesig und reicht von neuen geometrischen Körpern über Materialien bis hin zu Rendereffekten.
Heutzutage ist die enorme Flexibilität einer Software ein wichtiger Erfolgsfaktor. Die Entwickler des Programms können während der Implementierung nämlich unmöglich alle Anwendungsmöglichkeiten ihrer Software vorhersehen. Man ist sich also durchaus darüber im Klaren, dass spätere Änderungen und Erweiterungen nötig sein werden.
Dabei sollte die Entwicklung eines Plugins so einfach wie möglich sein. Andererseits darf man aber auch nicht die unglaubliche Komplexität vergessen, die ein Programm von diesem Umfang besitzt.
In diesem Kapitel soll nun also untersucht werden, wie die Architektur von 3D Studio Max® aufgebaut ist. Es werden viele Ähnlichkeiten zu dem vorgeschlagenen Entwurf aus Kapitel 2 erkennbar werden, aber auch einige Unterschiede.
Die Analyse des Programms wird schließlich durch die Entwicklung eines eigenen Plugins komplettiert, welches 3D Studio Max® Szenen so exportiert, dass sie von dem im Kapitel 2 entwickelten Raytracer geladen und gerendert werden können.

3.2 Die Architektur

Da 3D Studio Max® ein sehr komplexes Programm ist, kann hier leider nicht die gesamte Architektur näher erläutert werden. Vielmehr werden die nun folgenden Abschnitte lediglich die Teilaspekte der Software untersuchen, die auch bei der Entwicklung der Architektur im Kapitel 2 genauer erklärt wurden.
Der interessierte Leser sollte nach diesem Kapitel jedoch in der Lage sein, weiterführende Informationen relativ einfach aus den Hilfedokumenten des MaxSDKs zu extrahieren.

3.2.1 Abhängigkeiten und Animierbarkeit

Noch bevor auf die Repräsentation der Szene eingegangen werden kann, müssen andere wichtige Fakten erläutert werden, die später immer wieder auftreten werden.
In 3D Studio Max® können Abhängigkeiten zwischen verschiedensten Objekten definiert werden. Da sich die Eigenschaften der Objekte ständig ändern können, muss jedoch ein Apparat erschaffen werden, der die Modifikationen eines Eintrags an alle abhängigen Objekte weiterleitet. Dies geschieht durch die Klassen **ReferenceMaker** und **ReferenceTarget**.
Objekte, die vom Verhalten eines anderen Objekts abhängig sein sollen, werden von **ReferenceMaker** abgeleitet. Diese Klasse bietet unter anderem Funktionalität um auf die Modifikation des Ziels zu reagieren. Wenn eine neue Abhängigkeit erstellt wird, meldet sich das abhängige Objekt (**ReferenceMaker**) bei dem entsprechenden Zielobjekt (**ReferenceTarget**) an. Ab diesem Zeitpunkt wird es über alle Änderungen informiert und kann darauf reagieren.

Eine Modifikation an einem **ReferenceTarget** führt dazu, dass die Memberfunktion NotifyDependents() aufgerufen wird. In dieser Methode wird daraufhin durch alle abhängigen **ReferenceMaker** iteriert und jeweils die Methode NotifyRefChanged() aufgerufen.
Um dieses Prinzip noch allgemeiner und mächtiger zu machen, ist es möglich, dass der **ReferenceMaker** nicht nur von einem anderen Objekt abhängig ist, sondern von mehreren. Deswegen besitzt diese Klasse zusätzlich Routinen um auf alle Zielobjekte zuzugreifen. Im Kontext von geometrischen Objekten mag dies im ersten Moment nicht viel Sinn machen, aber für Modifikatorobjekte sind Abhängigkeiten eine mächtige Einrichtung. Man stelle sich zum Beispiel eine polygonale Kugel vor, die durch einen Rauschen – Modifikator verformt wird. Da die Kugel von dem Modifikator abhängt, erzeugt sie eine Abhängigkeit zum Rauschen - Objekt. Jedes mal wenn sich nun ein Parameter des Modifikators ändert, wird die Kugel informiert um ihre Repräsentation entsprechend anzupassen.
Da fast jedes Objekt in 3D Studio Max® animiert werden kann, wurde außerdem eine allgemeine Schnittstelle mit dem Namen **Animatable** in das System eingefügt. Sie bietet zum Beispiel die Möglichkeit animierbare Eigenschaften und Keyframes zu verwalten. Interessant ist jedoch die Hierarchie der drei beschriebenen Klassen, sie wird in Abbildung 3.1 dargestellt.

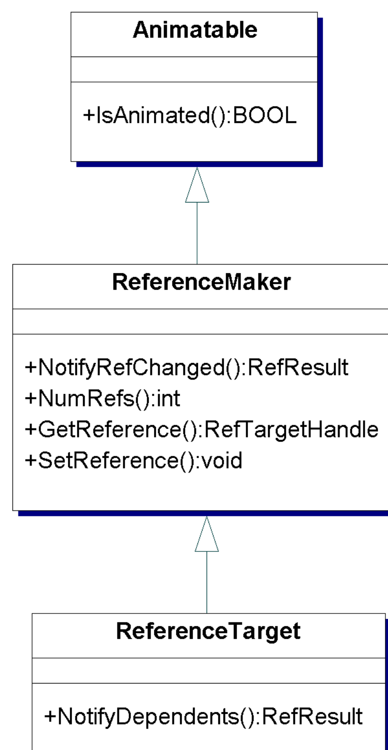

Abbildung 3. 1 – Abhängigkeiten und Animierbarkeit

3.2.2 Repräsentation der Szene

Eine Szene in 3D Studio Max® ist in Form eines Szenengraphs organisiert. Ein Szenengraph ist ein zyklenfreier gerichteter Graph, bei dem sich an jedem Knoten ein Objekt der Szene befindet. Die Transition von einem Knoten zu einem anderen, stellt eine hierarchische Beziehung dar. Wird also zum Beispiel ein Vaterknoten rotiert, so wird diese Transformation auch auf seine Kindknoten propagiert. Sollten diese Szeneneinträge wieder eigene Tochterobjekte besitzen, so wird die Rotation weiter rekursiv auf die neuen Kindkonten angewendet usw.

Der Ursprung des Graph ist durch den so genannten Sceneroot definiert. Von ihm aus können alle Objekte der Szene durch eine Traversierung des Graphen erreicht werden. Dieser Ursprungskonten ist mit der Klasse **CScene** des obigen Entwurfs vergleichbar. Anstatt zum Beispiel mit Hilfe von AddEntity() einen neuen Szeneneintrag hinzuzufügen, wird bei 3D Studio Max® das neue Objekt einfach als Tochterkonten des Sceneroots registriert.
Ein Knoten des Szenengraphs wird durch ein Objekt vom Typ **INode** repräsentiert. Diese Klasse kann mit dem Interface **I3dEntity** aus Kapitel 2 verglichen werden. **INode** besitzt zum Beispiel Methoden um die Weltmatrix oder die Hierarchie zu modifizieren.
Es ist jedoch nicht festgelegt, welchen Typ ein **INode** nun genau darstellt. Es kann Geometrie, eine Lichtquelle, eine Kamera oder etwas ganz anderes sein. Die Art des Szeneneintrags ist in der Spezialisierung der Klasse **Object** kodiert. So existiert zum Beispiel ein **GeomObject**, ein **LightObject** und ein **CameraObject**. Das konkrete **Object** einer **INode** kann durch die Methode EvalWorldState() geholt werden.

