Projekt-Protkoll - Computergrafik SS2015

David Derichs, Jonathan Stoye, Viktoria de Koning

26. 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.

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 Die Klasse Painter - 10.06.2015

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. Auß erdemm müssen die Pixel-Farben bekannt sein, damit diese später gezeichnet werdne können. Das Fenster soll ein Menü haben, welches zum „Schließ en“ des Programms und zum „Speichern“ des Bildes verwendet werden kann.

2.1 Variablen

Die Auflösung des Frames wird in den Variablen von 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.

2.2 Konstruktor

Im Konstruktor werden drei Werte übergeben für Breite, Höhe und Farbwerte aller Pixel. Diese werden dem Painter-Objekt zugewiesen. Auß erdem 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 ïmage”darstellen zu können, muss es aber an den Frame geknüpft werden. Dazu muss ein JPanel verwendet werden. Es wurde daher die Klassen-Variable „panel“ vom Typ „JPanel“ hinzugefügt.

In der Methode ßetupFrame”konnte nun das Panel erzeugt und dem Frame zugewiesen werden. Dazu wurde die Mathode ßetupPanel()ïmplementiert.

Auß erdem wurde die Methode ßetupMenuBar()ïmplementiert, um die Menüleiste zu konfigurieren.

2.3 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.

2.4 weitere Methoden

Es wurde noch weitere Methoden implementert, welche allerdings keine größ eren 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.

3 Die Klasse PaintDiagonal 11.05.2015

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.

4 Die Klasse ShowImage 28.04.15

Die Klasse ShowImage beinhaltet nur eine Variable vom Typ Buffered Image. 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.

5 Die Klassen Mat3x3, Vector3, Point3 und Normal3 01.05.15

Die Klassen wurden zunächst, wie im Klassendiagramm beschrieben deklariert.

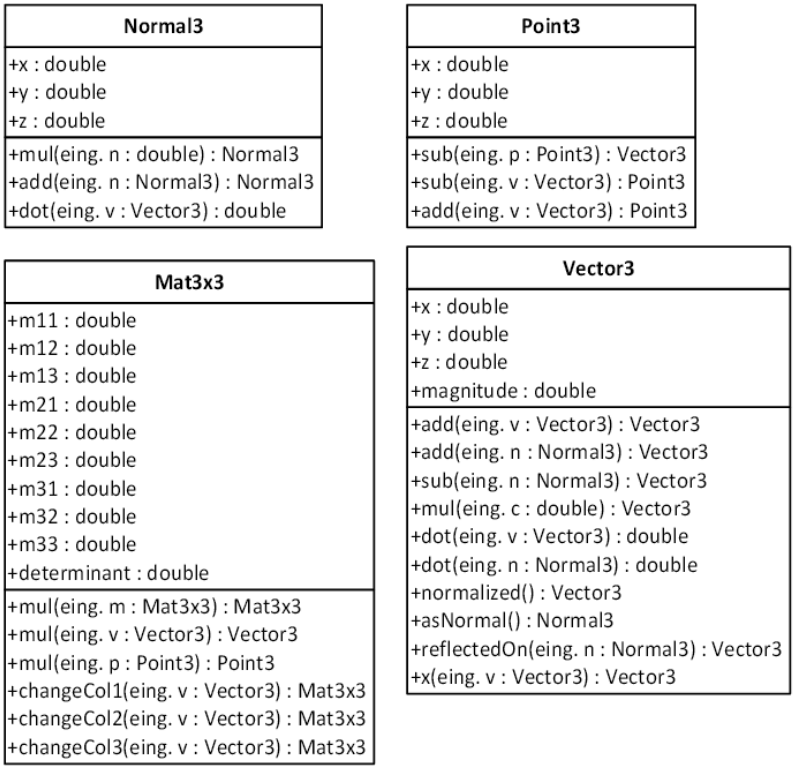

Abbildung 1: MatrixVector-Diagramm

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 erwateten Ergebnisse.

