SAARLAND UNIVERSITY
COMPUTER SCIENCE

Universität des Saarlandes
Lehrstuhl für Programmierung
Prof. Sebastian Hack
Dipl.-Inform. Christoph Mallon

# Projekt: Raytracer

## 1 Allgemeines

In diesem Projekt ist es Ihre Aufgabe einen Strahlenverfolger (engl. "Raytracer") zu schreiben. Ziel dieses Projektes ist, praktische Erfahrung im Bereich objektorientierter Entwurf und Programmierung zu sammeln. Sie erhalten das Programmierprojekt unter

```
https://prog2scm.cdl.uni-saarland.de/git/project4/$NAME
```

`$NAME` ist durch Ihren Benutzernamen zu ersetzen.

Sie könnnen das Projekt mittels *Eclipse* bearbeiten. Führen Sie folgende Schritte in Eclipse durch, um das Projekt zu importieren:

1. Wählen Sie *File->Import...*.
2. Wählen Sie *General->Existing Projects into Workspace*.
3. Geben Sie mittels *Browse...* das Verzeichnis, in das Sie das Projekt geklont haben.
4. Schließen Sie das Importieren mit *Finish* ab.

Zum Testen benutzen wir hier ein Testrahmenwerk für Java namens *JUnit*. Es führt Regressionstests mit Ihrer Implementierung aus und berichtet jeweils das Ergebnis. Die Tests können von der Benutzeroberfläche mit einem Rechtsklick auf das Paket *seamcarving.test.student* und anschließendem Wählen von *Run As* und danach *JUnit Test* gestartet werden.

## 2 Aufgaben

Im Folgenden sind die zu bearbeitenden Aufgaben aufgeführt. Sie sind jeweils mit Verweisen auf die betreffenden Abschnitte mit Erläuterungen versehen. Abschnitt 3 vermittelt hierzu die Grundlagen.

- Implementieren Sie die Klassen *Plane* (4.2.2) und *Sphere* (4.2.3). Sie können sich an der bestehenden Implementierung in der Klasse *Triangle* orientieren. Implementieren Sie dazu ebenfalls die Methoden der statischen Fabrik für Objekte *GeomFactory*.
- Implementieren Sie die Methode *hashCode()* und *equals(Object)* für die Klassen *Plane* und *Sphere*.
- Implementieren Sie die Methode *shade()* der Klassen *CheckerBoard* (4.5.2) und *Phong* (4.5.3). Sie können sich an der bestehenden Implementierung in der Klasse *SingleColor* orientieren. Für Shader gibt es eine statische Fabrik *ShaderFactory*.

- Implementieren Sie die Methode *read()* der Klasse *OBJReader* zum Einlesen von OBJ-Dateien (5.1).
- Implementieren Sie die Klasse *BVH* der Bounding-Volume-Hierarchy-Beschleunigungsstruktur (4.7).

### Hinweise

- Sie können den Raytracer als normales Programm starten. Als Ausgabe erscheint ein Bild, das (zunächst nur) ein gelbes Dreieck enthält.
- in *Main.main()* befinden sich mehrere boolesche Variablen, mit denen Sie weitere Objekte und Effekte in der Szene anschalten können, sobald Sie den entsprechenden Teil implementiert haben. Zum Beispiel nach der Implementierung der Klasse *Sphere* und setzen von *implementedSphere* auf *true* erscheinen zwei Kugeln. Auf diese Weise können Sie leicht ihren Fortschritt begutachten.
- In Anhang A befindet sich ein Referenzbild, welches nach der Implementierung aller relevanten Teile generiert wurde.
- Benutzen Sie für Vergleiche mit der Zahl 0 immer die Konstante *EPS* der Klasse *Constants*. Ein Vergleich der Form $i <= 0$ wird somit zu dem Vergleich $i <$ EPS. Testen sie Beispielsweise eine Zahl auf 0, können sie die Methode *isZero* dieser Klasse verwenden.
- Alle Teile, die zu vervollständigen sind, befinden sich im Ordner *impl*. Auch neue Klassen sind nur dort hinzuzufügen. Vegessen Sie nicht, neue Javadateien auch in git hinzuzufügen!

