ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Mentorierte Arbeit in Fachwissenschaftlicher Vertiefung
mit pädagogischem Fokus in Mathematik

Raytracing und Vektorgeometrie

Oliver Rietmann

Inhalt
In dieser Arbeit werden wir eine konkrete Anwendung der Vektorgeometrie in der Computergraphik studieren: Das Raytracing. Dabei handelt es sich um eine Methode zur Generierung realistischer 3D Bilder. Wir werden unser eigenes Raytracing-Programm in Python implementieren und dabei sehen, wie mächtig scheinbar abstrakte Mathematik im Zusammenspiel mit Physik und Informatik sein kann.

Zielpublikum
Gymnasiale Mittelschüler

Voraussetzungen
Vertrautheit mit der Vektorgeometrie wie sie am Gymnasium unterrichtet wird. Zudem werden grundlegende Programmierkenntnisse vorausgesetzt.

Form
Text mit Aufgaben und Lösungen. Die Python-Codes befinden sich im Anhang. Alternativ können sie online heruntergeladen werden. (Der Link befindet sich in der Arbeit.)

Betreuung
Christian Rüede

Datum
4. Januar 2020

Inhaltsverzeichnis

Einleitung

Dieses Skript richtet sich an gymnasiale Mittelschüler die mit den grundlegenden Begriffen der Vektorgeometrie im dreidimensionalen euklidischen Raum vertraut sind. Ziel ist es, die Vektorgeometrie zu veranschaulichen indem wir eine konkrete Anwendung in der Computergrafik betrachten, das Raytracing. Dabei handelt es sich um eine Technik zur Generierung von realistischen 3D Bildern, welche im nächsten Kapitel genauer erklärt wird. Wir werden den Raytracing-Algorithmus selber implementieren und damit solche Bilder generieren. Wir verwenden dazu die Programmiersprache Python, welche in Kapitel 2 kurz eingeführt wird. Grundkenntnisse im Programmieren werden dabei vorausgesetzt. Wer noch nie programmiert hat, kann das zum Beispiel mit einem der vielen online-Tutorials nachholen, bestenfalls gleich in Python.

1 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.

