

Projekt-Protokoll - Computergrafik SS2015

David Derichs, Jonathan Stoye, Viktoria de Koning

27. Juli 2015

Beschreibung

In diesem Dokument wurden Erfahrungen, allgemeine Herangehensweise und Schritte der Aufgabenbewältigung zusammengefasst. Die Ergebnisse der Arbeit wurden im Rahmen des Kurses „Computergrafik I“ erarbeitet. Als Programmierumgebung wurde IntelliJ genutzt und die Programmiersprache Java kam dabei zum Einsatz. Das Protokoll ist nicht immer chronologisch, dabei wurde eher auf aufeinander basierende Entwicklungen im Sinne des Codes geachtet. Einige Klassen sind zwar früher entstanden als andere, konnten aber erst später getestet werden.

1 Vorbereitung

Ziel der Vorbereitung ist der Umgang mit den Klassen „Buffered Image, ColorModel, Rendered Image, Writable Master und ImageIO“ von Java. Sie sollen es dem Programm ermöglichen, die vom Tracer erzeugten Bilder darzustellen.

2 05.06.15 Die Klasse World

Die Klasse World beinhaltet alle Objekte, die in der Szene zur Kalkulation verwendet werden sollen. Außerdem bietet sie eine Hintergrundfarbe, was bei der späteren Lichtberechnung (Strahlverfolgung) wichtig ist. Der Konstruktor nimmt eine Hintergrundfarbe beim Aufruf des Konstruktors entgegen. Außerdem stellt die Klasse eine Methode hit() zur Verfügung. Diese ermittelt für alle in der Szene vorkommenden Objekte einen Schnittpunkt, der dem Betrachter am nächsten liegt und gibt diesen zurück.

2.1 09.06.15 Änderungen an der Klasse World

Nach Erstellung der Klassen für die Lichtquellen ist auch die Signatur der Klasse World verändert worden. Innerhalb der Szene werden nun auch alle Lichtquellen beachtet. Diese werden bei der Initialisierung des Objekts berücksichtigt.

3 05.06.15 Die Klasse Raytracer

In der Klasse Tracer werden die Szenen mit den relevanten Objekten „geladen“. Danach wird die Szene berechnet. Dem Tracer werden bei seiner Initialisierung die Parameter zur Höhe und Breite der anzuzeigenden Szene übergeben. Des Weiteren wurden sämtliche Testszenarien in dieser Klasse in Form von Testmethoden implementiert.

3.1 Testmethoden

Dies dient zur besseren Orientierung beim auswählen der Testmethoden. Diese sind im Zuge der Bearbeitung der Pflichtaufgaben entstanden.

Methoden

testAllsimpleScenes()
testScene1()
testScene2()
testScene3()
testScene4()
testScene5()
testScene6()

testAllLightningScenes()
testLightning1()
testLambertLightning()
testPhongLightning()
testPhongLightningDirectional()
testPhongLightningSpotLight()
testPhongLightningSpotLightandAmbientLight()

testAllMaterialScenes()
testSphere()
testAAB()

testAllTransformations()
testSphereTransformation()
testBoxTransformations()

testGeometries()
testDisc()

Beschreibung

Alle Testszenen zu Aufgabe „Schnittberechnung“
1. Testszene zu Aufgabe „Schnittberechnung“
2. Testszene zu Aufgabe „Schnittberechnung“
3. Testszene zu Aufgabe „Schnittberechnung“
4. Testszene zu Aufgabe „Schnittberechnung“
5. Testszene zu Aufgabe „Schnittberechnung“
6. Testszene zu Aufgabe „Schnittberechnung“

Alle Testszenen zur Aufgabe „Beleuchtung - 1“
1. Testszene zur Aufgabe „Beleuchtung - 1“
2. Testszene zur Aufgabe „Beleuchtung - 1“
3. Testszene zur Aufgabe „Beleuchtung - 1“
4. Testszene zur Aufgabe „Beleuchtung - 1“
5. Testszene zur Aufgabe „Beleuchtung - 1“
6. Testszene zur Aufgabe „Beleuchtung - 1“

Alle Testszenen der Aufgabe „Beleuchtung - 2“
1. Szene der Aufgabe „Beleuchtung - 2“
2. Szene der Aufgabe „Beleuchtung - 2“

Alle Testszenen der Aufgabe „Szenengraph“
1. Testszene der Aufgabe „Szenengraph“
2. Testszene der Aufgabe „Szenengraph“

Alle Testszenen zu eigenen Geometrien
Testszene der Geometrie „Disc“

4 10.06.15 Die Klasse Painter

In der Klasse „Painter“ wird genutzt, um ein Bild zu erzeugen. Dabei wird ein Rahmen erzeugt, welcher pixelweise gefüllt wird. Jeder Pixel wird einzeln erzeugt und erhält einen Farbwert. Die Klasse Painter soll vom Typ „JFrame“ sein. Dieser Frame soll eine Auflösung, also einen Wert für Höhe und Breite haben. Außerdem müssen die Pixel-Farben bekannt sein, damit diese später gezeichnet werden können. Das Fenster soll ein Menü haben, welches zum „Schließen“ des Programms und zum „Speichern“ des Bildes verwendet werden kann.

4.1 Variablen

Die Auflösung des Frames wird in den Variablen vom Typ Integer `imageWidth` und `imageHeight` gespeichert. Als Variablen für diese Klasse wird ein Array benötigt, das die Pixel-Farben beinhaltet. Dazu wurde die Variable „`int[] pixels`“ deklariert.

Problem

Es können keine Pixel gezeichnet werden.

Lösung

Man benötigt dafür ein Objekt der Klasse „Buffered Image“, das in der Lage ist, an bestimmten Punkten bzw. Pixeln Farbwerte zuzuweisen und darzustellen. Diese Erkenntnis hat ein wenig Recherche-Arbeit im Internet gekostet. Also wurde eine weitere Variable vom Typ „Buffered Image“ namens `image` hinzugefügt.

4.2 Konstruktor

Im Konstruktor werden drei Werte übergeben für Breite, Höhe und Farbwerte aller Pixel. Diese werden dem Painter-Objekt zugewiesen. Außerdem wird der „Frame“ also das sichtbare Fenster erzeugt.

Das Fenster bzw. der „Frame“ wird mit der Methode „`setupFrame`“ erzeugt.

Problem

Das `BufferedImage` lässt sich nicht zum Frame hinzufügen

Lösung

Um `image` darstellen zu können, muss es aber an den Frame geknüpft werden. Dazu muss ein `JPanel` verwendet werden. Es wurde daher die Klassen-Variablen „`panel`“ vom Typ „`JPanel`“ hinzugefügt.

In der Methode `setupFrame` konnte nun das Panel erzeugt und dem Frame zugewiesen werden. Dazu wurde die Methode `setupPanel()` implementiert.

Außerdem wurde die Methode `setupMenuBar()` implementiert, um die Menüleiste zu konfigurieren.

4.3 `setupPanel()`

Die Methode „`setupPanel`“ erzeugt ein Panel, welches im Frame liegt. Es bekommt dieselbe Dimension wie der Frame basierend auf der Auflösung. Nach der Deklaration wird das Panel dem Frame zugewiesen.

Problem

Das JPanel war nach der Implementierung nicht sichtbar.

Lösung

Es wurde eine Private Klasse MyJPanel als Unterklasse von JPanel erzeugt, welche die Methode „paint()“ überschreibt. Innerhalb dieser Methode wird zusätzlich die Methode „drawImage()“ ausgeführt, welche das BufferedImage neu zeichnet. Danach wurde das Panel inklusive Inhalt sichtbar.

4.4 weitere Methoden

Es wurde noch weitere Methoden implementiert, welche allerdings keine größeren Probleme bereitet haben. Darunter „setupMenuBar()“, welche die Menüpunkte „save“ und „->save as png“ bereitstellen und die Methode „draw()“, welche das BufferedImage pixelweise mit den Farbwerten des Pixel-Arrays füllt.

5 11.05.15 Die Klasse PaintDiagonal

Die Klasse PaintDiagonal ist so implementiert, wie die Klasse Painter. Sie füllt das BufferedImage allerdings nicht anhand eines übergebenen Arrays mit Farbwerten sondern setzt an den Punkten, bei denen während der Iteration der x- und y-Wert gleich sind, einen roten Pixel. Die Hintergrundfarbe ist dabei Schwarz.

6 28.04.15 Die Klasse ShowImage

Die Klasse ShowImage beinhaltet nur eine Variable vom Typ BufferedImage. Der Konstruktor testet, ob ein Bild-Objekt übergeben wurde oder zeigt einen Dialog zum Öffnen einer Datei. Danach wird das Bild mit der Methode „drawImage“ auf dem Frame dargestellt. Es gab bei der Erstellung dieser Klasse keine Probleme.

7 01.05.15 Die Klassen Mat3x3, Vector3, Point3 und Normal3

Die Klassen wurden zunächst, wie im Klassendiagramm beschrieben deklariert.