# 3 Grundlagen

Das Wort „Raytracing“ heißt übersetzt „Strahlenverfolgung“. Die Grundidee ist, natürliche Seheindrücke nachzubilden. In der Realität senden Lichtquellen, wie die Sonne, Lichstrahlen in den Raum. Diese Lichtstrahlen werden auf vielerlei Weisen verändert. So werden sie von matten Oberflächen (z.B. Wänden) gestreut, von glatten Oberflächen (z.B. Spiegel) reflektiert oder von durchsichtigen Objekten (z.B. Wasser) gebrochen. Manche Lichtstrahlen treffen auf das Auge eines Beobachters und werden schlussendlich als Bild wahrgenommen.

## 3.1 Modell

Dies genau nachzubilden ist extrem aufwendig, da die meisten Lichtstrahlen nicht den Beobachter erreichen. Daher wird ein Ansatz in Gegenrichtung verfolgt: Zunächst wird der Beobachter als Kamera mit einer Projektionsebene, auf der das resultierende Bild gezeichnet wird, modelliert. Es werden nun „Sehstrahlen“ von der Kamera aus in die Szene geschickt. Diese treffen dort möglicherweise auf Objekte. Deren Farbe wird an den getroffenen Stellen berechnet und im resultierenden Bild (der Projektionsebene) eingetragen. Dies ist in Abbildung 1 veranschaulicht. Um Effekte wie Beleuchtung oder Reflektion nachzubilden, werden von den getroffenen Stellen aus weitere Strahlen (z.B. in Richtung von Lichtquellen zur Schattenberechnung) in die Szene geschossen.

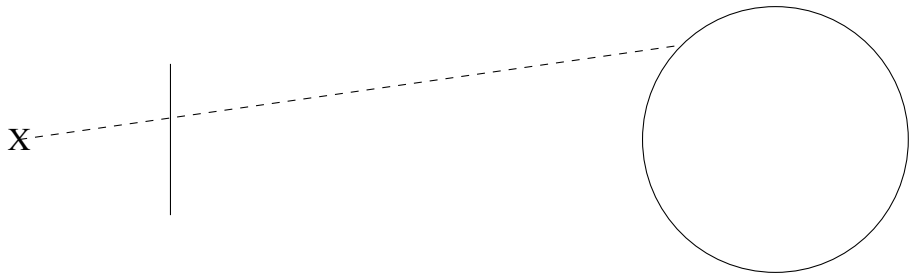


Abbildung 1: Ein Sehstrahl beginnend bei der Kamera durchwandert die Projektionsebene und trifft auf eine Kugel.

## 3.2 Lineare Algebra

In diesem Abschnitt werden die für das Projekt notwendigen Grundlagen in linearer Algebra kurz abgehandelt. Diese finden dann bei der Treffer- und Farbberechnung Anwendung.

### 3.2.1 Punkte

Ein Punkt bezeichnet einen Ort in einem Koordinatensystem relativ zu dessen Ursprung. Als Namen werden im Weiteren Kleinbuchstaben ($a$) verwendet. In einem dreidimensionalen Koordinatensystem wird ein Punkt eindeutig durch drei Zahlen beschrieben.

### 3.2.2 Vektoren

Vektoren geben mittels dreier Zahlen eine Verschiebung in einem dreidimensionalen Raum an. Vektoren werden als Kleinbuchstaben mit einem Pfeil darüber ($\vec{a}$) geschrieben. Im Folgenden sind die wichtigsten Operationen mit Vektoren aufgeführt:

- Die elementweise Summe zweier Vektoren ergibt eine kombinierte Verschiebung.
- Die elementweise Differenz zweier Punkte $a - b$ ergibt einen Vektor, der den direkten Weg von $b$ nach $a$ angibt.
- Die elementweise Multiplikation zweier Vektoren wird mit dem Operatorsymbol $*$ dargestellt.
- Ein Vektor kann elementweise mit einer Zahl $k$ multipliziert werden, um diesen zu stauchen ($|k| < 1$) oder zu dehnen ($|k| > 1$). Ist der Faktor negativ, so zeigt der resultierende Vektor in die Gegenrichtung. Können zwei Vektoren mit einer solchen Skalarmultiplikation in einander überführt werden, werden sie als linear abhängig bezeichnet.
- Ein Vektor elementweise addiert zu einem Punkt liefert einen um diesen Vektor verschobenen Punkt.
- Die Länge eines Vektors wird gemäß dem Satz von Pythagoras berechnet:

  $$|\vec{a}| = \sqrt{a_1^2 + a_2^2 + a_3^2} = \sqrt{\vec{a}^2}$$

  Die rechte Berechnungsvariante verwendet das Skalarprodukt, siehe Abschnitt 3.2.3. Vektoren der Länge $1$ werden als Einheitsvektoren bezeichnet.