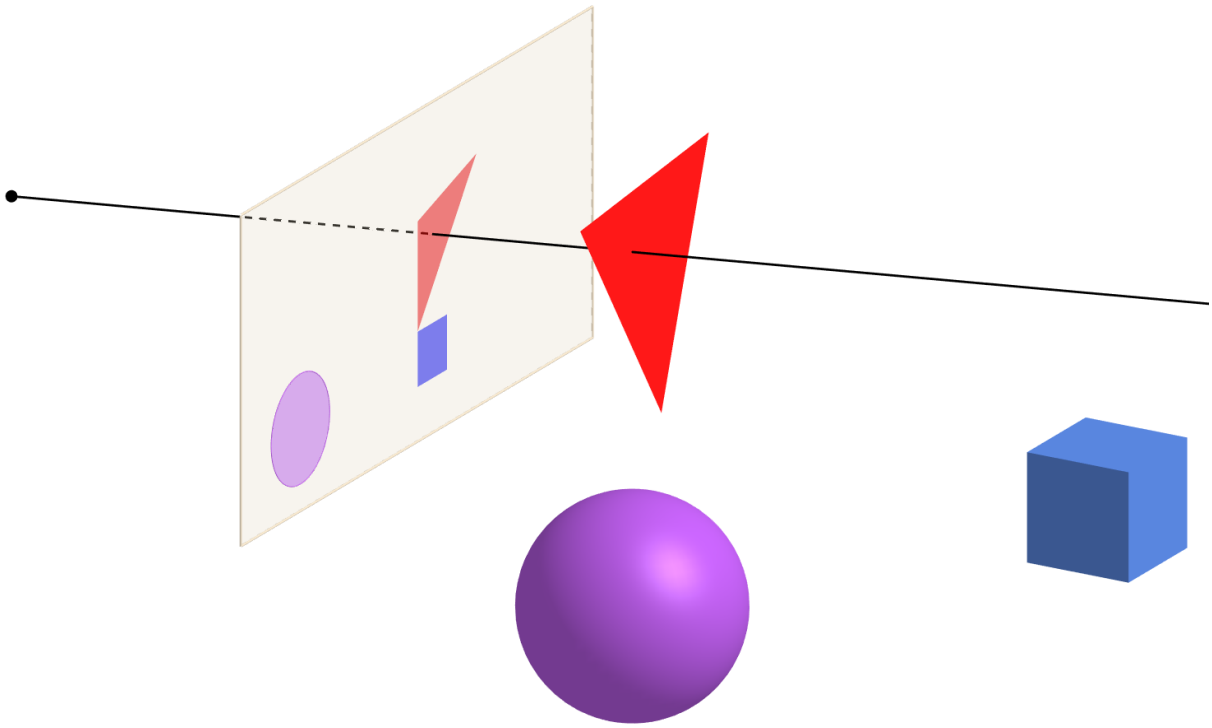

Abbildung 1: Raytracing: Es werden Strahlen ausgesendet und deren Schnittpunkte mit Objekten ermittelt. In diesem Fall wir der Durchstosspunkt auf der Bildebene in der Farbe des roten Dreiecks gefärbt.

2 Eine Einführung in Python 3

Nun installieren wir Python (Version 3.8 oder höher) und die von unserem Raytracing-Programm benötigten Packages `numpy` und `PIL`. Je nach Betriebssystem muss dabei anders vorgegangen werden.

Windows 10

Gehen Sie auf die offizielle Python-Webseite um von dort die neuste Python Version zu installieren. Danach überprüfen wir mit folgenden drei Schritten, ob die Installation erfolgreich war:

1. Drücken Sie die Windows-Taste um eine Suche zu starten.
2. Schreiben Sie dann `cmd` und drücken Sie Enter um die Kommandozeile zu öffnen.
3. In der Kommandozeile schreiben Sie `python` und drücken wieder Enter.

Nun sollte die Version der Python-Installation ausgegeben werden. Anschliessend können in der Kommandozeile mit

```
python -m pip install numpy Pillow --user
```

die Abhängigkeiten installiert werden. Von nun an kann ein Python-Script `script.py` in der Kommandozeile ausgeführt werden mit dem Befehl `python script.py`. Dazu muss aber zuerst mit dem Befehl `cd <Verzeichnis>` in das Verzeichnis navigiert werden, welches `script.py` enthält.

MacOS

Öffnen Sie einen Terminal, zum Beispliel indem Sie nach der Applikation Terminal suchen. Installieren Sie dann Homebrew indem Sie die Zeile (alles soll auf eine Zeile!)

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/master/install.sh)"
```

im den Terminal kopieren und mit Enter ausführen. Anschliessend können Sie Python mit dem Befehl `brew install python` installieren. Danach müssen noch mit

```
python3 -m pip install numpy Pillow --user
```

die Abhängigkeiten installiert werden. Von nun an kann ein Python-Script `script.py` im Terminal ausgeführt werden mit dem Befehl `python3 script.py`. Dazu muss aber zuerst mit dem Befehl `cd <Verzeichnis>` in das Verzeichnis navigiert werden, welches `script.py` enthält.

Linux

Öffnen Sie einen Terminal. Unter Ubuntu wird Python mit dem Befehl `sudo apt-get install python3` installiert, unter Fedora hingegen mit `sudo dnf install python3`. In beiden Fällen werden Administrator-Rechte benötigt. Anschliessend können mit

```
python3 -m pip install numpy Pillow --user
```

die Abhängigkeiten installiert werden. Von nun an kann ein Python-Script `script.py` im Terminal ausgeführt werden mit dem Befehl `python3 script.py`. Dazu muss aber zuerst mit dem Befehl `cd <Verzeichnis>` in das Verzeichnis navigiert werden, welches `script.py` enthält.

2.1 Listen und Schleifen

Öffnen Sie nun einen beliebigen Texteditor und erstellen Sie ein File namens `script.py` das nur folgende Zeile enthält:

```
print("Hello World")
```

Wenn Sie dieses im Terminal ausführen wie oben beschrieben, sollte die Nachricht `Hello World` erscheinen. Die Endung `.py` kennzeichnet das File als Python-Skript. Mit der Funktion `print` kann Text, eine Zahl, und vieles mehr im Terminal ausgegeben werden. Zum Beispiel können wir eine Liste von Zahlen ausgeben:

```
l = [4, 2, 5, 9, 6, 2]
print(l)
```

Mithilfe der Klammern `[ ]` haben wir hier die Variable `l` definiert und schliesslich mit `print` ausgegeben. Mit den Klammern `[ ]` kann man zudem auf die einzelnen Elemente der Liste zugreifen:

```
l = [4, 2, 5, 9, 6, 2]
print(l[0])
print(l[1])
print(l[2])
```

Die `for` Schleife erlaubt es eine Liste zu durchlaufen:

```
l = [4, 2, 5, 9, 6, 2]
for i in l:
    print(i)
```

Die Variable `i` durchläuft alle Werte der Liste `l`. Man beachte, dass die letzte Zeile eingerückt ist. Um aus einer Liste von Zahlen einen Vektor zu machen mit dem man rechnen kann, verwenden wir das Package numpy:

```
from numpy import array
l = [4, 2, 5, 9, 6, 2]
v = array(l)
print(2 * v)
```

So wird das `numpy.array`, bzw. der Vektor `v` mit dem Skalar `2` multipliziert. Man beachte, das die gewöhnliche Liste `l` diese Operation nicht unterstützt (probieren Sie!). Numpy stellt viele weitere Operationen aus der Vektorgeometrie zur Verfügung. Wie man auf diese zugreifen kann, schaut man am besten im Internet nach. In jedem Fall müssen diese aber zuerst importiert werden, so wie wir `numpy.array` in in der ersten Zeile importieren mussten. Nun folgen noch ein paar wichtige Beispiele:

```
from numpy import array, inner
from numpy.linalg import norm
v = array([1, 2, 3])
w = array([4, 5, 6])
print(w - 2 * v)
```

```
6 print(inner(v, w)) # Skalarprodukt von v und w
7 print(norm(v))      # Betrag des Vektors v
```

Der Hashtag # markiert einen Kommentar, das heisst alle darauffolgenden Zeichen werden ignoriert.

2.2 Funktionen und Klassen

Funktionen sind ein Weg um Code-Repetition zu verhindern. So könnte es zum Beispiel sehr oft nötig sein, einen Vektor zu normieren (das heisst auf Länge 1 zu bringen). In diesem Fall lohnt es sich, dafür eine Funktion zu machen:

```
1 from numpy import array
2 from numpy.linalg import norm
3
4 def normalize(v):
5     return v / norm(v)
6
7 w = array([1, 2, 3])
8 n = normalize(v)
9 print(n)
```

Wir haben also eine Funktion `normalize` definiert, welche einen beliebigen Vektor (ausser den Nullvektor) auf Länge 1 bringt. Wie schon beim der `for` Schleife ist auch hier die Zeile 5 eingerückt. Das Ende dieser Einrückung markiert das Ende der Definition der Funktion. Im Gegensatz zu vielen anderen Programmiersprachen hat das einrücken in Python eine syntaktische Bedeutung. Ein weiteres wichtiges Konzept ist die `Klasse`. Als Beispiel wollen wir eine Kugel beschreiben. Eine Kugel hat zwei definierende Eigenschaften: Einen Mittelpunkt und einen Radius. Daraus leiten sich andere Eigenschaften wie zum Beispiel ihr Volumen ab. Diese Information lässt sich in Code übersetzen indem man zum Beispiel eine Klasse `Sphere` definiert:

```
1 from numpy import pi
2
3 class Sphere:
4     def __init__(self, m, r):
5         self.m = m
6         self.r = r
7
8     def volume(self):
9         return 4 * pi * self.r**3 / 3
```

Dieser Klasse geben wir die zwei Funktionen `__init__` und `volume`. Ersteres setzt die Member-Variablen `m` und `r`, welche wir als Mittelpunkt bzw. als Radius interpretieren. Member-Funktionen wie `volume` können diese dann verwenden. Ähnlich wie zuvor bei den Funktionen kann der weitere Code diese Klasse wiederverwenden:

```
1 from numpy import array, pi
2
3 class Sphere:
4     def __init__(self, m, r):
5         self.m = m
6         self.r = r
7
8     def volume(self):
9         return 4 * pi * self.r**3 / 3
10
11 # Eine Kugel um [0, 0, 1] mit Radius 2
12 sphere = Sphere(array([0, 0, 1]), 2)
13
14 print("Mittelpunkt:", sphere.r)
```

```
print("Radius:", sphere.r)
print("Volumen: ", sphere.volume())
```

Die Variable `sphere` nennt man eine *Instanz* der Klasse `Sphere`. Mit dem Punkt `.` können auf die Member-Variablen und Member-Funktionen einer Instanz zugegriffen werden. Python Files können Funktionen und Klassen von anderen Files im selben Verzeichnis *importieren*. Sagen wir im File `normalize.py` sei die Funktion `normalize` definiert worden und im File `sphere.py` sei die Klasse `Sphere` definiert worden. Ein drittes File (im selben Verzeichnis wie die anderen beiden) könnte dann zum Beispiel so aussehen:

```
from numpy import array
from normalize import normalize
from sphere import Sphere

w = array([1, 2, 3])
n = normalize(v)
print(n)