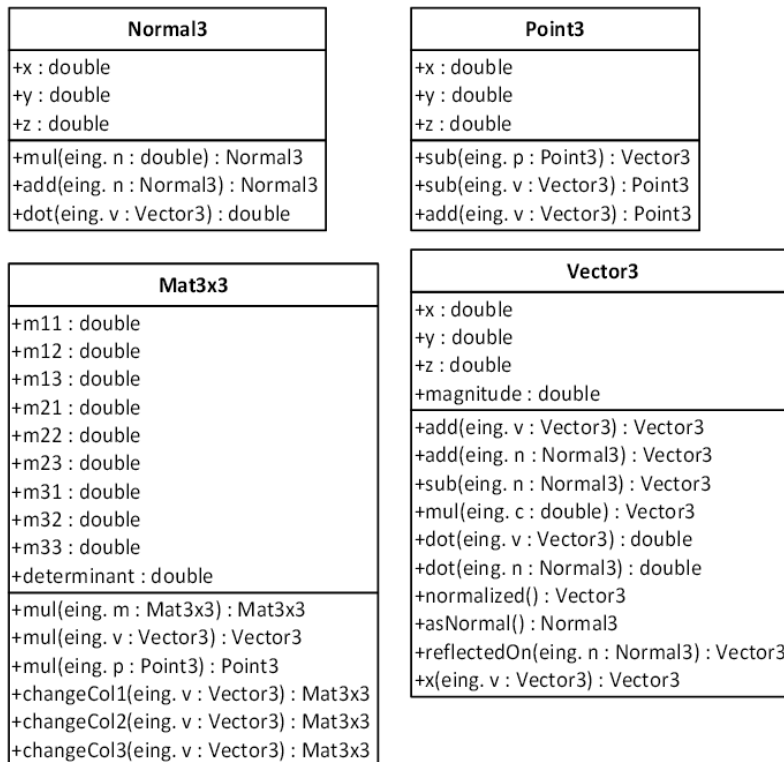


Abbildung 1: MatrixVector-Diagramm

Die Implementierung erfolgte anhand der allgemeinen Definitionen für die Rechenoperationen für Matrizen und Vektoren im 3-Dimensionalen Raum. Die Implementierung hat keine Probleme verursacht. Die Tests liefen bis auf einzelne Flüchtigkeitsfehler ohne Fehler durch und lieferten die erwarteten Ergebnisse.

8 05.06.15 Die Klasse Ray

Die Klasse Ray stellt einen Strahl dar. Sie hat einen Ursprung (Punkt - bzw. Ortsvektor) und einen Richtungsvektor. Außerdem kann ein Punkt auf dem Strahl mittels des Faktors t angegeben werden, welcher durch Multiplikation mit dem Richtungsvektor angegeben wird.

9 08.06.15 Die Klassen Camera, OrthographicCamera und PerspectiveCamera

Die Kamera-Klassen sind hierarchisch aufgebaut. Die Deklaration erfolgte anhand des folgenden Klassendiagramms.

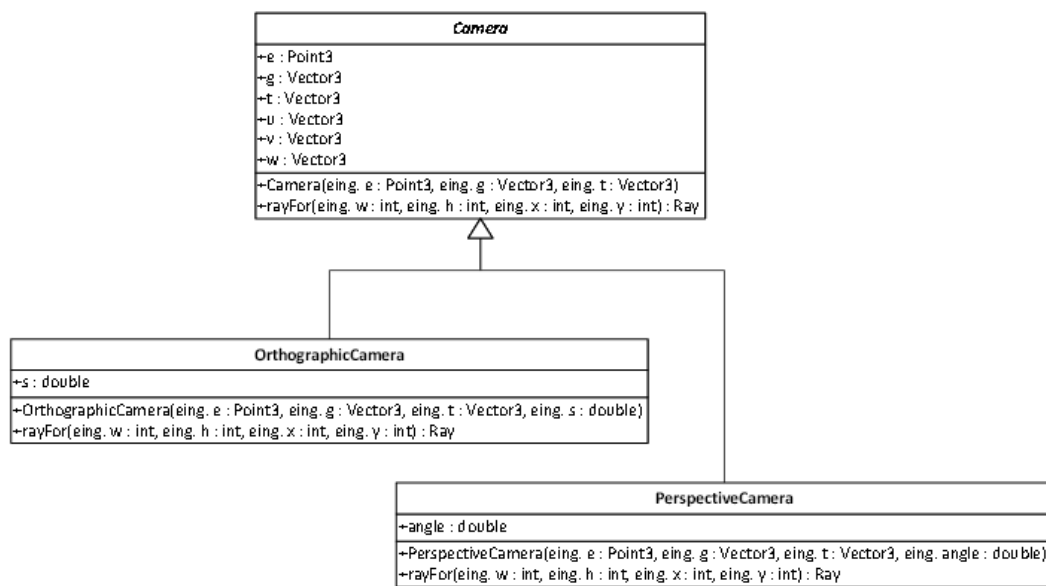


Abbildung 2: Camera-Diagramm

9.1 OrthographicCamera

Die Methode `rayFor()` soll einen Strahl zurückgeben, der auf den Kamera-Parametern basiert. Die Umsetzung folgender Formel hat lange gedauert und erforderte einige Umstellungen im Zuge der Implementierung:

$$\vec{o} = \vec{e} + a \cdot s \cdot \left(p_x - \frac{w-1}{w+1}\right) \cdot \vec{u} + s \cdot \left(p_y - \frac{h-1}{h+1}\right) \cdot \vec{v}$$

Für jeden Pixel, basierend auf der Auflösung der darzustellenden Szene, wird ein Strahl erzeugt, der den Regeln der orthographischen Formel unterliegt. Übergeben wird dabei die Höhe und Breite der Szene, sowie die Information, an welcher Position x und welcher Position y sich der Tracer gerade befindet. Für jede von x und y festgelegte Position wird hier ein einzelner Strahl, ausgehend von der orthographischen Kamera erzeugt und zurückgegeben.

Problem

Es wurde zunächst versucht, die Formel im Ganzen, wie in der Definition beschrieben, umzusetzen. Demzufolge gab es diverse Fehler und die Berechnung wurde sehr unübersichtlich.

Lösung

Es wurden diverse Variablen eingeführt, um die Formel übersichtlicher zu gestalten. Die Berechnung erfolgt am Ende der Methode rayFor() und alle Komponenten werden zusammengesetzt. Folgende Elemente wurden einzeln deklariert: $a = \frac{w}{h}$

$$xBracket = (p_x - \frac{\frac{w-1}{2}}{\frac{w-1}{2}})$$
$$yBracket = (p_y - \frac{\frac{h-1}{2}}{\frac{h-1}{2}})$$

Danach werden diese Teil-Formeln durch die Skalierung erweitert. Die Teil-Formel für die xBracket wurde zusätzlich um das Seitenverhältnis a erweitert. $xScalar = a \cdot s \cdot (p_x - \frac{\frac{w-1}{2}}{\frac{w-1}{2}})$

$$yScalar = s \cdot (p_y - \frac{\frac{h-1}{2}}{\frac{h-1}{2}})$$

Die Berechnung der Position wird dann gemäß der orthographischen Formel-Definition durchgeführt. $orthographicPosition = \vec{e} + (u \cdot xScalar) + (\vec{v}yScalar)$

Zur Übergabe wird ein Ray, also Strahl erzeugt. Sein Ursprung ist der errechnete Positions- bzw. Ortsvektor. Zusätzlich muss noch die Richtung angegeben werden, gegeben durch den umgedrehten Vektor des Kamera-Koordinatensystems \vec{w} .

9.2 PerspectiveCamera

Die Umsetzung der PerspectiveCamera war aufgrund der Erkenntnisse der Umsetzung der Orthografischen Kamera einfacher. Um die Probleme mit einer schwer nachvollziehbaren Formel im Code zu lösen, wurden die Klammern folgender Formel direkt aufgeteilt, in etwa passend zu dem Schema, wie in der Klasse OrthographicCamera. Folgende Formel liegt den Überlegungen zugrunde.

$$\vec{r} = -\vec{w} \cdot (\frac{\frac{h}{2}}{\tan \alpha}) + (p_x - \frac{w-1}{2}) \cdot \vec{u} + (p_y - \frac{h-1}{2}) \cdot \vec{v}$$

Es werden Variablen für folgende Teil-Formeln deklariert und initialisiert:

$$firstBracket = (\frac{\frac{h}{2}}{\tan \alpha}) \quad xBracket = (p_x - \frac{w-1}{2}) \quad yBracket = (p_y - \frac{h-1}{2})$$

Die Kalkulation des Ursprungsvektors wird dann mittels dieser Variablen durchgeführt.

$$r = -w \cdot firstBracket + xBracket + \vec{v} \cdot yBracket$$

Zurückgegeben wird dann ein Strahl mit dem Ursprung der Kamera und dem normierten Richtungsvektor r. Normiert deshalb, weil der Richtungsvektor mit einem Faktor multipliziert wird, um einen Punkt auf dem Strahl zu erreichen.