### 3.2.3 Skalarprodukt

Das Skalarprodukt ist eine Verknüpfung zweier Vektoren und liefert als Ergebnis eine Zahl, die angibt, welchen Winkel diese einschließen:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos \alpha$$

Da wir mit einem dreidimensionalen kartesischen Koordinatensystem arbeiten, berechnet sich das Skalarprodukt als Summe der elementweisen Produkte der Vektoreinträge:

$$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

Manchmal wird, analog zur skalaren Multiplikation, der Punkt weggelassen. Entsprechend wird $\vec{a}^2$ für das Skalarprodukt eines Vektors mit sich selbst geschrieben. Teilt man das Skalarprodukt durch das Produkt der Länge der Vektoren ergeben sich (gemäß dem Kosinus) folgende speziellen Werte:

- $1$: Die beiden Vektoren sind parallel und zeigen in dieselbe Richtung.
- $-1$: Die beiden Vektoren sind parallel und zeigen in entgegengesetzte Richtung.
- $0$: Die beiden Vektoren stehen senkrecht zueinander.

Werte dazwischen geben an, wie sehr die beiden Vektoren in dieselbe ($> 0$) oder in entgegengesetzte ($< 0$) Richtung zeigen.

### 3.2.4 Kreuzprodukt

Das Kreuzprodukt berechnet im dreidimensionalen Raum aus zwei Vektoren einen dritten, der senkrecht auf beiden Vektoren steht (Skalarprodukt mit den beiden Vektoren ist 0). Da die Länge des Ergebnisvektors abhängig vom Winkel ist, den die beiden gegebenen Vektoren einschließen, muss das Ergebnis üblicherweise noch normalisiert werden. Es wird folgendermaßen berechnet:

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

## 3.3 Koordinatensysteme

### 3.3.1 Weltkoordinaten

Wir verwenden im Projekt ein linkshändiges Koordinatensystem. D.h. die X-Richtung zeigt nach rechts, die Y-Richtung nach oben und die Z-Richtung weist vom Betrachter weg. Anschaulich ist dies – zusammen mit einer Erklärung der Bezeichnung „linkshändig“ – in Abbildung 2 zu sehen.

### 3.3.2 Texturkoordinaten

Während die Objekte sich in einer dreidimensionalen Szene befinden betrachten wir nur zweidimensionale Texturen. Legt man nun eine Textur auf ein Objekt, so muss man für den betreffenden Schnittpunkt die zugehörigen Texturkoordinaten berechnen. Diese werden üblicherweise mit $U$ und $V$ bezeichnet. Ein *Hit* liefert diese Koordinaten mittels *getUV()*.

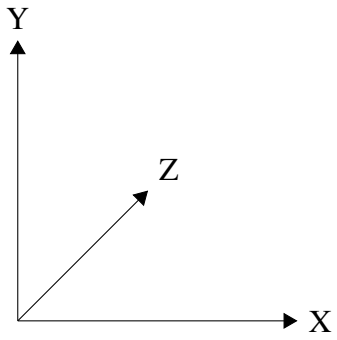


Abbildung 2: Ein linkshändisches Koordinatensystem: Der Daumen der linken Hand zeigt in X-Richtung, der Zeigefinger in Y-Richtung und der Mittelfinger in Z-Richtung.

# 4 Modellierung

## 4.1 Kamera

Eine Kamera beobachtet die Szene. Von ihr aus werden Strahlen in die Szene geschossen. Am Auftreffpunkt eines jeden Strahls wird abhängig vom getroffenen Objekt, seiner Oberfläche, den dort herrschenden Lichtverhältnissen usw. eine Farbe bestimmt. Hierfür werden häufig weitere Strahlen in die Szene geschossen, um z.B. zu bestimmen, ob eine Lichtquelle den Auftreffpunkt des ersten Strahls erreicht. Die so bestimmte Farbe wird in das resultierende Bild eingetragen wird. Wir verwenden im Projekt eine bereits fertige perspektivische Kamera.