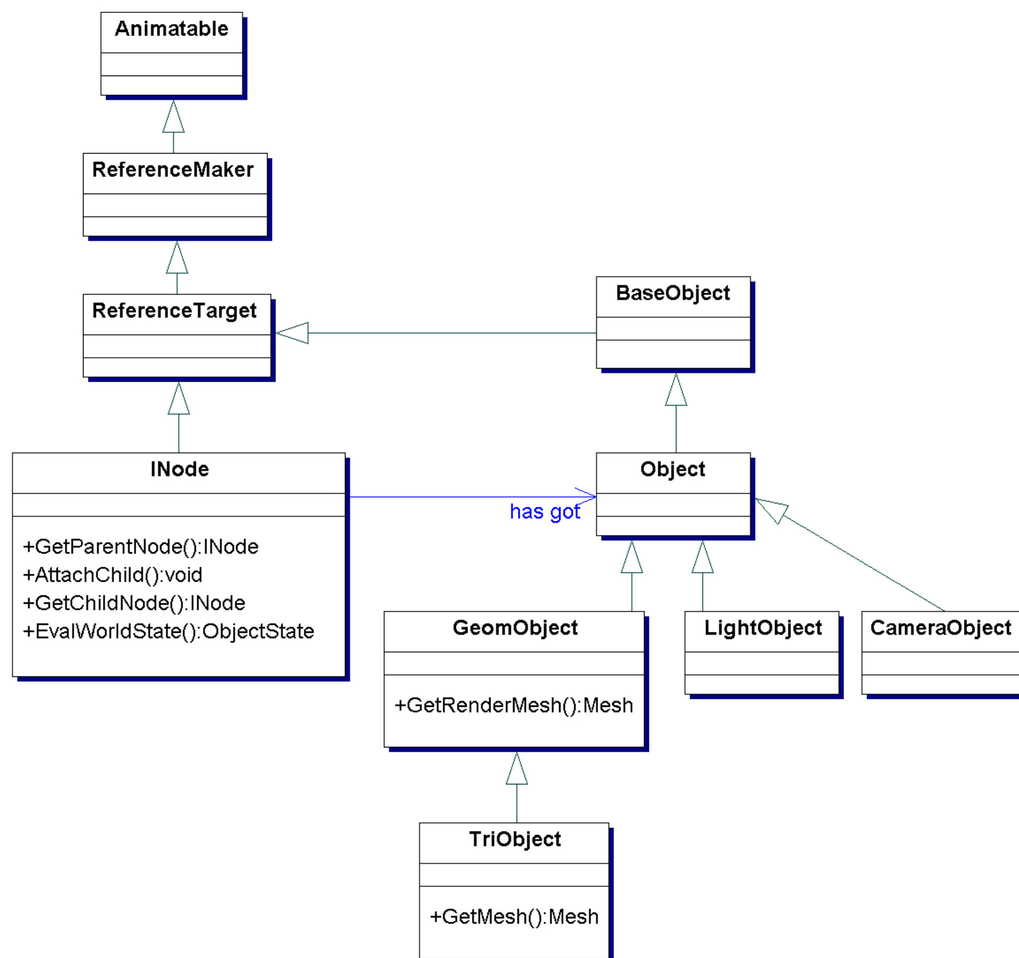

Abbildung 3. 2 – INodes und Objects

Dieses Vorgehen unterscheidet sich auf den ersten Blick sehr stark von der Architektur aus Kapitel 2. Dadurch dass 3D Studio Max® jedoch kein reiner Renderer, sondern auch ein Modellierungs- und Animationsprogramm ist, wurde diese Trennung nötig. Vor allem die Animierbarkeit führt zu starken Änderungen. Zu jedem Zeitpunkt besitzt ein **INode** genau ein aktuelles **Object**, dies kann mit EvalWorldState() berechnet werden. Diese **Object**s können sich aber über die Zeit ändern, d.h. zu zwei verschiedenen Zeitpunkten gibt ein und das selbe **INode** eventuell unterschiedliche **Object**s zurück. Man könnte sich

zum Beispiel ein Partikelsystem vorstellen, das beliebige Objekttypen unterstützt. Es kann also zum Beispiel geometrische Objekte genauso wie Lichtquellen in die Welt sprühen. Während ein einzelnes Partikel ein **INode** ist, so kann sich „sein“ **Object** jedoch ständig ändern. (zum Beispiel in Abhängigkeit der Zeit)
Bei einem Renderer sieht das etwas anders aus. Da sowieso immer nur genau ein Zeitpunkt bei der Bildberechnung betrachtet wird, ist die Art des Szeneneintrags für diesen Frame festgelegt. Dadurch wird die obige Trennung von Knoten und Repräsentation überflüssig.
Nun sollte noch näher auf die geometrischen Primitive eingegangen werden. Wie viele andere professionelle Modellierungs- und Renderprogramme, so ist auch 3D Studio Max® beim Rendern der Szene auf Dreiecksnetze beschränkt. In der Modellierungsphase können jedoch zusätzlich weitere Primitive, wie NURBS - Flächen oder Quadriks verwendet werden. Rein theoretisch werden sogar mathematische Primitive unterstützt, diese müssten dann aber, genau wie die NURBS – Objekte, vor der Bildberechnung in ein Dreiecksnetz konvertiert werden.
Dem einem oder anderem Leser ist eventuell schon aufgefallen, dass es in der Abbildung 3.2 zwei Klassen gibt, die ein Mesh – Objekt zurückgeben. Die Klasse **GeomObject** ist ein generelles geometrisches Objekt. Dies könnte also zum Beispiel auch ein mathematisches Primitiv sein. Damit dieser Szeneneintrag beim Rendern auch sichtbar ist, muss er trianguliert werden. Dies geschieht in der Methode GetRenderMesh(), welches das entsprechende Dreiecksnetz zurückgibt.
Ein **TriObject** ist ein polygonales Szenenobjekt. Diese Art von Geometrie besitzt ständig ein Mesh, auf das per GetMesh() zugegriffen werden kann.

