Projektdokumentation

– RayTracer –

Autoren: Daniel Gülzow,
Kolja Schötzau,
Steffen Heinrich,
Stephan Brumme,
Thomas Adam

Potsdam, den 2. September 2002

Bearbeitungsvermerk:

Vorwort

Aufgabe des Projektes ist es, eine auf Komponenten basierende Anwendung zu schreiben. In der Vorlesung „Computergrafik II“ des Herrn Professor Döllner haben wir einen RayTracer entwickeln müssen, in dessen Umsetzung viel Zeit investiert wurde. Daher erscheint uns gerade hier ein RayTracer als ein sehr passendes Beispiel. Schon aus dem Grund heraus, dass uns bei der Arbeit in der Computergrafik aus eigener Erfahrung klar wurde, dass ein RayTracer aufgrund seiner prinzipiellen Arbeitsweise ein extrem rechenaufwendiges Programm darstellt.

Aus dieser Erfahrung heraus haben wir den Entschluss gefasst, in dieser Projektarbeit einen verteilt rechnenden RayTracer zu entwerfen. Die zweite Herausforderung war zugleich, dass ganze in der Sprache C# umzusetzen. Sie half uns enorm, die Verteilung zu programmieren.
Dennoch soll die Komponentenprogrammierung hierbei im Mittelpunkt stehen. Beispielsweise kann der Benutzer entscheiden, ob er das Bild mittels OpenGL rendern lassen will oder mittels RayTracing. Diese Unterscheidung treffen wir, und natürlich der spätere Benutzer auch, während der Laufzeit des Programms, durch explizites dynamisches Laden einer OpenGL– oder RayTracing – Komponente.

Kapitel 1 umreißt in kurzen Zügen, was unsere Anwendung zu leisten im Stande ist. Das zweite Kapitel beschreibt, die theoretischen Grundlagen des RayTracing und gibt dazu ein paar anschauliche Beispiele. Im nächsten Kapitel werden die Anforderungen erläutert, die an das fertige Produkt – die verteilte Anwendung – gestellt werden. Im speziellen wird hier auf den RayTracer eingegangen. Kapitel 4 und 5 greifen dieses auf und leitet daraus die zugrunde liegende Architektur des Programms ab. An den richtigen Stellen werden verwendete Design Pattern genannt. Das 6. Kapitel erläutert schließlich die Realisierung der verteilten Berechnung unter Verwendung von Web Services. Das folgende Kapitel 7 geht dann auf den Test des Projektes und dessen technische Realisierung ein. Die beiden letzen Kapitel beantworten zum einen die Frage: warum wir uns für C# und .Net entschieden haben und geben zum anderen einen Ausblick auf Veränderungen, die gewünscht, aber noch nicht eingeflossen sind.
Im Anhang wird noch eine Möglichkeit vorgestellt, eine inhaltlich und visuell sehr ansprechende Dokumentation ohne eigenes Zutun zu gewinnen.

Inhaltsverzeichnis

1. Bedienung ..... Seite 5
1.1. Installation ..... Seite 5
1.2. Benutzung ..... Seite 5

2. Prinzipien des RayTracing ..... Seite 8
2.1. Berechnung ..... Seite 8
2.2. Schnittpunktberechnung ..... Seite 10
2.3. Beleuchtung ..... Seite 12

3. Anforderungsspezifikation ..... Seite 16

4. Architektur ..... Seite 19
4.1. Darstellungsunabhängige Klassen ..... Seite 19
4.2. Spezialisierte Klassen zur Darstellungsberechnung ..... Seite 21
4.3. GUI ..... Seite 25

5. OpenGL ..... Seite 29
5.1. Das Konzept ..... Seite 29
5.2. Geometriebasierter Ansatz ..... Seite 29
5.3. Was ist OpenGL ..... Seite 30
5.4. Einsatz ..... Seite 31

6. Web Services ..... Seite 33
6.1. Einleitung ..... Seite 33
6.2. Probleme traditioneller verteilter Anwendungen ..... Seite 33
6.3. Spezifikationen ..... Seite 34
6.4. Genereller Ablauf der Benutzung eines Web Services ..... Seite 35
6.5. ASP.NET und Web Services ..... Seite 36
6.6. Ein RayTracer als verteilte Anwendung ..... Seite 38

7. Testen ..... Seite 43
7.1. Grundlegendes ..... Seite 43
7.2. Klassentest ..... Seite 46
7.3. Integrationstest ..... Seite 51

8. Warum C# und .NET ..... Seite 56

9. Ausblick ..... Seite 60
9.1. Nutzung des UDDI – Services ..... Seite 60
9.2 Dynamisches Hinzufügen neuer Shapes ..... Seite 61

Anhang ..... Seite 62
A1. Das Dokumentationswerkzeug NDoc ..... Seite 62
A2. CD – ROM ..... Seite 64

1. Bedienung

1.1. Installation

Im Verzeichnis **binary** der beiliegenden CD–ROM ist eine lauffähige Version unseres Programmes zu finden. Als Voraussetzung muss das .NET–Framework installiert sein; wir empfehlen weiter das Servicepack 1. Der Internet Information Service (IIS) wird für einen erfolgreichen Einsatz der Web Services ebenfalls benötigt.

Zunächst empfiehlt sich die Installation der Web Services selbst. Dazu ist ein virtuelles Verzeichnis anzulegen, das auf das Verzeichnis **binary\service** verweist. Für Testzwecke haben wir auf unserem Rechner gleich drei solcher Verzeichnisse erzeugt, die alle auf das obige physikalische Verzeichnis zeigen. Wichtig ist, dass sie das Recht zum Ausführen von Skripten erhalten.

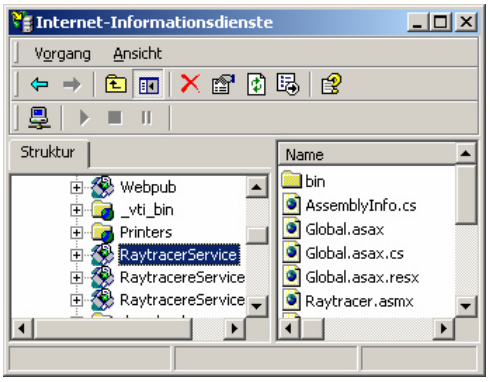


Abbildung 1: Virtuelle Verzeichnisse im IIS

In den Dateien `raytracer.exe.config` und `editor.exe.config` sind die letztlich gewählten Verzeichnisse festzuhalten:

```
<configuration>
 <appSettings>
  <add key="Worker"     value="3" />
  <add key="Raytracer1" value="http://localhost/RaytracerService1/Raytracer.asmx?WSDL" />
  <add key="Raytracer2" value="http://localhost/RaytracereServic2/Raytracer.asmx?WSDL" />
  <add key="Raytracer3" value="http://localhost/RaytracereServic3/Raytracer.asmx?WSDL" />
 </appSettings>
</configuration>
```

1.2. Benutzung

Der Dreh– und Angelpunkt des Projektes ist der Editor. Er verbirgt sich hinter der Datei `Editor.exe`. Beim Start präsentiert er sich in einem leeren Fenster; kann aber über den Menüpunkt **File/New** mit einer neuen Szene gefüllt werden:

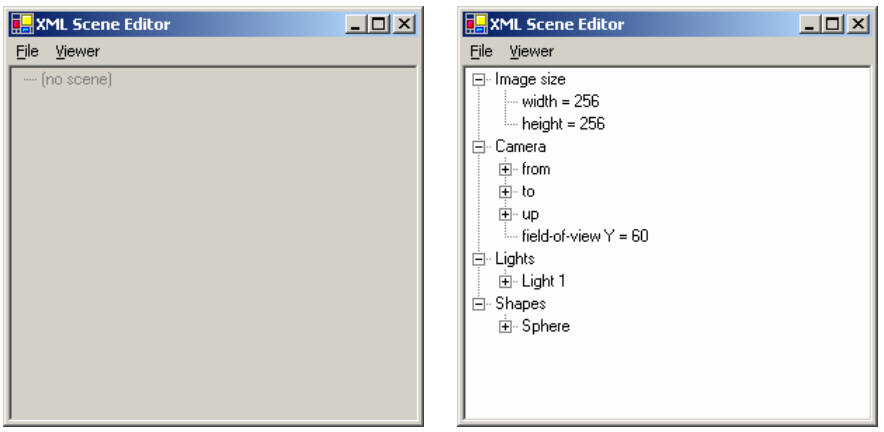


Abbildung 2: Der Editor direkt nach dem Start (links) bzw. beim Anlegen einer neuen Szene (rechts)

Über **Viewer/New Viewer Component** wählt man die gewünschte Darstellungskomponente aus. Zur Verfügung stehen hierzu die Komponenten `raytracer.dll` (nicht `.exe`!) und `opengl.dll`. Im ersten Schritt wird der RayTracer geladen.

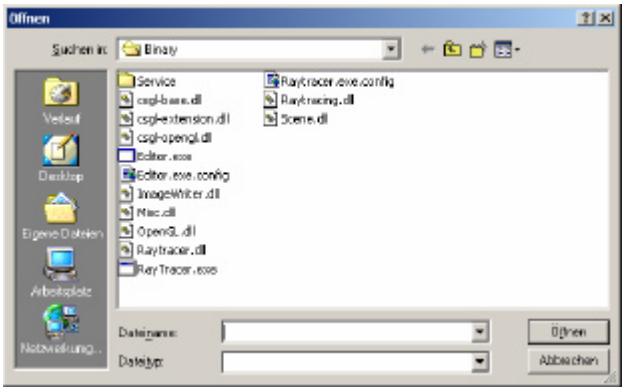


Abbildung 3: Auswahl einer Komponente

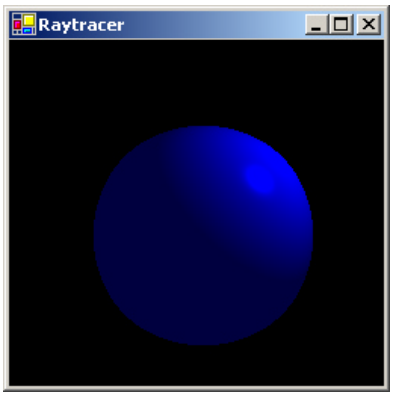


Abbildung 4: Das Ergebnis des RayTracer

Beim erstmaligen Aufruf des RayTracers muss zuvor noch das .NET–Framework starten, so dass selbst auf modernen Rechnern einige Sekunden vergehen, bis sich eine erste Reaktion einstellt.

Gerade in der Modellierungsphase bietet es sich daher an, die schnellere OpenGL–Vorschau zu benutzen. Dazu wählt man unter **Viewer/New Viewer Component** nun `opengl.dll` aus. Das Ergebnis ist innerhalb von Sekundenbruchteilen sichtbar. Leider zeigt sich aufgrund der unvollständigen Fremdbibliothek, die wir einsetzen, das Bild nur in Graustufen (vgl. Abbildung 5)

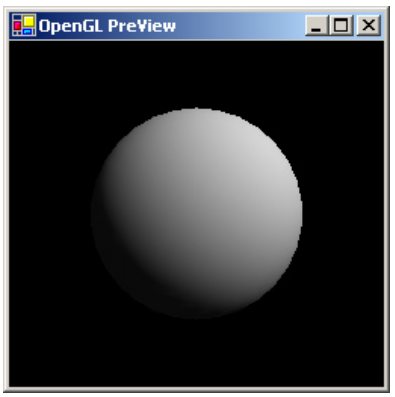


Abbildung 5: Die OpenGL–Vorschau

Die Szene kann im Editor beliebig verändert werden. Dazu klickt man einfach im „TreeView“ auf das zu bearbeitende Element. Je nachdem, ob es sich um einen skalaren Wert oder einen Vektor handelt, öffnet sich ein passender Dialog, der es erlaubt, genau diesen Typ zu ändern:

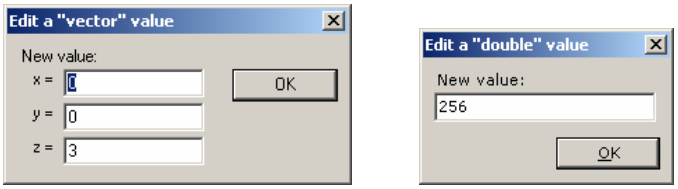


Abbildung 6: Bearbeiten eines Vektors (links) bzw. eines skalaren Wertes (rechts)

Über das Kontextmenü – aufrufbar über die rechte Maustaste – ist es des Weiteren möglich, neue Objekte oder Lichtquellen der Szene hinzuzufügen. Natürlich kann eine Szene jederzeit im Menü **File** gespeichert oder geladen werden.

2. Prinzipien des RayTracing

In diesem Kapitel geht es um die Grundlagen des RayTracing. RayTracing, zu Deutsch: Strahlenverfolgung, beschreibt ein Verfahren zur Erzeugung sehr realistischer Bilder. Der Algorithmus berechnet den Weg der für die Szene relevanten Lichtstrahlen unter Verwendung physikalisch approximierter Gesetze, wie beispielsweise Brechung und Reflektion. Die Verfolgung der Lichtstrahlen geschieht dabei zur Überraschung aller rückwärts. D.h. man wählt einen Lichtstrahl aus, der ins Auge des Betrachters fällt – gleichzusetzen mit dem Kamerastandpunkt – und ermittelt davon ausgehend seine Vergangenheit: Wo hat er seinen Ursprung und wieviel Licht trägt er dadurch mitsich?

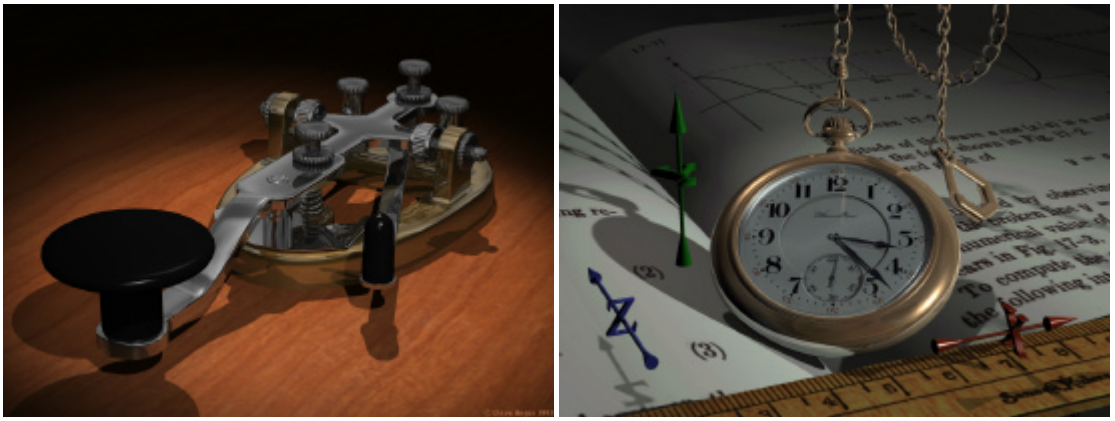


Abbildung 7: Zwei wirklich sehr realistische Bilder, die einem RayTracer entstammen.

2.1. Berechnung

Als Ergebnis liefert das RayTracing ein Bild – besser: ein zweidimensionales Feld mit Farbwerten – für jeden Punkt des sichtbaren Bereiches (vgl. Abbildung 8). Die Berechnung der Farbwerte selbst läuft dabei in mehreren Schritten ab.

Im ersten Schritt bestimmt man die Richtung der Strahlen, die ins Auge des Betrachters fallen und somit zurückzuverfolgen sind.

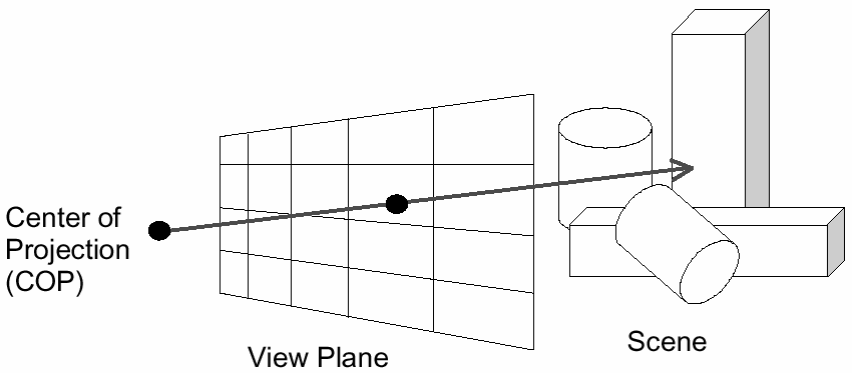


Abbildung 8: Prinzipielle Arbeitsweise des RayTracing[1]

[1] Dieses und weitere skizzenhaften Abbildungen entstammen den Skript zur Vorlesung „Computergrafik II“

Jeder Punkt der „View Plane“ (sichtbarer Bereich) wird mit einem Strahl „mittig getroffen“. Zusammen mit dem Wissen über den Betrachterstandpunkt (COP der Abbildung 8) kann man somit die Orientierung der Strahlen bestimmen. Die letztendliche Beleuchtung für einen bestimmten Punkt der „View Plane“ und damit des späteren Bildes hat zwei mögliche Ursachen:

- direkte Beleuchtung durch Lichtquellen,
- indirekte Beleuchtung durch Reflektion und Refraktion (Brechung).

Die einzigen Objekte, die dabei selbst Licht abstrahlen können, sind die Figuren der Szene selbst. Der Fall das der Beobachter direkt in eine Lichtquelle schaut, wird im RayTracing nicht berücksichtigt.

Nach Ermittlung der Richtung eines Strahls, untersucht man anschließend jeden Strahl auf Schnittpunkte mit etwaigen Szenenobjekten. In unserem Fall können das Kugel, Box oder Dreiecke sein. Dabei ist allerdings nur das erste Szenenobjekt, auf das der Strahl trifft, für den Betrachter von Bedeutung. Denn die restlichen Objekte werden vom ersten überdeckt.

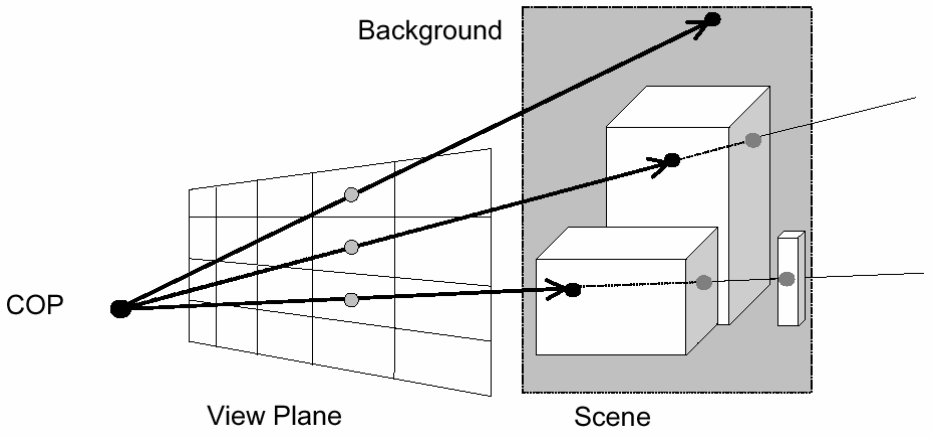


Abbildung 9: Schnittpunktberechnung mit den Szenenobjekten

Für jeden ermittelten Schnittpunkt wird nun die Lichtintensität berechnet, die im ersten Schritt durch den direkten Einfluss von Lichtquellen hervorgerufen wird. In unserem Fall kommt dazu das Beleuchtungsmodell nach Phong (siehe weiter unten im Text) zum Einsatz.
Zu den berechneten Werten wird dann die Intensität addiert, die aus der indirekten Beleuchtung resultiert. Ein Strahl wird unter Umständen an einer verspiegelten Oberfläche reflektiert, so dass der aktuell untersuchte Oberflächenpunkt weiteres Licht von anderen Objekten erhält. Theoretisch wird dieser Effekt durch die Bestimmung des Reflektionsstrahls am Schnittpunkt ermöglicht (vgl. Abschnitt: Beleuchtung). Für diesen neu gewonnenen Strahl werden alle bisher beschriebenen Schritte nochmals ausgeführt. D.h. Rekursion tritt zu Tage, womit sich die relative lange Rechenzeit pro Bild erklärt. Analog verhält es sich mit der Lichtbrechung (Refraktion), die ausschließlich bei transparenten Objekten auftritt. Ein endgültiger Farbwert für den Bildpunkt liegt vor, wenn eine maximale Rekursionstiefe erreicht wird und sie somit abbricht.

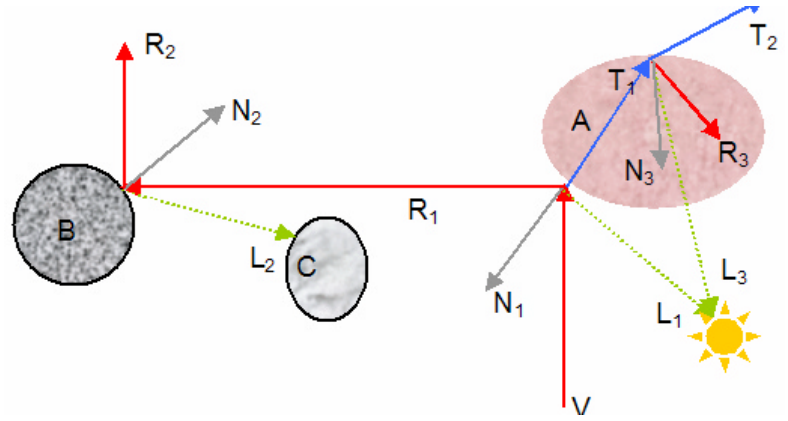


Abbildung 10: Strahlen n–ter Ordnung: Reflektion (rot), Refraktion (blau), Schattenfühler (grün)

Zusammenfassend kann man das Prinzip des RayTracing in fünf Schritte festhalten:

1. Ermittlung der Strahlen (Ursprung, Orientierung)
2. Berechnung der Schnittpunkte mit den Szenenobjekten
3. Ermittlung der Lichtintensität aus direkter Beleuchtung
4. Ermittlung der Lichtintensität aus indirekter Beleuchtung (rekursive Bestimmung)
5. Speichern des Farbwertes

2.2. Schnittpunktberechnung

Hinsichtlich der Schnittpunktberechnung werden bei unserem RayTracer drei Fälle unterschieden:

1. Schnitt Strahl – Dreieck
2. Schnitt Strahl – Box
3. Schnitt Strahl – Kugel

Für diese drei Fälle werden nun die Ideen zur Berechnung der Schnittpunke genauer beschrieben.

