World & Raytracer
Übung 2 // DOC

B. Bleckmann, A. Rezai, V. Risch

May 30, 2016

In den Klassen **World** und **Raytracer** findet die eigentliche Erstellung des Bildes statt. Die Welt hält eine Reihe von Objekten und implementiert selbst auch eine Hit Methode zur Schnitt Berechnung. In dieser wird dann über die Liste der Geometrie Objekte iteriert und das Hit Objekt mit dem kleinsten t zurück gegeben. So wird sicher gestellt, dass nur jene Objekte welche im Vordergrund liegen für die Berechnung der Farbe des Pixels berücksichtigt werden.
In der Klasse Raytracer wird über alle Pixel eines Writeable Raster Objekts iteriert und für jeden dieser Pixel ein Strahl für die Schnittberechung erstellt. Als Nächstes wird die Hit Methode der World aufgerufen und im Falle eines Treffers für ein Objekt mit dem kleinsten t des Farbe im Raster gesetzt. Wird kein Objekt geschnitten so wird für den Pixel die Farbe der Welt gesetzt.
Da unsere Color Klasse nur eine normalisierte abstrakte Form des RGB Farbmodels ist, in welchem sich die Werte für eine Farbe nur zwischen 0.0 und 1.0 bewegen dürfen, haben wir im Raytracer eine Instanz der java.awt.Color Klasse erzeugt und multiplizieren deren r,g und b Werte mit unseren.

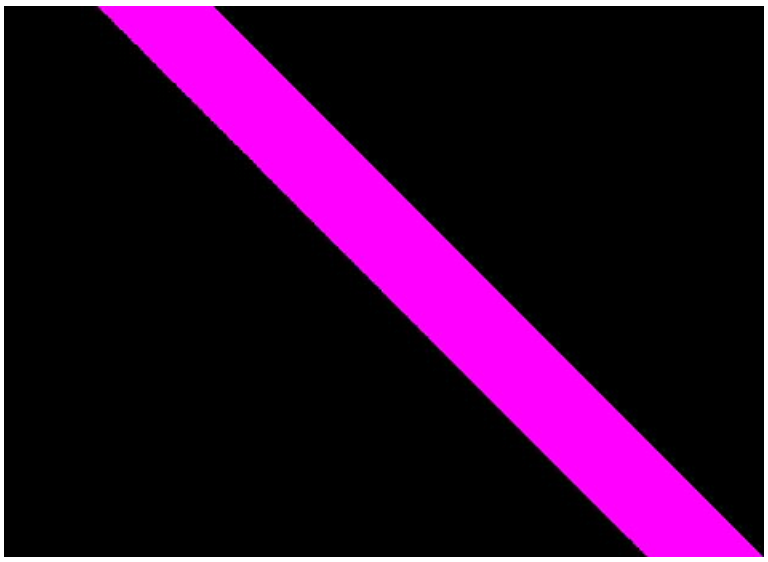
Geometrien

Die Implementierung der **Sphere** verlief größtenteils ohne Probleme und konnte ordnungsgemäß implementiert werden.

Bei der Implementierung der **Plane** stießen wir auf eine Abweichung, uns fiel auf, dass sie "gespiegelt" wurde (wenn $t < 0$ war Plane sichtbar, wenn $t > 0$ nicht). Die Ursache dafür: in der CG liegt der Ursprung (0/0 Koordinate) links unten im Bild, in unserer java GUI liegt der Ursprung (AnchorPane) rechts oben im Bild. Wir konnten das Problem lösen durch anpassen des y-wertes durch (umdrehen der y-indices) und -1 um beim Indize bei 0 anzufangen.

```
raster.setDataElements(x,raster.getHeight()-y-1, colorModel.getDataElements(color1.getRGB(),null)); // get rid of mirroring
```

Bei dem Implementieren des Triangle erhielten wir einen Fehler in der perspektivischen Ansicht



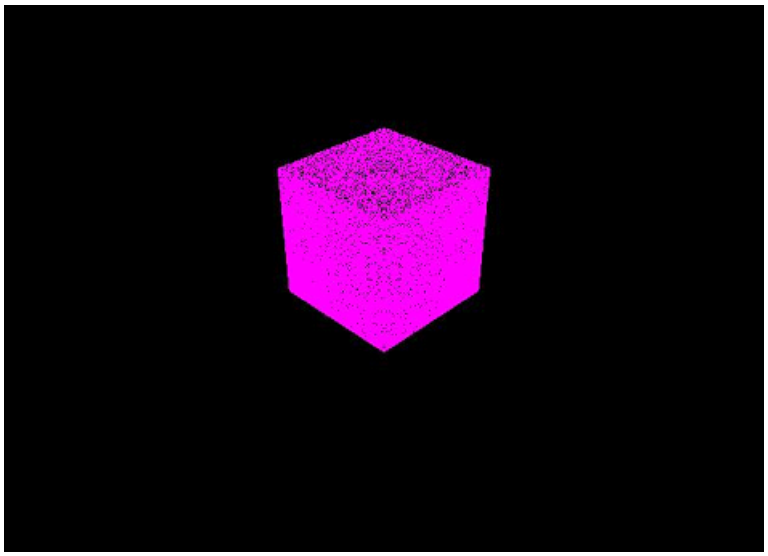
In der orthographischen Ansicht war dieser Fehler jedoch nicht erkenntlich (das Dreieck wurde korrekt angezeigt). Schlussendlich wurde der Fehler hervorgerufen durch einen falschen Wert in der Berechnung der Determinanten. Dies wurde uns klar, als wir einen neuen Testfall für die Determinante implementierten und dieser uns ein falsches Ergebnis lieferte.

```
java.lang.AssertionError:
Expected :0.0
Actual   :-96.0
 <Click to see difference>

<1 internal calls>
    at org.junit.Assert.failNotEquals(Assert.java:834) <2 internal calls>
    at de.beuth.cg1.dogeraytracer.junit.Mat3x3Test.calcDeterminante(Mat3x3Test.java:43) <29 internal calls>
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

Beim Evaluieren der Berechnung fiel auf, dass 1 Wert falsch gesetzt wurde.

Bei der Implementierung der **Axis-Aligned-Box** kam es zu einem Fehler der Darstellung, nicht alle Pixel wurden ordnungsgemäß angezeigt / getroffen.



Das Problem waren Rundungsfehler bei der Abfrage, ob ein Hit innerhalb der Axis-Aligned-Box liegt.
Die Lösung für dieses Problem führte uns schlussendlich zum definieren einer Variablen, die uns hilft Rundungsfehler zu vermeiden.

```
/**
 * DELTA is Used for round-off errors
 */
public static double DELTA = 0.00001;
```

Camera

Die Implementierung der Camera(s) verlief ohne Probleme.