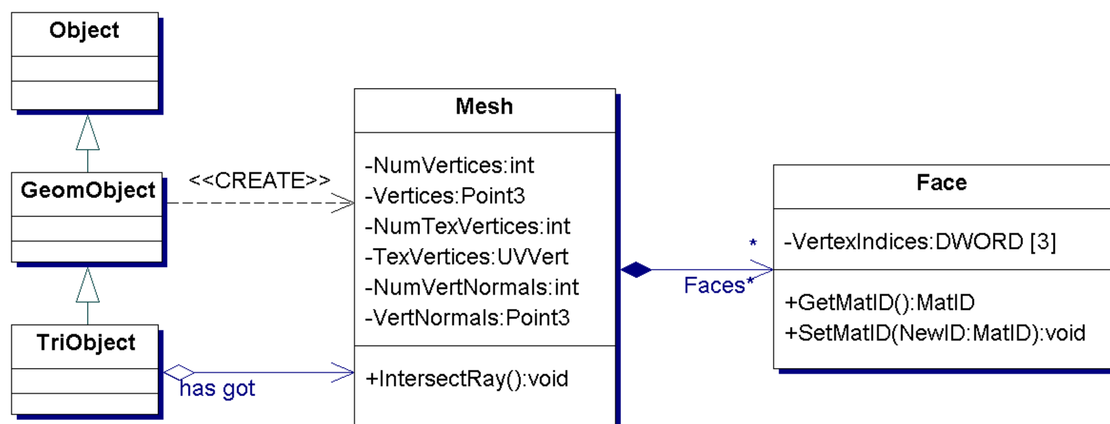

Abbildung 3. 3 - Mesh

Wie aus der obigen Abbildung ersichtlich wird, wurde bei der Realisierung des (Polygon) **Mesh** ein sehr ähnlicher Weg wie in der Architektur aus Kapitel 2 eingeschlagen. Die Unterschiede bestehen lediglich in der Art der Datenhaltung. In 3D Studio Max® existiert zum Beispiel keine Vertexklasse. Diese Daten werden in einem Array direkt im **Mesh** gehalten.
(Wichtige Anmerkung: Im folgenden wird von Instanziierung gesprochen. Dieser Begriff besitzt an dieser Stelle leider eine doppelte Bedeutung. Es ist jedoch ausschließlich die geometrische Instanziierung gemeint und nicht etwa die Ausprägung einer Klasse!)
Im Gegensatz zum Raytracer aus Kapitel 2 ermöglicht 3D Studio Max® auch die Instanziierung von Objekten. Dabei wird jedoch nicht das **INode** selbst instanziiert, sondern lediglich das entsprechende **Object**. Das hat zur Folge, dass die Instanz eines Objekts einen eigenen Namen und eine eigene Weltmatrix besitzt.

Es existieren jedoch zwei unterschiedliche Arten von Instanziierungen, die Instanz und die Referenz. Die eigentliche Instanz ist bidirektional. Wird also zum Beispiel eine Änderung am Mesh des Originalobjekts vorgenommen, so werden auch alle Instanzen geändert. Wird andererseits eine Änderung an einer Instanz ausgeführt, so wird sie auch auf das Original- und auf alle anderen Instanzobjekte angewandt. Programmiertechnisch kann man sich das so vorstellen, dass der Zeiger auf die Geometrie, also auf ein Objekt vom Typ **Mesh**, bei allen Instanzen der selbe ist.
Die Referenz wiederum ist unidirektional. Eine Modifikation des Originals führt somit zur Änderung der Referenzen. Wird jedoch eine Referenz bearbeitet, so werden diese Aktionen nicht auf das Original oder auf andere Referenzen übernommen.
Zum Schluss soll noch die Frage beantwortet werden, wo denn eigentlich der Sceneroot gespeichert wird. Dieser Konten wird in einer Klasse mit dem Namen **Interface** gehalten. Dies ist die Schnittstelle zur globalen Applikation. Man könnte diese Klasse mit **CRaytracer** vergleichen, die ja auch eine Art von Applikationsklasse darstellte.
Zusammenfassend kann also festgestellt werden, dass der generelle Aufbau einer Szene in 3D Studio Max® dem des Raytracers aus Kapitel 2 sehr ähnlich ist.

3.2.3 Daten des Schnitts

In 3D Studio Max® wurde das Prinzip der Schnittinformationsstruktur noch weiter ausgebaut. Die Entwickler der Software entschieden sich dafür, eine Schnittstelle bereitzustellen, aus dem jegliche Daten des Schnitts extrahiert werden können. Diese Klasse heißt **ShadeContext**. Sie ermöglicht einen lesenden Zugriff auf alle nötigen Details. Ein schreibender Zugriff wird jedoch nur bei wenigen Parametern gewährt. Eine dieser Parameter ist zum Beispiel die Normale des betrachteten Oberflächenpunkts, die ja durch die Anwendung von Bumpmapping noch während des Shadings modifiziert werden kann. Andererseits darf der Schnittpunkt nicht verändert werden.
Dieses Vorgehen hat seine Vor- und Nachteile. Einerseits ist dieser Weg sehr „sauber“, da zum Beispiel der Schnittpunkt nicht von einem Material modifiziert werden sollte. Andererseits kann es bei bestimmten Fällen notwendig werden, dass auf alle Attribute ein schreibender Zugriff möglich ist.

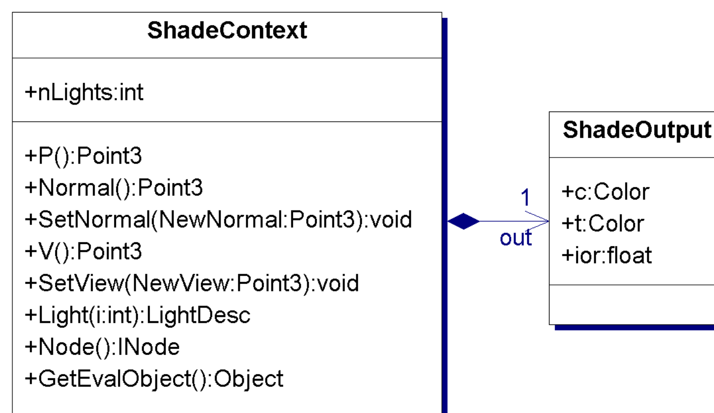

Abbildung 3. 4 – ShadeContext

In Abbildung 3.4 werden exemplarisch lediglich die wichtigsten Parameter dargestellt. Durch die Methode P() erhält man den aktuellen Schnittpunkt. Durch V() den momentanen Sichtvektor. Das Attribut nLights gibt die Anzahl der Lichtquellen an, die den Schnittpunkt potentiell beeinflussen könnten. Auf die entsprechenden Lichter kann durch die Funktion Light() zugegriffen werden. Mit Hilfe der Routinen Node() und GetEvalObject() werden das aktuelle **INode** und das entsprechende **Object** identifiziert.

Das Ergebnis des Shadings wird in der **ShadeOutput** – Klasse gespeichert, wobei das Attribut c die Farbe und t die Transparenz des Ergebnisses darstellt. Mit ior ist der momentane Brechungsindex gemeint.