Schnitt Strahl – Dreieck

Ein Dreieck ist bekannt, wenn seine drei Eckpunkte (e_1, e_2, e_3) gegeben sind. Der Strahl repräsentiert sich durch seinen Stützvektor sowie einen Richtungsvektor, der seine Orientierung im Raum wiedergibt.
Die Idee zur Schnittpunktberechnung liegt in der Verwendung von barizentrischen Koordinaten. Ein so spezifizierter Punkt in einem Dreieck ist durch folgende Gleichung eindeutig beschrieben:

$$P = (1 - u - v) \cdot e_1 + u \cdot e_2 + v \cdot e_3$$

wobei u und v die barizentrische Koordinaten darstellen, für die gilt: u, v liegen im Intervall [0, 1].

Die Strahlengleichung ist allgemein bekannt durch:

$$\text{Ray}: \vec{x} = \begin{pmatrix} O_x \\ O_y \\ O_z \end{pmatrix} + r \cdot \begin{pmatrix} D_x \\ D_y \\ D_z \end{pmatrix} \qquad ; r \in \Re$$

In der Strahlengleichung stellt O den Ortsvektor des Startpunktes dar bzw. der Vektor D die räumliche Orientierung des Strahls. Durch Gleichsetzen der Dreiecks– mit der Strahlengleichung gewinnt man den Schnittpunkt.

Schnitt Strahl – Box

Eine Box – ein achsenparalleler Quader – ist gegeben durch zwei Punkte, den vorderen linken und den rechten hinteren Punkt.
Zur Schnittpunkberechnung werden die Ebenen betrachtet, die die Box einschließen. Mit Hilfe der zwei jeweils zueinander parallelen Ebenenpaare gewinnt man den Abstand dieser zum Strahlursprung. Dadurch erhält man insgesamt vier Werte: 1 Minima (near) und 1 Maxima (far) für jedes Ebenenpaar.

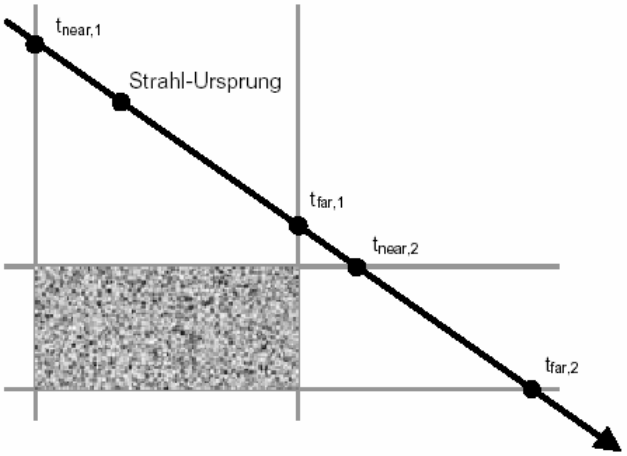


Abbildung 11: Zwei der Ebenenpaare (Draufsicht)

Es existiert nur dann ein Schnittpunkt, falls gilt:

$$\max(t_{near1}, t_{near2}) < \min(t_{far1}, t_{far2})$$

Schnitt Strahl – Kugel

Eine Kugel wird eindeutig durch die Angabe des Mittelpunkts $M(m,n,o)$ und einem Radius r größer Null spezifiziert. Die Schnittpunktberechnung geschieht durch Einsetzen der Strahlengleichung (vgl. weiter oben) in die Kugelgleichung in allgemeiner Lage.

$$\text{Sphere}: (x-m)^2 + (y-n)^2 + (z-o)^2 = r^2$$

Durch die Berechnung der Werte für x, y und z erhält man letztlich den Schnittpunkt von Strahl und Kugel.

2.3. Beleuchtung

Für jeden Schnittpunkt wird die Lichtintensität berechnet. Sie ist entscheidend dafür, welche Farbe sich am Punkt zeigt und schließlich gespeichert wird. In unserem RayTracer kommt, wie gesagt, das Beleuchtungsmodell von Phong zum Einsatz. Dieses Modell beschreibt nur die direkte Beleuchtung durch Lichtquellen. Der indirekte Beleuchtungsanteil, der durch Reflektion und Refraktion entsteht, schlägt sich durch weiteres Addieren in der jetzt hybriden Beleuchtungsgleichung wieder. Ein weiteres Phänomen ist die Schattenbildung, die hier ebenfalls kurz vorgestellt werden soll.

Das Phong–Beleuchtungsmodell

Mathematisch drückt sich das Beleuchtungsmodell wie folgt aus:

$$I = I_{Ambient} \cdot k_{Ambient} + I_{Diffuse} \cdot k_{Diffuse} \cdot (N \bullet L) + I_{Spekular} \cdot k_{Spekular} \cdot (R \bullet V)^n$$

$k_{Ambient}$, $k_{Diffuse}$ und $k_{Spekular}$ beschreiben dabei Konstanten und werden durch die Materialbeschaffenheit der Szenenobjekte definiert. Der erste Summand: $I_{Ambient} \cdot k_{Ambient}$ gibt die allgemeine Raumhelligkeit (ambiente Beleuchtung), die den Lichteinfall großer Öffnungen, wie Fenster etc. simulieren soll, wieder. Der zweite Summand: $I_{Diffuse} \cdot k_{Diffuse} \cdot (N \bullet L)$ stellt das Streulicht (diffuse Beleuchtung) dar, das an der Objektoberfläche in alle Richtungen gleichmäßig abgestrahlt wird. Matte Oberflächen sind meist stark diffus reflektierend. Der dritte und letzte Summand: $I_{Spekular} \cdot k_{Spekular} \cdot (R \bullet V)^n$ ruft das Glanzlicht / Spiegellicht (spekulare Beleuchtung) an Oberflächen hervor, das sich besonders in der Spiegelrichtung R konzentriert. Um die Spiegelrichtung korrekt bestimmen zu können, ist das Wissen über den Standort des Betrachters notwendig. Jener wird durch den Vektor V repräsentiert.

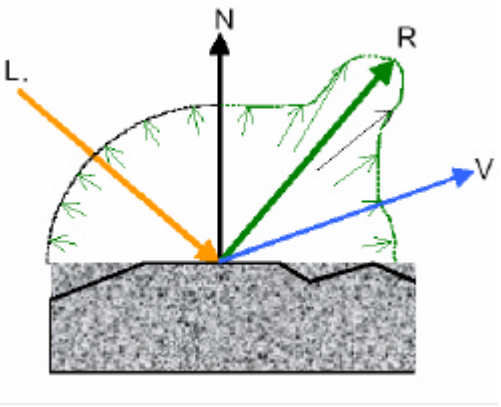


Abbildung 12: Phong–Beleuchtungsmodell

Umsetzung der Reflektion

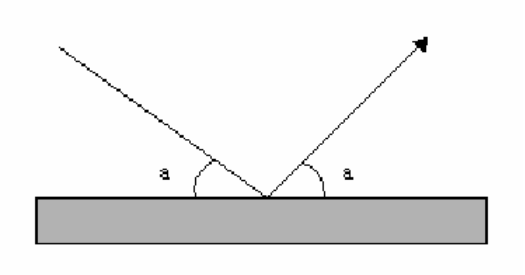


Abbildung 13: Einfache Reflektion

Durch die Reflektion wird ein Strahl erzeugt, dessen Orientierung durch den Einfallswinkel berechenbar ist. Denn bezüglich der Reflektion gilt: Einfallswinkel gleich Ausfallswinkel, so dass über diese Annahme sich die neue Richtung des Strahl mittels des Skalarprodukts bestimmen lässt.
Mit ihm wird dann der gesamte RayTracing–Algorithmus nochmals durchlaufen. Das Ergebnis ist wiederum eine Farbintensität, die mit der bisherigen (vgl. Phong–Beleuchtungsmodell) verrechnet wird.
Reflektionsstrahlen sind stets Sekundärstrahlen, d.h. sie werden erst ab einer Rekursionstiefe von zwei und mehr berücksichtigt.

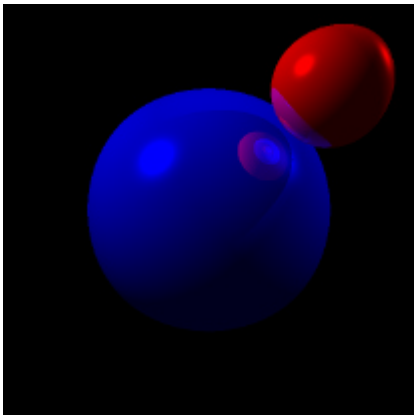


Abbildung 14: Ergebnis der durch Rekursion realisierten mehrfachen Reflektion

Umsetzung der Refraktion

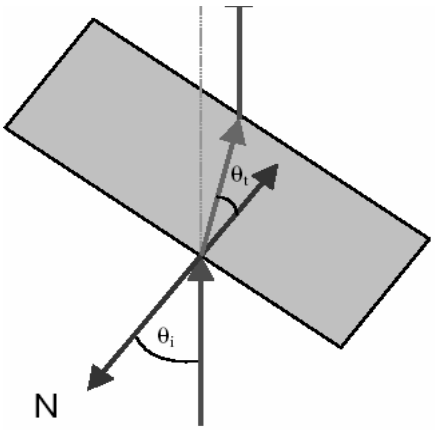


Abbildung 15: Refraktion / Lichtbrechung

Die Refraktion erfolgt nur, falls der Primärstrahl auf ein transparentes Objekt trifft. Ist dies der Fall, wird durch Einfallswinkel und Brechungsindices der Eintritts– und Austrittsmedien der Brechungsstrahl bestimmt. Die Berechung beruht auf dem Brechungsgesetz von Snellius, das folgendes beinhaltet:

$$\frac{\sin(\text{Einfallswinkel})}{\sin(\text{Ausfallwinkel})} = \frac{\text{Brechungsindex}_{\text{Mediumaustritt}}}{\text{Brechungsindex}_{\text{Mediumeinfall}}}$$

Auch mit dem berechneten Refraktionsstrahl wird erneut der RayTracing–Algorithmus durchlaufen. Damit zählen sie wie die Reflektionsstrahlen zu den Sekundärstrahlen.

Abbildung 16: Lichtbrechung an einer transparenten Kugel

Schatten

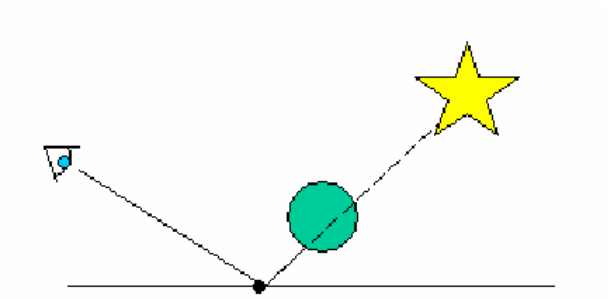


Abbildung 17: Prinzip der Schattenbildung

Die Prüfung, ob ein Schattenwurf vorliegt, erfolgt beim RayTracing anhand so genannter Schattenfühler. D.h. man entsendet einen Strahl vom Punkt, der aus dem Schnitt Objekt / Primärstrahl hervorgeht, in Richtung der Lichtquelle. Falls er auf seinem Weg dorthin ein anderes Objekt trifft und sich dieses zwischen Lichtquelle und Schnittpunkt befindet, liegt der Ausgangspunkt des Schattenfühlers im Schatten. Schattierte Bereiche einer Oberfläche erhalten nur den ambienten Term des Phong–Beleuchtungsmodells.

Was bis jetzt zum Thema: Schatten diskutiert wurde, besitzt nur solange Gültigkeit, wie das Objekt, welches sich zwischen Schnittpunkt und Lichtquelle befindet, nicht transparent, d.h. opak, ist. Ist das Objekt jedoch transparent, d.h. lichtdurchlässig, wirft es – aus unserer Anschauung begründet – gegenüber eines gleichem opaken Objekts einen helleren Schatten. Die Helligkeit jenes Schattens wird dabei durch den Grand der Transparenz bestimmt. Auch dieser Effekt kommt in unserem RayTracer Berücksichtigung zu.

Als Farbwert für ein Bildpunkt ergibt sich zusammenfassend nun folgendes:

Farbe = Farbwert, berechnet durch das Phong–Beleuchtungsmodell
+ Farbwert, gewonnen durch Reflektion
+ Farbwert, ermittelt durch Refraktion

Die letzten beiden Summanden beinhalten, wie gesagt, eine Rekursion, wodurch sich der hohe Rechenaufwand des RayTracings erklärt.

Anwendung

Anwendung findet das RayTracing bei der Erzeugung hochwertiger Bilder. Dass auch die Filmindustrie dieses Verfahren nutzt, zeigt die nachstehende Abbildung.

Abbildung 18: Ein durch RayTracing berechneter, wohl bekannter Trickfilm

3. Anforderungsspezifikation

Ziel ist es, ein System zu erstellen, dass ein Bild erzeugt. Der Inhalt des Bildes liegt in Form einer Szenenbeschreibung vor. Aus dieser Beschreibung, die der Anwender dem System übergibt, ist das Bild zu berechnen. Der Anwender soll dabei die Möglichkeit besitzen, Einfluss auf die Art der Erzeugung zu nehmen. Er kann dazu zwischen zwei Arten der Bilderzeugung (Rendering) wählen. Auf der einen Seite ist es das Rendering mittls OpenGL, das eine schnelle Vorschau zu liefern in der Lage ist, in der Qualität aber nicht an die eines RayTracers heranreicht. Auf der anderen Seite steht die Wahl des RayTracers, dessen Ergebnis ein qualitativ hochwertiges Bild ist; er dafür aber entsprechend mehr Zeit benötigt. Wie auch immer das Bild zustande kam, der Benutzer soll dieses mit Hilfe eines „Viewers“ betrachten können.

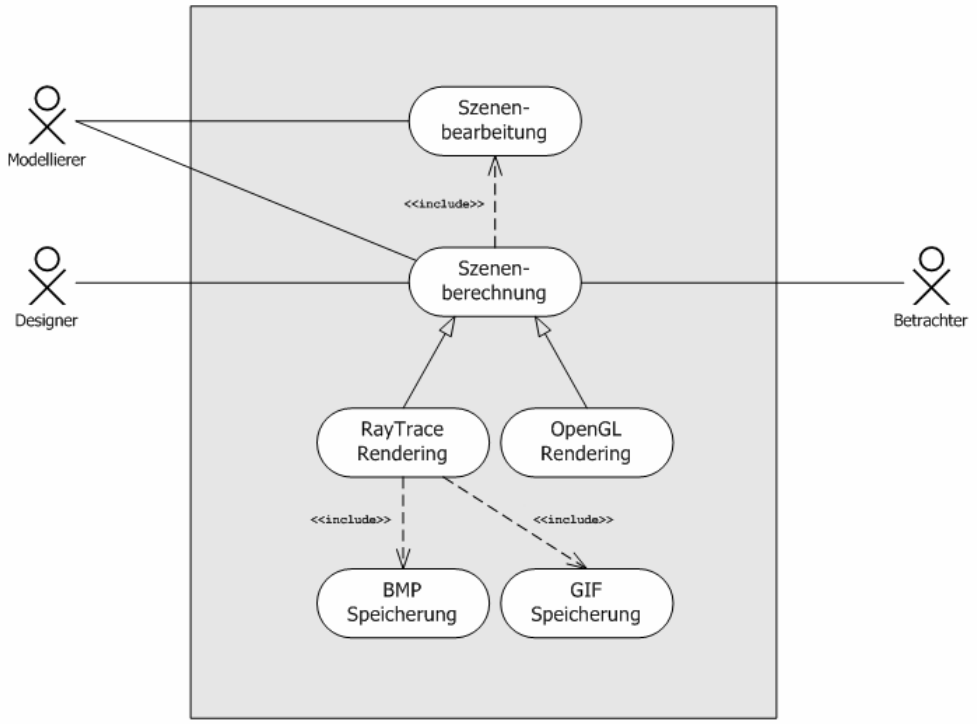


Abbildung 19: Anwenungsfalldiagramm

Abbildung 19 zeigt das zugehörige Anwendungsfalldiagramm. Der Modellierer / Designer stellt die Szene bereit und übergibt sie der Renderingpipeline. Des Weiteren hat der Akteur noch die Möglichkeit, zwischen den Methoden der Bilderzeugung – Anwendungsfall: RayTracer oder OpenGL – zu wählen. Zudem ist die Art, in der das durch den RayTracer erzeugte Bild gespeichert werden kann, nicht fest vorgegeben. Die Abbildung 19 zeigt dazu zwei Möglichkeiten, deren Anzahl damit noch nicht vollständig ist.

Der Anwendungsfall: OpenGL–Rendering spielt für das Projekt eine untergeordnete Rolle und dient lediglich als schnelle Vorschaufunktion. Somit liegt der Schwerpunkt auf der im Kapitel 2 vorgestellten RayTracing–Pipeline. Wie dort beschrieben, erfordert das Berechnen einer Szene viel Rechenzeit und somit liegt es nahe, die Aufgabe aufzuteilen und damit zu beschleunigen.

Ziel dieses Projektes ist es daher, einen verteilten RayTracer zu programmieren. Wir haben uns dazu folgende Spezifikation überlegt.

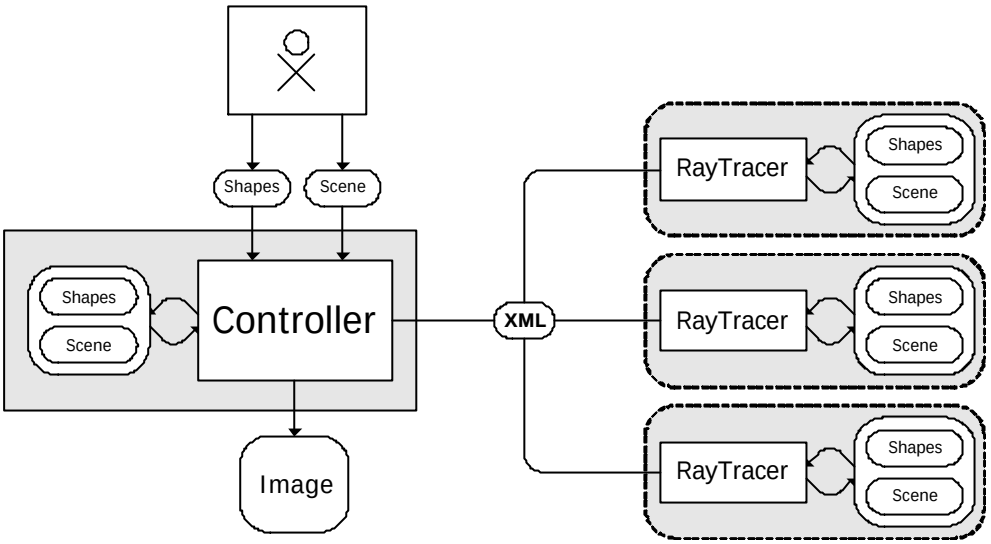


Abbildung 20: Grundlegender Aufbau des RayTracers

Abbildung 20 zeigt den generellen Aufbau unseres RayTracers.

Der **Benutzer** beschreibt die Szenen, die zur Bilderzeugung erforderlich sind, mittels eines Editors oder einer dem Editor verständlichen Sprache, beispielsweise XML. Die Szenenbeschreibung wird dem Controller übergeben. Eine weitere Aufgabe des Benutzers liegt darin, über das Wissen der Spezifikationen aller in der Szene vorkommenden Figuren, wie Box, Kugel oder Dreieck, zu verfügen und sie dem Controller ebenfalls zugänglich zu machen.

Der **Controller** kapselt die Szene und die Spezifikationen der in ihr enthaltenden Figuren. Dies tut er, indem er sie serialisiert. Das „serialisierte Paket“ übergibt er anschließend den einzelnen Workern (den RayTracern), zusammen mit einer Bereichsangabe des Bildes, den sie zu berechnen haben. Die Worker liefern daraufhin für die ihnen zugeteilten Bereichen die Farbwerte an den Controller zurück, der daraus das Gesamtbild generiert und in einem gegebenen Bildformat speichert.

Die **Worker** selbst sind als Web Services (vgl. Kapitel 6) implementiert und können daher auf verschiedenen Rechnern arbeiten. Sie erhalten vom Controller die im XML–Format kodierte Szenenbeschreibung, die sie sofort deserialisieren. Aus den enthaltenden Spezifikationen erzeugen die Worker auf das RayTracing spezialiserte Figuren.

Beim RayTracing reicht es nicht aus, alleinig zu definieren, dass ein Dreieck aus drei voneinander verschiedenen Punkten besteht; es muss auch eine Funktion geben, die die Intersektion zwischen Strahl und dem Dreieck bestimmt. Der Controller liefert dem entgegen nur Informationen über das grundlegenden Dreieck, d.h. nur die drei Punkte im Raum. D.h. der Rest muss den Workern aus der Szenenbeschreibung bekannt gemacht werden.

Daher enthält sie Informationen über:

- Position der Objekte
- Materialbeschaffenheit der Objekte, wie Reflektionseigenschaften oder Transparenz
- Beschreibung der Lichtquellen (Anzahl, Position, Farbe usw.)

Als Ergebnis erhält der Nutzer das fertige Bild in einer Datei, die er sich mittels eines geeigneten „Viewers“ betrachten kann.

4. Architektur

Der zentrale Aspekt beim Entwurf der Architektur ist, die Möglichkeit einzuräumen, dass der Benutzer eine Szene sowohl über RayTracing als auch mittels OpenGL darstellen lassen kann. Alle Daten– und Klassenstrukturen sollen dem Ansatz folgend vollkommen unabhängig davon sein, welche Darstellungstechnik der Anwender gerade gewählt hat.