10 05.06.15 Die Klasse Color

Die Klasse Color wurde gemäß des folgenden Klassen-Diagramms deklariert.

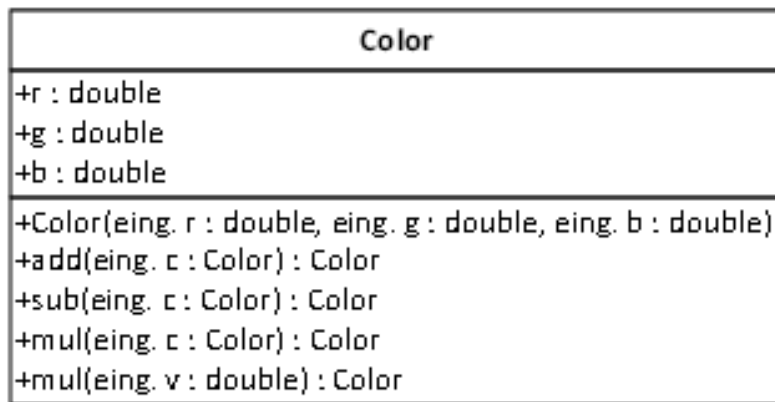


Abbildung 3: ColorDiagramm

Bei der Klasse Color ist festgelegt, in welcher Form die Daten für Farben im Tracer behandelt werden. In der Vorlesung wurden mehrere Modelle für die Umsetzung der RGB-Farbmodells beschrieben. Umgesetzt wurde die Variante, in der man ein einzelnes Array (für die darzustellenden Pixel) mit Farbwerten füllt, die in hexadezimaler Schreibweise repräsentiert werden. Dazu wurden in der Klasse Color Methoden implementiert, die die Farbwerte entweder einzeln oder insgesamt als Hexadezimal-Zahlen übergeben. Die RGB-Werte werden als Integer übergeben. Grobe Probleme sind bei der implementierung nicht aufgetreten.

11 06.06.15 Die Klassen zur „Geometrie“

Die Klassen zur Geometrie wurden nach folgendem Diagramm deklariert.

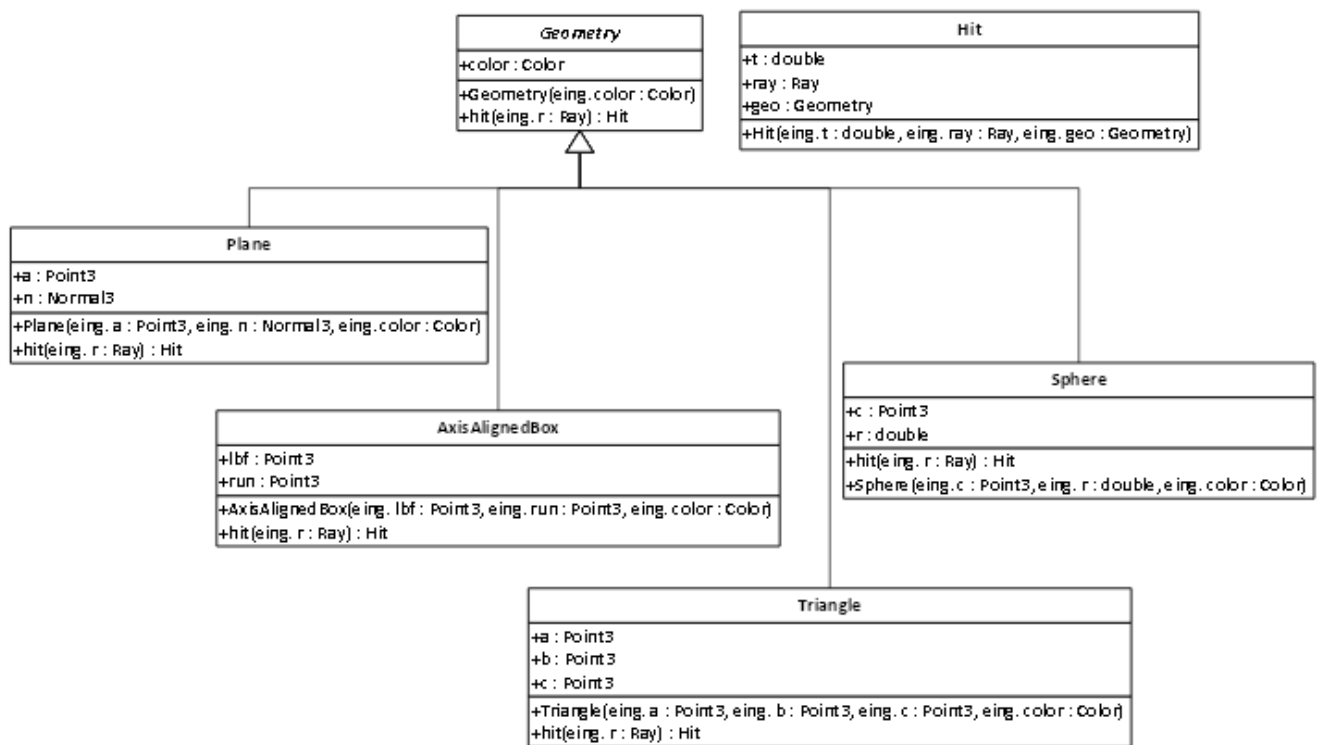


Abbildung 4: GeometrieDiagramm

Die Superklasse „Geometrie“ beerbt dabei folgende Unterklassen.

11.1 06.06.15 Die Klasse Plane

Die Klasse **Plane** repräsentiert eine Ebene im Raum. Die Schnittpunkte werden über die Ebene in implizierter Schreibweise ermittelt. Daher erhält jedes Objekt der Klasse **Plane** einen Punkt und eine Normale. Diese können zur Schnittpunkt-Berechnung genutzt werden. Außerdem erhält sie eine Farbe, später ersetzt durch das Material.

Um sicherzustellen, dass eine Berechnung überhaupt sinnvoll ist wurde folgendes geprüft.

$$\vec{d} \cdot \vec{n} \neq 0$$

Wenn \vec{d} (Richtung des Strahls) orthogonal zur Normalen, also parallel zur Ebene, dann gib es keinen Schnittpunkt.

Problem

Die Berechnung der Schnittpunkte ist recht schlecht verlaufen. Das lag an der falschen Umsetzung der Berechnung des Faktors t zur impliziten Ebene. Statt einer Multiplikation wurde das Kreuzprodukt verwendet.

Lösung

Das Kreuzprodukt wurde durch eine einfache Multiplikation ersetzt. Demnach wurde folgende Formel für die Schnittpunktberechnung benutzt.

$$t = \frac{(\vec{a}-\vec{o}) \cdot \vec{n}}{\vec{d} \cdot \vec{n}}$$

- o - Origin - Ursprung des Strahls
- n - Normal - Normale der Ebene
- a - Ortsvektor bzw. Punkt der Ebene
- d - Direction - Richtung des Strahls

Ist der Faktor t größer als 0, wird ein Hit-Objekt zurückgegeben. Diese Prüfung verhindert, dass es Schnittpunkte im Grenzbereich gibt.

11.2 06.06.15 Die Klasse AxisAlignedBox

Die Objekte der Klasse AxisAlignedBox bestehen aus einer Reihe von Ebenen, insgesamt 6, wie bei einem Würfel. Außerdem wird mittels eines Punktes spezifiziert, wo die linke, untere, weiter weg gelegene Ecke der Box ist. Das gleiche wird für die rechte, obere, nah gelegene Ecke festgelegt.

Im Konstruktor werden an diesen beiden Punkten jeweils drei Ebenen initialisiert. Deren Normalen zeigen jeweils in x-, y- bzw. z-Richtung. Das sorgt dafür, dass die Ebenen zusammengesetzt einen Würfel ergeben.

Problem

Um die Schnittpunkte zu berechnen bedient sich die Klasse einfach bei der Hit-Methode der Ebenen-Klasse. Aber es werden zu viele Treffer gefunden.

Lösung

Alle Hit-Objekte werden in eine Liste gespeichert. Danach werden alle Hits einzeln geprüft. Um die einzelnen Fälle voneinander trennen zu können, werden die jeweiligen Planes voneinander getrennt. Die erste Gruppe sind die Planes mit einer Normalen parallel zur x-Achse. Die Zweite Gruppe beinhalten die Planes mit einer Normalen in y-Richtung. Die dritte Gruppe dementsprechend mit einer Normalen in z-Richtung. Jeweils in positiver und auch in negativer Ausrichtung. Man hätte die Prüfung auch in einer einzigen Zeile bewerkstelligen können, aber das führt zu schier unlesbarem Code. Je nachdem wie die Ebenen im „Raum“ liegen, wird geprüft, ob der geschnittene Punkt noch innerhalb der „Grenzen“ der Box liegt. Dazu werden die jeweiligen x-, y- bzw. z-Werte des Schnittpunktes mit den Werten der beiden Initialisierungspunkte der Box verglichen. Wenn der Punkt innerhalb dieser beiden Punkte liegt, ist es ein möglicher sichtbarer Schnittpunkt.