3.2.4 Lichtobjekte und Beleuchtung eines Punktes

In der Abbildung 3.2 wurde ja bereits verdeutlicht, dass Lichtquellen eine Spezialisierung des allgemeinen **Object**s sind. Sie bestehen also aus einem Knoten im Szenengraph und aus einem **LightObject**. Die Entwickler von 3D Studio Max® haben sich jedoch nicht dafür entschieden die Illuminate() – Methode direkt in das Lichtobjekt zu integrieren. Sie trennen vielmehr die Informationen, die einerseits für den Modellierungsprozess und andererseits für den Rendervorgang benötigt werden. Dieses Vorgehen macht vor allem deswegen Sinn, weil während der Modellierung die Szene mit Hilfe von 3D – Grafikhardware dargestellt wird. Solche Karten sind auf sehr wenige unterschiedliche Typen von Lichtquellen beschränkt (Punktlicht, Direktionales Licht und Scheinwerfer). Außerdem unterstützen sie nicht alle Parameter der Lichtquellen. (z.B. Raytraceschatten)
Der dritte Grund für die Trennung stellt die GUI dar. In dem **LightObject** sind alle Attribute vorhanden, die für die Nutzerinteraktion benötigt werden. Dies können zum Beispiel Textfelder, Buttons, Comboboxen usw. sein. Diese Daten werden aber während des Renderings nicht gebraucht und müssen somit auch nicht im Hauptspeicher gehalten werden. Das heißt die unnötigen Informationen können vor der Bildberechnung ausgelagert und danach wieder in den Hauptspeicher geladen werden.
Informationen, die während des Renderprozesses zur Beleuchtung eines Punktes benötigt werden, sind in einer Lichtbeschreibung (**LightDesc**) gespeichert. Diese Klasse besitzt nun schließlich auch die Methode Illuminate().
Doch wie kommt „man“ an diese Lichtbeschreibung? Dafür ist wiederum das **LightObject** zuständig. Kurz vor der Berechnung eines Bildes wird die Routine CreateLightDesc() des Lichtobjekts aufgerufen. Dabei wird ein Objekt vom Typ **LightDesc** erzeugt und zurückgegeben. Mit diesem neuen Objekt arbeitet nun der Renderer. Wenn ein Oberflächenpunkt beleuchtet werden soll, so wird Illuminate() der neuen Lichtbeschreibung aufgerufen.
Für die interaktive Vorschau beim Modellieren existiert eine andere Informationsdatenstruktur, die aber fast nur Daten enthält, die auch für die Darstellung durch 3D – Grafikhardware benötigt werden. Diese Struktur heißt **LightState**. Der aktuelle Status einer Lichtquelle kann ebenfalls beim **LightObject** durch Aufrufen der Methode EvalLightState() abgefragt werden.
Um das Zusammenspiel der einzelnen Objekte zu verdeutlichen, soll ein Beispiel angeführt werden, wobei (imaginär) eine neue Lichtquelle in das Programm integriert werden soll. Das neue Lichtobjekt ist ein Flächenlicht. Um die Beleuchtung realistisch zu berechnen, muss über die gesamte Oberfläche der Lichtquelle integriert werden. Dieser Prozess ist jedoch sehr rechenintensiv und kann somit unmöglich im Vorschaufenster dargestellt werden. Dafür soll approximativ ein Punktlicht dienen.
Zunächst ist es notwendig eine neue Klasse einzuführen, die von **LightObject** erbt (nennen wir sie hier AreaLightObject). Außerdem wird eine neue Unterklasse von **LightDesc** benötigt (im folgenden als AreaLightDesc bezeichnet). Während der Modellierung der Szene kann der Benutzer von 3D Studio Max® jederzeit die Farbe des Abgestrahlten Lichts einstellen. Wenn eine Änderung vorgenommen wurde, wird der aktuelle Status der Lichtquelle vom AreaLightObject abgefragt. Die zurückgegebene **LightState** – Struktur enthält die Informationen, dass die neue Lichtquelle ein Punktlicht ist und welche Parameter sie besitzt, z.B. die Farbe. Mit diesen Daten kann 3D Studio Max® nun das Vorschaufenster aktualisieren.

Später soll die Szene in ein (photorealistisches) Bild transformiert, also gerendert werden. Der Renderer ruft vor der eigentlichen Berechnung die Methode CreateLightDesc() auf. Dabei wird ein neues Objekt vom Typ AreaLightDesc erzeugt, mit allen nötigen Informationen gefüllt und schließlich zurückgegeben. Soll nun ein Oberflächenpunkt beleuchtet werden, wird die Illuminate() – Funktion der neuen Lichtbeschreibung aufgerufen. Darin kann nun die aufwändige Integration durchgeführt werden.
Die Abbildung 3.5 stellt noch einmal alle involvierten Klassen dar.

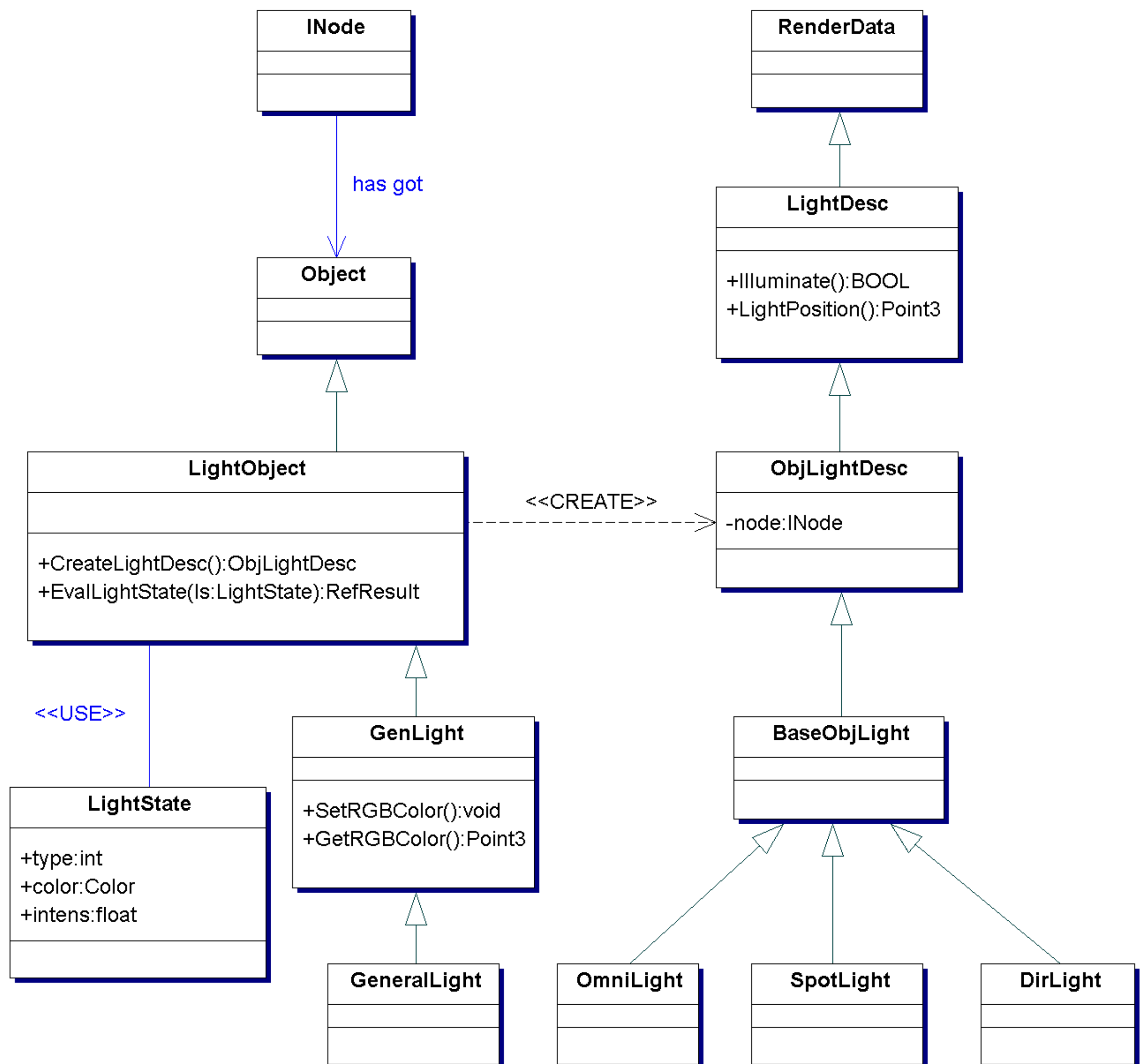

Abbildung 3. 5 – Das Lichtmodul

In der obigen Beschreibung wurde, aus Gründen der Verständlichkeit, noch nicht alle Klassen genannt.
GenLight steht für „generic light“ und stellt, zusammen mit der Klasse **GeneralLight**, eine einheitliche Schnittstelle für alle herkömmlichen Lichtquellen dar. Sie enthalten einige Standardmethoden, die dem Entwickler das Leben ein wenig leichter machen sollen. Zum Beispiel wird ein großer Teil der Nutzerinteraktion von diesen Klassen durchgeführt. Bei einer neuen Lichtquelle braucht der Programmierer lediglich die zusätzlichen Parameter zu integrieren und muss nicht alles komplett neu implementieren.
Die ebenso noch nicht genannten Klassen auf der Seite der Lichtbeschreibung dienen ebenfalls der einheitlichen Beschreibung der Standardlichtquellen.