## 4.2 Objekte

Als geometrische Primitive sind Ebenen (4.2.2) und Kugeln (4.2.3) beliebt, da die Berechnungen, ob sie an einer Stelle sichtbar sind, sehr einfach sind. Häufig werden auch Dreiecke angeboten, da diese üblicherweise die Grundlage von Polygonmodellen sind. Die Berechnung hierfür ist ähnlich der von Ebenen, benötigt jedoch weitere Prüfungen.

Objekte in der Szene besitzen die zwei wichtigen Methoden *hitTest()* und *shade()*. Mittels *hitTest()* wird festgestellt, ob ein Strahl das Objekt trifft. Ein weiteres wichtiges Resultat dieser Methode ist, in welcher Entfernung von der Kamera das Objekt getroffen wird. Das Objekt, das am nähesten zur Kamera getroffen wird, wird darauf zur Berechnung der Farbe des Bildpunktes auf der Projektionsebene verwendet. Dies geschiet mittels *shade()*, was in Abschnitt 4.5 behandelt wird.

### 4.2.1 Strahlen

Wir betrachten zunächst Strahlen, da wir diese im Weiteren mit den Objekten in der Szene schneiden, um Auftreffpunkte zu bestimmen. Ein Strahl wird eindeutig durch einen Punkt $p_s$, der den Startpunkt des Strahls angibt, und einen Vektor $\vec{v}_s$, der die Richtung des Strahls angibt, beschrieben. Alle Punkte $x$, die folgende Gleichung erfüllen, liegen auf dem Strahl:

$$x = p_s + \lambda \vec{v}_s, \lambda \geq 0$$

$\lambda$ gibt hierbei die Entfernung vom Startpunkt $p_s$ an. Für $\vec{v}_s$ wird ein Einheitsvektor verwendet.

### 4.2.2 Ebenen

Eine Ebene wird intuitiv durch drei Punkte, die nicht auf einer Geraden liegen, eindeutig beschrieben: Zum Beispiel kann man ein Buch (die Ebene) ohne Wackeln (eindeutig) auf drei Fingerspitzen (die Punkte) balancieren.

Diese Form eignet sich jedoch nicht zum Schneiden mit einem Strahl. Hierfür wird die Hesse-Normalform verwendet. Sie besteht nur aus einem Vektor $\vec{n}_e$, der senkrecht auf der Ebene steht, und dem Abstand $d_e$ der Ebene vom Ursprung. Jeder Punkt $x$, der die folgende Gleichung erfüllt, liegt in der Ebene:

$$x\vec{n}_e - d = 0$$

Aus den drei Punkten wird zunächst ein beliebiger Punkt $p_e$ gewählt und zwei Vektoren $\vec{u}_e$ und $\vec{v}_e$ (Weg vom gewählten Punkt zu den beiden übrigen Punkten) berechnet. Die beiden Vektoren werden normalisiert. Diese Darstellung spannt ebenso die Ebene auf:

$$x = p_e + \lambda\vec{u}_e + \mu\vec{v}_e, \lambda, \mu \in \mathbb{R}, |\vec{u}_e| = |\vec{v}_e| = 1$$

Nun wird mittels des Kreuzproduktes der Normalenvektor aus den beiden Vektoren berechnet:

$$\vec{n}'_e = \vec{u}_e \times \vec{v}_e$$

Das Ergebnis des Kreuzproduktes ist meistens kein Einheitsvektor, daher muss es noch normalisiert werden:

$$\vec{n}_e = \vec{n}'_e \frac{1}{|\vec{n}'_e|}$$

Nun ist nur noch der Abstand $d_e$ der Ebene vom Ursprung zu berechnen. Da $p_e$ in der Ebene liegt, geschieht dies durch Einsetzen in und Umformen der Hesse-Normalform:

$$p_e\vec{n}_e = d$$

Der Schnitt eines Strahls mit einer Ebene berechnet sich nun folgendermaßen:

$$\begin{aligned}
x &= p_s + \lambda\vec{v}_s && \texttt{(Strahl)} \\
0 &= x\vec{n}_e - d_e && \texttt{(Ebene)} \\
0 &= (\lambda\vec{v}_s + p_s)\vec{n}_e - d && \texttt{(Einsetzen)} \\
0 &= \lambda\vec{v}_s\vec{n}_e + p_s\vec{n}_e - d \\
\lambda\vec{v}_s\vec{n}_e &= d - p_s\vec{n}_e \\
\lambda &= \tfrac{d - p_s\vec{n}_e}{\vec{v}_s\vec{n}_e}, \vec{v}_s\vec{n}_e \neq 0 \wedge \lambda \geq 0
\end{aligned}$$