4.1. Darstellungsunabhängige Klassen

Geometrische Objekte

Jede Szene besteht aus geometrischen Objekten, Lichtquellen und einer Kamera. Das Programm kann in seiner ursprünglichen Version drei Kategorien geometrischer Objekte unterscheiden:

- Kugeln (Klasse: `Sphere`) besitzen einen räumlichen Mittelpunkt und einen Radius
- Dreiecke (Klasse: `Triangle`) bestehen aus drei voneinander verschiedenen Eckpunkten
- Boxen (Klasse: `Box`): sind achsenparallel; spezifiziert werden der vordere linke untere Eckpunkt und der hintere rechte obere Eckpunkt

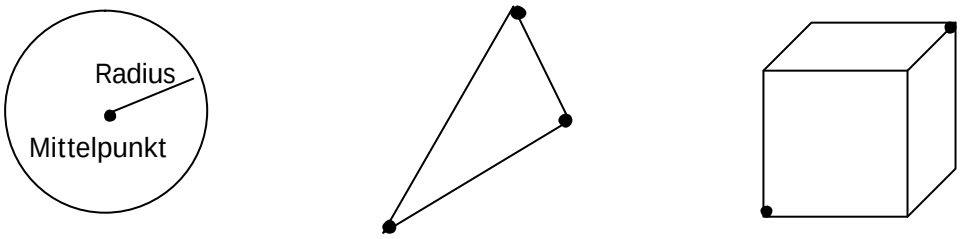


Abbildung 21: Spezifikation einer Kugel (links), eines Dreiecks (mittig) und einer Box (rechts)

Alle geometrischen Objekte erben von der Basisklasse `Shape`. Sie ist komplett leer; d.h. sie weist weder Methoden noch Attribute auf. Ihre Aufgabe ist es lediglich, eine Beziehung auf logischer Ebene herzustellen. Im Code ist man so in der Lage, jede Kugel, jedes Dreieck und jede Box einfach als ein `Shape` anzusprechen. In vereinfachter UML ausgedrückt, sieht dies wie folgt aus:

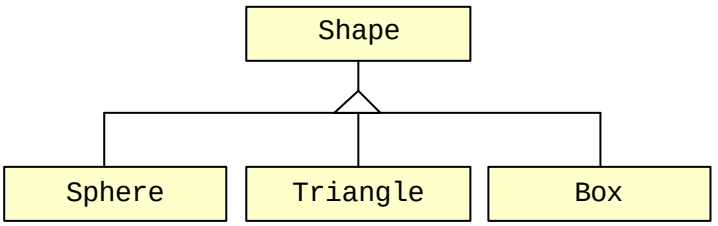


Abbildung 22: UML–Klassendiagramm geometrischer Objekte

Material

Die Beschreibung der Oberfläche eines geometrischen Objektes übernimmt die Klasse `Material`. Es kommen in ihr die ambienten, diffusen und spekularen Reflektionseigenschaften genauso zum Zuge, wie die Farbe, die Transparenz und der Brechungsindex.

Lichtquellen

Hinsichtlich der Lichtquellen gibt es keine weiteren Differenzierungen; sie alle sind Objekte der Klasse `Light` und werden in der Computergrafik als Punktlichtquellen aufgefasst. Ihre Parameter sind die Position im Raum, die Leuchtintensität von 0 (aus) bis 1 (maximale Leuchtstärke) und eine ambiente Grundhelligkeit (ebenfalls im Intervall von 0 bis 1).

Abbildung 23: Schwache Lichtquelle (Intensität 0,5)

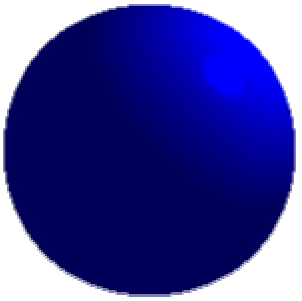


Abbildung 24: Starke Lichtquelle (Intensität 0,9)

Kamera

Es darf genau eine Kamera existieren. Sie entstammt der Klasse `Camera` und legt die Projektion der dreidimensionalen Szene auf den zweidimensionalen Bildschirm fest. Wir haben uns aus Gründen der Einfachheit für die Zentralprojektion entschieden. Einfluss haben der Kamerastandpunkt, die Blickrichtung, die Rotation um die Blickachse und das Verhältnis zwischen Höhe und Breite.

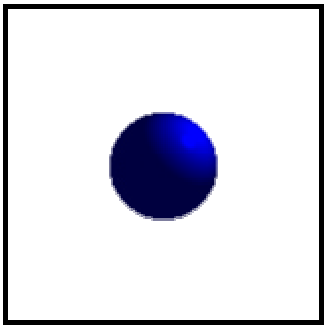


Abbildung 25: Große Entfernung zwischen Kamera und Szene

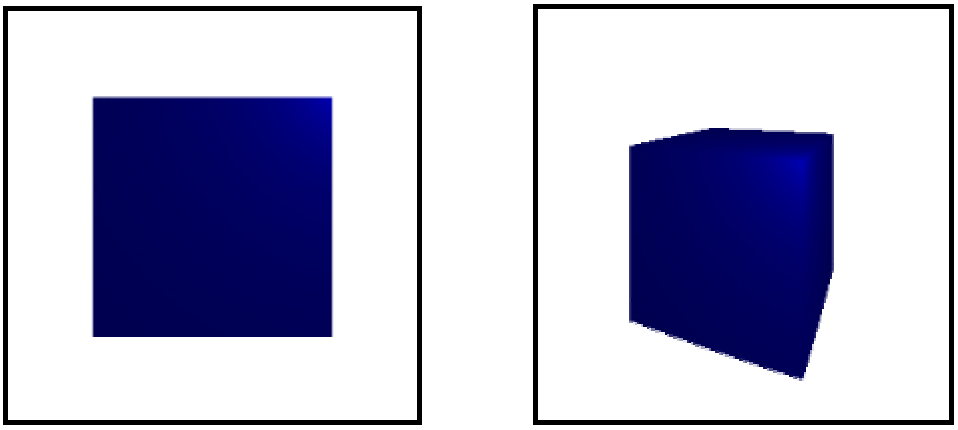


Abbildung 26: Frontale Ansicht einer Box (links) bzw. bei verschobener Kamera nach oben rechts (rechtes Bild)

Szenenbeschreibung

Die Szene als Ganzen fasst die Klasse `SceneDescription` zusammen. Sie stellt damit eine vollständige Beschreibung aller Eigenschaften des späteren Bildes dar. Im Prinzip dient sie nur der Datenhaltung. Darunter zählt natürlich auch die Persistenz.

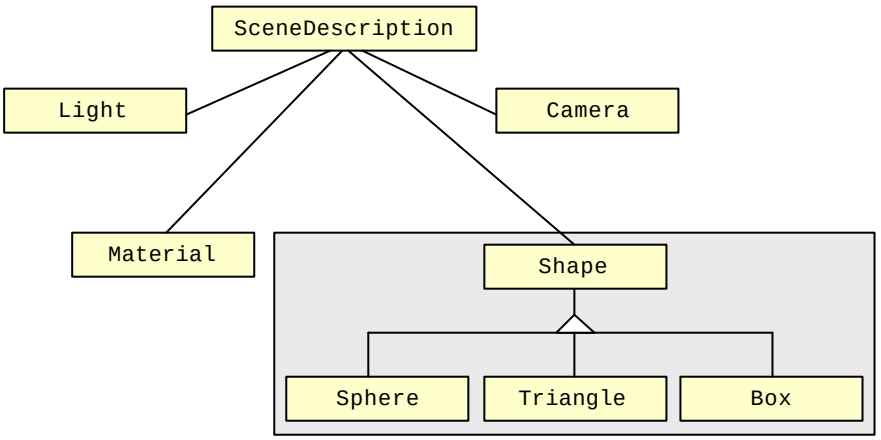


Abbildung 27: UML–Klassendiagramm von `SceneDescription`

4.2. Spezialisierte Klassen zur Darstellungsberechnung

Für die spätere Darstellung verfügt jedes geometrische Objekt über eine spezielle Methode, die es alleinig zur Aufgabe hat, die dazu benötigten Informationen bereitzustellen. Die Methode bezüglich OpenGL folgt dazu verständlicherweise einem komplett anderen Ansatz, als die für den Einsatz des RayTracing.
In der OpenGL–Fassung ist `DrawShape()` die ausschlaggebende Methode. Sie ist in der Lage, das komplette Objekt zu zeichnen, während die RayTracing–Version über die Methode `Intersect()` verfügt, welche den Schnittpunkt zwischen Strahl und Szenenobjekt berechnet.

In unserer Lösung sind zwei Interfaces definiert: `IRayTracer` und `IOpenGL`. Um ein geometrisches Objekt in OpenGL darstellen zu können, muss es von der Klasse `Shape` ableiten und zusätzlich das spezifische Interface `IOpenGL` implementieren. Ähnliches gilt es dann für das RayTracing.

Für ein Dreieck entsteht in der UML–Notation dann das folgende Klassendiagramm:

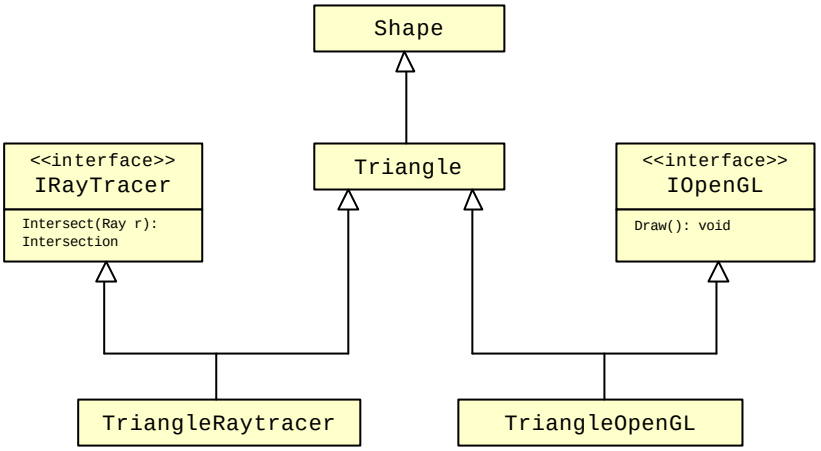


Abbildung 28: UML–Klassendiagramm spezialisierter Dreiecke (Triangles)

Um es noch einmal zu wiederholen: In der Klasse `SceneDescription` tauchen die spezialisierten Klassen nicht auf. Sie werden vielmehr erst zu dem Zeitpunkt erzeugt, wenn es tatsächlich um die Darstellung geht. In der Modellierungsphase einer Szene spielen sie eh kaum eine Rolle und müssen daher auch nicht vorliegen; erst in der finalen Berechnung des Bildes stellt sich ein Gebrauch ein.

Allerdings bietet die Klasse `SceneDescription` die Möglichkeit, aus den „Basisobjekten“ ihre spezialisierten Gegenstücke zu erzeugen. Die Methode `Downcast()` nutzt dazu Reflektion, um den aktuellen Klassennamen eines Shape–Objektes auszulesen. Auf rein textueller Basis wird dieser dann dazu benutzt, mittels eines Activators das neue Objekt zu generieren. Über Exception–Handlung wird zudem kontrolliert, ob der Vorgang des Erzeugens auch fehlerfrei vonstatten ging.

```
protected Shape Downcast(Shape shape, string strShapeNamespace, string strShapeSuffix)
{
    string strClassname = shape.GetType().ToString();

    if (strClassname.IndexOf(strShapeNamespace) != -1)
        return shape;

    strClassname = strClassname.Replace(strSceneNamespace+ ".", strShapeNamespace+ ".") +
                        strShapeSuffix;

    string strAssembly = strShapeNamespace + ".dll";

    try
    {
        Type newType = Type.GetType(strClassname);

        Object[] args = new Object[1];
        args[0] = shape;

        Object castedShape = (Activator.CreateInstanceFrom(strAssembly, strClassname, true, 0,
                                null, args, null, null, null)).Unwrap();

        return (Shape) castedShape;
    }
```

```
    catch (Exception e)
    {
        System.Console.WriteLine("Unknown class {0} - {1}.", strClassname, e.ToString());

        return shape;
    }
}

public void DowncastAllShapes(string strShapeNamespace, string strShapeSuffix)
{
    for (int i = 0; i < NumberOfShapes; i++)
        m_Shape[i] = Downcast(m_Shape[i], strShapeNamespace, strShapeSuffix);
}
```

Die Idee, dass der neue Klassenname textuell erzeugt wird, bietet eine ungeheure Flexibilität in Hinblick auf zukünftige Erweiterungen: Sollten einmal neue geometrische Objekte vorhanden sein, zum Beispiel ein Zylinder, so muss die Klasse `SceneDescription` nicht angepasst werden. Es ist lediglich darauf zu achten, dass die Namenskonvention eingehalten wird: Jede das Interface `IRaytracer` implementierende Klasse sollte auf „Raytracer“ enden, jede `IOpenGL` implementierende Klasse dementsprechend auf „OpenGL“.
Für ein Dreieck beispielsweise bedeutet dies, dass von der Basisklasse `Triangle` ausgehend die spezialisierten Klassen `TriangleRaytracer` und `TriangleOpenGL` heißen.

In der Sprache der Design Patterns stellt die Methode `Downcast()` eine Class Factory dar. Einem weiteren Design Pattern kommt ebenso Bedeutung zu: der Decorator. Die geometrischen Objekte werden um weitere Fähigkeiten, die der eigenen Darstellung nämlich, ergänzt.

Persistenzhaltung

Die Persistenzhaltung gehört ebenfalls zu dem Aufgabenbereich der Klasse `SceneDescription`. Sie ist in der Lage, ihren kompletten Zustand zu serialisieren und anschließend wieder zu deserialisieren. Unsere bisherigen Erfahrungen mit verschiedenen Frameworks zeigten, dass dies eine nicht immer triviale Aufgabe ist, die oft unverhältnismäßig viel Code erfordert. Gerade das .NET–Framework spielt in diesem Bereich jedoch seine Stärken aus, da es eine umfassende XML–Unterstützung mitbringt.

Da es also sehr einfach ist, unter .NET eine beliebige Klasse als „serialisierbar“ zu kennzeichnen, bieten die von uns programmierten Szenenobjekte dies gleich in zweierlei Hinsicht an: Sie können sowohl im SOAP–Format als auch im einfachen XML–Format serialisiert werden. Die dafür notwendigen Quelltextänderungen sind minimal: Für SOAP ist das Attribut `[Serializable]` zuständig. XML dagegen kann über Reflektion alle Typinformationen selbstständig ermitteln, beachtet dabei aber nur die als `public` deklarierten Variablen bzw. Properties.

Die Klasse `Triangle` ist dementsprechend implementiert. (Die für die Serialisierung verantwortlichen Anweisungen sind gelb unterlegt)

```
namespace Scene
{
    using Misc;
    using System;

    [Serializable]
    public class Triangle : Shape
    {
        public Vector[] Vertices = new Vector[3];
        public Vector Normal;

        public Triangle() : this(new Vector(), new Vector(), new Vector()) {}

        public Triangle(Vector a, Vector b, Vector c)
        {
            Vertices[0] = a;
            Vertices[1] = b;
            Vertices[2] = c;

            Normal = Vector.ClockwiseNormal(Vertices[0], Vertices[1], Vertices[2]);
        }
    }
}
```

Weil jede der bisher genannten Klassen nun potentiell serialisierbar ist, kann der eigentliche Vorgang dazu sehr kurz gehalten werden. Der `XmlSerializer` hilft hier entsprechend:

```
try
{
    Stream stream     = File.Open(strFilename, FileMode.Create);
    XmlSerializer xml = new XmlSerializer(typeof(SceneDescription));

    xml.Serialize(stream, this);
    stream.Close();
}
catch (Exception e)
{
    System.Console.WriteLine("XML-Serialize says NO!: {0}", e.ToString());
}
```

Das Resultat ist eine XML–Datei, die kompakt die Szene beschreibt und zugleich intuitiv verständlich ist. Es ist daher durchaus möglich, sie von Hand zu schreiben; wir empfehlen aber die Verwendung des Editors (nächster Abschnitt), da er Effizienz mit Komfort verbindet.

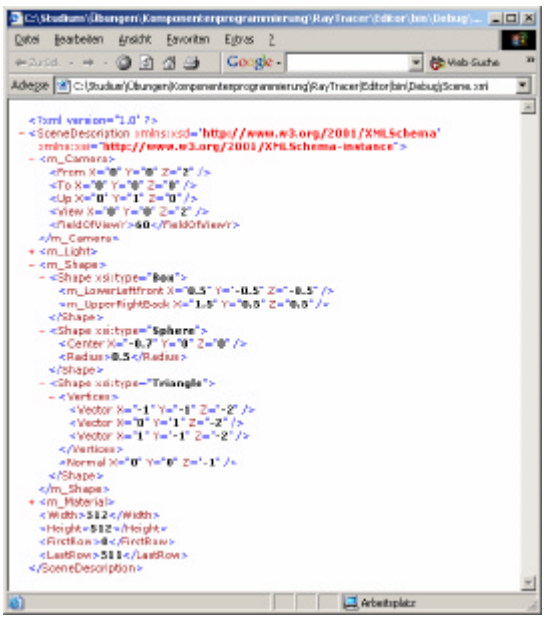


Abbildung 29: Auszug aus einer Szenenbeschreibung

Alternativ ist, wie gesagt, SOAP möglich. Allerdings wird diese Fähigkeit von unserem Programm nicht verwendet, obwohl es anfangs noch so war. Der Grund dafür liegt darin, dass das Resultat einer Serialisierung mittels SOAP eine ziemlich aufgeblähte Datei ist, die nur noch schwer von Menschen verstanden werden kann. Für Testzwecke kam der Code dennoch zum Zuge, so dass uns auch dieses Dateiformat vertraut wurde.
Hinsichtlich der Kommunikation mit Web Services wird dennoch SOAP verwendet, wenn auch nur indirekt. Allerdings kommt dabei nicht unsere selbstgeschriebene Serialisierungslogik zum Einsatz, sondern die die das .NET–Framework für uns automatisch generiert hat.

```
if (bUseSOAP)
{
    Stream stream        = File.Open(strFilename, FileMode.Create);
    IFormatter formatter = new SoapFormatter();

    formatter.Serialize(stream, this);
    stream.Close();
}
...
```

Man kann den gesamten Mechanismus auch als das Interpreter Design Pattern deuten.

4.3. GUI

Das MVC – Paradigma

Für die Benutzeroberfläche ist der Einsatz des Model–View–Controller–Paradigmas (MVC–Paradigma) ideal. Das Model, d.h. die Datenhaltung wird durch die Klasse `SceneDescription` repräsentiert. Sie ist vollkommen unabhängig vom Controller, dem der Editor am nahesten kommt. Dieser beinhaltet äußerst wenige Bestandteile des Views, denn die eigentliche Darstellung erfolgt durch separate Klassen, die das Interface `IPreView` implementieren. In unserem Programm existieren davon gleich zwei: `RenderOpenGL` und `RenderRaytracer`. Beide kann man im Editor dynamisch zur Laufzeit laden.

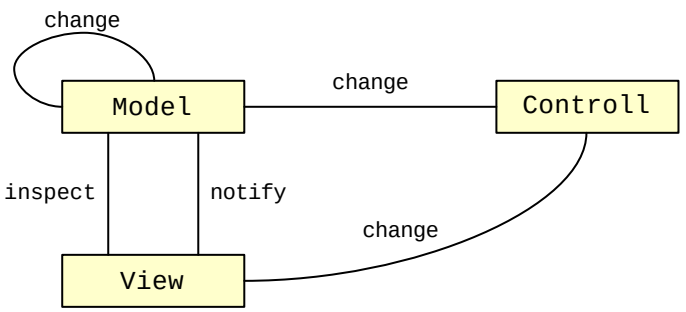


Abbildung 30: Beziehungen im MVC–Paradigma

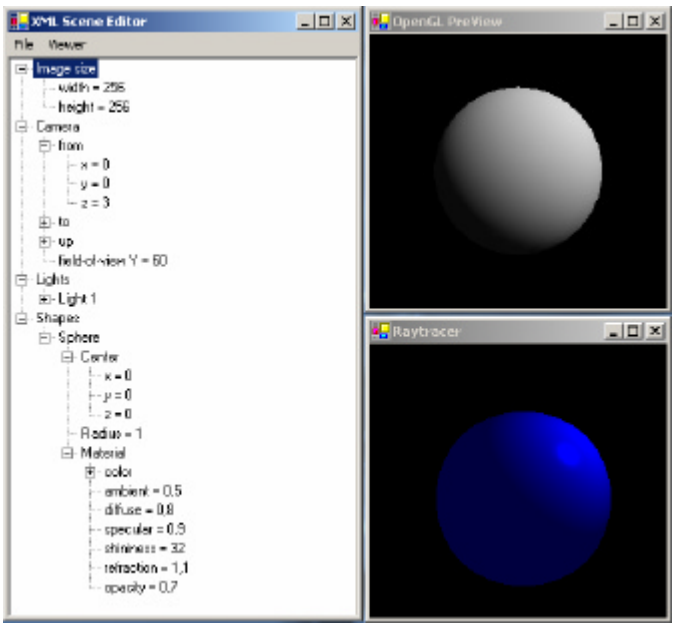


Abbildung 31: Zwei unterschiedliche Views im Einsatz

Das Einbinden erfolgt über Reflektion. Der Benutzer wählt zunächst die zu ladende Assembly (meist eine `DLL`). Aus dieser gewinnt das Programm alle exportierten Klassen. Wenn eine davon das `IPreView` – Interface implementiert, wird von dieser ein neues Objekt instanziiert. Es erhält die Szenenbeschreibung und kann dann das entsprechende Bild rendern.

Es können beliebig viele Views erzeugt werden, auch mehrere von derselben Klasse sind möglich. Sobald im Editor die Szene seitens des Benutzers modifiziert wird, zum Beispiel weil eine neue Lichtquelle hinzukommt, erhalten alle existierenden Views eine Mitteilung darüber sowie die komplett neue Szene.

```
//  load from assembly
IPreView viewer = null;

try
{
    Assembly asm = Assembly.LoadFrom(strComponent);

    Type[] asmTypes = asm.GetTypes();

    foreach(Type asmType in asmTypes)
    {
        if(asmType.GetInterface("IPreView") != null)
            viewer = (IPreView)(Activator.CreateInstanceFrom(strComponent,
                        asmType.FullName)).Unwrap();
    }
}
catch (Exception e)
{
    MessageBox.Show(e.ToString(), "Exception occured");

    return;
}
...

//  send scene to preview window
scene = MyTree.GetScene();
viewer.Render(scene);
```

Der Editor

Der Editor selbst besteht hauptsächlich aus einem TreeView–Darstellungsbereich. Wir vertreten die Auffassung, dass dieses Control die am besten geeignete Darstellung von hierarchischen Datenstrukturen, wie etwa einer Szene, darstellt. Alle Elemente der Szene zeigen sich so zusammen mit all ihren Eigenschaften, die zudem vom Benutzer einfach und zielgerichtet verändert werden können.

