CG Übung 3: Raytracer - Mathematik-Implementationen

Sascha Feldmann
sascha.feldmann@gmx.de

Inhaltsverzeichnis

1 Profil

1.1 Aufgabe

Implementation der Mathematik für den Raytracer.

1.2 Abgabe

05.06.2012

1.3 Autor

Sascha Feldmann - 778455

1.4 Beginn der Bearbeitung

22.05.2012

2 Dokumentation zur Aufgabe

2.1 Methodik

Zuerst wurden die bereitgestellten Interfaces und deren JavaDoc-Sources als jar-Dateien als libraries in das neue Projekt integriert.

Im Folgenden schaute ich mir die einzelnen Schnittstellen-Dokumentationen der Interfaces an, insbesondere hinsichtlich folgender Aspekte:

- **Was** leistet die Methode?
- **Welche** Paramter werden erwartet?
- **Wie** sind die Modellklassen miteinander verknüpft?

Im Anschluss implementierte ich die Interfaces. Entsprechend der Dokumentation sollten alle Instanzen final sein, daher boten sich für alle Klassen finale Attribute für die Modell-Komponenten an.

2.1.1 Modell-Komponenten

- **Vector3DImpl, NormalImpl, Point3DImpl** enthält x, y und z-Werte, bildlich

$$M = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

- **RayImpl** enthält den Ursprung als Point3DImpl und die Richtung als Vector3DImpl (Richtungsvektor).

- **Mat3x3Impl** enthält alle Elemente einer 3x3-Matrix als einzelne Attribute.

$$M = \begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix}$$

2.2 Modellklassen

Nachfolgend wird die konkrete Implementierung der Raytracer-Modell-Klassen erläutert. Dabei gehe ich insbesondere auf einzelne Berechnungsoperationen ein.

2.2.1 Mat3x3Impl

Die Funktion **getDeterminant()** rechnet die Determinante aus:

Listing 1: getDeterminant()-Methode

```
1 // First row
2 final double det = this.a11
3 * ( this.a22 * this.a33 - this.a23 * this.a32 )
4 - this.a12
5 * ( this.a21 * this.a33 - this.a23 * this.a31 )
6 + this.a13
7 * ( this.a21 * this.a32 - this.a22 * this.a31 );
8 return det;
```

Allgemein erfolgt die Berechnung von Determinanten zu Matrizen rekursiv mit dem Laplaceschem Entwicklungssatz:

1. Bilde je um eine Spalte bzw. Zeile verkürzte Matrizen.
2. Streiche dazu eine beliebige Zeile oder Spalte, bilde aus den den selektierten Koeffizienten eine Formel.
3. Diese Formel hat bei einer 3x3-Matix unter Entwicklung nach der 1. Zeile die Form

$$det(A) = a11 \cdot \begin{vmatrix} a22 & a23 \\ a32 & a33 \end{vmatrix} - a12 \cdot \begin{vmatrix} a21 & a23 \\ a31 & a33 \end{vmatrix} + a13 \cdot \begin{vmatrix} a21 & a22 \\ a31 & a32 \end{vmatrix}$$

Bei der 3x3-Matrix ist dieses Verfahren relativ simpel. Meine Implementation nutzt das Verfahren so aus, dass nach der ersten Zeile entwickelt wird.

Die Funktion **mul()** multipliziert 2 Matrizen. Dieses Verfahren gestaltet sich auf Papier relativ aufwendig, da die linke Matrix zeilenweise mit

den Spalten der rechten multipliziert wird, dabei muss die Anzahl der Zeilen der linken Matrix gleich der Anzahl der Spalten der rechten sein. Die resultierende Matrix hat stets die Größe der rechten Matrix.

