Was ist Raytracing?

Raytracing bezeichnet eine Methode zur Generierung von realistischen 3D Bildern. Dabei kommen unter anderem Reflexions- und Brechungsgesetze aus der Physik zur Anwendung. Vor allem aber beruht die Technik auf der Vektorgeometrie wie man sie am Gymnasium unterrichtet. Die Idee ist folgende: Wir verteilen einige Objekte (Kugeln, Würfel, etc.) im dreidimensionalen Raum. Wir nennen dies die Szene. Nun stellt man irgendwo eine Kamera auf. Vor der Kamera stellen wir uns eine rechteckige Fläche, die Bildebene, vor. Auf die Bildebene zeichnen wir nun die dahinter liegende Szene wie folgt: Ausgehend von der Kamera senden wir einen Strahl aus, der die Bildebene durchstösst und dann womöglich auf ein Objekt unserer Szene trifft. Trifft der Strahl zum Beispiel auf ein rotes Dreieck, so wird der Durchstosspunkt auf der Bildebene rot eingefärbt. Danach senden wir einen weiteren Strahl von der Kamera durch einen anderen Punkt auf der Bildebene und färben den Durchstosspunkt entsprechend ein. Wir wiederholen dies bis die ganze Bildebene eingefärbt ist. Das so auf der Bildebene generierte Abbild der Szene ist der Output unseres Programms. Diese Methode ein Bild zu generieren heisst Raytracing, denn man verfolgt den Strahl.

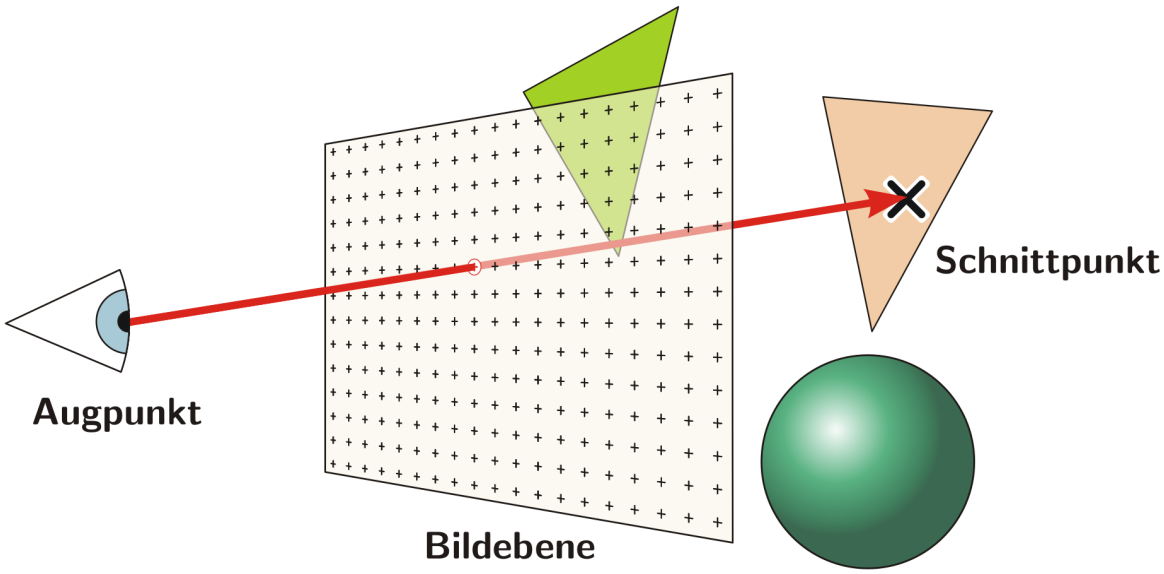

Abbildung 1: Raytracing: Es werden Strahlen ausgesendet und deren Schnittpunkte mit Objekten ermittelt. In diesem Fall wir der Durchstosspunkt auf der Bildebene in der Farbe des roten Dreiecks gefärbt.

Worum geht es im Folgenden?