3.2.5 Materialien und Texturen

In diesem Abschnitt soll es um die Beschreibung der Materialien und Texturen in 3D Studio Max® gehen. Dabei werden ebenfalls Ähnlichkeiten zu dem Entwurf aus Kapitel 2 erkennbar werden.
Wie oben schon erwähnt wurde, benötigen Materialien in der Regel keine Weltmatrix. Genau aus diesem Grund wurde das Basismaterial von 3D Studio Max® (**MtlBase**) von der Klasse **ReferenceTarget** abgeleitet und nicht von **INode**.
Die Entwickler haben außerdem die Verwendung von Maps von vorneherein mit in das Basismaterial integriert. Nun kann darüber gestritten werden, ob dieser Schritt sinnvoll ist oder nicht. Da aber sehr viele Materialien Texturobjekte benutzen werden, ist diese eher pragmatische Entscheidung doch vernünftig.
Eine Map wird in 3D Studio Max® als **Texmap** bezeichnet. Sie stellt eine Spezialisierung von **MtlBase** dar. Dies scheint im ersten Moment vielleicht ein wenig merkwürdig zu sein. Es sollte jedoch nicht vergessen werden, dass das Basismaterial noch keine Shade() – Methode enthält. Diese Klasse besitzt lediglich Grundfunktionalität, wie zum Beispiel die Verwaltung des Namens und der Subtexturen. Damit entspricht sie eher der abstrakten Klasse **IMapHolder** als **IMaterial**.
Die wichtigste Methode der **Texmap** ist sicherlich EvalColor(), was der GetColor() – Funktion des Entwurfs aus Kapitel 2 entspricht. Außerdem wurde von vorneherein die Möglichkeit des Bumpmappings, durch die Routine EvalNormalPerturb(), eingerichtet.
Die eigentliche Schnittstelle für Materialien wird durch die Klasse **Mtl** beschrieben. Sie besitzt nun auch eine Methode mit dem Namen Shade(), die genau wie in der Architektur aus Kapitel 2 das Shading durchführt. Der Ort der Farbberechnung wird durch den als Parameter übergebenen **ShadeContext** (Abschnitt 3.2.3) spezifiziert. Genau wie in der oben beschriebenen Architektur wird das Ergebnis des Shadings ebenfalls mit in der Informationsstruktur abgelegt. Hierbei kommen allerdings noch einige zusätzliche Daten hinzu. Zum Beispiel wird verlangt, dass die Transparenz des Ergebnisses berechnet und gesetzt wird. Dies ist dann wichtig wenn die berechneten Bilder beispielsweise für einen Compositing – Vorgang verwendet werden sollen.
Die Klasse **Mtl** besitzt außerdem noch Methoden um die ambiente, diffuse, spekulare und emittierende Farbe zu verwalten. Nun mag sich der eine oder andere Leser eventuell Fragen, warum das denn so ist. Schließlich benötigt zum Beispiel ein Lambertmaterial keine spekulare Farbe. Dieser Schritt kann allerdings gut begründet werden, denn die Materialien müssen auch im Vorschaufenster visualisierbar sein. Da diese Darstellung sich der 3D – Grafikhardware bedient und diese Karten nur die Standardattribute von Materialien unterstützen, wurden die genannten Methoden mit in **Mtl** integriert. Wie im letzten Abschnitt schon beschrieben wurde, musste für die Lichtquellen ein ähnlicher Weg eingeschlagen werden um die Darstellung während der Modellierung zu ermöglichen. Da die interaktive Vorschau sehr schnell Bilder berechnen muss, wurde an dieser Stelle auch keine Möglichkeit zur Anpassung der Materialien eingefügt. Wenn also zum Beispiel ein Lambertmaterial keine spekulare Farbe besitzt, dann liefert die Methode GetSpecularColor() eben einfach den Farbwert Schwarz zurück. Dadurch wird ebenfalls kein spekulares Glanzlicht dargestellt.
In der Shade() – Routine sieht das natürlich ein wenig anders aus. Hier kann sich der Entwickler eines Materials vollkommen frei entscheiden, welche Attribute er verwendet und welche nicht.
Von der Klasse **Mtl** selbst kann jedoch keine Instanz erzeugt werden, da sie abstrakt ist. Das erste konkrete Material hat den Namen **StdMat**. Wie der Name schon vermuten lässt, stellt diese Klasse ein Standardmaterial dar. Die Shade() – Methode enthält in etwa den selben Inhalt, wie das **CPhongMaterial**. Für das spekulare Glanzlicht kann jedoch auch der Algorithmus von Blinn verwendet werden.

Abbildung 3.6 verdeutlicht den Ableitungsbaum.