Ist der Nenner $0$, so ist der Strahl parallel zur Ebene und somit wird die Ebene nicht getroffen. Ansonsten gibt $\lambda$ die Entfernung vom Startpunkt des Strahls an, in der die Ebene getroffen wird. $\lambda$ darf weiterhin nicht negativ sein, sonst liegt der Schnittpunkt hinter dem Startpunkt des Strahls.

**Hinweis** Verwenden Sie die bereits implementierte Methode *geom.Util.computePlaneUV()*, um *getUV()* zu vervollständigen. Die Parameter von *computePlaneUV()* sind die Normale und der Aufpunkt der Ebene sowie der Auftreffpunkt.

In der *hit()* Methode werden Parameter *tmin* und *tmax* übergeben. Diese beschreiben einen Bereich als Abstand vom Startpunkt des Ray in dem ein hit "gültig" ist.

### 4.2.3 Kugeln

Alle Punkte $x$, die sich im Abstand $r_k$ vom Mittelpunkt $c_k$ einer Kugel befinden, bilden die Oberfläche einer Kugel:

$$|x - c_k| = r_k$$

Da $r_k$ nicht negativ ist, kann man die Quadratwurzelberechnung in der Längenbestimmung des Vektors durch quadrieren der Gleichung einsparen:

$$(x - c_k)^2 = r_k^2$$

Nach Einsetzen der Gleichung für einen Strahl erhält man:

$$\begin{aligned} r_k^2 &= (p_s + \lambda\vec{v}_s - c_k)^2 \\ 0 &= \lambda^2\vec{v}_s^2 + 2\lambda\vec{v}_s(p_s - c_k) + (p_s - c_k)^2 - r_k^2 \end{aligned}$$

$\lambda$ kann nun mit der üblichen Lösungsformel für gemischt quadratische Gleichungen bestimmt werden:

$$\begin{aligned} a &= v_s^2 = 1 \qquad \texttt{(Einheitsvektor)} \\ b &= 2\vec{v}_s(p_s - c_k) \\ c &= (p_s - c_k)^2 - r_k^2 \\ \lambda_{1,2} &= \tfrac{-b \pm \sqrt{b^2 - 4c}}{2} \end{aligned}$$

Ist die Diskriminante (Wert unter der Wurzel) nicht negativ, so schneiden sich der Strahl und die Kugel. Weiter darf die kleinere Lösung für $\lambda$ nicht negativ sein, sonst liegt der Schnittpunkt abermals hinter dem Startpunkt des Strahls. Die größere Lösung ist immer zu ignorieren, da sie dem Treffen der Kugel von innen auf der Rückseite entspricht.

**Hinweis** Verwenden Sie die bereits implementierte Methode *geom.Util.computeSphereUV()*, um *getUV()* zu vervollständigen. Der Parameter von *computeUV()* ist der Vektor vom Mittelpunkt der Kugel zum Auftreffpunkt.

In der *hitTest()* Methode werden Parameter *tmin* und *tmax* übergeben. Diese beschreiben einen Bereich als Abstand vom Startpunkt des Ray in dem ein hit "gültig" ist.

## 4.3 Bounding Box

Eine Bounding Box repräsentiert einen virtuellen Quader im Raum, der Objekte umschließt. Im Basisfall wird genau ein Objekt umschlossen, z.B. die Bounding Box einer Kugel hat als Seitenlängen genau den Durchmesser der Kugel. Um Schnitte mit einer Bounding Box schnell berechnen zu können, sind diese immer an den Achsen des Koordinatensystems ausgerichtet. Die Nutzung von Bounding Boxes zur Beschleunigung der Bildberechnung wird im Abschnitt 4.7 behandelt.

## 4.4 Lichtquellen

Zur Beleuchtung der Szene können verschiedenartige Lichtquellen verwendet werden, die jeweils andere Aspekte echter Lichtquellen annähern.

