# Bericht zur 02 Schnittberechnung

Marcus Bätz, Andreas Kiauka, Robert Dziuba

10. November 2015

# Inhaltsverzeichnis

# 1 Aufgabenstellung

Implementiert werden soll ein Raytracer, der eine gegebene Szene in ein Fenster rendert. Verwendet werden soll hierzu der Quelltext aus der ersten Übung.

## 1.1 Strahl

Es sollte eine Klasse implementiert werden die einen Strahl repräsentiert der von der Kamera aus geht.

## 1.2 Kamera

Es sollte eine abstrakte Oberklasse „Camera“ erstellt werden und davon abgeleitet eine orthographische und eine perspektivische Kamera.

## 1.3 Farbe

Weiter sollte eine Klasse „Color“ erstellt werden, welche die Farbe darstellt die letztendlich in den jeweiligen Pixel des Bildes geschrieben wird.

## 1.4 Geometrie

Es sollte eine abstrakte Oberklasse „Geometrie“ erstellt werden und davon abgeleitet die Klassen „Plane“, „Sphere“, „Triangle“ und „AxisAlignedBox“. Zusätzlich sollte eine Klasse Hit geschrieben werden, die als Behälter für einen Treffer zwischen Strahl und Geometrie dient, und sowohl den Strahl, als auch den Parameter des Schnittpunktes und die getroffene Geometrie enthält.

## 1.5 Welt

Als letztes sollte eine Klasse „World“ erzeugt werden, die eine Liste aller Geometrien enthält und die gegen alle Geometrien einen übergebenen Strahl testet ober er die geometrien trifft und die dann die Farbe der Geometrie die der Kamera am nächsten liegt oder die Background-Color der Welt zurückgibt. Des weiteren musste die Klasse „Image-Saver“ überarbeitet werden um nun das erwartete Bild darzustellen

## 1.6 Akzeptanzkriterien

Zur Erfüllung der Akzeptanzkriterien mussten verschiedene vorgegebene Szenen nachgebaut werden und sollten nach dem Rendern den Bilder aus den Akzeptanzkriterien entsprechen.

# 2 Lösungsstrategien

Zur besseren Bearbeitung der einzelnen Klassen, haben wir zu Beginn von allen Klassen gemäß des Klassen-Diagramms Dummy-Klassen erzeugt. Damit konnten die Klassen ohne Compiler-Fehler geschrieben werden. Wir haben sie gleichmäßig verteilt und auf unseren eigenen Git-Branche bearbeitet. Als alle Klassen fertig waren wurden sie auf den master gemerged und abschließend alle noch auftretenden Fehler beseitigt.

# 3 Implementierung

## 3.1 Camera

Die abstrakte Klasse „Camera“ erhält drei Parameter:

**Point3** $\vec{e}$ für den Ursprung der Kamera.

**Vector3** $\vec{g}$ für die Blickrichtung der Kamera.

**Vector3** $\vec{t}$ für die Orientierung um die Blickrichtung.

Diese werden in ein Kamera internes, rechtshändiges orthonormales Koordinatensystem ($\vec{w} \times \vec{u} \times \vec{v}$) umgewandelt.

```
1         this.w = this.g.normalized().mul(-1.0);
2         this.u = this.t.x(this.w).normalized();
3         this.v = this.w.x(this.u).mul(-1);
```

## 3.2 OrthographicCamera

Die orthographische Kamera leitet sich von Camera ab und implementiert eine Kamera zur orthographischen Darstellung. D.h. die Strahlen die die Kamera generiert verlaufen parallel. Dadurch gibt es keine perspektivischen Verzerrungen siehe „PerspectiveCamera“. Als zusätzlichen Parameter erhält sie

**double s** Ein Skalierungsfaktor, der festlegt, wie groß der Bildausschnitt ist.

```
1 public Ray rayFor(final int w, final int h, final int x,
     final int y) {
2 double aspectRatio = (double) w / (double) h;
3 double scalar1 = aspectRatio * s * (x - (w - 1) / 2) / (w -
     1);
4 double scalar2 = s * (y - (h - 1) / 2) / (h - 1);
5 final Point3 o = this.e.add(this.u.mul(scalar1)).add(
6 this.v.mul(scalar2));
7 final Vector3 d = this.w.mul(-1);
8 return new Ray(o, d);
9 }
```

## 3.3 PerspectiveCamera

Die perspektivische Kamera leitet sich von Camera ab und implementiert eine Kamera zur perspektivischen Darstellung. D.h. die Strahlen die die Kamera generiert entspringen alle dem selben Punkt und verlaufen dann in einem Winkel zu einander der, je weiter zwei Pixel auseinander liegen, immer größer wird. Als zusätzlichen Parameter erhält sie