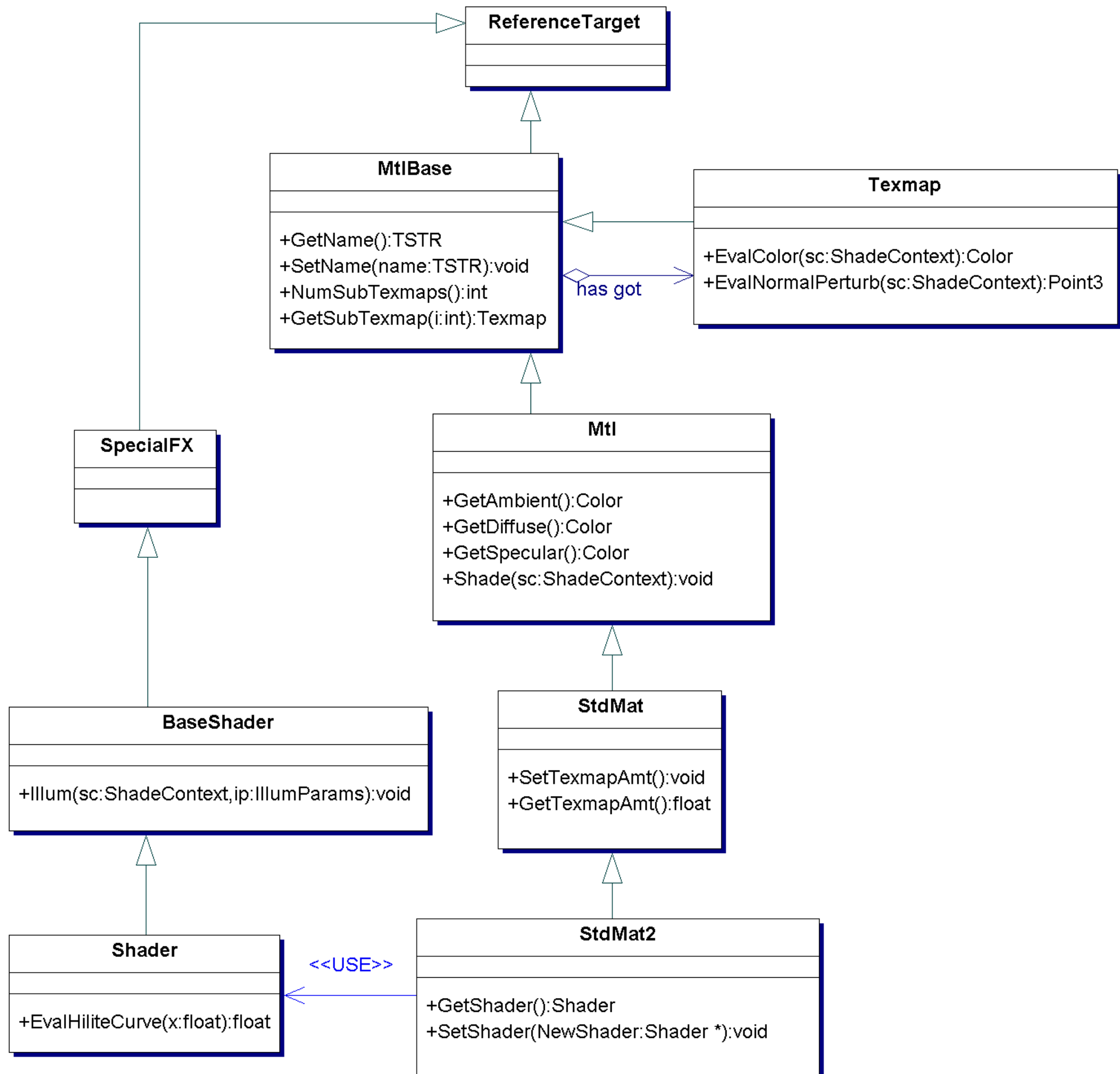

Abbildung 3. 6 – Das Materialmodul

Im Klassendiagramm sind noch einige weitere Klassen mit eingezeichnet. Auf sie soll nun kurz eingegangen werden. Es existiert ein erweitertes Standardmaterial, das so genannte **StdMat2**. Der einzige größere Unterschied zum „normalen" **StdMat** stellt die Verwendung eines **Shader**s dar. Mit dieser zusätzlichen Klasse wurde das Shading weiter vereinheitlicht und vereinfacht. Ein konkreter **Shader** implementiert genau eine Art der Farbberechnung an einem Oberflächenpunkt. Zum Beispiel wurde der Algorithmus von Phong durch die Klasse **Phong** integriert.
Diese zusätzliche Klassenhierarchie wurde eingeführt, weil beobachtet werden konnte, dass neue Materialien häufig die selben Algorithmen immer wieder neu implementierten. Das heißt, viele zusätzliche Plugin - Materialien verwendeten zum Beispiel wieder das Phong – Shading. Dafür schrieben die Programmierer den Quellcode jedes mal aufs neue. Ein Shader arbeitet auf Basis der Standardkanäle, also mit der ambienten, diffusen, spekularen und emittierenden Farbe. Der Entwickler eines Plugins kann sich hier voll und ganz auf den Shader verlassen und braucht sich lediglich um zusätzliche

Beleuchtungseffekte zu kümmern. In einem konkreten Material funktioniert dies nun folgendermaßen: Jedes Material vom Typ **StdMat2** besitzt ein aktuelles **Shader** – Objekt. Wenn nun die Shade() – Methode während des Renderings aufgerufen wird, so braucht der Entwickler des Materials lediglich die Standardkanäle des Shaders mit den momentanen Farben zu füllen und schließlich dessen Funktion Illum() aufzurufen. Nun berechnet das Shader – Objekt automatisch die benötigten Farben und legt sie in der Ergebnisstruktur ab.
Zusätzliche Effekte können im Material berechnet und danach inkrementell zum Ergebnis hinzuaddiert werden.
Dem Entwickler eines Plugin – Materials wird somit eine Menge Arbeit abgenommen.

3.2.6 Rendering

In 3D Studio Max® existieren zwei unterschiedliche Arten von Bildberechnungen. Einerseits wird die Szene während der Modellierung durch eine interaktive Vorschau dargestellt. Dabei werden die Möglichkeiten der modernen 3D Grafikhardware zur Beschleunigung der Berechnung ausgenutzt. Trotz neuer und mächtiger Prinzipien, wie zum Beispiel Vertex- oder Pixelshader, können in der Vorschau nicht alle Beleuchtungs- und Materialeffekte visualisiert werden. Deswegen beschränkt sich diese Art der Darstellung auch auf die Standardparameter der Lichtquellen (vgl. Abschnitt 3.3.4 - **LightState**) und Materialien (vgl. Abschnitt 3.4.5 - **Mtl**).
Die Bildberechnung erfolgt durch die Klasse **Mesh**, die dafür eine Methode mit dem Namen render() enthält. Darin werden die Dreiecke der Geometrie über einen Z – Buffer Algorithmus in das Vorschaufenster gezeichnet. Die Verwendung von 3D Grafikhardware erfolgt transparent. Momentan gibt es Unterstützung für einen Software-, OpenGL- und einen DirectX – Renderer. Diese Schnittstelle kann jedoch von einem Plugin - Programmierer kaum beeinflusst werden. Die Weiterentwicklung von diesem Interface haben sich die Entwickler von 3D Studio Max® selbst vorbehalten.
Anderseits kann die Szene auch mit einem „vollständigen“ Renderer in ein Bild transformiert werden, der das Prinzip des Raytracings verwendet. Hierbei kann nun jede Art von Berechnung ausgeführt werden. Dadurch ist diese Bildberechnung natürlich zeitintensiver als die erste Variante. Dafür sind die Ergebnisbilder dann auch photorealistisch.
Eine solche Art von Renderer kann nun auch von einem Plugin – Entwickler integriert werden. Ein sehr bekanntes Beispiel ist sicherlich Mental Ray® der Firma mental images. Die entsprechende Programmierschnittstelle ist die Klasse **Renderer**. Sie besitzt die Methoden Open(), Render() und Close(). In Open() kann die Szene auf die Bildberechnung vorbereitet werden. Zum Beispiel können räumliche Datenstrukturen initialisiert oder jede andere Form des Preprocessings durchgeführt werden. In Close() sollten alle Änderungen wieder Rückgängig gemacht werden, zum Beispiel kann allokierter Speicher wieder freigegeben werden.
Nachdem die Szene vorbereitet wurde, wird der eigentliche Renderprozess gestartet, indem die Funktion Render() aufgerufen wird. Darin sollte nun für jeden Pixel die Ergebnisfarben (Farbe, Transparenz, ...) berechnet und im Bild gesetzt werden. Standardmäßig besitzt 3D Studio Max® einen Scanlinerenderer.
Ein großes Problem stellt allerdings die Verwendung von sekundären Strahlen dar. Es existiert nämlich keine Klasse, die den Strahlenverfolgungsalgorithmus kapselt und damit editierbar macht. Im Entwurf aus Kapitel 2 wurde diese Anforderung durch die Klasse **CTracer** erfüllt.
Dadurch wird jedoch diese sehr kritische Stelle fast überhaupt nicht anpassbar. Standardmäßig verwendet 3D Studio Max® für sekundäre Strahlen einen Algorithmus, der einer Brute – Force – Implementierung sehr nahe kommt. Es wird nämlich lediglich