Die objektorientierte Programmierung erlaubt hier einen außergewöhnlich eleganten Ansatz: Gewisse Einträge im TreeView, wie zum Beispiel skalare Werte oder Vektoren, sollen direkt manipulierbar sein. Aus diesem Grund heraus müssen sie entsprechend auf Ereignisse reagieren können. Zu diesem Zweck existieren die Klassen `TreeNodeDouble` und `TreeNodeVector`, die direkt von `TreeNode` ableiten. Sie sorgen für die Datenhaltung eines `double` bzw. `Vector` und steuern die Anzeige im TreeView. Somit stehen die Klassen in folgender Beziehung:

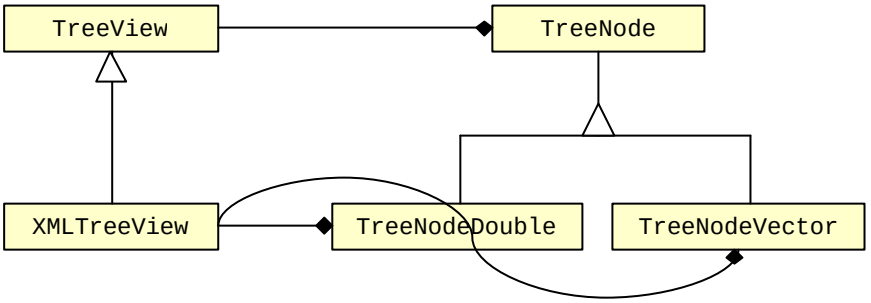


Abbildung 32: Klassendiagramm Editor

Sobald ein Eintrag im `XMLTreeView` angeklickt wird (`OnSelect()`), überprüft das Programm zur Laufzeit den Typ des jeweiligen `TreeNode`. Handelt es sich dabei um ein `TreeNodeDouble`, so kann mit Hilfe des `EditDoubleDialog` der Wert verändert werden, bei einem `TreeNodeVector` wird dementsprechend `EditVectorDialog` gerufen.

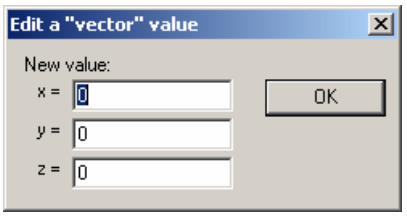


Abbildung 33: Ein `EditVectorDialog` in Aktion

Die beiden Dialoge besitzen die Aufgabe, eine vom Typ abhängende Editierung zu erlauben. Sie nehmen aber keinerlei Gültigkeitsüberprüfungen vor.
Da ihr Aufbau dermaßen einfach ist, stammen sie direkt aus dem GUI–Designer der Visual Studio .NET Entwicklungsumgebung. Keine einzige ihrer Zeilen wurde daher explizit programmiert; alles entstammt einer automatischen Erzeugung.

Eine aufwändige Arbeit ist aber das Einlesen einer Szene. Zwar ist es kein Problem, sie aus einer XML–Datei in ein Objekt der Klasse `SceneDescription` zu überführen, aber die anschließende Transformation in ein `XMLTreeView` erwies sich dagegen als ein nicht unproblematisches Unterfangen.

Als Lösungsidee kam uns erneut die Reflektion von C# und .NET zur Hilfe. Unter Benutzung der erwähnten spezialisierten `TreeNode` – Klassen ist es möglich, dass alle Informationen einer `SceneDescription` zu 100% in ein `XMLTreeView` abbildbar sind. Dessen Methode `SetScene()` kann sehr einfach die Beleuchtung und die Kamera platzieren. Schwieriger wird es bei den geometrischen Objekte: es ist nun mal nicht im Vorfeld bekannt, welche Typen eine Szene genau beinhaltet. Ebenso soll der Editor auch mit neuen Objekten umgehen können, die bis jetzt dem Projekt überhaupt noch nicht bekannt sind (wie zum Beispiel ein Zylinder).

Glücklicherweise bietet die .NET–Klasse `FieldInfo` einen Lösungsansatz. Sie setzt beim Auslesen aller Attribute der geometrischen Objekte an und erlaubt es, für `double`– bzw. `Vector` – Attribute die entsprechenden `TreeNodes` spezifisch zu erzeugen.

Einen ähnlichen Weg geht auch `GetScene()`. Die umgekehrte Aufgabe, die Transformation eines `XMLTreeView` in eine `SceneDescription`, erkennt über Reflektion den Typ des jeweiligen `TreeNodes` und kann dann wieder die korrekten Objekte erzeugen.

Als kleinen Gimmick ist über das Kontextmenü das Hinzufügen neuer Objekte bzw. Lichtquellen möglich. Da dieses Feature erst in einer sehr späten Projektphase hinzukam, wurde das Löschen leider nicht mehr umgesetzt.

5. OpenGL

5.1. Das Konzept

Das RayTracing ist eine so genannte Point–Sampling–Technik, die zum Ausdruck bringt, dass an einzelnen Punkten im Raum nach speziellen Verfahren Stichproben genommen werden, mit deren Hilfe man versucht, das Gesamterscheinungsbild ableitet. In unserem RayTracer wird die Stichprobe für ein Pixel durch den Strahl bestimmt, der vom Betrachter aus durch die Mitte desselben verläuft.

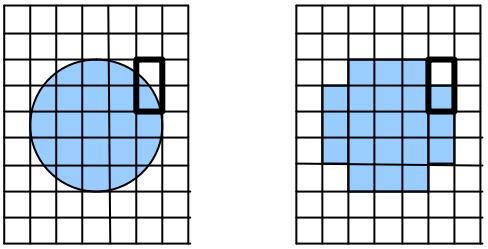


Abbildung 34: Point–Sampling einer Kugel

Für die Stichprobe kann man exakt die Umgebungseinflüsse, wie Beleuchtung oder Spiegelung berechnen. Allerdings muss die Stichprobe nicht repräsentativ sein. In der obigen Abbildung werden dazu zwei Pixel exemplarisch betrachtet. Das Zentrum des oberen wird nicht von der Kugel überdeckt, er bleibt daher weiß. Der untere wiederum erhält die blaue Farbe, da dessen Zentrum sehr wohl von der Kugel getroffen wird.
Aber genau genommen, sind beide Pixel falsch: Sie müssten eine Mischung aus blau und weiß sein, wobei natürlich der obere heller als der untere wäre.

Wenn man aber die Anzahl der Stichproben pro Pixel erhöht, erhöht sich auch die Genauigkeit. Leider steigt dann auch der Rechenaufwand linear an. In der Praxis bewährt haben sich 9 bis 16 Stichproben pro Pixel, was mitunter zu mehreren Stunden Rechenzeit pro Bild führen kann, so dass wir in unserem RayTracer gerne darauf verzichten.

5.2. Geometriebasierter Ansatz

Interaktive Systeme beruhen meist auf einem geometriebasierten Ansatz. Sie zeichnen zuerst alle Objekte und prüfen dann, welche sichtbar sind und welche nicht. Beim RayTracing geht man genau den umgekehrten Weg.

Etablierte Schnittstellen, wie OpenGL und Direct3D, verfügen über Algorithmen, um primitive Elemente zeichnen zu können. Dazu zählen Punkte, Linien und Dreiecke. Darauf aufbauend können weitere, komplexere Bildelemente erzeugt werden. Da die grundlegenden Primitive einfach in Hardware umzusetzen sind, stellen geometriebasierte Ansätze derzeit die einzige Möglichkeit dar, interaktive Systeme zu realisieren.

5.3. Was ist OpenGL?

OpenGL, eine Abkürzung für Open Graphics Language, ist das weltweit am häufigsten benutzte Grafikinterface, wenn es um portable und interaktive 2D– bzw. 3D–Grafik geht. Auf nahezu allen Betriebssystemen (MS Windows, Linux, BeOS, etc.) existiert eine OpenGL–Implementierung.

Silicon Graphics entwarf diesen Standard 1992 unter dem Gesichtspunkt der Performance bei gleichzeitig hoher visueller Qualität und Genauigkeit. Diese Ansprüche konnten zu einem hohen Maß erfüllt werden, denn OpenGL kommt in vielen Bereichen zum Zuge: von CAD/CAM über wissenschaftliche Visualisierung, die Generierung von Animationen, bis hin zu Spielen setzen nahezu alle marktführenden Anwendungen im Grafiksektor auf diesen Industriestandard.

Silicon Graphics rief das OpenGL Architecture Review Board ins Leben, um eine verbindliche Spezifikation für OpenGL zu erstellen. Dieser gehören etliche große Firmen aus dem Hard– und Softwarebereich an. In der Folge existiert eine solide Dokumentation und eine Vielzahl guter Sekundärliteratur, was uns auch beim Einstieg in die Materie entscheidend half.

Um die Portabilität im gewünschten Ausmaß erreichen zu können, wählte man die Programmiersprache C. Mittlerweile entstanden Anbindungen an Java, Fortran und viele mehr. Auf moderne Entwicklungen, wie objektorientierte oder komponentenbasierte Programmierung, verzichtete man bewusst.
Da selbst eine GUI–Verwaltung fehlt, integrieren (nahezu) standardisierte Zusatzbibliotheken wie die GLU (Graphics Language Utilities) und GLUT (GLU Toolkit) diese Schnittstellen.

Der interne Aufbau von OpenGL entspricht einer Pipeline.

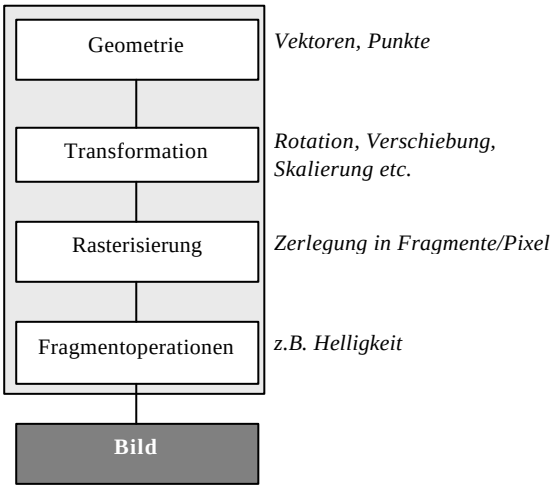


Abbildung 35: Aufbau der OpenGL–Pipeline

Es muss stets der gesamte Funktionsumfang von OpenGL in jeder Implementierung gewährleistet sein. Heutzutage wird ein großer Teil der Pipeline in Hardware verarbeitet. Alles, was nicht durch Hardware direkt darstellbar ist, muss dann in Software umgesetzt werden. Dies geht dann zwar stark zu Lasten der Performance, sichert aber die Konformität zum Standard ab.

Grundsätzlich ähnelt Direct3D in sehr vielen Konzepten OpenGL. Mehrere Programme, wie zum Beispiel die Unreal–Engine, verfügen daher über die Möglichkeit, die Bildschirmausgabe zur Laufzeit zwischen beiden Systemen umzuschalten. Leider verhindert die mangelnde Portabilität eine Verbreitung von Direct3D über die Grenzen von Windows hinaus.

5.4. Einsatz

In der Win32–API ist OpenGL fest integriert. Leider hat Microsoft es versäumt, auch im .NET–Framework eine Anbindung vorzusehen. Wir entschieden uns daher, die Open–Source–Lösung Namens CsGL (`http://csgl.sourceforge.net`) zu verwenden, die im Prinzip lediglich einen C# – Wrapper um OpenGL darstellt.

Dieser ist aber noch nicht zu 100% fertig gestellt und weist momentan einige Schwächen, insbesondere bei der Verwendung von Farben und Beleuchtung auf; alle Objekte zeigen sich lediglich in Graustufen. Für unser Projekt ist dieser Mangel dennoch vertretbar.

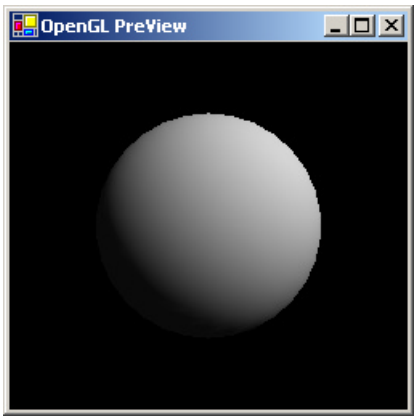


Abbildung 36: OpenGL–Darstellung einer Kugel

Die Syntax entspricht weitgehend OpenGL unter C. Ein kleiner Unterschied besteht dennoch darin, dass die C# – Klasse unbedingt von `OpenGLControl` erben muss. Ebenso sind alle OpenGL–Funktionen als statische Methoden der Klasse GL implementiert.

Ein Dreieck kann sehr einfach dargestellt werden:

```
public void DrawShape()
{
    GL.glPushMatrix();

    //  define shape
    GL.glBegin(GL.GL_TRIANGLES);
        //  set normal and vertices
        GL.glNormal3d(Normal.X, Normal.Y, Normal.Z);

        for(int i = 0; i < 3; i++)
            GL.glVertex3d(Vertices[i].X, Vertices[i].Y, Vertices[i].Z);
    GL.glEnd();

    GL.glPopMatrix();

    //  finish
    GL.glFinish();
}
```

Die oben dargestellte Kugel ist in OpenGL nicht unmittelbar verfügbar. Da die Darstellung stets auf einem diskreten Raster – meist der Bildschirm – erfolgt, kann man sich einer Kugel annähern: Man zerlegt sie in viele Dreiecke, wobei die Anzahl der Dreiecke über die Detailgenauigkeit entscheidet. Sobald die Dreiecke höchstens ein halbes Pixel überdecken (Nyquist–Rate), ist das Ergebnis visuell nicht mehr von einer echten Kugel zu unterscheiden. Eine Box wird ebenso anhand von Dreiecke simuliert (6 Seiten mit je 2 Dreiecken). Das alles geht natürlich zu Lasten der Geschwindigkeit, aber moderne Grafikkarten können bereits über 100 Millionen Dreiecke pro Sekunde zeichnen.

6. Web Services

6.1. Einleitung

Laut Firmen wie Microsoft oder IBM stellen Web Services die große Weiterentwicklung für Internetapplikationen dar, und werden zukünftig einen Standard zur Informationsverarbeitung via Internet bieten. Web Services basieren dabei – anders als bisherige Technologien – auf einem einheitlichen Übertragungsprotokoll, welches sowohl die Veröffentlichung der eigenen Applikationslogik, als auch die Anbindung weiterer Internetapplikationen sehr vereinfacht.

Eine technische Definition des Begriffes „Web Service“ geben die Autoren von „Professional ASP.NET“[2], indem sie sagen: Ein Web Service ist nichts anderes als programmierbare Applikationslogik, zugreifbar über standardisierte Internetprotokolle.
Dabei ist es unerheblich, in welcher Art und Weise diese Applikationslogik implementiert ist, solange die XML–Unterstützung gesichert ist. Der Transport der Daten erfolgt dann über das SOAP–Protokoll (Simple Object Access Protocol).

Ziel ist es somit, den Nutzern von Web Services die Kommunikation mit diesen via standardisierter SOAP–Nachrichten zu ermöglichen, ohne ihnen Implementierungsdetails über entsprechend genutzte Plattformen, Objektmodelle oder Programmiersprachen zu geben.

6.2. Probleme traditioneller verteilter Anwendungen

Nicht ohne Grund wird das Konzept der Web Services als ein zukünftiger Standard für Internettechnologien angepriesen. Die bisherigen Lösungsansätze zur Veröffentlichung programmierter Applikationslogik und weiterer Details litten doch unter einigen Schwächen. Web Services sollen versuchen, eben diese Engpässe zu überbrücken.

- Kleinste Änderungen der Applikationslogik können weitereichende Veränderungen der gesamten Implementation des Clients nach sich ziehen, bzw. machen die Veränderung eines möglicherweise vorgegebenen XML–Schemas zu einer undankbaren Aufgabe.
- Wie ist es möglich, vorhandene Services zu finden?
- Oft ist eine ausführliche Beschreibung des Services nötig, um ihn überhaupt benutzbar zu machen. So sind beispielsweise Auskünfte über unterstützte Protokolle, die Art und Weise der Serialisierung, die unterstützten Datentypen oder verlangter Schemata notwendig.
- Viele der herkömmlichen Protokolle wie DCOM oder CORBA nutzen TCP/IP Ports, welche verschlossen sind, oder zumindest den Zugang einschränken. Viele Administratoren benutzen zudem „Firewalls“, um nur die Ports 80 und 443 (HTTP / HTTPS) der Öffentlichkeit freizugeben. Wie baut

[1] Anderson, R.; Francis, B.; Homer, A.; Howard, R.; Sussman, D.; Watson, K.; Professional ASP.NET Wrox Press 2001

man aber dann erfolgreich verteilte Anwendungen, wenn benötigte Ports vorsorglich gesperrt wurden?

6.3. Spezifikationen

Um die bisher beschriebenen Probleme verteilter Anwendungen zu überwinden, arbeiten Firmen wie Microsoft, IBM, Intel, HP und weitere Spezifikationen aus, die sich grob in drei Kategorien unterteilen lassen:

- *Discovery* – Es gibt zwei Spezifikationen, die das Entdecken von Web Services adressieren; zum einen das UDDI (Universal Description, Diccovery and Integration), welches als den Clients bekanntes globales Verzeichnis zum Publizieren und Auffinden öffentlicher Web Services dient und zum anderen das DISCO (von Discovery abgeleitet), als Microsoft–spezifische Technologie zum Auffinden von Web Services. Hier wird eine Liste aller verfügbaren WSDL–Dateien (siehe nächsten Anstrich) veröffentlicht. Auf die Funktionsweise von UDDI wird zu einem späteren Zeitpunkt noch genauer eingegangen werden.
- *Description* – Die WSDL (Web Service Description Language) definiert eine XML–Grammatik zur Beschreibung von Web Services. Sie beinhaltet Informationen über die Zuordnung zu bestimmten Internetadressen (URLs), über Methoden und Eigenschaften des vorliegenden Web Services, über Datentypen und die Bindung an bestimmte Protokolle. WSDL–Beschreibungen enthalten demnach alle relevanten Informationen, um den jeweiligen Web Service zu definieren. Tools benutzen WSDL–Beschreibungen zur Generierung von Stellvertreter–Objekten – so genannten Proxies – welche Clients zur Kommunikation mit dem Web Service nutzen können.
- *Protocol* – ASP.NET unterstützt drei Protokolle: HTTP–GET, HTTP–POST und SOAP. Die beiden Ersteren dienen der Wahrung der Rückwärtskompatibilität zu herkömmlichen Web–Anwendungen. Das Standardprotokoll für Webservices ist aber SOAP. SOAP ist ein auf XML basierendes und auf HTTP aufsetzendes Protokoll für den Austausch strukturierter Daten. Wobei gesagt werden muss, dass HTTP diesbezüglich nicht das einzig denkbare Transportprotokoll darstellt, aber das einzig bisher standardisierte ist. Durch den SOAP–Standard sind definiert:
    - der generelle Aufbau von SOAP–Nachrichten (Envelope, Header und Body),
    - der Mechanismen zum (De–) Serialisieren von abstrakten Daten,
    - RPC–Konventionen,
    - die Bindung an das HTTP.

6.4. Genereller Ablauf der Benutzung eines Web Services

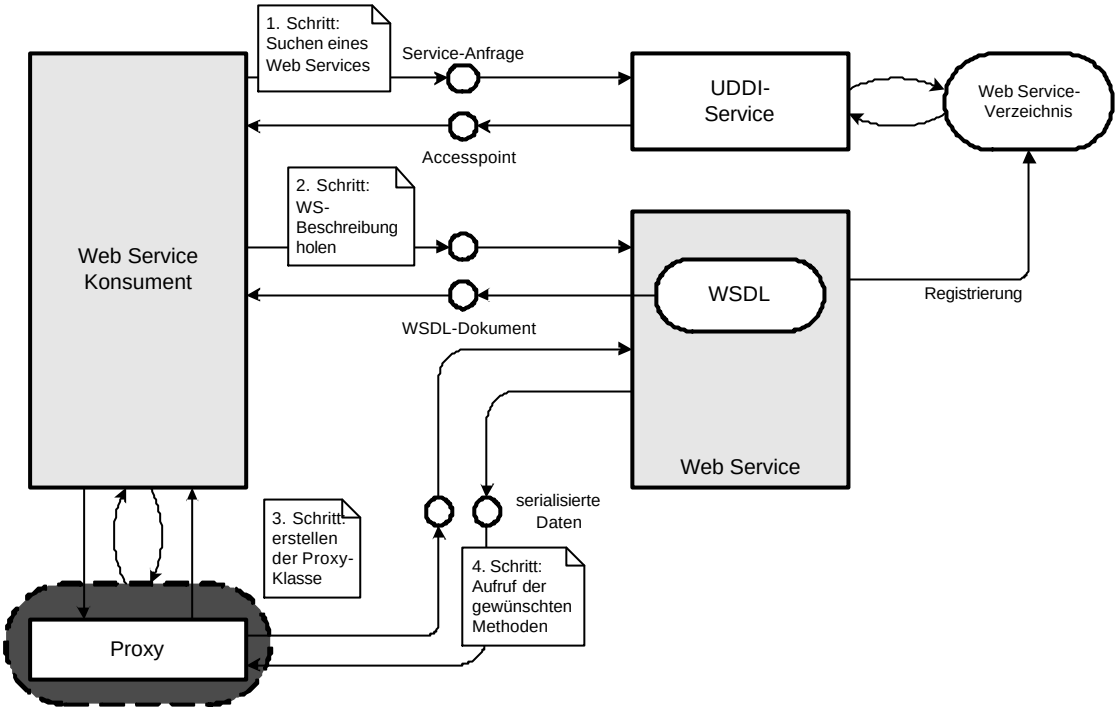


Abbildung 37: Auffinden eines Webservices und Erstellung einer Proxy–Klasse

1. Erfragen eines bestimmten Services unter Nutzung bereitgestellter öffentlicher UDDI Web Services, oder mittels eines Browsers bei einem UDDI Knoten.
2. UDDI antwortet mit einer Liste von erreichbaren (eingetragenen) Services.
3. Die Liste dieser Services beinhaltet entweder URIs auf Basis von DISCO– oder WSDL–Dokumente. In diesem Fall soll angenommen werden, dass auf ein DISCO verwiesen wird.
4. Zusätzlich zu den programmtechnischen Details von UDDI sind auch Informationen zu finden, die der Provider bezüglich seines Services anbietet.
5. Folgt man nun der URI zum DISCO–Dokument, so findet man darin eine Auflistung von „Orten“, an denen sich WSDL–Dokumente befinden.
6. Diese enthalten Informationen, die das Erstellen eines Proxies zu Nutzung des Web Services ermöglichen.
7. Nach dem Parsen des DISCO–Dokuments, kann man schließlich der angegebenen URI zum eigentlichen WSDL–Dokument folgen.
8. Es ist nun möglich, nur mit Hilfe der im WSDL–Dokument enthaltenen Beschreibungen eine Stellvertreterklasse zu erstellen, die sich auf dem Clientrechner exakt so verhält, als wäre sie der Web Service selbst.