6 Die Klasse Ray 05.06.15

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

7 Die Klassen Camera, OrthographicCamera und PerspectiveCamera

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

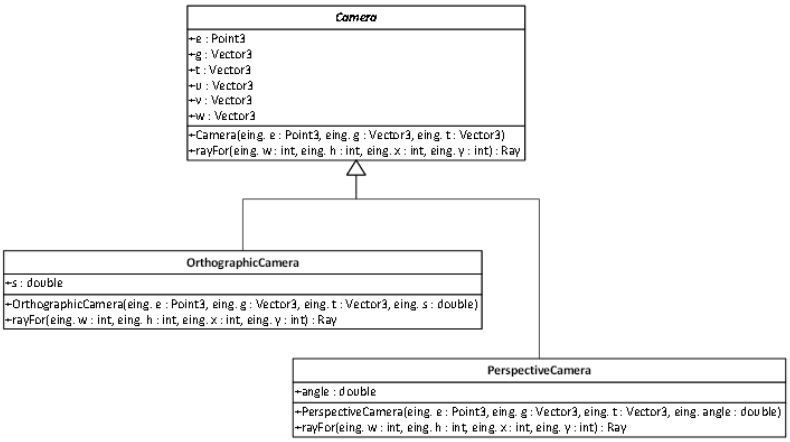

Abbildung 2: Camera-Diagramm

7.1 OrthographicCamera

Probleme
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:

$$\vec{o} = \vec{e} + a \cdot s \cdot (p_x - \frac{\frac{w-1}{2}}{w-1}) \cdot \vec{u} + s \cdot (p_y - \frac{h-1}{2}) \cdot \vec{v}$$

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

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

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 $\vec{w}$.

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

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.

8 Die Klasse Color 05.06.15

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

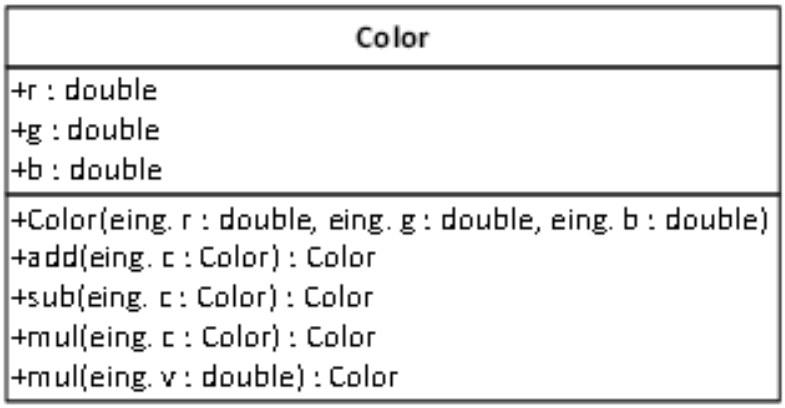

Abbildung 3: ColorDiagramm

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.

9 Die Klassen zur „Geometrie“ 06.06.15

Die Klassen zur Geometrie wurden nach folgendem Diagramm deklariert.