Punktlichtquellen senden von einer Stelle im Raum gleichmäßig in alle Richtungen Licht aus. Sie eignen sich zur Simulation von z.B. Glühbirnen. Beleuchtung wird weiter im Abschnitt 4.6 beschrieben.

Umgebungslichtquellen erhellen gleichmäßig jede Stelle eines Objektes. Sie sind eine krude Simulation von Streulicht. So ist ein Raum, in den Licht nur durch ein Fenster fällt, nicht nur an der Stelle, an der das Licht auftrifft, beleuchtet. Durch Lichtstreuung sind andere Teile leicht erhellt. Diesen Effekt realitätsnahe zu simulieren ist sehr aufwendig, daher begnügt man sich manchmal mit dem einfachen Trick einer Umgebungslichtquelle. Da an verschiedenen Stellen einer Szene verschiedene Umgebungslichtverhältnisse herrschen können wird dieser Aspekt nicht als Lichtquelle, sondern als Teil eines Shaders, hier Phong (4.5.3), umgesetzt.

## 4.5 Shader

Shader verleihen den Oberflächen von Objekten ihr Aussehen. Wir verwenden in diesem Projekt drei typische Shader.

### 4.5.1 Einfache Farbe

Der einfachste Shader liefert stets eine konstante Farbe unabhängig von anderen Gegebenheiten, wie z.B. Lichtquellen. Dieser kann dann als Grundlage für andere Shader verwendet werden. Er ist aufgrund seiner Einfachheit auch gut zum Testen und zur Fehlersuche geeignet.

### 4.5.2 Schachbrett

Ein beliebter Shader ist das Schachbrett. Hierbei werden auf einem Raster abwechselnd zwei weitere Shader angezeigt.

Welcher Shader zu verwenden ist folgt aus den gegebenen Texturkoordinaten $u$ und $v$ sowie einem Skalierungsfaktor $s$, der die Größe des Rasters angibt:

$$x = \lfloor u/s \rfloor + \lfloor v/s \rfloor$$

Ist $x$ gerade, so wird der erste Shader verwendet, ansonsten der zweite.

### 4.5.3 Phong

Der Phong-Shader, benannt nach seinem Erfinder, gibt einer Oberfläche ein plastikartiges Aussehen. Sein Resultat berechnet sich aus drei Teilen:

$$I_{\texttt{Phong}} = I_{\texttt{ambient}} + I_{\texttt{diffuse}} + I_{\texttt{specular}}$$

**Umgebungslicht ($I_{\texttt{ambient}}$)** Zur Beleuchtung wird eine feste Farbe als Beitrag des Umgebungslicht auf die Resultatfarbe addiert.

**Diffuse Streuung ($I_{\texttt{diffuse}}$)** Hiermit wird Streuung an einer rauhen Oberfläche simuliert. Das Ergebnis hängt von der Farbe der Lichtquelle $c_l$, der Farbe des darunterliegenden Shaders $c_{\texttt{sub}}$, dem Auftreffwinkel (je steiler desto heller) des Strahls auf die Oberfläche und einer Materialkonstante $k_{\texttt{diffuse}}$ ab. Dieser Betrag wird für jede Lichtquelle, die diese Stelle erreicht, berechnet. Das bedeutet, dass der Strahlweg zur jeweiligen Lichtquelle nicht verdeckt ist (4.6) und das Licht von vorne (Skalarprodukt aus Normale der Oberfläche $\vec{n}$ und Vektor zur Lichtquelle $\vec{v}$ größer 0) auf die Oberfläche trifft. Beide Vektoren müssen normalisiert sein.

$$I_{\texttt{diffuse}} = (c_l * c_{\texttt{sub}}) k_{\texttt{diffuse}} \mathit{max}(0, \vec{n}\vec{v})$$

**Spiegelung ($I_{\texttt{specular}}$)** Dieser Aspekt simuliert spiegelnde Reflektion von Licht. Das Ergebnis ist abhängig von der Farbe der Lichtquelle $c_l$ und dem Winkel, den der Vektor $\vec{v}$ vom Auftreffpunkt zur Lichtquelle und der an der Normalen der Oberfläche gespiegelte Sehstrahl $\vec{r}$, einschließen. Weiter wichtig sind die Materialkonstanten des Reflektionsfaktors ($k_{\texttt{specular}}$) und der Glanzfaktor ($n$). Der Glanzfaktor („shininess“) beschreibt die Glätte der Oberfläche. Je kleiner der Wert ist, desto rauher erscheint die Oberfläche, $n = \infty$ entspricht einem perfekten Spiegel.