Danach wird geprüft, welcher Schnittpunkt dem Betrachter am nächsten liegt und ob der Abstand zum Objekt groß genug ist, um es überhaupt erkennen zu können. Abschließend wird das darauf zutreffende Hit-Objekt, sofern es existiert zurückgegeben.

11.3 06.06.15 Die Klasse Triangle

Um die Schnittpunkte auf einem Dreieck zu finden wurde das Prinzip der Baryzentrischen Koordinaten verwendet. Diese sind Faktoren, die vor den jeweiligen Vektoren stehen, die zu einem der drei Eckpunkte des Dreiecks führen. Die Linearkombination dieser Vektoren bilden je nach Faktorisierung (Grenzwerte sind festgelegt) einen Punkt auf dem Dreieck. In der Vorlesung gab es zur Schnittpunktberechnung ein Lineares Gleichungssystem mit den drei Unbekannten β , γ und t .

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_0 \\ y_a - y_0 \\ z_a - z_0 \end{bmatrix}$$

Durch Berechnungen basierend auf der Determinante dieses in Form einer Matrix repräsentierten Gleichungssystems erhält man die drei Unbekannten Werte. Zunächst wird im Code überprüft, ob die Determinante der Matrix 0 ist, sprich ob das Gleichungssystem überhaupt lösbar ist. Zum Lösen wurde die Cramer'sche Regel verwendet. Dazu wurde folgendes Beispiel von Wikipedia als Anhaltspunkt verwendet.

Lineares Gleichungssystem 3. Ordnung

Diesem Beispiel liegt das folgende lineare Gleichungssystem zu Grunde:

$$82x_1 + 45x_2 + 9x_3 = 1$$

$$27x_1 + 16x_2 + 3x_3 = 1$$

$$9x_1 + 5x_2 + 1x_3 = 0$$

Die erweiterte Koeffizientenmatrix des Gleichungssystems ist dann:

$$(A \ b) = \left(\begin{array}{ccc|c} 82 & 45 & 9 & 1 \\ 27 & 16 & 3 & 1 \\ 9 & 5 & 1 & 0 \end{array} \right)$$

Nach der Cramerschen Regel berechnet sich die Lösung des Gleichungssystems wie folgt:

$$x_1 = \frac{\det(A_1)}{\det(A)} = \frac{\det \begin{pmatrix} 1 & 45 & 9 \\ 1 & 16 & 3 \\ 0 & 5 & 1 \end{pmatrix}}{\det \begin{pmatrix} 82 & 45 & 9 \\ 27 & 16 & 3 \\ 9 & 5 & 1 \end{pmatrix}} = \frac{1}{1} = 1$$

$$x_2 = \frac{\det(A_2)}{\det(A)} = \frac{\det \begin{pmatrix} 82 & 1 & 9 \\ 27 & 1 & 3 \\ 9 & 0 & 1 \end{pmatrix}}{\det \begin{pmatrix} 82 & 45 & 9 \\ 27 & 16 & 3 \\ 9 & 5 & 1 \end{pmatrix}} = \frac{1}{1} = 1$$

$$x_3 = \frac{\det(A_3)}{\det(A)} = \frac{\det \begin{pmatrix} 82 & 45 & 1 \\ 27 & 16 & 1 \\ 9 & 5 & 0 \end{pmatrix}}{\det \begin{pmatrix} 82 & 45 & 9 \\ 27 & 16 & 3 \\ 9 & 5 & 1 \end{pmatrix}} = \frac{-14}{1} = -14$$

Abbildung 5: Wikipedia - Cramersche Regel

Die Lösung im Code wurde anhand dieses Beispiels geschrieben. Dabei wurden jeweils die drei Spalten durch den sog. `changeVector` getauscht und danach die Berechnungen mittels der Determinante durchgeführt. Nachdem die Berechnung der drei Parameter abgeschlossen ist, wird geprüft, ob die Werte den Kriterien eines Schnittpunktes auf dem Dreieck gerecht werden. Dazu gehört, dass der Wert t , welcher den Abstand vom Betrachter zum Dreieck repräsentiert eine gewissen festgelegte Mindestgröße hat. Außerdem müssen die Faktoren γ und β in der Summe genau 1 ergeben. Beider Werte müssen größer 0 sein.

```
Mat3x3 linearEquationSystem = new Mat3x3(
//x-Values
this.a.x - this.b.x, this.a.x - this.c.x, ray.direction.x,
//y-Values
this.a.y - this.b.y, this.a.y - this.c.y, ray.direction.y,
//z-Values
this.a.z - this.b.z, this.a.z - this.c.z, ray.direction.z
);
if (linearEquationSystem.determinant == 0){
return null;
}
Vector3 changeVector = (this.a.sub(ray.origin));
Mat3x3 matrixA1 = linearEquationSystem.changeCol1(changeVector);
Mat3x3 matrixA2 = linearEquationSystem.changeCol2(changeVector);
Mat3x3 matrixA3 = linearEquationSystem.changeCol3(changeVector);
final double beta = matrixA1.determinant / linearEquationSystem.determinant;
final double gamma = matrixA2.determinant / linearEquationSystem.determinant;
final double t = matrixA3.determinant / linearEquationSystem.determinant;
if (beta < 0 || gamma < 0 || Math.ceil(beta+gamma) != 1.0 || t<0.001) {
return null;
}
final Normal3 hitNormal = normalOnA.mul(1 - beta - gamma).add(normalOnB.mul(beta)).add(normalOnC.mul(gamma));
return new Hit(t, ray, this, hitNormal);
```

Unklarheit

```
hitNormal=NormalA.mul(1-beta-gamma).add(NormalB.mul(beta).add(NormalC.mul(gamma)));
```

Dieser Programmcode wurde so umgesetzt, allerdings ohne die Eigentliche Funktion dieser Formel zu verstehen. Um möglichst gute Ergebnisse bei der Lichtberechnung zu erhalten, wird hier meines Erachtens nicht die einfache Normale eines der Punkte des Dreiecks berechnet, sondern eine Art Mittelwert-Normale. Leider ist die Funktionsweise dieser Formel nicht ganz klar gewesen. Laut Stephan Rehfeld ist sie aber korrekt. Alternativ funktioniert für grundlegende Darstellungen auch einfach die Normale eines der drei Eckpunkte.

Bemerkung

Zur Umsetzung der Klasse Dreieck wurde folgendes Dokument zu Hilfe genommen.

<http://www.uninformativ.de/bin/RaytracingSchnitttests-76a577a-CC-BY.pdf>]Raytracing: Einfache Schnitttests - P. Hofmann, 22. August 2010

11.4 06.06.15 Die Klasse Sphere

Die Hit-Methode der Kugel-Klasse Sphere wurde anhand der im Script angegebenen Formeln umgesetzt. Es gibt drei Variablen a, b und c welche bekannt sein müssen, damit der Faktor t für den Strahl bzw. das Hit-Objekt ermittelt werden kann. Die Berechnung erfolgt, wie im Script nach folgenden Formeln.

$$a = \vec{d} \cdot \vec{d}$$

```
double a = ray.direction.dot(ray.direction);
```

d ist dabei der Richtungsvektor des Strahls

$$b = \vec{d} \cdot (2 \cdot (\vec{o} - \vec{c}))$$

```
double b = ray.direction.dot(ray.origin.sub(this.c).mul(2.0));
```

o ist dabei der Ursprung des Strahls und c ist der Mittelpunkt der Kugel.

$$c = (\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c}) - r^2$$

```
double c = ((ray.origin.sub(this.c)).dot(ray.origin.sub(this.c))) - (this.r * this.r);
```

r ist der Radius der Kugel

Um den Faktor t zu berechnen wird zunächst die Variable d errechnet, für die gilt:

$$d = b^2 - 4 \cdot a \cdot c$$

```
final double d = (b*b) - (4*a*c);
```

Die eigentlich wichtige Formel ist die folgende.

$$t = \frac{-b \pm \sqrt{d}}{2 \cdot a}$$

Die Variable d wird daraufhin geprüft. Sie gibt Aufschluss darüber, wie viele Schnittpunkte gefunden wurden. Gibt es mehrere, also wenn die Variable d größer 0 ist, dann wird ein Hit-Objekt mit dem kleinsten Faktor t übergeben.

Ist die Variable d kleiner als 0 dann gibt es keinen Schnittpunkt.

```
final double tNegative = ( (b*-1.0) - Math.sqrt(d) ) / (2.0*a);
final double tPositive = ( (b*-1.0) + Math.sqrt(d) ) / (2.0*a);
if (tNegative > 0.001){
    return new Hit(tNegative, ray, this, ray.at(tNegative).sub(this.c).normalized().asNormal());
} else if (tPositive > 0.001){
    return new Hit(tPositive, ray, this, ray.at(tPositive).sub(this.c).normalized().asNormal());
}
if (d==0){
    final double tValue = -b / (2*a);
    if (tValue < 0.001){
        return null;
    }
    return new Hit(tValue, ray, this, ray.at(tValue).sub(this.c).normalized().asNormal());
}
return null;
```

11.5 05.06.15 Die Klasse Hit

Die Klasse Hit wurde nur deklariert anhand des Klassendiagramms. Jedes Hit-Objekt erhält einen Strahl, einen Faktor t, auf dem ein bestimmter Punkt auf dem Strahl „markiert“ wird und eine Geometrie, die getroffen wird. Außerdem erhält das Hit-Objekt eine Normale, die später zur Lichtstärke-Berechnung genutzt wird.