**double angle** Der Öffnungswinkel der Kamera in **RAD**.

```
1 public Ray rayFor(final int w, final int h, final int x,
     final int y) {
2 final Vector3 summand1 = this.w.mul(-1).mul((h * 1.0 / 2) /
     Math.tan(angle / 2));
3 final Vector3 summand2 = this.u.mul(x - ((w - 1.0) / 2));
4 final Vector3 summand3 = this.v.mul(y - ((h - 1.0) / 2));
5 final Vector3 r = summand1.add(summand2).add(summand3);
6
7 return new Ray(this.e, r.normalized());
8 }
```

## 3.4 Color

Die Color-Klasse erhält 3 Parameter:

**double r** Repräsentiert den Rot-Anteil der Farbe und liegt zwischen 0 und 1.

**double g** Repräsentiert den Grün-Anteil der Farbe und liegt zwischen 0 und 1.

**double b** Repräsentiert den Blau-Anteil der Farbe und liegt zwischen 0 und 1.

Weiter implementiert sie 4 Methoden zur Farbberechnung.

**+add(c:Color)** Addiert die Werte einer Farbe zu der Farbe die durch das Objekt repräsentiert wird und erzeugt ein neues Objekt.

**+sub(c:Color)** Zieht die Werte einer Farbe von der Farbe die durch das Objekt repräsentiert wird ab und erzeugt ein neues Objekt.

**+mul(c:Color)** Multipliziert die Werte einer Farbe mit der Farbe die durch das Objekt repräsentiert wird und erzeugt ein neues Objekt.

**+mul(v:double)** Multipliziert einen double Wert mit der Farbe die durch das Objekt repräsentiert wird und erzeugt ein neues Objekt.

## 3.5 Ray

Ray stellt einen Strahl dar, der von der verwendeten Kamera für einen bestimmten Pixel des zu generierenden Bildes erzeugt wurde. Dabei besteht Ray aus zwei Parametern:

**Point3 o** Repräsentiert den ursprung des Strahls.

**Vector3 d** Repräsentiert die Richtung des Strahls.

Außerdem enthält Ray auch zwei Methoden:

**+at(t:double)** Gibt den Punkt zurück der durch $o + t * d$ repräsentiert wird.

**+tOf(p:Point3)** Gibt das t zurück das durch $\frac{|p-o|}{|d|}$ repräsentiert wird.

## 3.6 Geometrie

Die abstrakte Oberklasse Geometrie wird von allen Geometrien geerbt die der Raytracer unterstützt. Dabei erhält sie als Parameter:

**Color color** welche die Farbe der Geometrie repräsentiert.

Weiter implementiert sie die Methode:

**+hit(r:Ray)** Wird von allen erbenden Klassen überschrieben und gibt zurück ob ein Strahl die Geometrie trifft oder nicht.

## 3.7 Plane

Die einfachste Geometrie ist die Plane. Diese ist unendlich groß und wird lediglich durch 2 Parameter beschrieben:

**Point3 a** Gibt einen Punkt an der garantiert auf der Ebene liegt

**Normal3 n** Stellt die Normale der Ebene dar.

```
1 final double nenner = r.d.dot(n);
2 if (nenner != 0) {
3   final double t = n.dot(a.sub(r.o)) / nenner;
4   if (t > 0) return new Hit(t, r, this);
5 }
6 return null;
```

## 3.8 Sphere

Die Sphere stellt eine perfekte Kugel dar und wird ebenfalls durch zwei Parameter bestimmt:

**Point3 c** Der Ursprung bzw. Mittelpunkt der Sphere

**double r** Der Radius der Sphere.

```
1
2  final double a = r.d.dot(r.d);
3  final double b = r.d.dot(r.o.sub(c).mul(2));
4  final double cn = r.o.sub(c).dot(r.o.sub(c)) - (this.r *
       this.r);
5  final double d = (b * b) - (4 * a * cn);
6
7  if (d > 0) {
8    final double t1 = (-b + Math.sqrt(d)) / (2 * a);
9    final double t2 = (-b - Math.sqrt(d)) / (2 * a);
10   if (t1 >= 0 && t2 >= 0) {
11     return new Hit(Math.min(t1, t2), r, this);
12   } else if (t1 >= 0) {
13     return new Hit(t1, r, this);
14   } else if (t2 >= 0) {
15     return new Hit(t2, r, this);
16   }
17 } else if (d == 0) {
18   final double t = -b / (2 * a);
19   if (t >= 0) {
20     return new Hit(t, r, this);
21   }
22 }
23
24 return null;
```