$$I_{\texttt{specular}} = c_l k_{\texttt{specular}} \mathit{max}(0, \vec{r}\vec{v})^n$$

## 4.6 Beleuchtung

Bevor eine Lichtquelle zur Beleuchtung eines Objekts herangezogen wird, muss getestet werden, ob das Licht der Lichtquelle das Objekt überhaupt erreichen kann. Hierzu wird ein weiterer Strahl – ein Schattenstrahl – vom berechneten Auftreffpunkt des Sehstrahls in Richtung der Lichtquelle geschossen. Wird auf dem Weg ein anderes Objekt getroffen, so ist die Lichtquelle von diesem verdeckt und trägt damit nicht zur Beleuchtung des Objektes bei.

## 4.7 Beschleunigungsstruktur

Jeder Strahl, der in die Szene geschossen wird, muss auf Schnitt mit jedem Objekt in der Szene getestet werden. Bei komplexen Szenen ist dies sehr langsam. Mit einer einfachen Überlegung kann man dies enorm beschleunigen: Man packt mehrere Objekte in eine Kiste, die jene umgibt. Trifft ein Strahl die Kiste nicht, so erübrigt sich der Test der einzelnen Objekte in der Kiste. Diese Idee kann nun rekursiv weiter verfeinert werden. Die Kiste wird wieder in zwei etwa gleich große Teil-Kisten zerschnitten, die jeweils einen Teil der Objekte enthalten. Nun kann man sie immer weiter unterteilen, bis nur noch wenige Objekte in jeder Kiste enthalten sind.

So enthält jede Kiste entweder zwei Teil-Kisten oder eine kleine Anzahl Objekte. Auf diese Weise müssen statt linear vieler Objekte nur etwa logarithmisch viele getestet werden. Diese Aufgabe bewältigen wir in mehreren Teilschritten.

Zuerst wird die Größe der zu unterteilenden Kiste berechnet in *calculateMinMax()* berechnet. Hier wird die kleinste und größte Koordinate berechnet, die eine Ecke (wir verwenden *getMin()*) aller Bounding Boxes der enthaltenen Objekte einhüllt.

Dann bestimmen wird die Dimension, die unterteilt wird, in *calculateSplitDimension()*. Deren Ergebnis ist der Index der Achse, die beim übergebenen Vektor am größten ist.

Darauf werden die Objekte in *distributeObjects()* in zwei neue BVHs verteilt. Ist die *getMin()*-Ecke einer Bounding Box eines Objekts in der zuvor bestimmten Unterteilungsdimensi-

on kleiner als der Mittelpunkt zwischen dem zuvor bestimmten Minimum und Maximum, so wird das Objekt in die erste Unter-BVH eingefügt, ansonsten in die zweite.

Die zwei gefüllten BVHs werden bei Bedarf weiter unterteilt. Sie werden zuletzt als einzige Objekte in der BVH vermerkt.

# 5 Modelle

Es ist unpraktisch eine Szene fest im Quelltext des Raytracers zu verankern. Daher wurden viele Austauschdateiformate erfunden. Unser Raytracer soll eines davon (auszugsweise) einlesen können.

## 5.1 Wavefront OBJ-Dateien

```
# Kommentar
v 0.0 0.0 0.0
v 1.0 0.0 0.0
v 0.0 1.0 0.0
v 1.0 1.0 0.0
f 1 2 3
f 3 2 4
```

Abbildung 3: Eine OBJ-Datei, die ein Quadrat bestehend aus zwei Dreiecken beschreibt.

Wavefront OBJ-Dateien beschreiben Polygonmodelle. Wir beschränken uns auf Modelle bestehend aus Dreiecken.

OBJ-Dateien bestehen aus eine Liste von Definitionen für Eckpunkte, Seiten, Normalen, Texturkoordinaten usw. Jede Definition nimmt eine Zeile ein und beginnt mit einem Wort, das die Art der Definition beschreibt. Ein Beispiel einer OBJ-Datei, die ein Quadrat bestehend aus zwei Dreiecken beschreibt, ist in Abbildung 3 zu sehen. Wir betrachten nur Eckpunkte und Seiten, deren Aufbau im Folgenden beschrieben ist. Alle anderen Definitionen und Kommentarzeilen (beginnend mit einem #) sind zu ignorieren.