6.5. ASP.NET und Web Services

Im Folgenden soll auf Implementierungsdetails eingegangen werden, die nicht nur den generellen Aufbau eines Web Services wiedergeben, sondern auch spezielle, vor allem technische Details darlegen sollen.

Da unser verteilter RayTracer Web Services nutzen soll und zudem Microsoft's .NET–Technologie zum Einsatz kommen soll, fiel die Wahl der entsprechenden Middleware zwingend auf ASP.NET.

ASP.NET ist der „Nachfolger" des recht erfolgereichen ASP 3.0; eines serverseitigen Script–Interpreters, in etwa mit den Sprachen PHP oder Perl vergleichbar.
Der ASP 3.0 Scriptcode war in herkömmliche HTML–Seiten eingebettet; durch die Dateiendung `.asp` wurde dem Server signalisiert, dass die vorliegende Seite Scriptcode enthält, den es zu interpretieren gilt. Das Ergebnis der Ausführung wurde dann anstelle des Codes in der vom Server ausgelieferten HTML–Seite eingefügt.
Der Ansatz war gut, doch waren die Möglichkeiten relativ beschränkt; etwa gleichzusetzen mit den Scriptsprachen JScript und VBScript, nur mit dem Zusatz, dass sich ASP 30 durch COM–Komponenten erweitern lies.

ASP.NET stellt dagegen einen echten Schritt nach vorne dar. ASP–Seiten haben jetzt Zugriff auf die gesamte .NET–Klassenbibliothek; außerdem sind die aus dem .NET–Framework her bekannten Hochsprachen uneingeschränkt einsetzbar, so dass sich HTML–Elemente direkt programmieren lassen. D.h. sie sind als so genannte „HTML–Control" – Objekte durch den Code unmittelbar ansprechbar und somit auch programmierbar. Die Möglichkeiten für den Einsatz dynamischer Webseiten lassen sich an dieser Stelle nur erahnen.

Durch die so bezeichnete „code behind" – Programmierung in ASP.NET, womit HTML–Elemente vom Programmcode getrennt werden können, was einer `Include` – Anweisung gleichkommt, erhalten wir die für uns interessante Möglichkeit, Abarbeitungsergebnisse unseres Servercodes sowohl als HTML an einen Browser zu senden, als auch diese Ergebnisse durch einen Web Service zur Verfügung zu stellen. Da für unsere Zwecke kein HTML–Client gebraucht wird, findet sich in unserer Implementierung eines verteilt rechnenden RayTracers nur die erste Möglichkeit wieder.

Ein ASP.NET Web Service ist dabei nur eine einfache (C# –) Quellcodedatei mit der Dateierweiterung `.asmx`; beispielsweise die Datei `Raytracer.asmx.cs`.

Was zeichnet einen Web Service aus?

Am Kopf einer solchen `.asmx` – Datei steht immer die Web Service–Direktive, die der .NET–Engine zu verstehen gibt, dass dieser Code (Klasse) einen Web Service implementiert. Sie kann unterschiedliche Attribute beinhalten, welche die Arbeitsweise genauer beschreiben. In unserem Beispiel eines einen RayTracer implementierenden Web Services sieht diese Direktive folgendermaßen aus: (Datei: `Raytracer.asmx`)

```
<%@WebService Language="C#" Codebehind="Raytracer.asmx.cs"
Class="RaytracerService.SRaytracer"%>
```

Man sieht, dass innerhalb dieser Datei nur diese Web Service–Direktive zu finden ist. Aber wo ist dann unsere programmierte Applikationslogik?
Visual Studio .NET legt automatisch beim Erstellen eines ASP .NET Web Service Projektes zwei Dateien an. Die eine Datei enthält die obige Web Service–Direktive und die andere den Code für unsere Applikationslogik. Dieses Prinzip der Codierung nennt sich, wie schon erwähnt, „code–behind“. Ganz im Gegensatz zur `Inline` – Codierung, bei der sich der gesamte Code der Applikationslogik in einer `.asmx` Datei befindet.
Die Web Service–Direktive kann mehrere Attribute enthalten. In unserem Fall sagen diese Attribute aus, dass unser Quellcode der Programmiersprache C# entstammt, dass sich dieser Quellcode in der Datei `Raytracer.asmx.cs` befindet und dass die Klasse, welche als einzige Webmethoden beinhalten darf, `RaytracerService.SRaytracer` heißt.

Um weiterhin die Applikationslogik des RayTracers als Web Service zu etablieren, werden dem Code unterschiedliche Attribute hinzugefügt. Attribute in .NET sind deklarativer Code, der unserem programmierten Code zusätzliches Verhalten hinzufügt, ohne diesen per Hand programmieren zu müssen.
Das wichtigste Attribut dabei ist das WebMethod–Attribut. Dieses Attribut befähigt unsere Applikation dazu, über Rechnergrenzen hinweg aufrufbar zu sein. Webmethoden unterstützen zudem die Serialisierung und Deserialisierung nach und von XML, die Abbildung von XML–Werten auf korrekte Datentypen und den Transport von Anfragen und Antworten. Methoden einer Klasse, welche mit dem WebMethod–Attribut deklariert sind, sind später auch die einzigen Methoden, die nach außen hin sichtbar sind und von einem Client oder einer anderen Applikation genutzt werden können. Man kann daher sagen: Sie sind die `public` – Klassen des Webservices.
Bemerkenswerterweise genügt das Hinzufügen der beiden genannten Attribute `<WebService>` und `<WebMethod>`, um aus einer „normalen“ Klasse einen Web Service zu machen. Die erforderliche Arbeiten – das Erstellen eines WSDL–Dokuments und das Generieren der für die (De–) Serialisierung zuständigen Routinen – nimmt uns das .NET–Framework ab.
Sobald mit Visual Studio .NET ein ASP Web Service Projekt angelegt wird, erstellt es für uns ein „Web“ auf der lokalen Maschine (IIS muss installiert sein), das den Web Service bereitstellt. Die WSDL–Beschreibung des Services ist zu diesem Zeitpunkt schon vorhanden und kann über eine URL der Form:

```
http://computername/Webservice/WebserviceName.asmx?WSDL
```

angefordert werden.

Der Webserver erkennt bei etwaigen Requests anhand der angeforderten Datei, ob es sich um einen Web Service handelt (Endung `.asmx`). Wenn ja, dann gibt er die Kontrolle an das .NET–Framework weiter. Webserver und Framework kommunizieren dazu über die ISAPI–Schnittstelle des IIS.

6.6. Ein RayTracer als verteilte Anwendung

Der generelle Aufbau der Applikation wurde schon in Kapitel 2 beschrieben. Wir wollen nun einen Blick darauf werfen, wie das Zusammenspiel zwischen dem Controller und den RayTracern genau funktioniert. Die als Web Services implementierten RayTracing–Engines werden im Folgenden schlicht „Worker“ genannt. Als „roter Faden“ soll uns Abbildung 37 dienen.

Auffinden der Services

Zunächst muss der Controller die Adressen der Worker kennen, mit denen er zusammenarbeiten will. Die Technologie dieses zu ermöglichen ist, wie schon erwähnt, UDDI. Ursprünglich wollten wir diesen Weg auch gehen, jedoch war keine Implementation des UDDI–Services für Entwicklungszwecke verfügbar, welche man auf dem lokalen Rechner ohne größeren Aufwand installieren konnte. Es gibt zwar einen Testdienst seitens Microsoft, der öffentlich und frei benutzbar ist, doch ist dafür eine Registrierung notwendig bzw. war er für die Dauer der Entwicklung unseres Systems zu umständlich. Dennoch soll nicht darauf verzichtet werden, näheres über diesen Dienst zu erfahren. Das Kapitel „Ausblick“ wird später auf seine Benutzung genauer eingehen.

So dann entschieden wir uns dafür, die Adressen der benutzten Worker im Quellcode fest zu codieren.
Um die Integration einer späteren Nutzung des UDDI–Dienstes zu erleichtern, haben wir den Vorgang zum Auffinden der Worker in einen eigenem Namensraum `UDDIService` gekapselt. In ihm sind zwei Klassen zu finden, `UDDIInfo` und `FindServices` mit Namen. Die Klasse `UDDIInfo` enthält Angaben über den Namen, den Accesspoint und den ServiceKey eines Workers. Dieselben Angaben würde auch eine Abfrage des UDDI–Dienstes zurückliefern. Die Methode `Find()` der Klasse `FindServices` liefert eine `ArrayList` zurück, die die oben genannten UDDI–Informationen der einzelnen Worker beinhaltet.

Erstellen der Worker

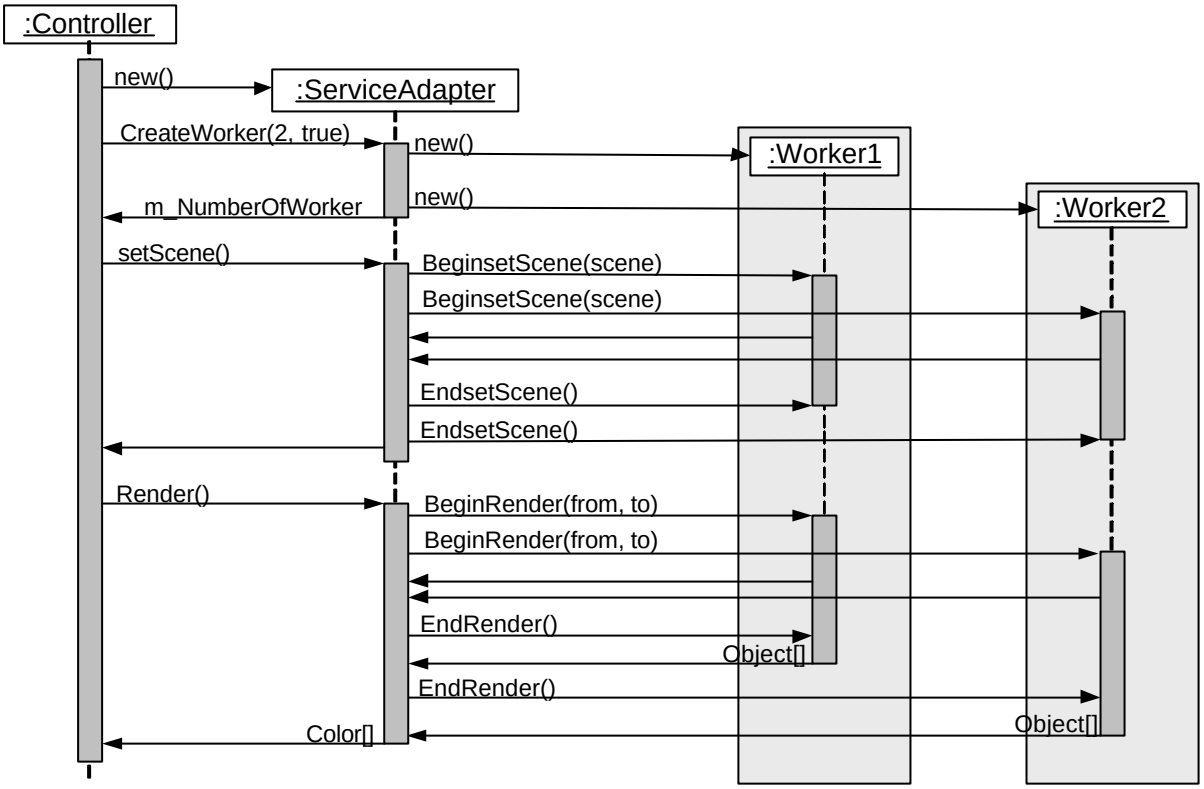


Abbildung 38: Sequenzdiagramm zum Ablauf des Erstellens der Worker und Aufruf der Webmethoden

Im Visual Studio .NET wird einem das Benutzen von Web Services sehr vereinfacht. Durch das Hinzufügen einer Webreferenz, verbunden mit der Eingabe einer Service–URL, wird von der Entwicklungsumgebung automatisch das WSDL–Dokument des betreffenden Web Services ausgelesen und eine passende Stellvertreterklasse[3] erstellt, in der die Kommunikation mit demselben komplett gekapselt ist. Diese Klasse enthält alle Methoden mit identischer Signatur, die der Web Service als „WebMethods“ exportiert, jedoch unterscheiden sich ihre Implementierungen stark von den ursprünglichen Methoden. Der neu generierte Code übernimmt die Kommunikation sowie die (De–) Serialisierung, ohne dass der Programmierer sich selbst darum kümmern muss. Die neugewonnenen Proxy–Klassen passen immer nur zu einem bestimmten Web Service. Sie ermöglichen die Benutzung des Services, als wäre dieser eine lokale Klasse.

Da wir mehrere Web Services benutzen, diese aber identische Schnittstellen besitzen, braucht es nur eines expliziten Setzens des jeweiligen Accesspoints, den wir für jeden Worker spezifisch aus der Klasse `UDDIInfo` gewinnen können, um mit seiner Hilfe den empfangenden RayTracer–Service zu bestimmen. Wir haben folglich nur eine einzige Proxy–Klasse für viele RayTracer.

[3] Dieser Vorgang kann auch manuell mit Hilfe des `wsdl.exe` Tools durchgeführt werden.

In der Klasse `ServiceAdapter` ist die Methode `CreateWorker()` für das Instanziieren der Worker zuständig. Sie bekommt als Argument die Anzahl der gewünschten Worker übergeben und ruft sie daraufhin in Form von Objekten der RayTracer–Proxyklasse ins Leben. Dabei wird normalerweise bei jedem Stellvertreterobjekt dem Attribut `Url` ein anderer Accesspoint zugewiesen.
Ist jedoch die geforderte Anzahl von Workern höher als die der verfügbaren Web Services und zugleich das Flag `multiple` gesetzt, so können auch mehrere Objekte mit ein und demselben Accesspoint erstellt werden.

Benutzung der RayTracer–Web Services

Wir haben nun schon Schritt 3 der Abbildung 37 erreicht; alle Worker sind bereit zum Einsatz gelangt. Bleibt noch zu klären, was im Schritt 4 geschieht – der tatsächlichen Benutzung des Web Services.

Im RayTracer–Projekt existieren genau zwei Webmethoden. Sie sind in der Klasse `SRaytracer`[4] zu finden und heißen: `setScene()` bzw. `render()`. Dem Controller obliegt es, sie bei Bedarf aufzurufen.
Die Methode `setScene()` hat die Aufgabe, die Szenenbeschreibung vom Controller in Empfang zu nehmen. Parallel existiert eine gleichnamige Methode in der ServiceAdapter–Klasse, die für die Verteilung der Szenenbeschreibung an alle beteiligten Worker zuständig ist.

Asynchroner Aufruf von Web Services

Das Setzen der Szenenbeschreibung auf Web Service – Seite geschieht aus dem Grund heraus, dass eine Übertragung einer großen Szenenbeschreibung einige Zeit in Anspruch nehmen kann, asynchron.
Wie aber kann man Web Services asynchron aufrufen, wo doch auf jeden HTTP–Requests – SOAP–Nachrichten basieren bekanntlich auf HTTP – genau eine Response folgt?
Glücklicherweise unterstützt uns auch hier das .NET–Framework tatkräftig. Wie schon erwähnt, sind in der Proxy klasse alle Methoden des Web Services ebenfalls definiert. Weiter existieren zu jeder Methode zwei weitere, die alleinig dafür sorgen, einen asynchronen Aufruf zu realisieren. Die Signaturen dieser zusätzlichen Methoden folgen immer dem nachstehenden Schema:

```
System.IAsyncResult BeginMethodenname(..., System.AsyncCallback callback,
     object asyncState);

[Rückgabetyp] EndMethodenname(System.IAsyncResult asyncResult);
```

Wird die Methode `Begin[irgendwas](...)` aufgerufen, so wird ein Request an den Web Service geschickt und die Methode kehrt <u>sofort</u> zum Aufrufer zurück. Zusätzlich zu den vorhandenen Übergabeparametern existieren noch zwei weitere von gewisser Bedeutung. Der eine `callback` ist dabei ein Delegator, der erst nach Beendigung der Aufgabe aufgerufen wird. Der andere erlaubt es, der Methode ein beliebiges Objekt zu übergeben[5].

[4] Diese stellt den eigentlichen Web Service dar.
[5] Wird oft zur Übergabe des Proxy–Objektes genutzt.

Im Grunde entsprechen Aufrufe von „WebMethods“ in asynchroner Weise viel eher dem asynchronen Naturell von HTTP[6]. Tatsächlich werden die Antworten eines Web Services, denen ein vermeintlich „synchroner“ Methodenaufruf vorausgegangen ist, genauso über Callback–Methoden abgewickelt, wie dies auch im asynchronen Fall geschieht, nur wird jener Sachverhalt oft vor dem Programmierer versteckt. Asynchrone Aufrufe sind also nichts weiter als etwas weniger transparent gehaltene „normale“, synchrone Aufrufe.

Der Controller ruft somit für jeden beteiligten Worker die Methode `BeginsetScene()` neu auf. Eine Callback–Methode, welche ein Ergebnis entgegennimmt, benötigen wir nicht, folglich werden beide Parameter auf `null` gesetzt. Stattdessen warten wir auf die Fertigstellung aller Aufrufe (`WaitAll`). Nachdem wir alle „WaitHandles“ gespeichert haben, rufen wir für jeden Worker einzeln die Methode `EndsetScene()`[7] auf. Durch diesen kleinen Trick können wir sicher sein, dass alle Services ihre Szenenbeschreibung auch wirklich empfangen haben.

Datentypen und Web Services

Unsere Szenenbeschreibung ist im XML–Format kodiert. In der Klassenstruktur existiert eine entsprechende Klasse `SceneDescription`, die jeweils eine Methode zur Serialisierung und Deserialisierung anbietet. Nach der Deserialisierung liegt das `SceneDescription` – Objekt wieder in seiner ursprünglichen Form vor. Die Methode `setScene()` des Workers nimmt als Parameter für die Szene einen `string` – Wert entgegen, womit sich unmittelbar die Frage anschließt: Warum übergibt man nicht einfach das gerade deserialisierte `SceneDescription` – Objekt?

Alle Übergabeparameter an Webmethoden müssen nach XML serialisiert und in eine SOAP–Nachricht verpackt werden, damit man überhaupt Netzwerkfähigkeit gewinnt. Hierbei werden alle primitiven Typen wie `string`, `int`, `double` usw. des .NET–Frameworks unterstützt; einschließlich Arrays dieser Datentypen. Eigene Typen wie Klassen und Strukturen (`struct`) werden auch ohne weiteres akzeptiert, allerdings nur solange, wie deren Member als `public` deklariert sind und sie ihrerseits primitive Datentypen darstellen.

Enthält eine Klasse alles andere als diese primitiven Datentypen, wie beispielsweise die Klasse `SceneDescription`, so gibt es noch die Möglichkeit, die Serialisierungsvorgänge durch Hinzufügen von Attributen so zu steuern, dass sie in der gewünschten, korrekten Form als XML vorliegen.
Dies kann aber bei relativ komplexen Datenstrukturen, wie in unserer Szenenbeschreibung vorzufinden, bisweilen recht mühsam sein, weshalb wir kurzerhand die Methode `sceneToString()` in der Klasse `SceneTool` aufnahmen, die den gesamte Inhalt der Szenenbeschreibung einfach in einen String konvertiert. Dieser ist dann ein primitiver Datentyp und kann ohne weiteres serialisiert bzw. verschickt werden.
Auf der Web Service – Seite rufen wir dann die (statische) Deserialisierungsmethode auf, die es zur Aufgabe hat, aus dem String die Szenenbeschreibung wieder herzustellen.

[6] Auf jeden Request folgt ein Response, zwischen beiden können, wie bei Web Services durchaus möglich, Tage vergehen.
[7] Die Methode hat als Rückgabewert das Resultat des Aufrufes, weshalb eine Callback–Routine nicht unbedingt nötig ist.

Zustandsbehafteter Webservice?

Zurück zu unserem Raytracer. Die zweite von den Workern bereitgestellte Webmethode ist `render()`, die ebenso wie `setScene()` ein gleichnamiges Pendant in unserem ServiceAdapter besitzt. Diese Methode ist für die Verteilung der Rendering–Abschnitte zuständig; die Szene wird zeilenweise gerendert.

Der Aufruf funktioniert genauso wie beim Setzen der Szene selbst. Die Worker erhalten reihum jeweils die Anweisung, eine Zeile der Szene, welche sie als Szenenbeschreibung zuvor erhalten haben, zu berechnen.
Aber warum muss die Szenenbeschreibung nur einmal zu Beginn an den Worker übergeben werden, zumal auf der Web Service – Seite jeder Request getrennt von jedem davor oder danach empfangenen bearbeitet wird?
Die Antwort ist einfach: Bei jedem Request wird eine neue Instanz des Web Service erstellt und nach Beendigung des Aufrufes wieder zerstört. Die Szenenbeschreibung muss daher in einer statischen Klassenvariablen vorliegen, was einem gewissen Zustand gleichkommt.

ASP.NET verfügt über einen so genannten Application–State, welcher applikationsweit genutzte Einstellungen und Daten beinhaltet. In diesem wird auch unsere Szenenbeschreibung verwaltet. Der Application–State war schon Bestandteil der Version ASP 3.0, jedoch erfuhr er in ASP.NET Erweiterung und wurde in die objektorientierte Welt des Frameworks integriert.
Andere ASP–Seiten, die zur gleichen Web–Applikation wie der Web Service gehören, haben ebenfalls vollen Zugriff auf den Application–State, weshalb es etwa denkbar wäre, über das Internet eine `.aspx` – Seite über einen Browser aufzurufen, die dann mitteilt, welche Szene gerade gerendert wird. ASP.NET ist hier einen weiteren Schritt in Richtung „richtiger“ Applikationsentwicklung gegangen.

Eine andere Möglichkeit wäre für unsere Zwecke die Nutzung des Session–States, welcher ähnlich aufgebaut ist, wie der Application–State, jedoch den Fokus auf die aktuelle Session – also die momentan ablaufende Interaktion mit einem bestimmten Client – besitzt.
Um eine Session, die aus verschiedenen, voneinander unabhängigen Requests besteht kann, zu identifizieren, ist es notwendig, eine Session–ID entweder als Cookie auf dem Client zu speichern, oder bei jedem Request an die URL anzuhängen.