12 14.06.15 Lichter

Die Klassen für die Lichtberechnung wurde nach folgendem Diagramm deklariert.

Bei der Umsetzung haben die illuminates-Methoden einen gewissen Aufwand produziert.

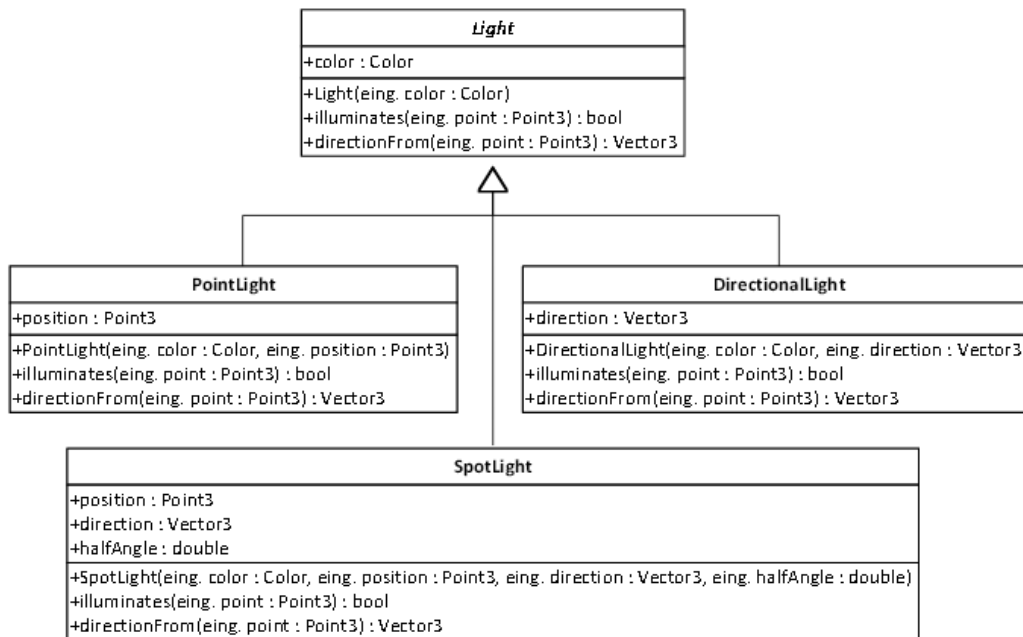


Abbildung 6: Lichter - Klassendiagramm

12.1 14.06.15 Die Klasse PointLight

Die Klasse **PointLight** soll ein Licht repräsentieren, dass in alle Richtungen strahlt und sich an einem bestimmten Punkt befindet. Deshalb hat sie nur eine Variable für die Position. Die `illuminates`-Methode gibt daher immer `true` zurück.

```

public boolean illuminates(Point3 point) {
    return true;
}

```


12.2 09.06.15 Die Klasse Spotlight

Das Spotlight soll in eine bestimmte Richtung strahlen. Daher kommen hier noch die Variablen für Richtung und Winkel hinzu. Der Winkel ist ähnlich wie bei der Perspektivischen Kamera. In der Methode `illuminates()` wird ein Vektor erzeugt, der von der Kamera zum Punkt zeigt, der geprüft werden soll.

Des weiteren muss der Winkel zwischen diesem Vektor und dem Strahl ausgerechnet werden. Dazu dient folgende Formel.

$$\alpha = \cos\left(\frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|}\right)$$

Durch Prüfung des Winkels wird nun festgestellt, ob die Lichtquelle den Punkt anstrahlt. Dabei wird der berechnete Winkel mit dem halben Öffnungswinkel des Lichts verglichen und bei entsprechendem Ergebnis `false` bzw. `true` zurückgegeben.

```
public boolean illuminates(Point3 point) {
    final Vector3 diffVector = this.directionFrom(point).mul(-1.0);
    double angle = (diffVector.dot(this.direction)) / (diffVector.magnitude * this.direction.magnitude);
    angle = Math.acos(angle);
    if (angle > this.halfAngle){
        return false;
    } else return true;
}
```

12.3 09.06.15 Die Klasse DirectionalLight

Eine Lichtquelle der Klasse `DirectionalLight` hat keine Position. Das Licht kommt immer von oben, was durch einen Richtungsvektor beschrieben wird. Solange der Abstand zwischen Lichtquelle und Objekt groß genug ist, wird der Punkt angestrahlt.

```
public boolean illuminates(Point3 point) {
    if(this.position.sub(point).magnitude/1>0.00001){
        return true;
    } else return false;
}
```

13 14.06.15 Material

Die Klassen zum Material wurde zunächst anhand des folgenden Diagramms deklariert.

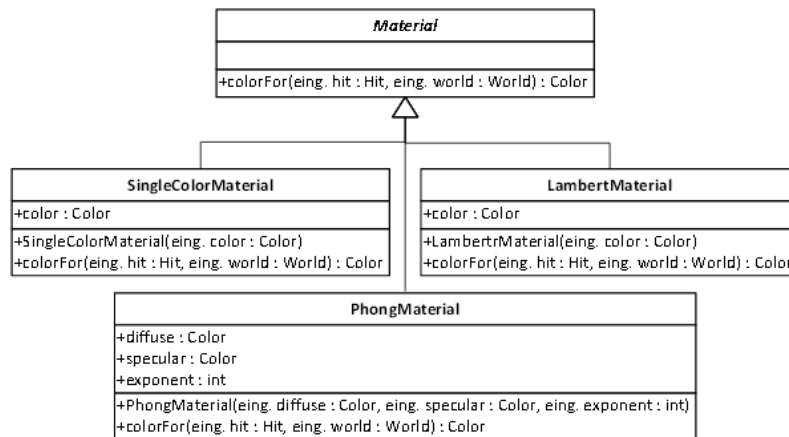


Abbildung 7: MaterialDiagramm

13.1 15.06.15 Die Klasse SingleColorMaterial

Die Klasse erhält nur eine Variable von Typ Color. Sie hat keine weiteren Eigenschaften.

13.2 09.06.15 Die Klasse PhongMaterial

Die PhongMaterial-Klasse wird verwendet um Glanzpunkte „korrekt“ bzw. „schön“ darzustellen. Dazu erhält das Material eine diffuse Farbe, eine Spekulare Farbe und einen Exponenten, der den Glanzpunkt verkleinern bzw. vergrößern kann. Zur Berechnung der Farbe auf dem Phong-Material wurde folgende Formel verwendet. Zugrunde liegt dieser Formel ein bestimmter Winkel, der für die Intensität der Lichtstärke des Glanzpunktes wichtig ist.

$$c = c_r \cdot (c_a + c_l \cdot \max(0, \vec{n} \cdot \vec{l}) + c_l \cdot \max(0, \vec{e} \cdot \vec{r})^P)$$

```
Color phongColor = ((cd.mul(c1)).mul(Math.max(0, n.dot(l)))) + (cs.mul(c1).mul(Math.pow(Math.max(0, e.dot(rn)), exponent)));
```

Zunächst wird die Farbe des Lichts mittels des diffusen Farbwerts, multipliziert mit der Farbe des ambienten Lichts errechnet. Alle bekannten Werte werden aufgrund der besseren Lesbarkeit in entsprechende Variablen gespeichert.

```
Normal3 n = hit.n;
Vector3 e = hit.ray.origin.sub(hit.ray.at(hit.t)).normalized();
Color cd = this.diffuse;
Color ca = world.ambientLight;
Color cs = this.specular;
Color c = cd.mul(ca);
```

Danach wird für jede Lichtquelle innerhalb der Szene geprüft, ob der Punkt angestrahlt wird. Falls ja, werden die noch nicht bekannten aber relevanten Vektoren \vec{l} und \vec{r} berechnet.

\vec{l} : Vektor vom Punkt zur Lichtquelle

```
Vector3 l = currentLight.directionFrom(hit.ray.at(hit.t)).normalized();
```

\vec{r} : An Normale \vec{n} reflektierter Vektor \vec{l}

```
Vector3 rn = l.reflectedOn(n);
```

Die Abschließend berechnete phongColor wird dann auf die bereits ermittelte diffuse Farbe addiert. Abschließend wird der neue Farbwert zurückgegeben.