### 5.1.1 Eckpunkte

Definitionen für Eckpunkte beginnen mit dem Wort *v* (Vertex), worauf drei Gleitkommazahlen folgen, die die X-, Y-, und Z-Koordinaten des Eckpunktes angeben. Die Eckpunkte werden in eine beim Index 1 beginnende Liste für spätere Verwendung in Seiten eingetragen.

### 5.1.2 Seiten

Definitionen für Seiten beginnen mit dem Wort *f* (Face). Darauf folgen mindestens drei Indizes in die Liste der Eckpunkte, welche die Eckpunkte des Polygons darstellen. Um einen Eckpunkt in einer Seite zu verwenden, muss dieser vor der Seite definiert worden sein. Wir beschränken uns im Projekt auf Dreiecke, d.h. es werden nur Seiten mit drei Indizes verwendet.

**Hinweis** Verwenden Sie zum einlesen der Datei *java.util.Scanner*. Nachdem ein Scanner erzeugt wurde, rufen Sie die Methode *useLocale()* mit dem Argument *Locale.ENGLISH* auf, da sonst Punkte in der einzulesenden Datei nicht als Dezimaltrennzeichen erkannt werden.

# A Referenzbild

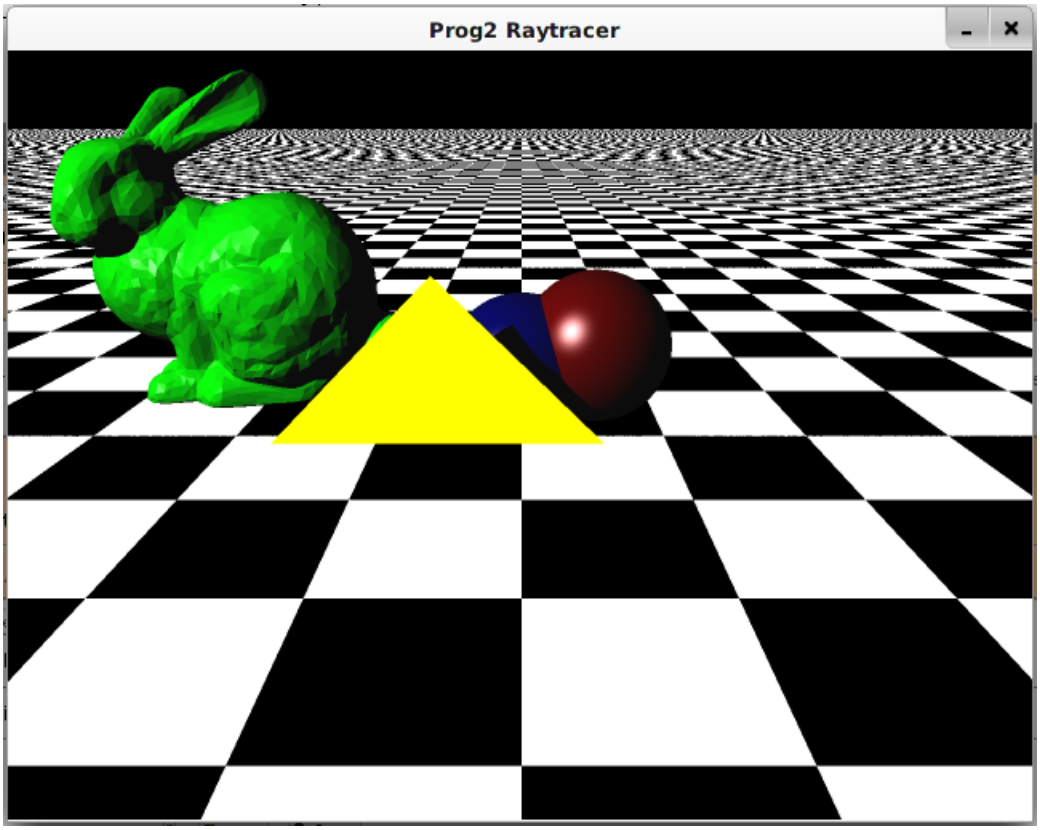


Abbildung 4: Referenzbild