ein einfacher Boundingboxtest durchgeführt, um zu entscheiden, ob ein Objekt von dem spezifizierten Strahl getroffen wurde oder nicht. Sollte der Test positiv ausfallen, werden sequentiell alle Dreiecke der Geometrie getestet. Wie man sich sicherlich vorstellen kann, ist diese Art der Schnittpunktbestimmung sehr langsam.
Aus diesem Grund entwickelte die Firma Blur Studio ein Plugin, um diesen Vorgang zu beschleunigen. Blur Studio ist eigentlich ein professionelles Animationsstudio, das aber auch eine Abteilung besitzt, die Plugins für die verwendeten Programme schreibt. Das entwickelte RayFX - Plugin benutzt ein Voxelgitter um den Schnitt von sekundären Strahlen zu beschleunigen.
Diese Entwicklung wurde mit großer Begeisterung in der Entwicklergemeinde von 3D Studio Max® aufgenommen. Mittlerweile gehört das RayFX – Plugin mit zu den Standardelementen des Programms, welche mit jeder Version von 3D Studio Max® ausgeliefert werden.
Die Integration ging sogar so weit, dass die Schattenberechnung der Standardlichtquellen mittlerweile mit Hilfe des RayFX – Plugins berechnet werden.
Tatsächlich sehen sehr viele Entwickler die Nichteditierbarkeit des Strahlenverfolgungs-algorithmus als das größte Manko von 3D Studio Max® an.
Zur Verteidigung muss jedoch gesagt werden, das zur Zeit der Implementierung von 3D Studio Max R1® nur wenige Forschungsarbeiten existierten, die das Thema der Beschleunigung von Raytracing aufgriffen. Auf Grund der Abwärtskompatibilität musste bei der Entwicklung der nächsten Versionen immer wieder die alte Form beibehalten werden.

3.2.7 Serialisierung

3D Studio Max® besitzt das binäre Dateiformat .max um Szenen zu speichern. Da bestimmte Objekttypen eigene Daten besitzen können, wird eine Möglichkeit zur Serialisierung benötigt. Es kann zum Beispiel passieren, dass ein Plugin – Programmierer eine neue Lichtquelle integriert, die aber Daten benötigt, die nicht von den Entwicklern von 3D Studio Max® vorhergesehen wurden. Somit wäre es vollkommen unflexibel, wenn zum Beispiel nur die Standardparameter gespeichert werden könnten.
Damit nun jedes Plugin eigene Daten speichern kann, werden Methoden wie Load() und Save() benötigt. Diese lassen sich auch tatsächlich finden und zwar in der Klasse **ReferenceMaker** (vgl. Abschnitt 3.2.1). Jedes Objekt, welches als Plugin in 3D Studio Max® integriert werden kann und das potentiell eigene Informationen besitzt, erbt von dieser Klasse. Es ist mehr oder weniger die Basisklasse für fast alle Plugins.
Nun braucht der Entwickler eines solchen Zusatzmoduls also nur noch die zwei genannten Methoden zu überschreiben und schon kann er eigene Daten speichern.
Der Funktion Save() wird dabei ein Zeiger auf die (globale) Klasse **ISave** übergeben, mit der die Informationen in den Ausgabestrom geschrieben werden können.
Analog dazu wird Load() ein Zeiger auf eine Klasse mit dem Namen **ILoad** übergeben, mit dessen Hilfe die Daten wieder geladen werden können.
Die Abbildung 3.7 veranschaulicht dies noch einmal.

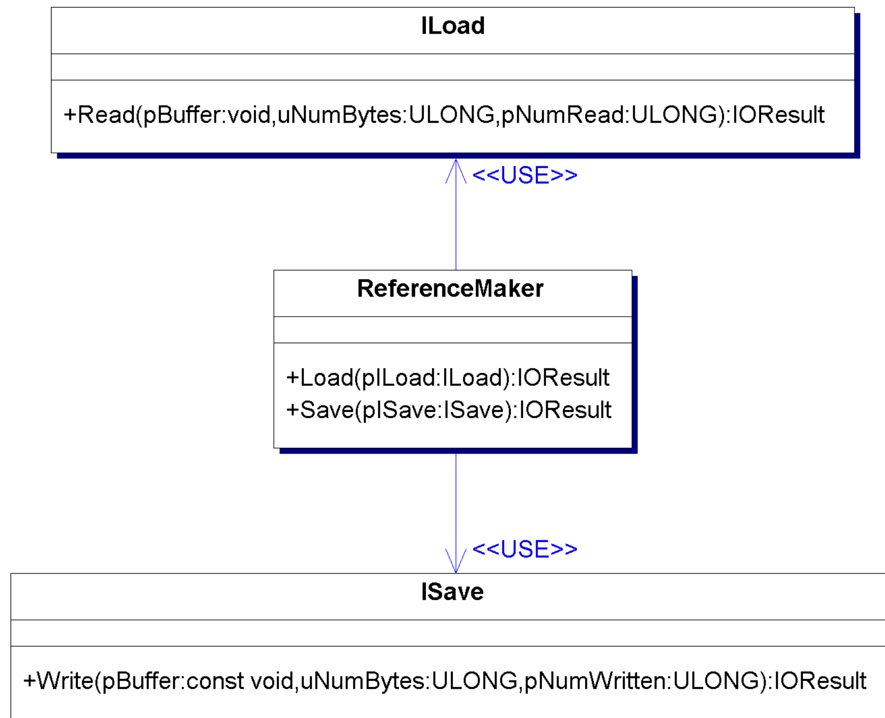

Abbildung 3. 7 – Serialisierung

3.2.8 Import und Export von Szeneninformationen

Die Möglichkeit verschiedene Dateiformate zu unterstützen ist heutzutage eins der wichtigsten Voraussetzungen für ein professionelles Modellierungs- und Animationsprogramm. Das Vorgehen in 3D Studio Max entspricht dabei fast vollständig dem aus Kapitel 2. Deswegen soll hier auch nur kurz darauf eingegangen werden.
Die Schnittstellen für den Import bzw. Export von Szeneninformationen werden durch die Klassen **SceneImport** und **SceneExport** beschrieben. Die wichtigste Methode der Importschnittstelle heißt DoImport() und ist dafür zuständig, die Daten aus der spezifizierten Datei zu laden und 3D Studio Max® hinzuzufügen. Dementsprechend besitzt die Exportschnittstelle die Funktion DoExport(), die natürlich die Informationen der Szene in die genannte Datei speichern soll.

3.2.9 Implementierung eines Exporters

Um die Entwicklung eines Plugins für 3D Studio Max® verständlicher zu gestalten, wurde im Rahmen dieses Belegs ein Exporter entwickelt. Dieses Modul speichert die aktuelle Szene in eine externe Datei, die dann von dem Raytracer aus Kapitel 2 geladen werden kann. Die Daten werden in Form einer Textbeschreibung (ASCII) gespeichert. Das hat den Vorteil, dass bestimmte Parameter auch noch im nachhinein mit Leichtigkeit editiert werden können. Alles was dafür benötigt wird, ist ein Texteditor.
Ein Nachteil dieser Art der Speicherung ist allerdings, dass die Dateien unter Umständen sehr groß werden. Eine binäre Speicherung wäre hier effektiver.
Der Quelltext des Exporters umfasst rund 2000 Zeilen C++ Code. (2500 Zeilen wenn Kommentarzeilen mitgezählt werden.)
Viele der im Kapitel 3 erläuterten Fakten wurden in dem Exporter aktiv verwendet. Dem einen oder anderen Leser dient es eventuell als leicht verständliche Einführung in das MaxSDK.