13.3 15.06.15 Die Klasse LambertMaterial

Das Material der Klasse LambertMaterial wird genutzt, um je nach Einfallswinkel des Lichts eine höhere bzw. schwächere Helligkeit auf einer Oberfläche zu erzeugen. Der Methode zur Farbberechnung nutzt dabei folgende Formel.

$$c = c_r \cdot (c_a + c_l \cdot \max(0, \vec{n} \cdot \vec{l}))$$

```
Color lambertColor = (cd.mul(cl)).mul(Math.max(0, n.dot(l)));
```

Das Ergebnis der Berechnung wird, wie beim PhongMaterial auf die diffuse Farbe (inkl. ambientes Licht) addiert und zurückgegeben. Die Berechnung wird für jede Lichtquelle der Szene durchgeführt. Die Variablen wurden ähnlich dem PhongMaterial genutzt bzw. ermittelt.

```
Normal3 n = hit.n;
Color cd = this.color;
Color ca = world.ambientLight;
Color c = cd.mul(ca);
Vector3 l = currentLight.directionFrom(hit.ray.at(hit.t)).normalized();
Color cl = currentLight.color;
```

14 06.07.15 Schatten

Die Konstruktoren der Lichtquellen erhalten ein Attribut `castShadows`, welches spezifiziert, ob die Lichtquelle Schatten werfen soll oder nicht. Des wird der `illuminates()` Methode, welche feststellt, ob ein Punkt angestrahlt wird oder nicht, nun zusätzlich mit dem Attribut `World` aufgerufen. Dadurch kann festgestellt werden, ob zwischen Licht und Punkt ein Objekt liegt, dass einen Schatten wirft.

14.1 Änderungen an der `illuminates` Methode

Nur wenn das Licht-Objekt das attribut `castShadows` auf `true` gesetzt hat, wird ein Schatten geworfen. Es befinden sich Prüfungen in der Methode, die mittels eines Strahls gemacht werden, der vom potenziell angestrahlten Punkt aus Richtung Lichtquelle geworfen wird. Befindet sich ein Objekt zwischen Lichtquelle und Strahl, wird der jeweilige Punkt „nicht“ angestrahlt.

14.1.1 `PointLight`

Es wird nun geprüft ob die Lichtquelle einen Schatten werfen soll (`castShadows`). Falls dies zutrifft, wird ein Strahl vom jeweiligen Punkt richtung Lichtquelle geworfen. Gibt es einen Schnittpunkt mit einem Objekt, wird geprüft, ob das Objekt zwischen Licht und Punkt liegt. Falls ja, liegt ein Objekt zwischen dem Punkt und der Kamera, dadurch würde der Punkt „nicht“ angestrahlt. Um festzustellen, ob ein Objekt dazwischen liegt, wird der Abstand zwischen Punkt und Kamera und der Abstand zwischen Punkt und Schnittpunkt verglichen.

```
public boolean illuminates(Point3 point, World world) {
    if (!this.castShadows) return true;
    Ray ray = new Ray(point, directionFrom(point));
    Hit hit = world.hit(ray);
    double t = this.position.sub(point).magnitude/1;
    if (hit != null) {
        if (hit.t < t && hit.t > 0.001) {
            return false;
        } else return true;
    } else return true;
}
```

14.1.2 SpotLight

Dieselben Mechanismen werden in der Klasse SpotLight eingesetzt. Das SpotLight wirft immer einen Schatten, wodurch das Attribut castShadows hier nicht explizit geprüft wird.

```
public boolean illuminates(Point3 point, World world) {
    final Vector3 diffVector = this.directionFrom(point).mul(-1.0);
    double angle = (diffVector.dot(this.direction)) / (diffVector.magnitude * this.direction.magnitude);
    angle = Math.acos(angle);

    if (!this.castsShadows) {
        if (angle > this.halfAngle) {
            return false;
        } else return true;
    } else {
        Hit hit = world.hit(new Ray(point, diffVector));
        double t = diffVector.magnitude/1;
        if (angle > this.halfAngle || (hit != null && hit.t < t)){
            return false;
        } else return true;
    }
}
```

14.1.3 DirectionalLight

Ursprünglich wurde nur geprüft, ob der Abstand zwischen Lichtquelle und Punkt groß genug ist, damit er angestrahlt werden kann. Zusätzlich dazu wird nun auch geprüft, ob sich ein Objekt zwischen Punkt und Lichtquelle befindet und ob überhaupt ein Schatten geworfen werden soll.

```
public boolean illuminates(Point3 point, World world) {
    if (!this.castsShadows) return true;
    Ray ray = new Ray(point, directionFrom(point));
    Hit hit = world.hit(ray);
    if(hit == null){
        return true;
    } else return false;
}
```

15 25.06.15 Reflexion

Um die Reflexion möglich zu machen, muss eine Rekursion stattfinden. Dazu wurde die Klasse RecursiveTracer erzeugt. Des Weiteren wurden die Material-Klassen verändert, indem der Tracer in der colorFor() Methode übergeben wird. Je nachdem welcher Tracer übergeben wird, findet eine Reflexion statt oder nicht. Folgendes Diagramm zeigt die neue Klassen-Beschreibung.

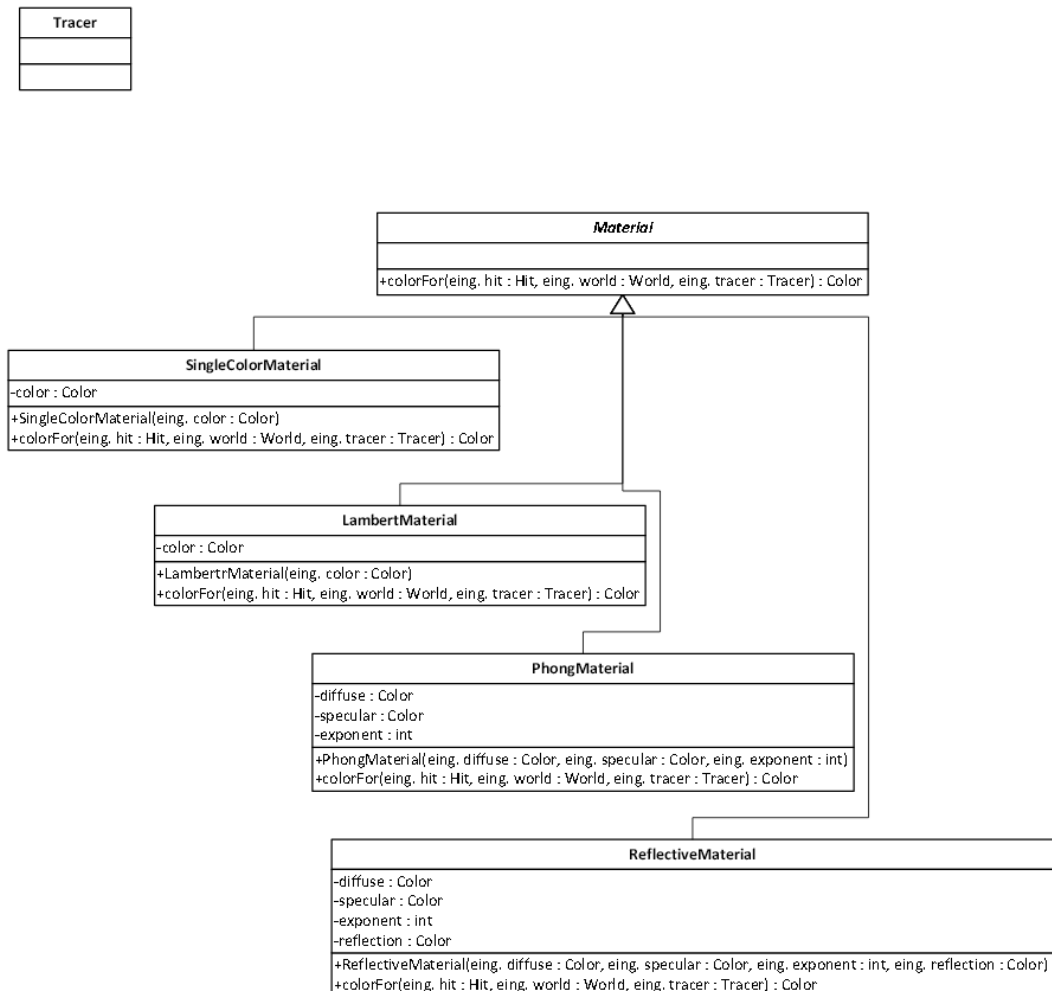


Abbildung 8: ReflexionMaterialDiagramm

15.1 25.06.2015 Die Klasse RecursiveTracer

Dem Konstruktor der Klasse RecursiveTracer wird ein World-Objekt übergeben, welches alle Objekte der Szene beinhaltet. Außerdem erhält der RecursiveTracer eine Art Limit in Form der Variable recursion. Diese sorgt dafür, dass ab einer bestimmten Anzahl keine weiteren Reflexionen mehr ausgeführt werden. Innerhalb einer Reflexierenden Oberfläche bzw. einem entsprechenden Material kann die Methode trace() dieser Klasse aufgerufen werden. Sie wirft bei Erreichen des Limits die Hintergrundfarbe der Welt zurück (Gleiches gilt auch, wenn kein Objekt getroffen wird). Ansonsten wird die Methode colorFor() des jeweiligen Objekt's ausgeführt, welche dann wiederum die Methode trace() aufruft usw. usw.

```
public Color trace(Ray r) {
    if (recursion < 1) {
        return world.backgroundColor;
    }
    if (world.hit(r) == null) {
        return world.backgroundColor;
    }
    this.recursion=this.recursion-1;
    return world.hit(r).geo.material.colorFor(world.hit(r), world, this);
}
```