Da in unserer Implementation zwei 3x3-Matrizen multipliziert werden sollen, bietet es sich an, die Berechnung auf die implementierten Vektor3D-Klassen zu reduzieren. Die Spalten der linken Matrix (im Parameter) werden dabei zu 3 Vektoren, die jeweils mit den Zeilenvektoren der rechten Matrix multipliziert werden. Daraus entstehen neue Spaltenvektoren, die eine neue 3x3-Matrix ergeben.

2.2.2 RayImpl

Die Funktion **tOf()** rechnet den Punkt zu einem Parameter t aus.

Geraden in der Parameterform bestehen aus einem Ursprungspunkt, zu dem das Vielfache eines Richtungsvektors addiert wird. Der Parameter t nimmt die Rolle des Skalars an. In 3D-Vektor-Darstellung lautet die Geradengleichung:

$$\vec{g} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + t \begin{pmatrix} xx \\ yy \\ zz \end{pmatrix} \cdot$$

Ein Punkt liegt auf einer Geraden, falls für alle Koeffizienten t gleich ist. Der Algorithmus funktioniert also folgendermaßen:

1. Rechne t für jeden Koeffizient durch Umstellung aus.
2. Prüfe, ob t für jeden Koeffizient gleich ist.
3. Falls 2 gilt, liefere dieses t.
4. Falls 2 nicht gilt, liegt der Punkt nicht auf der Geraden und eine Exception wird geworfen.

Listing 2: tOf-Methode

```
9  public double tOf( Point3D point ) {
10          if(point == null)
11           throw new IllegalArgumentException(
12          "Illegal Argument for value point
13  ________in the method tOf (RayImpl)." );
14
15          // Calculate ts for each coordinate
16          final double xT = (point.getX( )
17          - this.origin.getX( ))
```

```
18          / this.direction.getX( );
19          final double yT = (point.getY( )
20          - this.origin.getY( ))
21          / this.direction.getY( );
22          final double zT = (point.getZ( )
23          - this.origin.getZ( ))
24          / this.direction.getZ( );
25
26          // Check if point lays on the ray
27          if(xT != yT && yT != zT) {
28                  throw new IllegalArgumentException(
29                  "The point must lay on the ray in
30 ________________the method tOf (RayImpl)." );
31                  }
32                  return xT;
33          }
```

2.2.3 Vector3DImpl

Die primitiven Operationen bedürfen keiner weiteren Erklärung, da sie Vektor-Grundrechenarten darstellen.

Nach der Implementation dieser Methoden reicht eine Zeile für die Methode **normalized()**:

Listing 3: normalized-Methode

```
34 /**
35           * This method returns the so-called
36            "Einheitsvektor"
37           */
38          public Vector3D normalized() {
39          return this.div(this.getMagnitude( ) );
40          }
```

2.3 JUnit- UnitTests

Bei allen Testfällen verifizierte ich die erwarteten Rechenergebnisse auf Papier und gerade bei Matrizenoperationen durch vorhandene Tools im Internet.

Ich verfolgte nicht den Ansatz des ExtremeProgrammings, sondern entwickelte die Testfälle erst nach der Implementation.

Zum Testen der Mat3x3-Klasse entwickelte ich Testunits für folgende Fälle:

- Berechnung der Determinanten.

- Multiplikation einer Matrix mit einem Vektor.

- Multiplikation zweier Matrizen.

Zum Testen der Ray-Klasse entwickelte ich Testunits für folgende Fälle:

- Berechnung des ts zu einem Punkt.

- Berechnung des Punktes zu einem t.

Zum Testen der Vector3D-Klassen entwickelte ich Testunits für folgende Fälle:

- Addition und Subtraktion mit anderen Vektoren.

- Multiplikation und Division mit Skalaren.

- Das Skalarprodukt zweier Vektoren.

- Die Berechnung des Einheitsvektors.

- Das Kreuzprodukt zweier Vektoren.

Generell sind die Klassen Point2DImpl und NormalImpl äquivalent zur Point3DImppl-Klasse. Somit sind keine weiteren Testfälle notwendig.