3.2.10 Zusammenfassung und Ausblick

Im dritten Kapitel wurde exemplarisch die Architektur einer professionellen Software untersucht. Die Fallstudie wurde an 3D Studio Max® durchgeführt. Dabei wurde vor allem auf die Aspekte näher eingegangen, die auch im Kapitel 2 erläutert wurden.
Bei der Analyse der Programmierschnittstelle des Programms wurden viele Ähnlichkeiten aber auch einige Unterschiede zum Entwurf aus Kapitel 2 erkenntlich. Viele der Unterschiede lassen sich auf die umfangreichere Aufgabenstellung von 3D Studio Max® zurückführen. Es kann eben nicht nur zum Rendern von Szenen, sondern auch zum Modellieren und Animieren verwendet werden.
Um eben dies flexibel zu ermöglichen, müssen viel mehr Überlegungen in die Struktur der Software investiert werden. Andererseits ist häufig die Abwärtskompatibilität ein Hemmnis für einen neuen flexibleren Entwurf.
Trotzdem ist es schon sehr beeindruckend, wie trotz einiger Schwachstellen in der Programmierschnittstelle, die Anforderungen der vielen Nutzer erfüllt werden können. Das heißt, die Architektur ist immerhin so flexibel, dass vieles im nachhinein noch integriert werden kann.
Zum aktuellen Zeitpunkt (Juni 2003) befindet sich 3D Studio Max® in der Version 5. Eins der wichtigsten neuen Features ist sicherlich die Erweiterung des Beleuchtungsmodells zu einen globalen Modell. Dabei werden zwei verschiedene Ansätze unterstützt. Einerseits die Finite Elemente Methode Radiosity und eine Variante des Monte Carlo Raytracings.
Es ist zu erwarten, das 3D Studio Max® auch in Zukunft eine große Rolle auf dem internationalen Markt spielen wird. Deswegen wird es wohl früher oder später eine vollständige Neuentwicklung des Programms geben. Dadurch kann dann der Ballast der Abwärtskompatibilität zurückgelassen werden.
Außerdem ist 3D Studio Max® momentan nur auf das Betriebssystem Microsoft Windows® beschränkt. In der Zukunft wird wahrscheinlich immer wichtiger werden, das Programme auch unter einer heterogenen Umgebung arbeiten können.

Danksagung

Ich möchte mich bei einigen Personen bedanken, ohne die der Beleg heute nicht in dieser Form vorliegen würde:
Erst einmal bedanke ich mich bei meiner Familie und ganz speziell bei meinen Eltern.
Außerdem möchte ich Matthias Bräuer dafür danken, dass er mir viele hilfreiche Verbesserungsvorschläge gab.
Ganz besonderen Dank gilt auch dem Lehrstuhl für Computergrafik der Technischen Universität Dresden. Ich wünsche Prof. Dr. Deussen viel Erfolg in Konstanz!

Schlusswort

In der Einleitung wurde die Bedeutung von computergenerierten Bildern in der heutigen Zeit schon erwähnt. In diesem Zusammenhang sollte auch auf die Verantwortung der Medien hingewiesen werden. Gerade jetzt, wo es möglich ist Bilder und Animationen zu erstellen, die fast so real wie echte Videoaufnahmen aussehen, ist die Medienkompetenz von integraler Bedeutung.
Jede Technik kann nämlich auch immer für falsche Zwecke missbraucht werden. Die Medien besitzen einen großen Einfluss auf unser alltägliches Leben. Sie beeinflussen uns manchmal mehr, als man das zugeben will.
Durch die Möglichkeit photorealistische Bilder zu erzeugen, kann die Wirklichkeit verzerrt, ja sogar komplett verändert werden. Dies mag zwar in Hollywoodfilmen noch ein aufregender Effekt sein, in der Realität ist hier ein gewisses Risikopotential vorhanden. Die Meinung der Massen kann dann in eine bestimmte Richtung verschoben werden. Dies könnte wiederum dafür missbraucht werden, dass einige Entscheidungen als legitim erscheinen, obwohl sie das nicht sind.
Die Medien sollten sich also stets vor Augen halten, dass sie dafür da sind die Leute zu Informieren nicht aber sie zu kontrollieren!
Nun soll dieser Beleg aber nicht mit dem erhobenen Zeigefinger beendet werden. Schließlich sollte nicht vergessen werden, wie viel Spaß Computergrafik macht!
Im Kino werden aufregende Welten gezeigt, die ohne moderne Software nicht machbar wären. Die Freunde der interaktiven Unterhaltung erfreuen sich zudem an den neusten Computerspielen, die von Jahr zu Jahr immer aufwendigere und schönere Grafik zeigen.
Doch die Computergrafik leistet noch viel mehr! Es wird endlich möglich abstrakte Sachverhalte zu illustrieren und damit leichter verständlich zu machen. Riesige Datenmengen werden erst jetzt für den Mensch verarbeitbar, weil sie eben nicht als endlos lange Zahlenreihe, sondern als Diagramm oder Bild dargestellt werden.
Die Liste der Anwendungen ist endlos.
Computergrafik ist wohl eins der schönsten Forschungsgebiete der Informatik.

Weiterführende Literatur

Rendering / Raytracing / Global Illumination:

Jose Encarnacao, Wolfgang Straßer, Reinhard Klein
Graphische Datenverarbeitung I und II
ISBN: 3-486-23469-2

Allen Watt, Mark Watt
Advanced Animation and Rendering Techniques – Theory and Practice
ISBN: 0-201-54412-1

Michael F. Cohen, John R. Wallace
Radiosity and Realistic Image Synthesis
ISBN: 0-12-178270-0

Andrew Glassner:
Principles of Digital Image Synthesis
ISBN: 1-55860-276

Andrew Glassner
An Introduction to Ray Tracing
ISBN: 0-12-286160-4

Henrik van Jensen
Realistic Image Synthesis Using Photon Mapping
ISBN: 1-56881-147-0

Internetressourcen:

Rendering / Raytracing:

http://www.graphics.cornell.edu	Cornell University
http://www.graphics.stanford.edu	Stanfort University
http://www.povray.org	Open – Source - Raytracer

3D Studio Max®:

http://www.discreet.com	Offizielle Hompage
http://www.3dmax.de	Deutschsprachiges Forum
http://www.3dcafe.com	Ressourcen und Plugins
http://www.3dtotal.com	Galerie, Ressourcen und Plugins
http://www.highend3d.com	Galerie, Ressourcen und Plugins

Programmierung:

http://www.codeguru.com	Allgemeine Ressource
http://www.programersheaven.com	Allgemeine Ressource
http://www.flipcode.com	Hauptsächlich Echtzeit 3D Grafik
http://msdn.microsoft.com	Windowsprogrammierung

Quellennachweis

James D, Foley, Andries van Dam, Steven K. Feiner, John F. Huges
Computer Graphics – Principles and Practice (Second Edition in C)
ISBN: 0-201-84840-6

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software
ISBN: 3-8273-1862-9

Hilfe des MaxSDK
In jeder Version von 3D Studio Max® enthalten.
Online abrufbar unter: http://sparks.discreet.com