Wir wollen nun unser eigenes Raytracing-Programm in Python programmieren. Genauer gesagt ist der Raytracing-Algorithmus im Kern schon implementiert. Allerdings kennt diese Implementierung noch keine Objekte wie Kugeln, Würfel, Ebenen und so weiter. Hier kommen Sie als Leser ins Spiel. Sie werden unter Anleitung diese fehlenden Teile ergänzen. Es geht darum dem Raytracing-Programm zu sagen, was zum Beispiel eine Kugel ist, wie man deren Schnittpunkt mit einem Strahl berechnet und vieles mehr. Am Ende werden wir Bilder wie in Abbildung 2 generieren können. Dabei müssen Sie

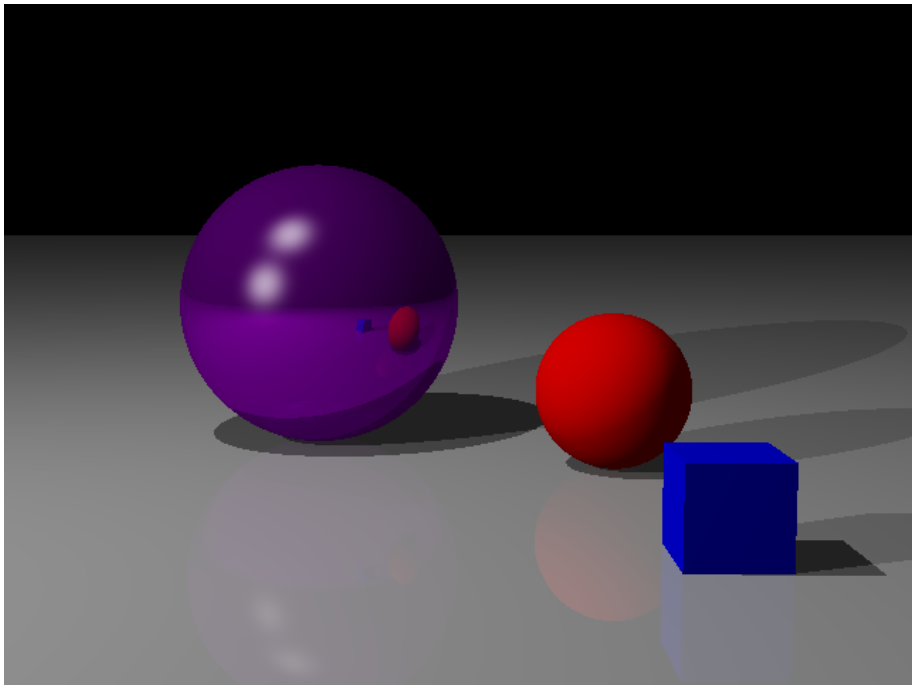

Abbildung 2: Dieses Bild wurde mit unserem Raytracing-Programm generiert.

als Leser ausschliesslich die Python-Files im Verzeichnis `myobject` editieren. Die darin enthaltenen Klassen werden dann die verschiedenen Objekt-Typen wie Kugeln, Ebenen und dergleichen beschreiben. Ohne hier weiter ins Detail zu gehen, werden wir nun unser erstes Bild generieren.

Die Kugel

Als Einstieg generieren wir ein Bild bestehend aus folgender Szene: Eine Kamera befindet sich an den Koordinaten $(-1, 0, 1)$ und schaut in Richtung des Punktes $(0, 0, 1)$, das heisst entlang der x-Achse. Zudem platzieren wir eine Kugel mit Mittelpunkt $(1, 0, 1)$ und Radius 1. Damit schaut die Kamera genau auf die Kugel. Diese Szene entspricht dem Python-File `examples/example1.py`, welches wir nun ausführen. Anstatt eine Kugel sehen wir aber nur ein schwarzes Bild. Um die Kugel auch zu sehen, müssen wir zuerst das File `myobject/sphere.py` bearbeiten. Genauer gesagt, muss die Funktion `intersect(self, ray)` vervollständigt werden. Diese soll zu einem gegebenen Strahl dessen nächstgelegenen Schnittpunkt berechnen. Man betrachte dazu Abbildung 3.

Aufgabe 1. Gegeben Sei ein Strahl mit Ursprung $\vec{v}$ und Richtung $\vec{w}$. Überlegen Sie sich die Kugel mit Mittelpunkt $\vec{m}$ und Radius r schneidet. Wie erhalten Sie in diesem Fall den Parameter $t > 0$, so dass $\vec{v}+t\vec{w}$ gerade dem näherem der beiden Schnittpunkte entspricht?