7. Testen

7.1. Grundlegendes

Das Testen objektorientierter Software zeigt sich erstaunlicher Weise als nicht–triviales Unterfangen. Typische Charaktere der Objektorientierung, die das intuitive Verständnis erschweren können, sind:

- *Dynamisches Binden* – Hierunter ist zu verstehen, dass eine Nachricht erst zur Laufzeit eines Programms einer konkreten Operation zugeordnet wird, die diese Nachricht dann interpretiert.
- *Polymorphismus* (Vielgestaltigkeit) – Heißt, dass gleichlautende Nachrichten an kompatible Objekte unterschiedlicher Klassen ein unterschiedliches Verhalten bewirken können. Beim dynamischen Polymorphismus wird eine Nachricht nicht zur Compilezeit, sondern erst beim Empfang zur Programmlaufzeit einer konkreten Operation zugeordnet. Voraussetzung hiefür ist das dynamische Binden.
- *Gestiegene Komplexität*, hervorgerufen durch Vererbungsbeziehungen
- *Abstrakte– / Interface– / Parametrisierbare Klassen* (Templates) – Von einer abstrakten Klasse werden niemals Objektexemplare erzeugt; sie wird bewusst unvollständig gehalten und bildet somit die Basis für weitere Unterklassen, die dann Exemplare haben können. Interfaces sind besondere Erscheinungen abstrakter Klassen; sie definieren ausschließlich abstrakte Operationen. Eine parametrisierbare Klasse hingegen ist eine mit generischen formalen Parametern versehende Schablone, mit der „gewöhnliche“ Klassen erzeugt werden können. Die generischen Parameter dienen als Stellvertreter für die aktuellen Parameter, die Klassen oder einfache Datentypen repräsentieren.

Um den letzten Punkt nochmals aufzugreifen: Aus der Tatsache heraus, dass einer abstrakten Klasse kein Objekt entstammen kann, ergibt sich unmittelbar das Problem, diese Klasse ausreichend zu testen. Erst mit Hilfe von ihr abgeleitenden Klassen, also Unterklassen, räumt sich eine entsprechende Möglichkeit ein. Ähnliche Engpässe lassen sich selbstverständlich auch für die anderen Punkte zeigen, doch soll darauf verzichten werden.

Wie soll aber vorgegangen werden? In jedem Falle kann die Modularisierung des Programms einen ersten Ansatz bieten. Besser noch, man macht sich das Konzept der Klassen sowie der daraus resultierenden Abgeschlossenheit zu Nutze. Dieses sollte dann aber konsequent durchgesetzt bzw. beibehalten werden, so dass man niemals Gefahr läuft, die „naturgegebene“ Abgeschlossenheit zu durchbrechen. Besonders die Programmiersprache C++ bringt diesbezüglich problematische Sprachbestandteile (bspw. `friend`) mit. Doch wird dieser Sprachbestandteil im vorliegendem Programm kaum in Erscheinung treten können, da die Umsetzung mittels C# erfolgte und derartige Ausdrücke durch die Sprache nicht verstanden werden. Dennoch gehört zum Verständnis der Abgeschlossenheit hinzu, dass Zugriffe auf etwaige Attribute der Klasse nur über den dafür konzipierten Operationen erfolgen dürfen.
Denn nur so werden annähernd Voraussetzungen geschaffen, die es erlauben, einzelne Operationen (Methoden) einer Klasse ebenso wie „normale“ Funktionen aus der prozeduralen Welt zu überprüfen.

Eine erste Möglichkeit ist das „CodeReview“. Dieses geschieht eher intuitiv während der eigentlichen Entwicklung der Anwendung und kann daher immer genannt werden.

Der nächste Schritt gilt der Klasse selbst. Eine Klasse ist die Definition der Attribute, Operationen und der Semantik für eine Menge von Objekten. Alle Objekte einer Klasse entsprechen eben dieser Definition. Demzufolge müsste ein Klassentest dieser Semantik folgen, damit die starke Bindung, die zwischen den Attributen und Operationen herrscht, in den seltensten Fällen durchbrochen wird.

Ist die Funktionalität der Basis, d.h. der Klassen, nachgewiesen, geht es daran ihr Zusammenwirken im Ganzen zu testen. Jener Test wird als Integrationstest verstanden. Gegenstände der Untersuchung können beispielsweise die Schnittstellenspezifikationen[8] der Operationen sein.

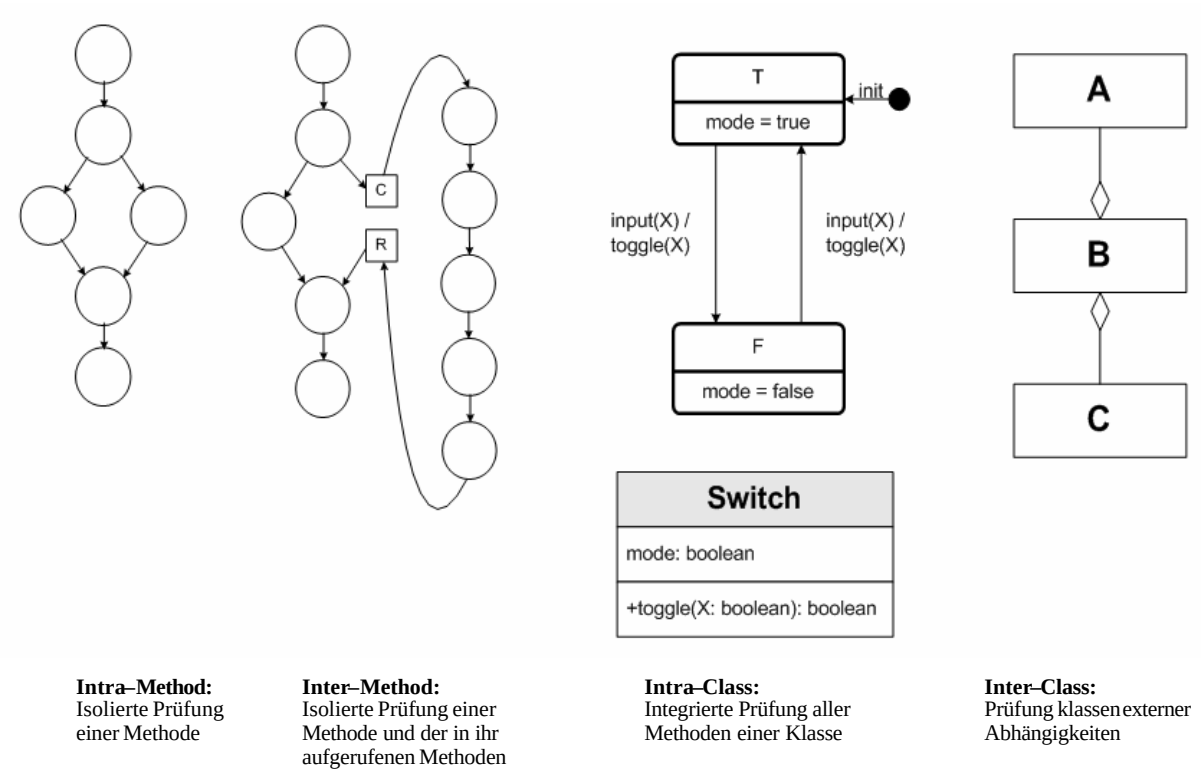


Abbildung 39: Ebenen objektorientierter Modulprüfung[9]

Die Abbildung 39 fasst die vorherige Diskussion prägnant zusammen. Die dort angedeutete Teststrategie ist unter der Bezeichnung „Bottom Up – Test“ bekannt.

Er setzt an den Klassen geringster Komplexität an; d.h. diejenigen Bausteine, die die Basis für andere aufsetzende Bausteine darstellen. Im Rahmen des vorliegenden Programms sind das beispielsweise Klassen wie: `Box`, `Sphere`, `Triangle`, `Material`, `Ray` oder `SceneDescription`.

Natürlich sollte man darin vertrauen können, dass alle der Framework Class Library (kurz FCL) entstammenden Klassen als ausreichend getestet gelten und daher mit Recht den Status einer „reifen Klasse“[10] beschreiben.

[8] Ein geeigneter Test ist das Ansetzen von Äquivalenzklassen.

[9] Doktorandenkolloquium vom 24. Mai 2002; Dipl.–Ing. Christopher Robinson–Mallett

[10] Von zentraler Bedeutung ist hierbei die Frage, wann eine Klasse als wieder verwendbar gilt (Klassenbibliothek). Anders als bei anwendungsspezifischen Klassen, die mit Beginn der Betriebs– und Wartungsphase als fertig gestellt betrachtet werden, gelten Klassen, die zur Wiederverwendung bestimmt sind, zunächst als „unreif“; sie müssen noch eine gewisse Stabilität erfahren. Dazu wird eine unreife Klasse über die Zeit beobachtet und erst nach zahlreichen erfolgreichen Einsätzen in unterschiedlichen Umgebungen in den Status „reif“ erhoben.

Auch innerhalb von Teilsystemen (Bsp. Komponente) sollte der Idee des „Bottom Up – Tests“ nachgegangen werden. Demzufolge sind zunächst alle Klassen innerhalb einer Komponente zu prüfen, bevor sie selbst einem Test unterzogen wird. Ein Beispiel dafür ist die Komponente `RayTracing.dll` und alle in ihr befindlichen Klassen, wie `RayTracer`, `BoxRayTracer` oder `SphereRayTracer`. Auf diese Weise arbeitet man sich von dem Einfachen zum Komplexen vor.

7.2. Klassentest

Nachdem die Ansatzmöglichkeiten eines Tests sowie das Vorgehen desselben ausreichend beleuchtet wurden, soll nun der zweiten Ebene des Testens – der Klassentest – Aufmerksamkeit zukommen. Weshalb die erste Ebene – die des CodeReviews – nicht näher beschrieben wird, liegt darin begründet, dass diese Testmöglichkeit, wie schon weiter oben gesagt wurde, während der Programmierung eher automatisch geschieht; zum Beispiel bei auftretenden Kompilationsfehlern.

Dazu sollte im Vorfeld weiterer Überlegungen geklärt werden, was denn überhaupt sinnvoller Weise zu testen ist. Um die Frage zufrieden stellend zu beantworten, hilft ein kleiner Rückblick weiter: Die zugrunde liegende Anwendung beschreibt in ihrem Hauptanliegen einen RayTracer. Das Ergebnis allen Handelns ist demnach ein computergrafisches Bild, indem allgemeingültige Figuren (Shapes) sichtbar sind. Das Wichtigste ist daher das korrekte Darstellen aller Bildbestandteile – also der Shapes.

Nun aus der vorgestellten Architektur lassen sich drei verschiedene Arten von Shapes identifizieren.

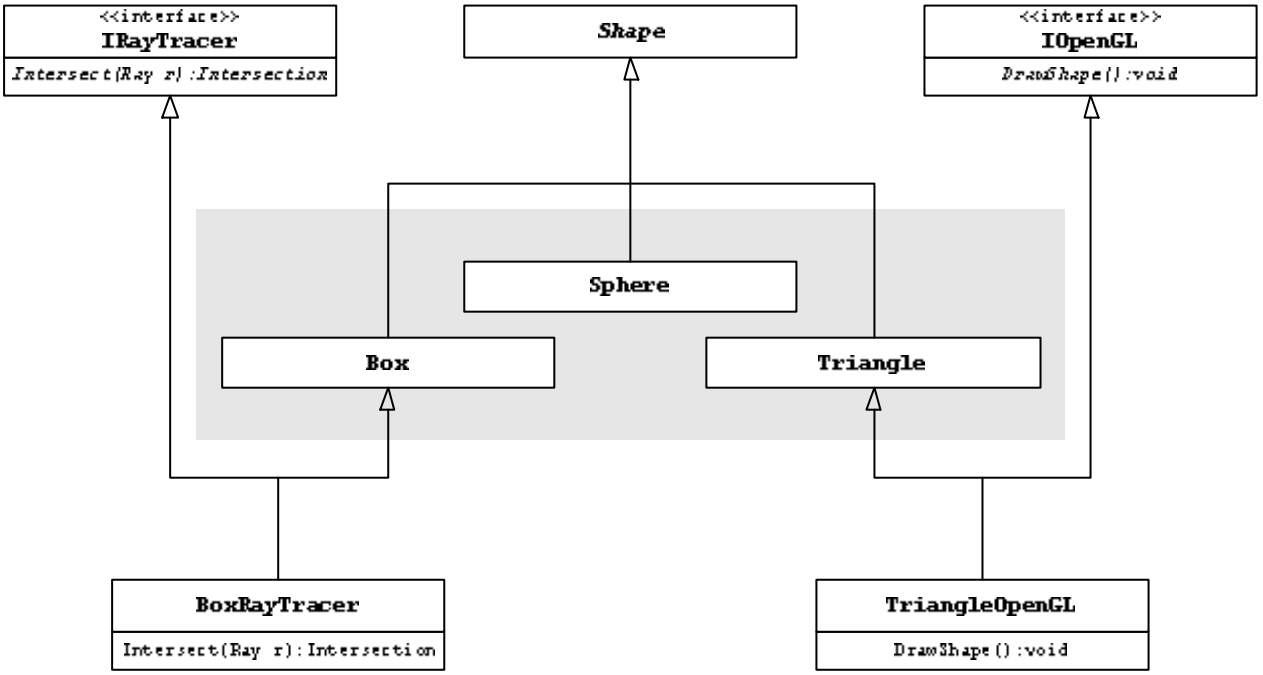


Abbildung 40: Die wesentlichen Vererbungsbeziehungen (Auszug der Klassenhierarchie)

Im Ersten sind das Shapes, die außer ihrer geometrischen / mathematischen Repräsentation nichts weiter aussagen und daher einen sehr begrenzten Ansatz zum Testen bieten. D.h. natürlich nicht, dass man eine Kugel (Klasse: `Sphere`) nicht durch Angabe eines räumlichen Kugelmittelpunkts sowie eines Radius größer Null instanziieren kann, aber in irgendeiner Weise sichtbar, was das eigentliche Anliegen ist, wird sie dadurch nicht. Mit anderen Worten: Der Test muss schon deswegen eine Hierarchieebene weiter tiefer greifen, um eine Darstellungsmöglichkeit zu gewinnen.

So ergeben sich zwei Alternativen dieses zu realisieren. Sie werden durch zwei neue Arten von Shapes repräsentiert. Man betrachte dazu nochmals das Klassendiagramm.
Eine Art von Shapes (rechts abgebildet) gewinnt die Fähigkeit sich darzustellen, in dem sie die OpenGL–Grafikbibliothek (`www.opengl.org`) verwendet. Jene Klassen erben zum einen die geometrischen Eigenschaften der allgemeingültigen Shapes und implementieren zum anderen das Interface `IOpenGL`. Das Interface `IOpenGL` verlangt alleinig die Umsetzung der Methode `DrawShape()`, die die Aufgabe hat, mittels OpenGL–Befehle das betreffende Shape zu zeichnen. Die obige Abbildung 40 zeigt beispielhaft, wie ein Dreieck unter Benutzung von OpenGL zum Darstellen gebracht werden kann. Die resultierende Klasse heißt dann `TriangleOpenGL`.

Doch braucht diese Art der Repräsentation nicht notwendiger Weise einem Test unterzogen werden. Wie bereits gesagt, ist OpenGL im Einsatz. OpenGL selber existiert schon seit Jahren als freiverfügbare Grafikbibliothek, so dass zahlreiche Anwendungen – v.a. Spiele –darauf aufsetzen und man den Herstellern darin vertrauen kann, dass sie die Funktionalität hinreichend getestet haben.

Die andere Möglichkeit besteht in der Bildberechnung mittels RayTracing (links abgebildet). Das berechnete Bild ist somit die eigentliche Darstellung. Da das Testen der OpenGL verständlichen Klassen aus den obigen Gründen wegfällt, bleibt nur hier ein Ansatz zum Testen übrig. Das ist auch praktikabel; denn neben dem Test der geometrischen Eigenschaften der unmittelbar höheren Hierarchieebene hat man zudem die Möglichkeit seine visuelle Vorstellung von der Szene in einem auf komplexem Wege bestimmten Bild zu überprüfen.
Das Interface `IRayTracer` legt dazu die verbindliche Implementation der Methode `Intersect(Ray r)` fest. Mit Hilfe dieser Methode wird mathematisch ermittelt, ob ein gegebener Strahl (Ray) die sichtbare Oberfläche des Shapes tatsächlich trifft (Schnittpunktberechnung).

Nach dem Klären der Frage: „Was getestet wird?“ schließt sich unmittelbar das „Wie?“ an. Ein erster, weit verbreiteter Weg zu Testen besteht darin, im Quelltext ausreichend viele Ergebnis– und Kontrollanzeigen zu verankern. Diese werden dann beim Ausführen des Programms über die Standardausgabe (in der Regel die Konsole) ausgegeben. Zu dieser Sparte gehört ebenfalls das Führen einer Log–Datei, die die sonst in der Standardausgabe gezeigten Testdaten persistent festhält.
Der Einsatz eines Debuggers stellt eine zweite, komfortablere Testmethodik dar. Er ermöglicht die Analyse und Manipulation einzelner Ausdrücke, ohne das Programm als Ganzes neu kompilieren zu müssen. Zudem ist es unter Umständen sehr hilfreich, den gesamten Quelltext zum Zwecke der Analyse vor sich zu sehen. Jenem Bedürfnis wird ein Debugger gerecht.

Obwohl die zwei vorgestellten Verfahren in der Praxis weite Anwendung finden, besitzen sie doch einen vielleicht entscheidenden Nachteil. Beide benötigen den Menschen zur Kontrolle und Steuerung. Sei es nur schon deshalb, die einzelnen Ausgaben im Laufe des Testens auswerten zu wollen. Weitaus schlimmer wiegt die Tatsache der in den Verfahren begründeten Unhandlichkeit. Um zwei Beispiele zu geben:

- Setzt man den Debugger des Microsoft Visual Studios ein und hat entsprechende „BreakPoints“ definiert, braucht es immer eines Tastendruckes, bis zur nächsten logisch folgenden Quelltextzeile gesprungen wird. Im Falle des Microsoft Debuggers ist das Aufgabe der Taste F10.

- Das Verfahren, welches das Festhalten der Ergebnis– und Kontrollanzeigen beinhaltet, ist nicht nur aufwendig, sondern birgt zudem die Gefahr der so genannten „Scroll Blindness“; was zur Bedeutung hat, dass die Konsole (Standardausgabe) mit Ausgaben nur so überflutet wird, dass jedes Verständnis verloren geht.

Wohin die Diskussion führen soll ist, den Vorteil darin zu erkennen, ein generisches Test–Framework einzusetzen. Solch ein Test–Framework führt den eigentlichen Test und was weitaus entscheidender ist, die Auswertung der Ergebnisse voll automatisch durch. Eine menschliche Interaktion wird so weitestgehend überflüssig, so dass der Aufwand für den Menschen auf ein Minimum reduziert wird.

Ein guter Vertreter eines Test–Frameworks ist **NUnit** (`www.nunit.org`). NUnit ist gewissermaßen das Gegenstück des aus der Java–Welt bekannten Frameworks JUnit; halt nur auf das Testen .NET–fähiger Sprachen spezialisiert.

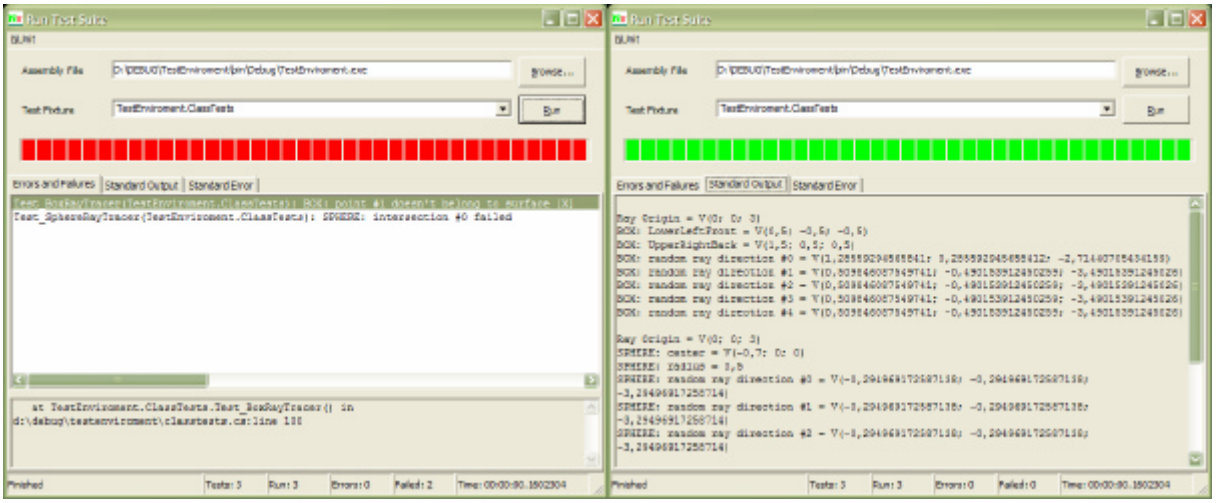


Abbildung 41: GUI von NUnit

Die Abbildung 41 zeigt die grafische Oberfläche von NUnit. Links ist ein Fall zu sehen, der tunlichst zu vermeiden ist. Der rote Balken signalisiert dort, dass ein Fehler während des Testens aufgetreten ist. Im Ausgabebereich direkt darunter ist die Ursache dafür gezeigt. Der Fehler war, dass ein Schnittpunkt bestimmt wurde, der nicht zur Oberfläche des untersuchten Shapes (Box) gehört. Im rechten Bild dagegen ist alles „gut gegangen“ (man betrachte den grünen Balken); es traten keine Fehler auf.

Um das Framework NUnit nutzen zu können, sind folgende Schritte notwendig:

- Es ist eine Unterklasse zu definieren, die von der Klasse `TestCase` erbt. Diese Unterklasse beherbergt die für den individuellen Anwendungsfall relevanten Testfälle.
- Beim Testen kommt es des Öfteren vor, dass verschiedene Testszenarien auf ein und dieselbe wohldefinierte Menge an zu testende Objekte basieren. Da immer die Gefahr besteht, dass ein Testszenarium die Menge manipuliert und dadurch die Ergebnisse anderer später folgender Szenarien verfälschen kann, bietet NUnit die Möglichkeit mittels Überschreiben der Methode `SetUp()` die identische Ausgangssituation für jeden einzelnen Testfall bereitzustellen. Falls dabei

auf besonderem Wege Speicher alloziiert wurde, kann dieser durch Neudefinieren der Methode `TearDown()` wieder freigegeben werden.