## 3.9 Triangle

Das Triangle stellt die mächtigste Geometrie im Sortiment dar. Mit seiner Hilfe lassen sich nahezu alle möglichen Geometrien erzeugen. Leider ist die Berechnung etwas aufwändiger als bei den anderen Geometrien, siehe AxisAlignedBox. Als Parameter benötigt das Triangle folgende Werte:

**Point3 a** Stellt eine der drei Ecken dar.

**Point3 b** Stellt eine der drei Ecken dar.

**Point3 c** Stellt eine der drei Ecken dar.

```
1  final Vector3 v = new Vector3(
2  a.x - r.o.x,
3  a.y - r.o.y,
4  a.z - r.o.z);
5  final Mat3x3 m = new Mat3x3(
6  a.x - b.x, a.x - c.x, r.d.x,
7  a.y - b.y, a.y - c.y, r.d.y,
8  a.z - b.z, a.z - c.z, r.d.z
9  );
10 final double detA = m.determinant;
11 final double detA1 = m.col1(v).determinant;
12 final double beta = detA1 / detA;
13 if (beta >= 0 || beta <= 1) {
14   final double detA2 = m.col2(v).determinant;
15   final double gamma = detA2 / detA;
16   if ((beta > 0 && gamma > 0) && beta + gamma <= 1) {
17     final double detA3 = m.col3(v).determinant;
18     final double t = detA3 / detA;
19     if (t>0) return new Hit(t, r, this);
20   }
21 }
22 return null;
```

## 3.10 AxisAlignedBox

Die AxisAlignedBox ist eine Sonderform. Obwohl sich eine Box mittels 12 Triangles erstellen ließe, gibt es auch eine einfachere und schneller zu berechnende Form, zumindest wenn man einige Einschränkungen macht. Die Box muss nämlich genau parallel zu den Achsen des Koordinatensystems liegen. Dadurch braucht man auch lediglich 2 Parameter.

**Point3 run** „Right-Upper-Near“ Stellt die rechte, obere, vordere Ecke dar.

**Point3 lbf** „Left-Bottom-Far“ Stellt die linke untere hintere Ecke dar.

```
1  final Plane[] planes = new Plane[6];
2
3  planes[0] = new Plane( // front layer
4  run,
5  new Normal3(0, 0, 1),
6  color
7  );
8  .
9  .
10 .
11 planes[5] = new Plane( // down layer
12 lbf,
13 new Normal3(0, -1, 0),
14 color
15 );
16
17 Hit max = null;
18
19 for (final Plane plane : planes) {
20   final double condition = r.o.sub(plane.a).dot(plane.n);
21
22   if (condition > 0) {
23     final double t = plane.a.sub(r.o).dot(plane.n) / r.d.
         dot(plane.n);
24     if (max == null || t > max.t) {
25       max = new Hit(t, r, this);
26     }
27   }
28 }
29
30 return comparison(max);
```

## 3.11 Hit

Hit repräsentiert lediglich einen Behälter für den Strahl und das von ihm getroffene Objekt sowie den t-wert des Schnittpunktes.

## 3.12 World

World stellt die eigentliche Szene dar und enthält bis auf die Kamera alle wichtigen Objekte. Sowohl eine Liste aller Geometrien als auch die Hintergrundfarbe der Szene. Außerdem enthält sie auch eine Methode:

**+hit(r:Ray)** Durchläuft alle Geometrien und prüft ob der Strahl trifft. Gibt die Farbe des der Kamera am nächsten liegenden Objektes oder die Hintergrundfarbe zurück.

```
1 Hit hit = null;
2 for (Geometry g : geometries) {
3   final Hit h = g.hit(r);
4   if (hit == null || (h != null && h.t < hit.t)) hit =
        h;
5 }
6 return hit != null ? hit.geo.color : backgroundColor;
```

# 4 Probleme bei der Bearbeitung

Während der Bearbeitung sind mehrere Probleme aufgetreten.

1. Der Formel für die perspektivische Kamera fehlte ein wichtiges Element, nämlich das der Winkel halbiert werden musste. Zwar wurde das während der Vorlesung erwähnt, aus der Formel war das aber nicht zu erkennen.

2. Die AxisAlignedBox wurde nicht durchgängig gerendert sondern hatte an vielen stellen Löcher. Das Problem wurde durch die Ungenauigkeit von double erzeugt und lies sich durch einen winzigen Wert, der im Vergleich mit verrechnet wurde wieder aufheben.