Lösung. Ein beliebiger Punkt auf dem Strahl ist von der Form $\vec{v} + s\vec{w}$ für ein $s > 0$. So ein Punkt liegt auf der Kugeloberfläche genau dann wenn

$$\|\vec{v} + s\vec{w} - \vec{m}\|^2 = r^2,$$

also wenn er den Abstand r zum Mittelpunkt hat. Dies ist eine quadratische Gleichung in s, das heisst sie ist von der Form

$$as^2 + bs + c = 0$$

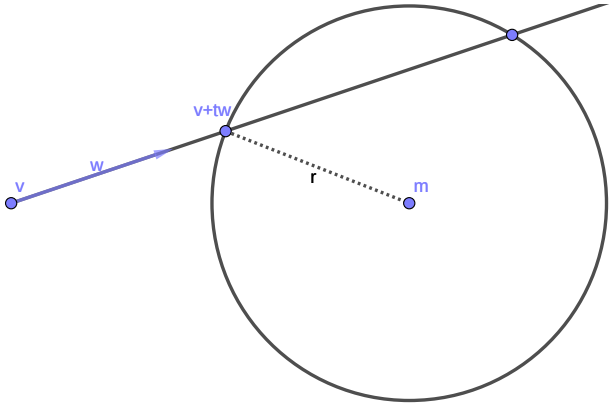

Abbildung 3: Schnittpunkt von Strahl und Kugel.

für reelle Zahlen a, b und c. Durch einen Koeffizientenvergleich erhält man

$$a = \|\vec{w}\|^2, \quad b = 2\vec{w} \cdot (\vec{v} - \vec{m}), \quad c = \|\vec{v} - \vec{m}\|^2 - r^2.$$

Falls $b^2 - 4ac > 0$, so existieren genau zwei Schnittpunkte $v + t_1 w$ und $v + t_2 w$, wobei

$$t_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad \text{und} \quad t_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}.$$

Wir sind aber nur an positiven Lösungen interessiert, denn wir beschreiben einen Strahl und keine Gerade. Sind t_1 und t_2 beide negativ, so schneidet der Strahl die Kugel nicht. Andernfalls ist die kleinste positive Lösung der quadratischen Gleichung unsere Wahl für t. Der nächstgelegene Schnittpunkt ist entsprechend $\vec{v} + t\vec{w}$.

Aufgabe 2. Öffnen Sie nun das File `myobject/sphere.py` und implementieren Sie die Funktion `intersect(self, ray)` gemäss Ihren Überlegungen aus Aufgabe 1. Lassen Sie anschliessend das Skript `examples/example1.py` nochmals laufen. Nun sollten Sie die Kugel sehen.

Lösung. Die Lösung könnte zum Beipiel so aussehen:

```
2     def intersect(self, ray):
3         v = ray.origin
4         w = ray.direction
5
6         a = inner(w, w)
7         mv = v - self.m
8         b = 2.0 * inner(w, mv)
9         c = inner(mv, mv) - self.r**2
10        d = b**2 - 4.0 * a * c
11
12        if d >= 0.0:
13            t0 = (-b - sqrt(d)) / (2.0 * a)
14            if t0 > 0.0 and inner(ray(t0) - self.m, w) < 0.0:
15                return t0
16            t1 = (-b + sqrt(d)) / (2.0 * a)
17            if t1 > 0.0 and inner(ray(t1) - self.m, w) < 0.0:
18                return t1
19        return inf
20
```

../object/sphere.py

Zusätzlich zu unserer Lösung von Aufgabe 1 haben wir hier noch überprüft, ob der Strahl am Schnittpunkt in die Kugel eintritt (und nicht etwa austritt). Nur diese Lösung lassen wir zu. Wir werden später sehen, warum das nützlich ist. Das so generierte Bild ist in Abbildung 4 gezeigt.

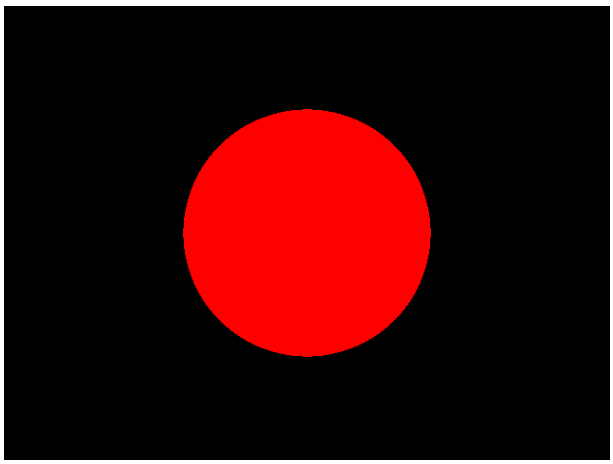

Abbildung 4: Lösung der Aufgabe 2.