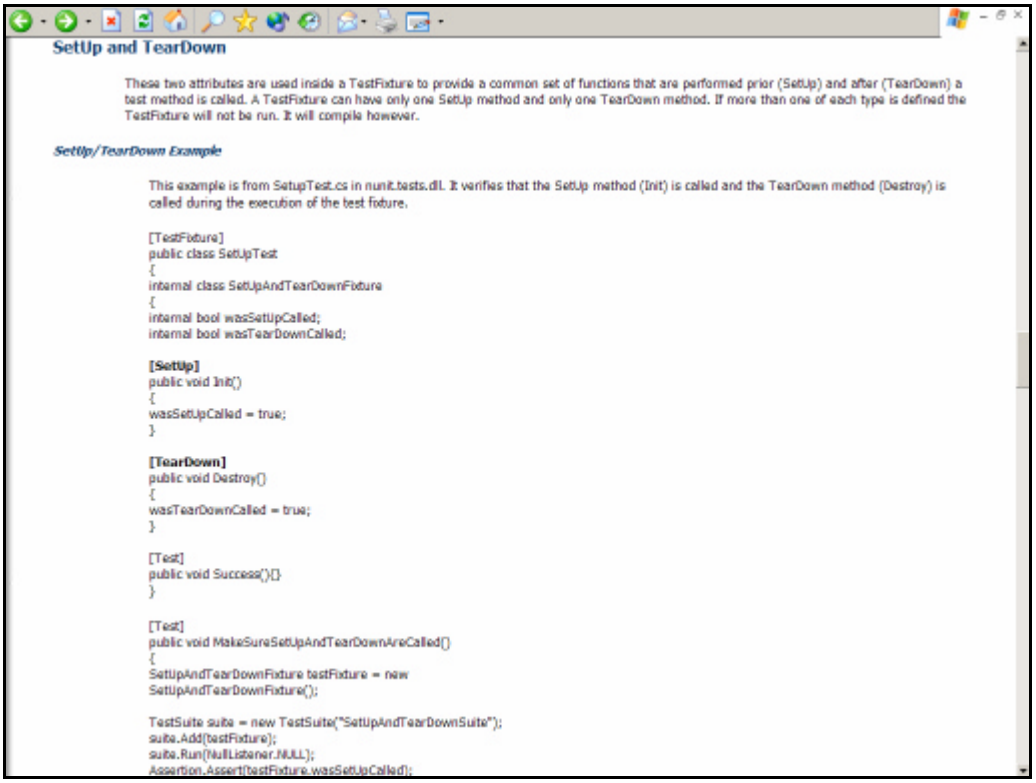
- Die Testfälle selbst sind in eigene Methoden zu kleiden. Dabei ist zu beachten, dass diese Methoden das Sichtbarkeitsattribut `public` besitzen, damit ein Zugriff seitens NUnits auf sie stattfinden kann. Hinsichtlich der Namensgebung ist man relativ frei. Dennoch ist ein Präfix „Test“ vorteilhaft, da NUnit solch benannte Methode über Reflektion selbstständig erkennt und ausführt. Andernfalls muss die gewählte Bezeichnung der Methode NUnit explizit bekannt gemacht werden. Das kann mit Hilfe der Methode `RunTest()` geschehen. In ihrem Rumpf sind dann die selbstgewählten Methodenbezeichnungen aufzuführen.

Ein Beispiel:

```
public class ClassTests : TestCase
{
    //  add an instance variable for each part of the fixture
    public BoxRaytracer      testBox;
    public SphereRaytracer   testSphere;
    public TriangleRaytracer testTriangle;

    public ClassTests(string method) : base(method) {}

    //  initialize the variables
    protected override void SetUp() {
        testBox      = new BoxRaytracer();
        testSphere   = new SphereRaytracer();
        testTriangle = new TriangleRaytracer();
    }

    public void Test_BoxRayTracer() {
        //  ARE THE VERTICES DIFFERENT?
        Assert("BOX: vertices aren't different",
            !testBox.m_LowerLeftFront.Equals(testBox.m_UpperRightBack));
        ...
    }
}
```

Der obige Quelltext zeigt beispielhaft, wie eine eigene „Testklasse“ aufgebaut werden kann. Zu sehen ist das Überschreiben der Methode `SetUp()`, so dass für jeden Testfall alle drei Shapes – BoxRaytracer, SphereRaytracer und TriangleRaytracer – neu instanziiert werden. Somit wird eine identische Ausgangssituation garantiert. Unmittelbar folgend ist ein erster Testfall angedeutet. Das Shape, was einer Untersuchung unterliegt, ist eine Box. Der Präfix „Test“ in der Namensgebung der Methode ist deutlich zu erkennen. Die Methode selbst besteht im Großen und Ganzen aus Assert–Anweisungen. Die im Text gezeigte überprüft, ob die die Box spezifizierenden Punkte auch verschieden voneinander sind. Denn das wird laut Definition gefordert, weil sonst die Box sich in einem einzigen Punkt repräsentiert, was völliger Blödsinn ist.

In der neuesten Version[11] sind die Richtlinien für eine korrekte Verwendung NUnits hinsichtlich der Benutzbarkeit stark verbessert worden. Vieles, was zuvor nur das Überschreiben vordefinierter Methoden erlaubte, kann jetzt einfach und elegant über Attribute spezifiziert werden. Der Übersichtlichkeit kommt dies enorm entgegen.



SetUp and TearDown

These two attributes are used inside a TestFixture to provide a common set of functions that are performed prior (SetUp) and after (TearDown) a test method is called. A TestFixture can have only one SetUp method and only one TearDown method. If more than one of each type is defined the TestFixture will not be run. It will compile however.

SetUp/TearDown Example

This example is from SetupTest.cs in nunit.tests.dll. It verifies that the SetUp method (Init) is called and the TearDown method (Destroy) is called during the execution of the test fixture.

```
[TestFixture]
public class SetUpTest
{
internal class SetUpAndTearDownFixture
{
internal bool wasSetUpCalled;
internal bool wasTearDownCalled;

[SetUp]
public void Init()
{
wasSetUpCalled = true;
}

[TearDown]
public void Destroy()
{
wasTearDownCalled = true;
}

[Test]
public void Success(){}
}

[Test]
public void MakeSureSetUpAndTearDownAreCalled()
{
SetUpAndTearDownFixture testFixture = new
SetUpAndTearDownFixture();

TestSuite suite = new TestSuite("SetUpAndTearDownSuite");
suite.Add(testFixture);
suite.Run(NullListener.NULL);
Assertion.Assert(testFixture.wasSetUpCalled);
```

Abbildung 42: Auszug aus den Neuerungen NUnits

[11] vom 16. August 2002 (RC1)

7.3. Integrationstest

Das Testen der Klassen – somit der einzelnen Shapes – ist erfolgreich abgeschlossen, so dass es jetzt daran geht, ihr Zusammenwirken zu untersuchen. Der Begriff „Zusammenwirken“ soll sich hierbei soweit zu verstehen geben, dass damit die Beziehungen zwischen den Shapes gemeint ist; zum Beispiel: zeigen sich Spiegelbilder, werden gerade Kanten aufgrund von Linseneffekte verzerrt dargestellt usw.

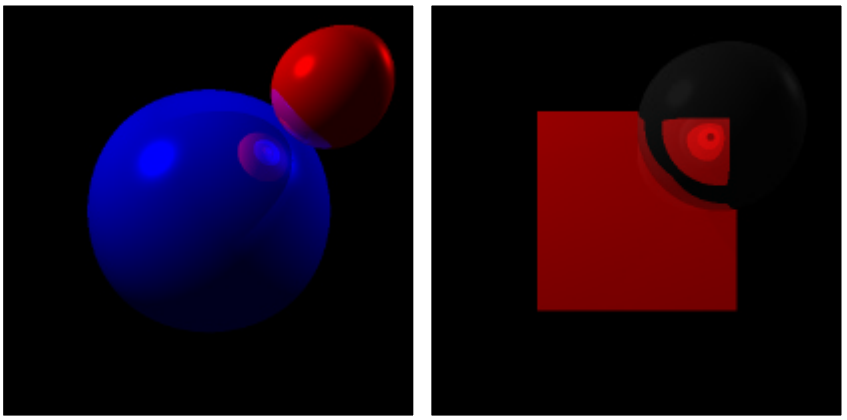


Abbildung 43: Spiegelbild (links) und Linseneffekt (rechts)

Das Problem, welches sich einem stellt, ist eine angemessene Beurteilung des Gesamtergebnisses. Schon mehrmals wurde herausgestellt, dass das Resultat aller Berechnungen ein computergrafisches Bild ist. Demnach sollte ein Bewertungskriterium genau hier ansetzen.

Eine erste Bewertung kann der Mensch, besser: der Szenendesigner, unmittelbar selbst treffen. Denn gewöhnlich beschreibt er die Szene mit Hilfe eines entsprechenden Editors und hat daher das beste Verständnis darüber, was im späteren Bild zu sehen sein wird. Somit ist es ihm möglich, sein Szenenverständnis – seine Erwartungshaltung – mit dem resultierenden Bild zu vergleichen.

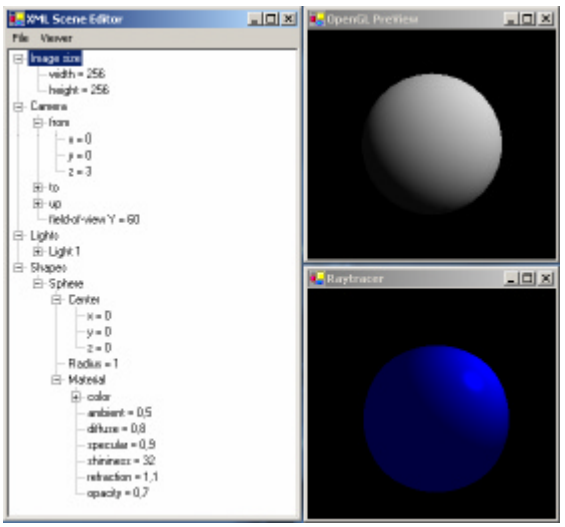


Abbildung 44: Vergleich der beschriebenen Szene (Editor links) und dem resultierenden Bild (oben: OpenGL; unten: RayTracing)

Wird dann das Erwartete nicht sichtbar, ist ein Indiz gegeben, dass etwas schiefgelaufen ist.

Ein anderer Weg ein Bild zu beurteilen, liegt in dem Vergleich desselben mit einem zweiten die gleiche Szene darstellenden Referenzbild. Für den vorliegenden Fall könnte das das Resultat eines RayTracers sein, der in C++ umgesetzt wurde. Solch ein RayTracer war Gegenstand einer Übungsaufgabe der Vorlesung „Computergrafik II“ des letzten Wintersemesters 2001 / 2002. Er ist ausreichend getestet und kann folgerichtig als Referenz herangezogen werden.

Das hier berechnete Bild liegt in einem Format vor (PPM – Format), dass es leicht macht, einzelne Bytes, die die Farbwerte der einzelnen Pixel kodieren, auszulesen und mit gleichgesinnten Bytes des Referenzbildes zu vergleichen. Leider ist ein Vergleich auf Identität nicht praktikabel, da die Werte der Farbkomponenten auf Variablen beruhen, die vom Typ `double` sind. Es dürfte bekannt sein, dass sich die Rechengenauigkeit von Prozessor zu Prozessor unterscheidet; vor allem bei der Berechnung von `double` – Werten. D.h. man müsste, um diesen Umstand zu berücksichtigen, eine gewisse Varianz zulassen, mit der die Bytes im resultierenden Intervall immer noch Gültigkeit besitzen.

```
public Intersection Intersect(Ray ray)
{
    Vector point     = new Vector(0, 0, 0);
    Vector normal    = new Vector(0, 0, 0);
    double dDistance = 9999999999.9;

    //  ray origin
    double x = ray.Origin.X;
    double y = ray.Origin.Y;
    double z = ray.Origin.Z;

    //  ray orientation
    double i = ray.Direction.X;
    double j = ray.Direction.Y;
    double k = ray.Direction.Z;

    //  sphere properties
    double l = Center.X;
    double m = Center.Y;
    double n = Center.Z;
    double r = Radius;

    //  calculate parameters a, b, c for equation: at²+ bt+ c= 0
    double a = i* i +
               j* j +
               k* k;
    double b = 2 * (i * (x - l) +
                    j * (y - m) +
                    k * (z - n));
    double c = l* l +
               m* m +
               n* n +
               x* x +
               y* y +
               z* z -
               2 * (l* x + m* y+ n* z) - r* r;

    //  calculate D= b²- 4ac
    double D = b* b – 4 * a * c;

    //  analyse cases of intersections
    //  no intersection
    if (D < 0.0) return new Intersection();
```

```
    //  ray parameter
    double t = 0.0;
    //  exactly one intersection
    if (D == 0.0)
    {
        //  calculate ray parameter
        t = -b / 2 * a;
    }
    //  two intersections
    else
    {
        // calculate ray parameter
        double t1 = (-b + Math.Sqrt(D)) / 2* a;
        double t2 = (-b - Math.Sqrt(D)) / 2* a;

        //  looking for nearest intersection
        t = Math.Min(t1, t2);
    }

    //  work around for shadows
    if(t <= 0.001) return new Intersection();

    //  define return values
    point  = ray.Origin + t * ray.Direction;
    normal = (point- Center) / r;

    dDistance = (point - ray.Origin).Length;

    return new Intersection(point, normal, ray, dDistance);
}
```

Dieser Quelltext soll nachmals zum Ausdruck bringen, dass verschiedene Prozessoren unterschiedlich genau rechnen. Im obigen Quelltext wird dem dadurch entgegengewirkt, indem man innerhalb der Schnittpunktberechung eine entsprechende Abfrage definiert. Ist t kleiner gleich 0,001, also vor allem negativ, liegt der Punkt aufgrund der Rechenungenauigkeiten innerhalb des Shapes und wird bei der Schattierung nicht berücksichtigt.

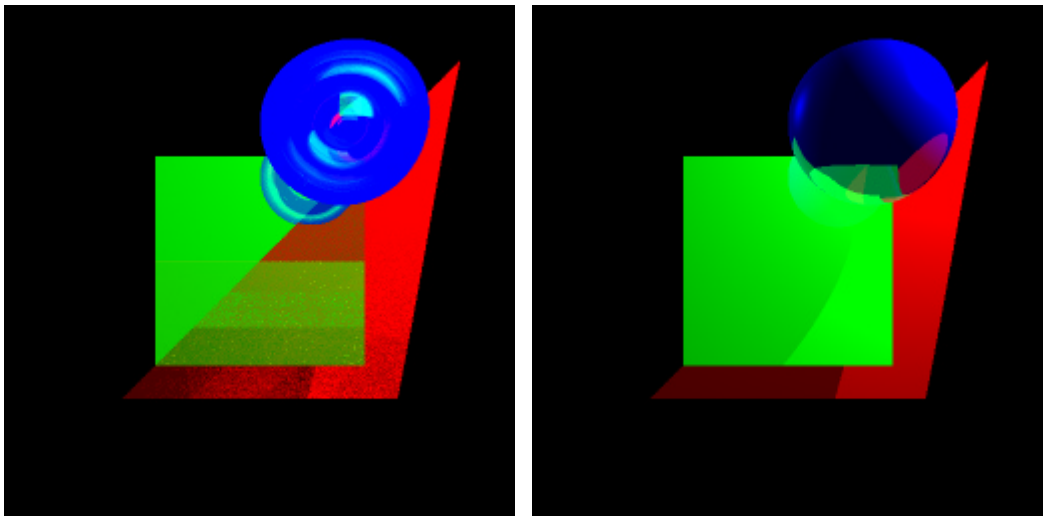
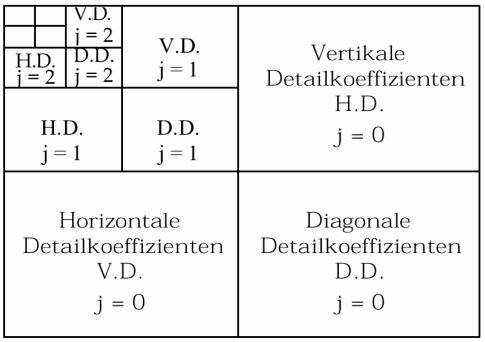


Abbildung 45: Fehlerhafte Schattierung / Färbung (links); Korrektur der Rechenungenauigkeiten (rechts)

Ein wissenschaftlicher und daher sehr genauer Ansatz ist die Anwendung von Bildmetriken. Ein Weg, ein Bild dahingehend zu untersuchen, soll im Weiteren vorgestellt werden.

Die Wavelet–Transformation

Um mittels der Wavelet–Transformation Bildverarbeitung betreiben zu können, muss sie auf zwei Dimensionen erweitert werden. Die Ursache dafür liegt in der Auffassung der Bilder als zwei dimensionale Felder, wobei jeder Position im Bild ein Farbwert zugeordnet wird. Ein Farbwert selbst ist oft eine mehrdimensionale Größe, die die verschiedenen Komponenten (RGB) des Farbwertes definiert. Zur Verarbeitung sind dann diese Komponenten zu extrahieren, so dass sie einzeln angesprochen werden können.



Zum Verständnis der Vorgehensweise ist die nebenstehende Abbildung zu betrachten. Bei der Zerlegung des Bildes wird in jedem Schritt zuerst eine Transformation in horizontaler Richtung und anschließend in vertikaler Richtung durchgeführt. Das ursprüngliche Bild wird so in jeder Richtung um die Hälfte verkleinert. Im Rest des Bildes, immerhin dreiviertel der Bildfläche, sind die so genannten Detailkoeffizienten abgelegt. Für den ersten Transformationsschritt ist das beispielsweise j = 0. Nun kann in einem nächsten Schritt (j = 1) das bereits gefilterte Bild einer weiteren Transformationsstufe unterzogen werden.

Ein typisches Anwendungsgebiet, in dem man sich die Eigenschaften zu Nutze macht, ist das Auffinden von Bildern in Datenbanken oder Archiven. Dieses auf den Inhalt beruhende Abfragen von Bildern mittels einer Vorlage – die das zu suchende Bild charakterisieren soll – ist eine nicht triviale Aufgabe.
Gründe dafür sind schon angeklungen; die wesentlichen aber werden hier nochmals zusammengefasst:

- Vorlage– und Zielbild liegen in unterschiedlichen Auflösungen vor
- Farben sind eventuell verschoben
- Form und Position stimmen nicht exakt überein.

Aus diesen und vor allem Performanzgründen, ist ein 1:1 Vergleich (L^1 – Metrik) nicht anzustreben. Es ist vielmehr eine Metrik vorzuziehen, die in der Lage ist, wesentliche Bildcharakteristika zu identifizieren und auf entsprechende Koeffizienten abzubilden.

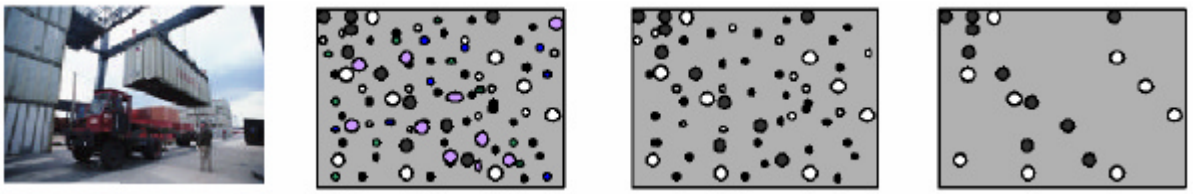
Charles Jacobs[12] gelang es, solch eine Metrik zu entwickeln. Die wesentlichen Merkmale sind (vgl. dazu Abbildung 46):

- *Wavelet–Tranformation:* Das Bild durchläuft eine Wavelet–Transformation. Aus dem Bild wird eine Auflösungsebene in der Größenordnung 128 x 128 Pixel verwendet. Dies entspricht 128^2 Koeffizienten für jede Farbkomponente. Um nicht alle Koeffizienten zu verwenden, werden in dem folgenden Schritt die Anzahl der Werte reduziert.

[12] C.E. Jacobs, A. Finkelstein, D. Salesin: Fast Multiresulotion Image Querying. In Proceddings of SIGGRAPH ’95, 277-286. ACM, New York, 1995.

- *Truncation:* Es werden nur die Detailkoeffizienten mit dem größten Werten verwendet. Erfahrungsgemäß genügen ca. 40 bis 60 Werte.
- *Quantisierung:* Es werden nicht mehr die Koeffizientenwerte selbst gespeichert, sondern nur ob ein großer negativer oder großer positiver Wert vorliegt.

In einer Anwendung werden nun alle Bildkandidaten nach diesem Algorithmus hin untersucht und mit einem „Merkmal – Stempel“ versehen.



a) b) c) d)

Abbildung 46: Erzeugung eines „Merkmal – Stempels“: Das ursprünglich Bild (a) wird in Wavelet– Koeffizienten (b) zerlegt. Es werden nur die größten Werte berücksichtigt (c). Diese werden im letzten Schritt (d) quantisiert.

Dieses Prinzip könnte mühelos auf unser Problem – eine faire Bewertung von Bildinhalten zu finden – übertragen werden. Die Lösung hieße dann:

- Generieren von zwei die gleiche Szene beschreibenden, aber von unterschiedlichen RayTracern berechneten Bildern
- Analyse jener Bilder nach dem oben vorgestellten Algorithmus
- Vergleich der resultierenden Wavelet–Koeffizienten

Wie gesagt, die Wavelet–Transformation stellt nur eine Möglichkeit dar, Bildmetriken mit dem Testen von Bildern in Beziehung zu setzen. Das Verfahren selbst, ohne in die mathematischen Tiefen abstürzen zu wollen, ist leicht nachzuvollziehen und scheint auch logisch. Es sollte auch nur der Anstoß gegeben werden, sich diesbezüglich mit Bildmetriken auseinander zu setzen.

8. Warum C# und .NET

Das Anliegen des gesamten Projektes ist es, ein computergrafisches Bild geteilt zu berechnen. „Geteilt“ soll zum Ausdruck bringen, dass mehrere Workerprozesse – in ihrem Ursprung RayTracer – auf verteilten Rechnern einen bestimmten Bereich des Bildes zugeteilt bekommen, den sie dann mit Farbwerten zu füllen haben. Das Ergebnis senden sie nach Fertigstellung des Bildausschnittes an den Auftrageber zurück.

Die Sprache C# macht es einem diesbezüglich einfach, verteiltes Rechnen zu realisieren. Es glänzt mit einer sehr guten Integration von Web Services. Schon ein mittels des „Application Wizards“ generiertes Projekt, das auf Web Services beruht, zeigt schnell ein einfaches „Hello World!“. Dabei hat man doch nichts ahnend nur mit OK betätigt.