15.2 25.06.2015 Die Klasse ReflectiveMaterial

So, wie die Klassen PhongMaterial, ist auch diese Material-Klasse in der Lage, Glanzpunkte zu errechnen, genauso wie die Abnahme der Helligkeit bei sich veränderndem Winkel. Außerdem ist ihre Methode zur Farb-Berechnung eines Punktes in der Lage, eine Rekursion durchzuführen, um Reflexionen zu errechnen. Folgende Formel liegt diesem Sachverhalt zugrunde.

$$c = c_d \cdot c_a + \sum_{i=1}^n [[c_d \cdot c_l \cdot \max(0, \vec{n} \cdot \vec{l}) + c_s \cdot c_l \cdot \max(0, \vec{e} \cdot \vec{r}_l)^P] + c_r \cdot f_r(\vec{p}_r, \vec{r}_d)]$$

Nach einmaliger Berechnung der Farbe durch dieselben Mechanismen, wie bei der PhongMaterial-Klasse, wird diese Rekursion durchgeführt. Dazu wird ein Vektor erzeugt, welcher durch Reflexion des Richtungsvektors zwischen Betrachter und Punkt an der Normalen des zu prüfenden Punktes erzeugt wird. Mittels dieses Vektors und des Punktes wird ein Strahl erzeugt, welcher nun erneut zur Kalkulation der Farbe verwendet wird. Die daraus ermittelte Farbe wird mit dem Attribut reflection multipliziert (Stärke der Reflexion). Abschließend wird die Reflexions-Farbe zu der ursprünglichen Objekt-Farbe addiert und zurückgegeben.

```
....
    Vector3 rd = e.reflectedOn(hit.n);
    Point3 point3 = hit.ray.at(hit.t - 0.001);
    Ray ray = new Ray(point3, rd);
    Color color = tracer.trace(ray);
    Color cr = this.reflection.mul(color);
    c = c.add(cr);
.....
```

16 11.07.15 Transformationen

Die Transformationen sind wie in diesem Diagramm beschrieben deklariert worden.

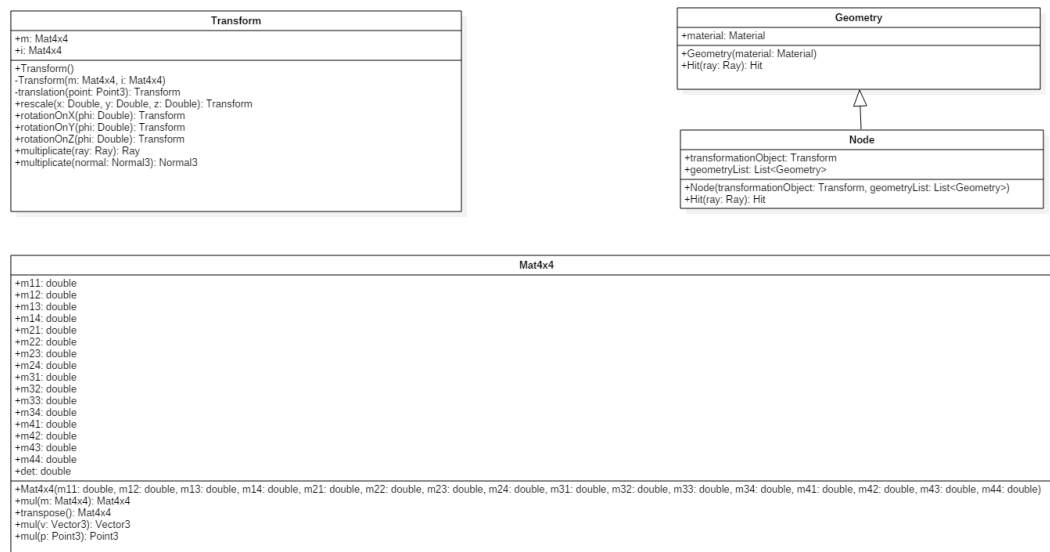


Abbildung 9: UMLMat4x4TransformGeometryNode

16.1 11.07.15 Die Klasse Mat4x4

Um im späteren Verlauf Transformationen mit den Objekten durchführen zu können, ist die Klasse Mat4x4 erzeugt worden. Für die Umsetzung der Methoden zur Multiplikation wurden die allgemein gültigen mathematischen Formeln verwendet.

16.2 13.07.15 Die Klasse Transform

Mittels der Klasse Transform sollen Objekte transformiert werden, also rotiert, skaliert, gedreht etc. Das wird mittels der bereits bekannten Matrizen durchgeführt.

16.2.1 Translation translation()

Zur Umsetzung der Translation kommt eine Translationsmatrix der Klasse Mat4x4 zum Einsatz. Sie hat folgende Syntax.

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Methode erstellt eine Transform anhand der Koordinaten eines übergebenen Punktes und gibt die neue Translations-Transform zurück.

```

public Transform translation(final Point3 point){
    Mat4x4 transformationMatrix = new Mat4x4(1.0, 0.0, 0.0, point.x, 0.0, 1.0, 0.0, point.y, 0.0, 0.0, 1.0, point.z, 0.0, 0.0, 0.0, 1.0);
    Mat4x4 transformationInverse = new Mat4x4(1.0, 0.0, 0.0, -point.x, 0.0, 1.0, 0.0, -point.y, 0.0, 0.0, 1.0, -point.z, 0.0, 0.0, 0.0, 1.0);
    return new Transform(m.mul(transformationMatrix), i.mul(transformationInverse));
}

```


16.2.2 Skalierung rescale()

Die Skalierung streckt eine Form in x-, -y bzw. z-Richtung. Dazu werden der Methode drei Parameter übergeben, die als Skalierungsfaktoren dienen. Die Skalierungsmatrix hat folgende Form.

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Methode erzeugt eine Transform, die für Skalierungen genutzt werden kann und übergibt diese.

```
public Transform rescale(final double x, final double y, final double z) {
    Mat4x4 tm = new Mat4x4(x, 0.0, 0.0, 0.0, 0.0, y, 0.0, 0.0, 0.0, 0.0, z, 0.0, 0.0, 0.0, 0.0, 1.0);
    Mat4x4 ti = new Mat4x4(1.0 / x, 0.0, 0.0, 0.0, 0.0, 1.0 / y, 0.0, 0.0, 0.0, 0.0, 1.0 / z, 0.0, 0.0, 0.0, 0.0, 1.0);
    return new Transform(m.mul(tm), i.mul(ti));
}
```

16.2.3 Rotation rotationOnX() rotationOnY() rotationOnZ()

Rotationen werden mit Hilfe der Rotationsmatrizen durchgeführt. Die Methoden innerhalb der Klasse Transform erstellen eine Mat4x4 Matrize, die für die jeweilige Rotation genutzt werden kann. Sie haben folgenden Aufbau.

xRotation:	y-Rotation:	z-Rotation:
$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos & -\sin & 0 \\ 0 & \sin & \cos & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos & 0 & \sin & 0 \\ 0 & 1 & 0 & 0 \\ -\sin & 0 & \cos & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos & -\sin & 0 & 0 \\ \sin & \cos & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Beispiel anhand der x-Rotation.

```
public Transform rotationOnX(final double phi) {
    Mat4x4 tm = new Mat4x4(1.0, 0.0, 0.0, 0.0, 0.0, Math.cos(phi), -Math.sin(phi), 0.0,
        0.0, Math.sin(phi), Math.cos(phi), 0.0, 0.0, 0.0, 0.0, 1.0);
    Mat4x4 ti = new Mat4x4(1.0, 0.0, 0.0, 0.0, 0.0, Math.cos(phi), Math.sin(phi), 0.0,
        0.0, -Math.sin(phi), Math.cos(phi), 0.0, 0.0, 0.0, 0.0, 1.0);
    return new Transform(m.mul(tm), i.mul(ti));
}
```

16.2.4 Multiplikationen multiply()

Die Methode multiply() nimmt entweder einen Strahl oder eine Normale (Rücktransformation) entgegen und transformiert diese, indem . Anschließend werden die transponierten Objekte zurückgegeben. Das dient dazu, in späteren Aufrufen, z.B. einen transformierten Strahl wieder in seinen Ursprungszustand zurückzuführen. Später wird dieser Mechanismus in der Klasse Node verwendet.

17 13.07.15 Szenengraph (Darstellung der Transformationen)

Um die im Kapitel Transformationen erläuterten Funktionen nutzen zu können, müssen die Transform-Objekte in die Szene gebracht werden. Dazu dient eine neue von Geometry abgeleitete Klasse Node. Zur Erinnerung hier noch einmal das entsprechende Diagramm.