Unsere Kugel sieht momentan eher wie eine Kreisscheibe aus, weil wir der Kugel an jedem Punkt den selben Farbwert geben. Das wollen wir nun ändern indem wir eine diffuse Lichtreflexion simulieren. Das Python-File `examples/example2.py` platziert zu diesem Zweck eine Punktförmige Lichtquelle an den Koordinaten $(0, 0, 10)$. Diese wollen wir nun in die Berechnung der Farbwerte miteinbeziehen: Die Punkte auf der Kugeloberfläche, welche der Lichtquelle zugewandt sind, sollen heller sein. Die wichtigsten Begriffe sind das *Skalarprodukt* und der *Normalenvektor* auf die Kugeloberfläche. Man betrachte dazu Abbildung 5. Diese zeigt eine punktförmige Lichtquelle am Ort $\vec{v}$. Wir wollen den Farbwert am Punkt $\vec{p}$ auf der Kugel berechnen. Der Vektor $\vec{n}$ an diesem Punkt soll rechtwinklig zur Kugeloberfläche sein, nach aussen Zeigen und Länge 1 haben. Wir nennen $\vec{n}$ den Normalenvektor auf die Kugel im Punkt $\vec{p}$. Wir ordnen dem Punkt $\vec{p}$ eine Zahl $c \in [0, 1]$

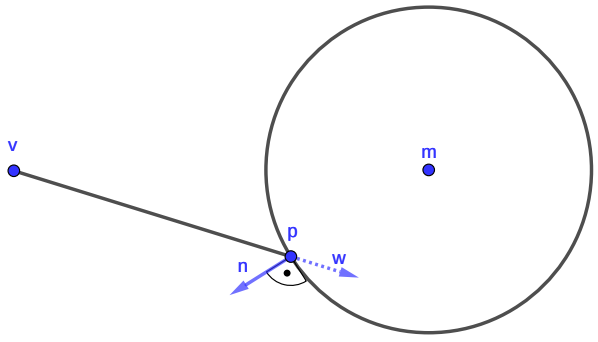

Abbildung 5: Die Kugel wird von einer Punktförmigen Lichtquelle in $\vec{v}$ beleuchtet.

zu, welche die Helligkeit an diesem Punkt beschreibt. Dabei bedeutet $c = 1$ maximale Helligkeit und $c = 0$ minimale Helligkeit, also Schwarz:

$$c = \max\left(-\vec{n} \cdot \vec{w}, 0\right), \quad \vec{w} = \frac{\vec{p} - \vec{v}}{\|\vec{p} - \vec{v}\|}. \tag{1}$$

Aufgabe 3. Warum misst c in Gleichung (1) die Helligkeit? Erklären Sie.

Lösung. Da sowohl $\vec{n}$ als auch $\vec{w}$ Länge 1 haben, gilt wie verlangt $c \in [0, 1]$. Der Vektor $\vec{w}$ beschreibt gerade die Einfallsrichtung des Lichtes am Punkt $\vec{p}$. Ist diese rechtwinklig auf die Kugeloberfläche, so haben wir $c = 1$, also maximale Helligkeit. Je kleiner der Einfallwinkel des Lichtes, desto kleiner wird c, bis schliesslich alle Punkte auf der der Lichtquelle abgewandten Seite die Helligkeit $c = 0$ haben.

Aufgabe 4. Öffnen Sie nun das File `myobject/sphere.py` und implementieren Sie die Funktion `get_normal(self, p)`, nach aussen zeigenden Normalenvektor $\vec{n}$ der Lange 1 am Punkt $\vec{p}$ zurück gibt. Lassen Sie anschliessend das Skript `examples/example2.py` laufen.

Lösung. Die Lösung könnte zum Beipiel so aussehen:

```
2     def get_normal(self, p):
3         return normalize(p - self.m)
4
```

../object/sphere.py

Das generierte Bild in Abbildung 6 sieht schon viel interessanter aus. Die Lichtquelle befindet sich über der Kugel und beleuchtet nur deren obere Hälfte.

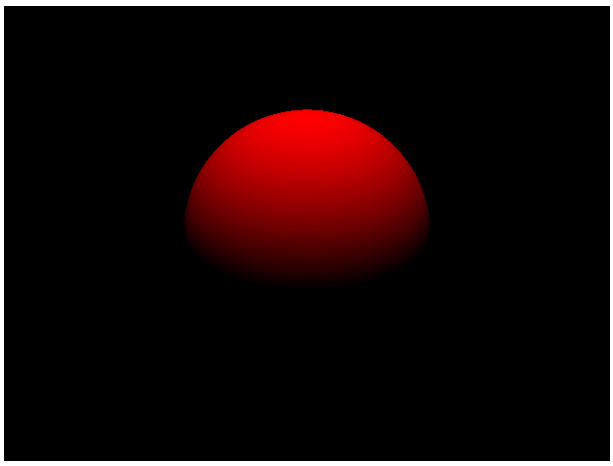

Abbildung 6: Lösung der Aufgabe 4.