Für die notwendige Kommunikation kann beispielsweise XML als Metasprache zum Einsatz kommen. Auch dies ist in C# ohne weiteres möglich, so dass es sich gerade zu anbietet, die Szenenbeschreibung, die jedem Worker, so weit er auch entfernt sein mag, bekannt gemacht werden muss, im XML – Format zu kodieren.

```
<?xml version="1.0" ?>
- <SceneDescription xmlns:xsd= ... >
- <m_Camera>
  <From X="0" Y="0" Z="2" />
  <To X="0" Y="0" Z="0" />
  <Up X="0" Y="1" Z="0" />
  <View X="0" Y="0" Z="2" />
  <FieldOfViewY>60</FieldOfViewY>
  </m_Camera>
- <m_Light>
- <Light>
  <Position X="1" Y="1" Z="1" />
  <Intensity>1</Intensity>
  <Ambient>0.5</Ambient>
  </Light>
  </m_Light>
- <m_Shape>
- <Shape xsi:type="Triangle">
- <Vertices>
  <Vector X="-1" Y="-1" Z="-1" />
  <Vector X="1" Y="-1" Z="-1" />
  <Vector X="1" Y="1" Z="0" />
  </Vertices>
  <Normal X="0" Y="-0.44721359549995793"
    Z="0.89442719099991586" />
  </Shape>
  </m_Shape>
...
```

Abbildung 47: Auszug einer Szenenbeschreibung im XML – Format

Schon diese beiden Gründen sprechen für sich – für den Einsatz von C#. Ein anderer ist die höhere Performanz gegenüber einer Alternativimplementierung in Java, in der sicherlich ein Zusammenspiel mit CORBA erfolgt wäre.

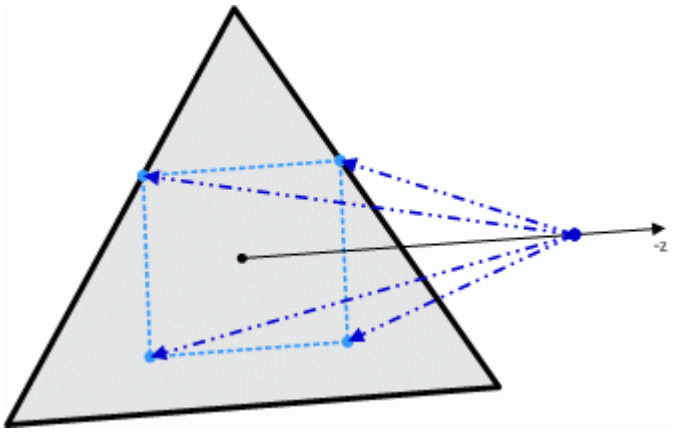
Um die Behauptung noch zu untermauern, wurden Leistungstests durchgeführt. Der Test beruhte auf gleichartige, aber in verschiedenen Programmiersprachen umgesetzte Implementierungen eines RayTracers.

Die Wahl der Programmiersprachen viel auf:

- C++ als Referenzlösung,
- selbstverständlich C# als Realisierungsplattform und
- letztlich Java.

Der Aufbau des Tests ist denkbar einfach (vgl. auch nachstehenden Quelltext in C#). Auf ein festkodiertes Referenzshape – ein Dreieck – werden mehrmals Teststrahlen geschickt. Diese sind so orientiert, dass sie mit mathematischer Gewissheit das Dreieck treffen; folglich mit dem Dreieck Schnittpunkte besitzen.
Warum gerade ein Dreieck? Ein Dreieck bot sich deshalb zum Einsatz an, weil mögliche Schnittpunkte mit Hilfe barizentrischer Koordinaten bestimmt werden. D.h. das Verfahren nutzt zahlreiche Eigenschaften von Vektoren, wie das Normalisieren (Einheitsvektor), das Berechnen des Kreuz– und Skalaproduktes; jedenfalls rechenintensive Operationen, die Zeit benötigen.

Die nebenstehende Abbildung zeigt den Vorgang. Die Strahlen treffen abwechselnd auf verschiedene Punkte des Dreiecks, die zusammen genommen, im Aussehen einem Quadrat gleichkommen. Zwei der Punkte sind sogar so gewählt, dass sie direkt auf Dreieckskanten liegen und infolgedessen immer noch zum Dreieck gehören. Dadurch gewinnt man die Möglichkeit, zu untersuchen, ob die vorliegende Programmiersprache große Rechenungenauigkeiten mit sich bringt.



Die dazugehörige Implementierung in C#:

```
namespace Performance_CSharp
{
    class MainClass
    {
        const int MAX_TEST_CASES = 999999;

        /// <summary>
        /// the main entry point for the application
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //  initialize the shape
            Triangle testShape = new Triangle(new Vector(-4.0d,-2.0d, 0.0d),
                                              new Vector( 0.0d, 2.0d, 0.0d),
                                              new Vector( 4.0d,-2.0d, 0.0d));

            //  initialize the ray
            Ray testRay = new Ray(new Vector( 0.0d, 0.0d,-5.0d), new Vector());

            //  define points that belong to triangle's surface
            Vector[] testPoints = {  new Vector(),
                                     new Vector(-1.0d, 1.0d, 0.0d),
                                     new Vector( 1.0d, 1.0d, 0.0d),
                                     new Vector( 1.0d,-1.0d, 0.0d),
                                     new Vector(-1.0d,-1.0d, 0.0d)  };
```

```
            int successful = 0;
            int failed     = 0;
            int index      = 0;

            //  start timer
            DateTime startTesting = DateTime.Now;

            for(int testCase = 0; testCase < MAX_TEST_CASES; testCase++)
            {
                //  calculate actual ray direction
                testRay.Direction = testPoints[index++] - testRay.Origin;

                //  prove intersection
                if(testShape.intersect(testRay))
                    successful++;
                else
                    failed++;

                //  for fluctuation
                if(index >= 5) index = 0;
            }

            //  stop timer
            DateTime endTesting = DateTime.Now;

            //  calculate elapsed time
            TimeSpan duration = endTesting - startTesting;

            Console.WriteLine("C# Performance...");
            Console.WriteLine("Time to do {0} test cases is {1}!",
                MAX_TEST_CASES, duration.ToString());
            Console.WriteLine("[+ {0}; - {1}]", successful, failed);
        }
    }
}
```

Und hier das Ergebnis des Leistungstests. Es ist die Zeit dargestellt, die von den einzelnen Implementierungen benötigt wurde, die geforderte Anzahl von Testfällen zu erfüllen.

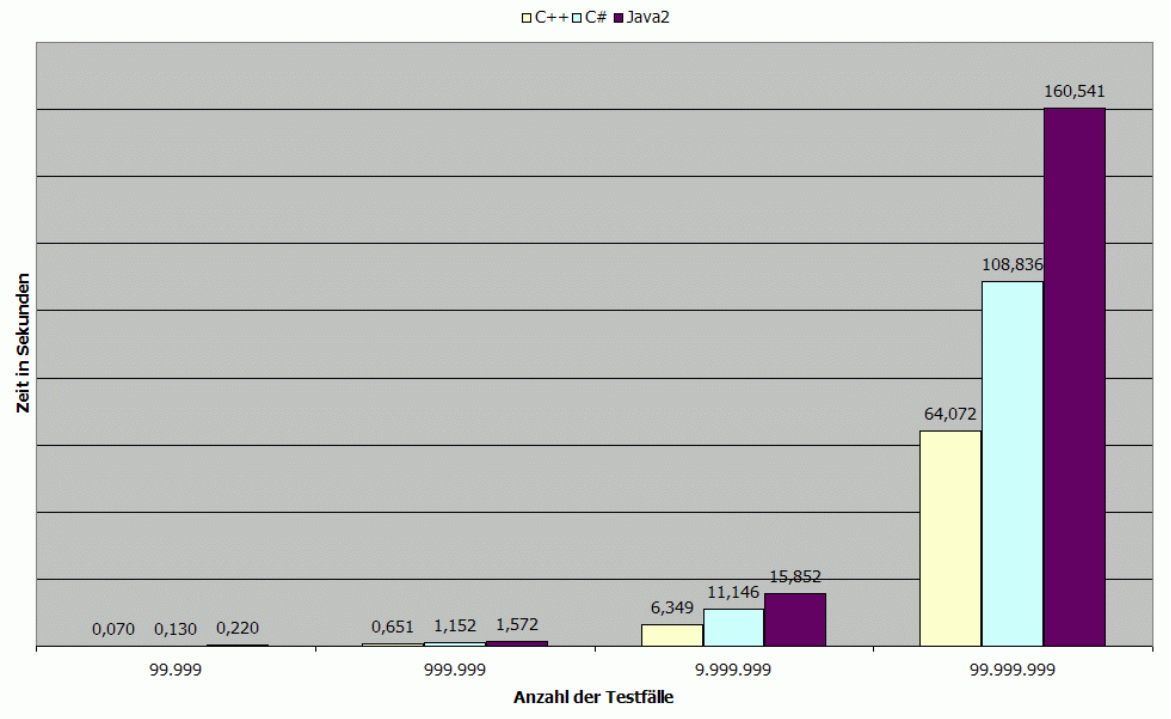


Abbildung 48: Ergebnis des Leistungstests

(System: AMD 900MHz; 256 MB SDRAM; XP)

Wie zu sehen ist, steht die Implementation in Java immer hinten an. Die Ursache dafür, so die grobe Vermutung, liegt in dem langen Laufzeitverhalten der Virtual Machine, die den Java–Code erst interpretieren muss. Natürlich, so wird man jetzt argumentieren, liegt auch C# eine den Code interpretierende Instanz zugrunde, doch scheint sie, so zeigt es die Abbildung 48, weitaus performanter zu sein.

Während den einzelnen Testdurchläufen wurden auch die Intersektionsergebnis festgehalten; d.h. ob eine Intersektion zwischen Strahl und Shape stattfand oder nicht. Da die Strahlen, wie beschrieben, so gewählt sind, dass stets eine Intersektion vorliegt, müsste die Anzahl der „geglückten“ Intersektionen (Variable: `successful`) gleich der der Testfälle insgesamt sein.
Bei den Programmiersprachen C++ und C# ist das auch der Fall gewesen. Nur in Java schlug jede fünfte Intersektion (Variable: `failed`) fehl, was theoretisch unmöglich ist. Somit zeigt sich die Rechenungenauigkeit in der Java–Umsetzung besonders stark.

Und zu guter letzt, der wohl entscheidende Grund für die Wahl der Sprache C#. Die Entwicklungsumgebung „Visual Studio .NET“ gestaltet das Programmieren mit zahlreichen Tools, Wizards, Syntaxhervorhebungen und „just in time“ – Fehlermeldungen einfach. Zudem ist das grundlegende Arbeitsprinzip, welches in der neuen Version nur Verbesserung erfuhr, bereits aus der Vorabversion 6.0 her bestens bekannt.

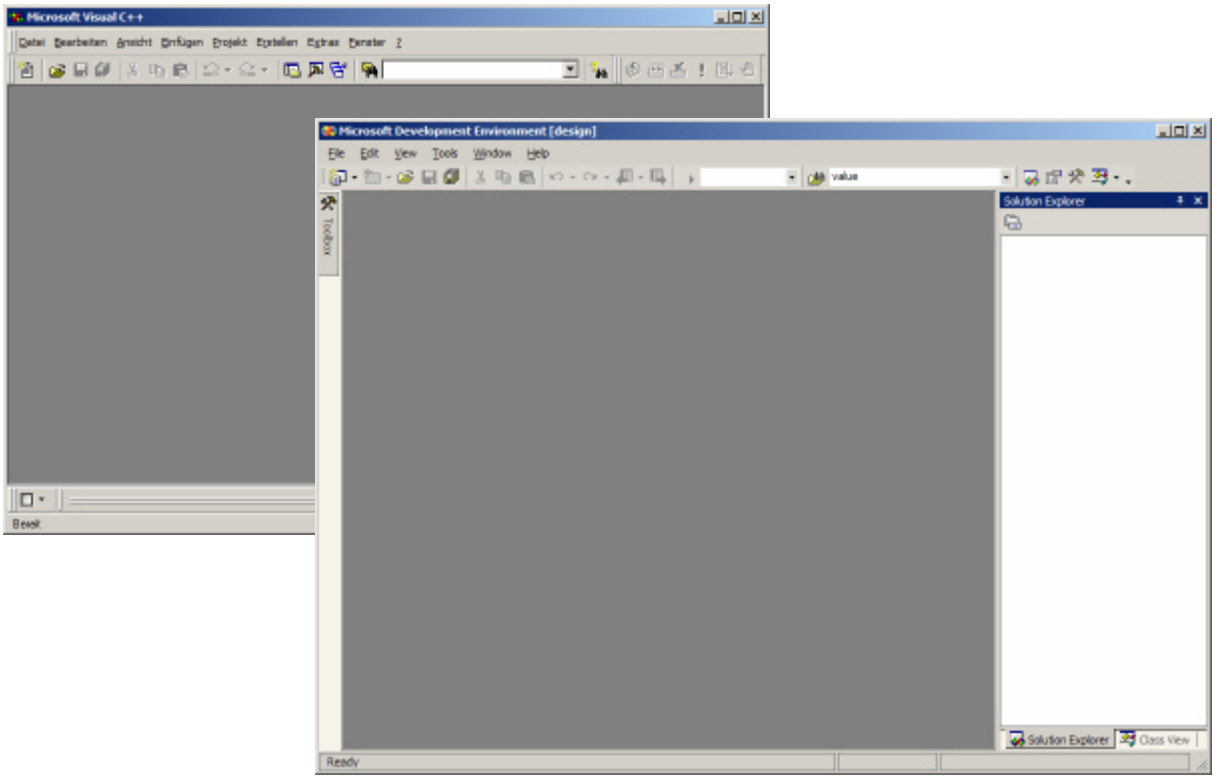


Abbildung 49: Vertraute Entwicklungsumgebungen (VS 6.0 links; VS .NET rechts)

9. Ausblick

9.1. Nutzung des UDDI – Services

Wie schon angedeutet, sollte das Auffinden unserer Worker nicht mittels fest kodierter URL's geschehen, sondern vielmehr durch die intensive Nutzung aller Möglichkeiten, die uns der UDDI – Dienst bietet.
Dieser Dienst ist selbst als Web Service implementiert und stellt eine Art „Gelbe Seiten" für weitere dar. Es existieren schon jetzt brauchbare Implementationen von zahlreichen Firmen, wie beispielsweise Microsoft, IBM und HP.

UDDI ermöglicht den Zugriff auf Informationen, die angeben, welche konkreten Web Services angeboten und welche WSDL–Dokumente zu ihrer Beschreibung benutzt werden. Zur besseren Auffindbarkeit sind diese Informationen in Kategorien geordnet.

Zunächst müssen die Web Services im UDDI–Verzeichnis registriert werden. Die Eingabe der relevanten Informationen kann entweder manuell per Webinterface geschehen, oder automatisch über die UDDI – API, die zum einen das programmatische Registrieren und zum anderen das Auffinden von Web Services unterstützt. Diese API ist auch für .NET erhältlich.

Für unser Projekt besonders interessant wäre ein bestimmter organisatorischer Aspekt des UDDI–Verzeichnisses. UDDI „merkt" sich zu jedem Web Service seine Schnittstellenbeschreibung, unabhängig von den Angaben, die ein UDDI – Nutzer zu einem konkreten Web Service führt (hauptsächlich der Accesspoints). Die Schnittstellenbeschreibung wird als so genanntes tModel (für Technologie Model) gespeichert. Zu jeder Schnittstelle kann es natürlich, genau wie in der objektorientierten Programmierung, verschiedene Implementationen geben. Die Zuordnung des tModels zu den Web Services, die eine gegebene Schnittstelle implementieren, erfolgt über so bezeichnete Bindings.

Für uns wäre es nun wünschenswert, wenn man lediglich die Schnittstellenbeschreibung im UDDI registriert (was durchaus möglich ist) und später über Abfrage der an unseren tModel geknüpften Bindings die Accesspoints der entsprechenden Web Services identifiziert. Neben dem leichteren Auffinden unserer Worker wäre es ebenso möglich, andersartige Implementationen in das System aufzunehmen. Wenn man seine Gedanken etwas schweifen lässt, wird dadurch ein weltweites RayTracing–Netzwerk denkbar, welches auf nur einer Schnittstellenbeschreibung basiert.

9.2. Dynamisches Hinzufügen neuer Shapes

Weiterhin wäre es wünschenswert, unserem RayTracing–System weitere Shapes hinzufügen zu können. Dies ist auf der Controllerseite relativ unproblematisch, aber wie sollen die neuen Shapes den Workern bekannt gemacht werden?

Zunächst ist es ohne Probleme machbar, dass ein Worker dem Controller meldet, dass ihm ein oder mehrere Shapes fehlen. Da die Szenenbeschreibung in ihrer XML–Form die verwendeten Shapes beinhaltet, wäre es denkbar, dass die Worker zunächst die darin klassifizierten Shapes auslesen und sie mit einem vorhandenen Shape–Repertoire zu vergleichen. Fehlt dann ein Shape (Exception), so melden sie dies dem Controller.

Dieser wiederum könnte daraufhin den Workem eine entsprechende `.dll` – Datei zusenden. Da die `.dll` – Dateien selbst im Binärformat vorliegen, müsste man den BinaryFormatter des .NET eigenen XML–Serialisierers zur Konvertierung nach XML einsetzen, um sie nutzbar zu machen.
Ist die `.dll` dann erst einmal auf der Worker–Seite vorhanden, kann die darin enthaltene Shape–Klasse dynamisch geladen und instanziiert werden. Somit wäre das fehlende Shape einsatzbereit.

Anhang

A1. Das Dokumentationswerkzeug **NDoc**

NDoc (`http://ndoc.sourceforge.net/`) ist ein kleines Tool, das aus einer Datei, die im XML – Format vorliegt, und einer Projektdatei der Visual Studio .NET – Entwicklungsumgebung eine Dokumentation wahlweise im HTML – Format oder als CHM – Datei automatisch generiert.
Mittels dieses kleinen Tools ist es einem möglich, im mitdokumentierten Quelltext einfach und schnell von Methoden– oder Klassennamen zu den entsprechenden Implementierungen zu springen.

Für eine korrekte Anwendung sind geringfügige Konventionen einzuhalten.

Vor allem ist es von Bedeutung, den Quelltext ausreichend zu kommentieren. Diese „Hinweise“ werden von NDoc ausgelesen und finden sich so in der späteren Dokumentation wieder. Wichtig dabei ist, dass der Stil der Kommentare der des Visual Studio .NET ist. D.h. Kommentare „der neuen Version“ sehen dann wie folgt aus:

```
///<summary>;
/// Dies ist die Klasse RayTracer.
///</summary>
```

Die Kommentare werden dabei in Tags eingeschlossen. Insgesamt sind neun Kategorie zu unterscheiden:

Tag	Beschreibung / Verwendung	Attribute
`example`	Zeigt ein Anwendungsbeispiel für das jeweilige API – Element; verwendet meist das `<code>` tag	keine
`exception`	Beschreibt eine Ausnahmesituation, die auftreten kann.	cref *
`include`	Bindet eine externe XML – Datei ein.	file, path
`param`	Beschreibt einen Parameter einer Methode oder Eigenschaft.	name *
`permission`	Gibt die Zugangsberechtigungen bzw. Gültigkeit (Scope) eines Typmitglieds wieder.	cref *
`remarks`	Beschreibt einen Typ oder eine Klasse.	keine
`returns`	Gibt den Rückgabewert einer Methode wieder.	keine
`seealso`	Äußert einen Verweis auf verwandte Themen.	cref *
`summary`	Beschreibt Typmitglieder wie Methoden, Eigenschaften und Felder.	keine

Zur Formatierung stehen weitere elf Tags zur Verfügung:

Tag	Beschreibung / Verwendung	Attribute
`c`	Text wird als einzeiliger Quelltextauszug formatiert	keine
`code`	Für mehrzeilige Quelltextauszüge	keine
`description`	Beschreibung	keine
`item`	Punkt in einer Liste	keine
`list`	Liste	type
`listheader`	Listenkopfzeile	keine
`para`	Fügt einen Absatz ein (vergleichbar mit `<p>` in HTML)	keine
`paramref`	Zeichnet einen Text als Parameter aus	name *
`see`	Fügt einen Verweis innerhalb eines XML – Kommentars ein	cref *
`term`	Begriff, der durch `<description>` definiert wird	keine
`value`	Beschreibt eine Eigenschaft	keine

Hat man die Kommentare in den Quelltext einfügt und in der Projekteigenschaft aktiviert, dass während der Kompilierung ein XML – Datei erzeugt werden soll, kann NDoc mit seiner Arbeit beginnen.

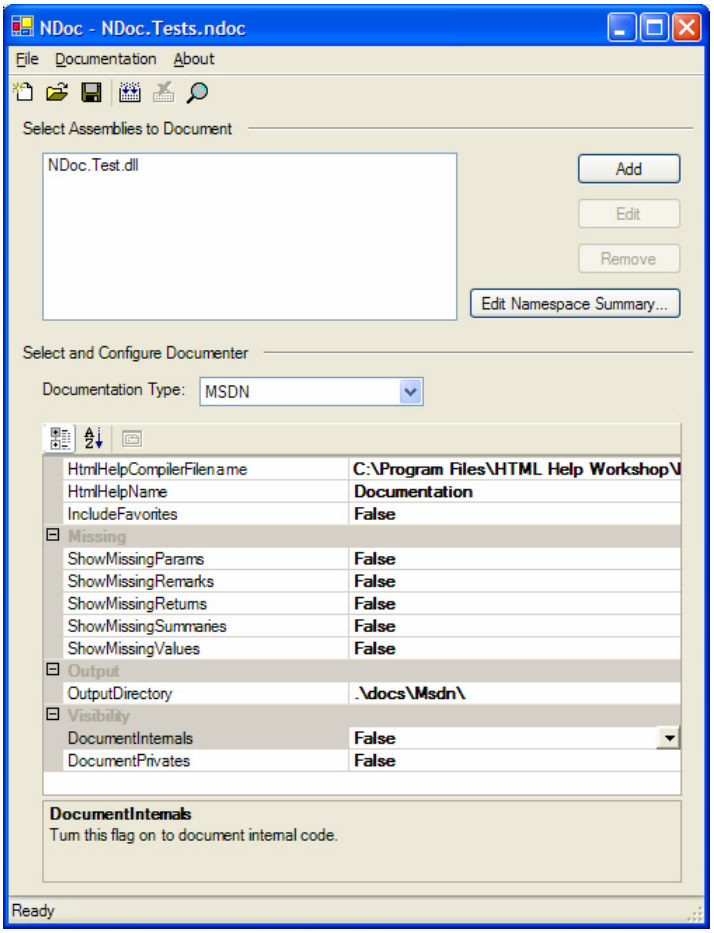


Abbildung 50: GUI von NDoc

Als Ergebnis erhält man eine Dokumentation im MSDN Stil (HTML) oder CHM Format.

A2. CD – ROM