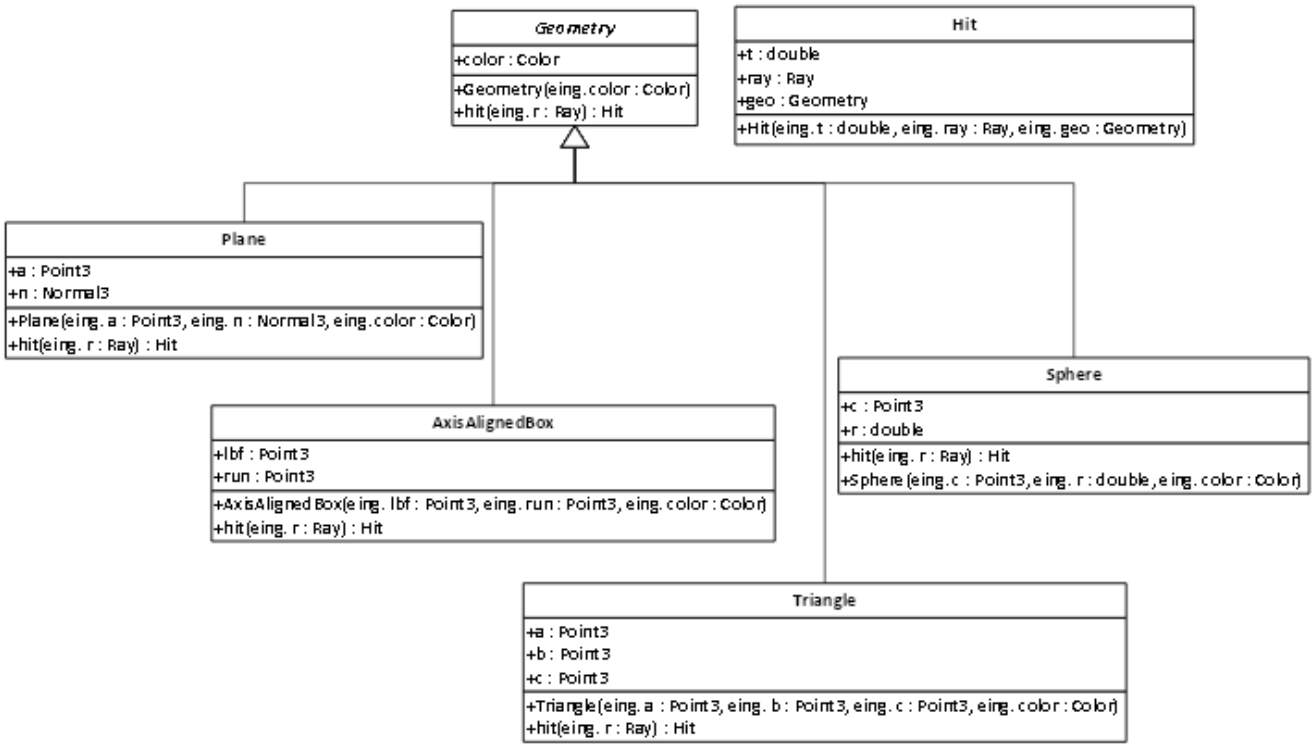

Abbildung 4: GeometrieDiagramm

Die Superklasse „Geometrie“ beerbt dabei folgende Unterklassen.

9.1 Die Klasse Plane 06.06.15

Die Klasse Plane repräsentiert eine Ebene im Raum. Die Schnittpunkte werden über die Ebene in implizieter 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 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.

9.2 Die Klasse AxisAlignedBox 06.06.15

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.

9.3 Die Klasse Triangle 06.06.15

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 aber geprüft, 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:

$$\begin{aligned} 82\,x_1 + 45\,x_2 + 9\,x_3 &= 1 \\ 27\,x_1 + 16\,x_2 + 3\,x_3 &= 1 \\ 9\,x_1 + 5\,x_2 + 1\,x_3 &= 0 \end{aligned}$$

Die erweiterte Koeffizientenmatrix des Gleichungssystems ist dann:

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

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

9.4 Die Klasse Sphere 06.06.15

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;
```

9.5 Die Klasse Hit 05.06.15

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.

10 Lichter 14.06.15

Die Klassen für die Lichtberechnung wurde nach folgendem Diagramm deklariert.
Bei der Umsetzung haben die illuminates-Methoden einen gewissen Aufwand produziert.

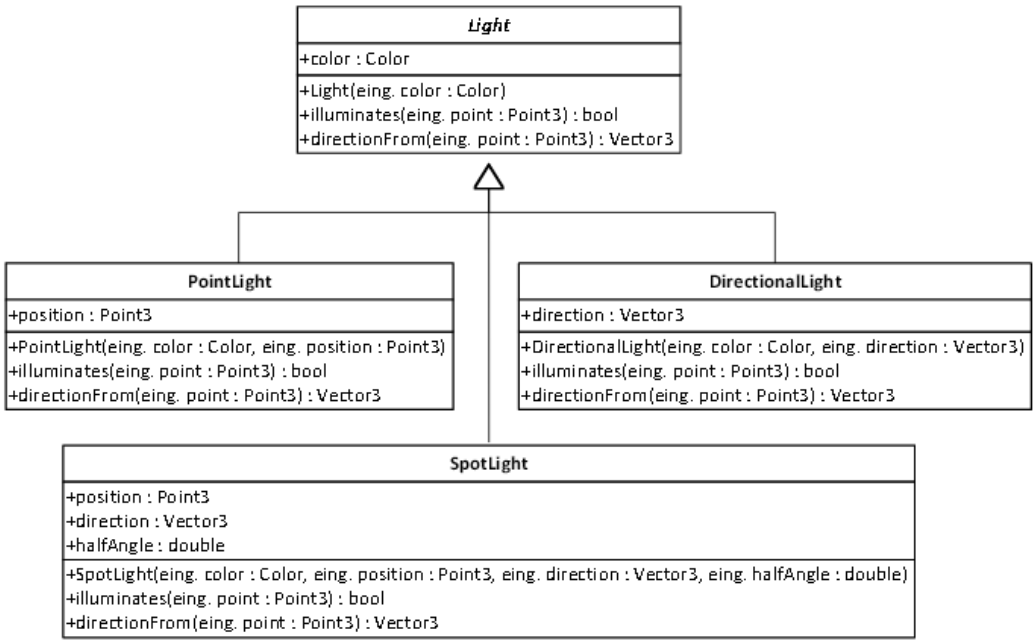

Abbildung 6: Lichter - Klassendiagramm

10.1 Die Klasse PointLight 14.06.15

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 erstellt einen Strahl, von dem Punkt aus, der geprüft werden soll und einem Richtungsvektor, der von diesem Punkt hin zur Lichtquelle zeigt. Gibt es einen Schnittpunkt, wird true zurückgegeben.

```
public boolean illuminates(Point3 point, World world) {
    if (!this.castsShadows) {
        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;
}
```

10.2 Die Klasse SpotLight 09.06.15

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

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. Außerdem muss der Abstand zur Lichtquelle korrekt sein, denn wenn das Hit-Objekt näher am Punkt liegt also die Lichtquelle, ist ein anderes Objekt „im Weg“, wodurch ein Schatten geworfen wird und der Punkt nicht angestrahlt 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.magn
    angle = Math.acos(angle);
    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;
}
```

10.3 Die Klasse DirectionalLight 09.06.15

Eine Lichtquelle der Klasse DirectionalLight hat keine Position. Das Licht kommt immer von oben, was durch einen Richtungsvektor beschrieben wird. Dadurch ist die Prüfung eines potenziell angestrahlten Punktes einfacher, aber ähnlich der illuminates() Methode der Klasse SpotLight. Wird kein Hit-Objekt gefunden, also liegt kein Objekt zwischen „oben“ und dem Punkt, wird dieser Punkt angestrahlt.

```
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;
}
```

11 Material 14.06.15

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

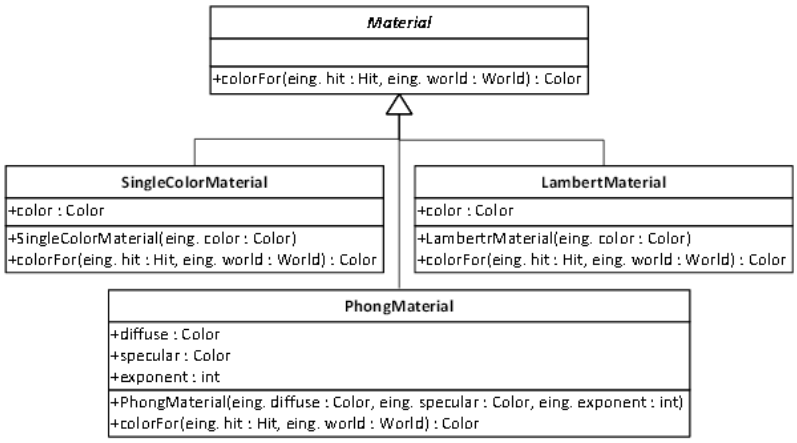

Abbildung 7: MaterialDiagramm

11.1 Die Klasse SingleColorMaterial 15.06.15

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

11.2 Die Klasse PhongMaterial 09.06.15

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 Farbei 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(cl)).mul(Math.max(0, n.dot(l)))).add(cs.mul(cl).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.

11.3 Die Klasse LambertMaterial 15.06.15

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;
```