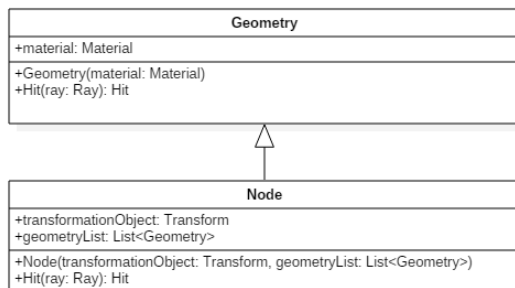


Abbildung 10: UMLNode

17.1 13.07.15 Die Klasse Node

Die Klasse Node dient zur Umsetzung der in Transform vorbereiteten Methoden zu Skalierung, Rotation, Translation etc. Dazu nimmt der Konstruktor ein Transformations-Objekt, z.B. zur Rotation entgegen. Zusätzlich dazu erhält der Konstruktor eine Liste mit zu Transformierenden Geometrien, wie z.B. eine Sphere (Kugel) und eine AAB (Box). Die Objekte werden nicht, wie üblich an die Szene übergeben und führen ihre Hit-Methoden aus, sondern werden durch das Node-Objekt „ersetzt“, welches seinerseits die Hit-Methode bereitstellt. Dadurch können die Objekte beliebig transformiert werden. Dabei ist die Hit-Methode der Klasse Node beachtenswert.

17.1.1 Die Methode hit()

Die Methode der Geometry-Klasse wird hier, wie auch bei den anderen von Geometry abgeleiteten Klassen überschrieben. Hier wird allerdings der Strahl zunächst mittels des in der Node gespeicherten Transform ein transformierter Strahl erzeugt. Danach wird über die Liste der Geometrien (ebenfalls in der Node gespeichert) iteriert. Die Hit-Methode arbeitet nach dem bekannten Prinzip des sichtbaren Schnittpunkts und gibt diesen auch zurück. Allerdings wird beim übergebenen Hit-Objekt nicht der transformierte Strahl zurückgegeben, sondern der untransformierte, damit die aufrufende Klasse (i.d.R. World) nicht „merkt“, dass eine Transformation stattgefunden hat.

```

public Hit hit(Ray ray) {
    Hit cachedHit = null;
    Hit currentHit = null;
    Ray transformedRay = transformationObject.multiply(ray);
    for (Geometry geometry : geometryList) {
        currentHit = geometry.hit(transformedRay);
        if (currentHit != null && cachedHit != null) {
            if (currentHit.t < cachedHit.t) {
                cachedHit = currentHit;
                continue;
            }
        }
        if (currentHit != null) {
            cachedHit = currentHit;
        }
    }
    if (cachedHit != null) {
        return new Hit(cachedHit.t, ray, cachedHit.geo, transformationObject.multiply(cachedHit.n));
    }
    return null;
}
  
```

18 25.07.15 Die Klasse Disc

Die Klasse Disc wurde im Zuge der Bearbeitung einer Klasse-C-Zusatzaufgabe erstellt. Die Klasse Disc erbt von der Klasse Plane, denn prinzipiell ist die Disc zunächst eine Ebene, welche durch einen zusätzlichen Parameter zum Radius des Kreises erweitert wird. Als Mittelpunkt des Kreises dient der bei der Initialisierung der ohnehin übergebene Ortsvektor. Selbiges gilt für die „Ausrichtung“ im Raum, welche durch die Normale der Ebene beschrieben werden kann.

18.1 Die Methode hit()

Die Methode hit() der Klasse Disc ermittelt, falls vorhanden, einen Schnittpunkt mit der Ebene. Dieser wird in Form eines Point3 Objektes gespeichert. Anschließend wird geprüft, ob dieser Punkt innerhalb des Kreisradius liegt. Dazu wird der Abstand des Mittelpunktes zum Punkt berechnet und mit der Länge des Radius verglichen. Um Vorzeichenfehler zu vermeiden, wird der Betrag beider Parameter verglichen bzw. einfach das Quadrat der jeweiligen Werte.

```
public Hit hit(Ray ray){
    if (ray.direction.dot(n) == 0) {
        return null;
    }
    Vector3 v = a.sub(ray.origin);
    Double b = v.dot(n);
    Double c = ray.direction.dot(n);
    Double t = b/c;
    Point3 p = ray.origin.add(ray.direction.mul(t));
    if(Math.pow(a.sub(p).magnitude, 2) < Math.pow(radius, 2)) {
        if (t > 0) {
            return new Hit(t, ray, this, n);
        }
    }
    return null;
}
```

19 20.07.15 Raytracer Fortschrittsbalken

Im Zuge der Umsetzung eines „Ladebalkens“ zur Anzeige des Rendering-Fortschritts, wurde die Methode `render()` innerhalb der Klasse `Raytracer` hinzugefügt. Diese Methode nimmt die Szene in Form des `world`-Objekts und eine Liste aller Kamera-Objekte entgegen. Die Methode erzeugt ein Objekt der Klasse `Painter`, um die Szene zu „zeichnen“. Anschließend wird ein Frame erzeugt, welcher den Inhalt der Fortschrittsberechnung anzeigen soll. Ein Panel liefert innerhalb dieses Frames die nötigen Darstellungsfelder und Beschriftungen. Der Maximalwert der „ProgressBar“ wird über die Auflösung festgelegt.

```
private void setupProgressBar() {
    this.progressBarFrame = new JFrame("Rendering...");
    this.progressBarFrame.setAlwaysOnTop(true);
    this.progressBarFrame.setSize(300, 110);
    this.progressBarFrame.setResizable(false);
    this.progressBarFrame.setLocationRelativeTo(null);
    JPanel panel = new JPanel();

    this.timeRunningLabel = new JLabel("Time Running: 00:00");
    this.timeTillFinishLabel = new JLabel("Estimated Time remaining: 00:00");
    this.progressBar = new JProgressBar(0, this.pixels.length);
    this.progressBar.setBounds(30, 15, 240, 60);
    this.progressBar.setStringPainted(true);
    panel.add(this.timeRunningLabel);
    panel.add(this.timeTillFinishLabel);
    panel.add(this.progressBar);
    this.progressBarFrame.add(panel);
    this.progressBarFrame.setVisible(true);
}
```

Zur Berechnung der noch zu erwartenden verbleibenden Zeit wird ein Durchschnittswert genutzt. Dieser wird bei Initialisierung über die Auflösung und Anzahl der in der Szene vorhandenen Elemente kalkuliert. Nachdem erste Berechnungen durchgeführt wurden, wird dieser Wert kontinuierlich angepasst. Es wird nach jedem Linien-Durchlauf, also eine pixelbreite Linie horizontal, die dafür benötigte Zeit gespeichert und der Fortschrittsbalken und die aktuelle Laufzeit werden angepasst (erhöht). Die Durchschnittsbearbeitungszeit für eine Linie dient bei der Kalkulation der noch verbleibenden Zeit als Maßstab. Die verbrauchte Zeit der letzten horizontalen Linie wird immer auf diesen Parameter addiert und danach wird, je nach Veränderung, die Anzeige angepasst.

Sobald der Maximalwert der ProgressBar erreicht wurde, wird der Frame geschlossen.

```
private void render(Camera camera, World world) {
    Painter p = new Painter(this.width, this.height, this.pixels);
    int start = (int) (System.currentTimeMillis() / 10);
    int progress = 0;
    this.setupProgressBar();
    int averageTime = 0;
    double lastAverageTime;
    if(world.lights == null){
        lastAverageTime = (this.width * this.height * world.objects.length + 100000) / 100000;
    } else if (world.objects == null) {
        lastAverageTime = (this.width * this.height * world.lights.size() + 100000) / 100000;
    } else lastAverageTime = (this.width * this.height * world.lights.size() * world.objects.length + 100000) / 100000;

    int intervall = 60;

    for(int y = 0; y < this.height; y++) {
        int timePerLine = (int) (System.currentTimeMillis());
        ....
        Pixel Berechnung Horizontal
        ...
        timePerLine = (int) (System.currentTimeMillis() - timePerLine);
        progress = y * this.width + 800;
        this.progressBar.setValue(progress);
        p.draw();
        int s = (int) (System.currentTimeMillis() / 10) - start;
        String currentTime = "Time running: " + String.format("%d:%02d:%02d", s/3600, (s%3600)/60, (s%60));
        this.timeRunningLabel.setText(String.valueOf(currentTime));
        averageTime = averageTime + timePerLine;
        int s2 = 0;
        if (lastAverageTime - averageTime / intervall < 3){
            lastAverageTime = averageTime / intervall;
            s2 = (int) (lastAverageTime * (this.height - y - 1) / 10);
        }
        else {
            s2 = (int) (lastAverageTime * (this.height - y - 1) / 10);
        }
        String remainingTime = "Estimated Time remaining: " + String.format("%d:%02d:%02d", s2/3600, (s2%3600)/60, (s2%60));
        this.timeTillFinishLabel.setText(remainingTime);

        /**
         * hides progressBarFrame if finished
         */
        if (this.progressBar.getMaximum() - progress < this.width * 2) {
            this.progressBarFrame.setVisible(false);
        }
    }
}
```