# Eine Kugel um [0, 0, 1] mit Radius 2
sphere = Sphere(array([0, 0, 1]), 2)

print("Mittelpunkt:", sphere.r)
print("Radius:", sphere.r)
print("Volumen: ", sphere.volume())
```

Auf diese Weise kann ein einziges Programm in viele verschiedene Files aufgeteilt werden. Damit beenden wir unsere Einführung und vertrauen darauf, dass der Leser sich weitere Programmierkenntnisse selber aneignet, wenn er sie denn benötigen sollte.

3 Unser eigenes Raytracing Programm

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

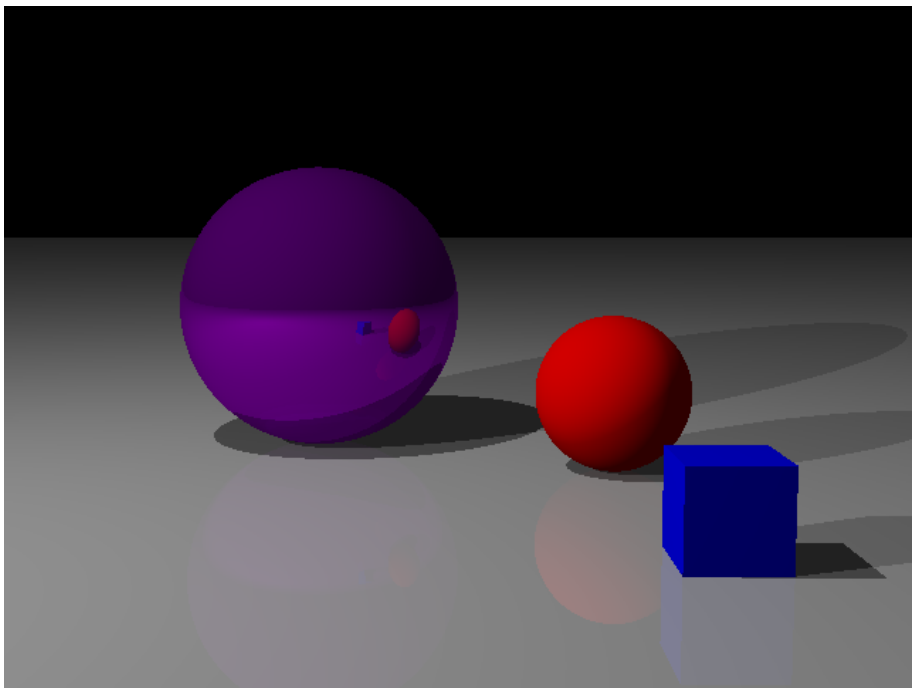

Abbildung 2: Dieses Bild wurde mit unserem Raytracing-Programm generiert.

man deren Schnittpunkt mit einem Strahl berechnet und vieles mehr. Laden Sie dazu unter `https://gitlab.math.ethz.ch/rioliver/raytracing` die Python-Codes herunter. Um zu testen ob alles funktioniert, können Sie dann das File `main.py` ausführen. Dies sollte das Bild aus Abbildung 2 generieren. In `main.py` wird eine Szene beschrieben, das heisst es werden Objekte wie Kugeln und Ebenen platziert. Diese entsprechen wiederum Klassen, welche in den Verzeichnissen `object` und `myobject` definiert werden. Der eigentliche Raytracing-Code befindet sich im Verzeichnis `core`. Beim Ausführen ruft `main.py` die Funktionen aus `core` auf um schliesslich aus der Szene ein Bild zu generieren.

3.1 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 `example1.py`.

```python
from numpy import array, pi

from core.camera import Camera
from core.renderer import Renderer
from myobject.sphere import Sphere

# Platziere eine Kamera im Punkt [-1.0, 0.0, 1.0] welche in Richtung
# des Punktes [0.0, 0.0, 1.0] schaut.
camera = Camera(array([-1.0, 0.0, 1.0]), array([0.0, 0.0, 1.0]), 0.25 *
    pi, 640, 480)

# Renderer ist eine Klasse zum Generieren der Bilder
renderer = Renderer(camera)

# Eine Kugel um [5.0, 0.0, 1.0] mit Radius 1 (in der Farbe rot)
sphere = Sphere(array([5.0, 0.0, 1.0]), 1.0)

# Man übergiebt dem Renderer eine Liste von Objekten (die Kugel)
renderer([], [sphere], photo_exposure=0.0)

# Hier wird das Bild generiert und abgespeichert
renderer.save_image("example1.png")
```

../raytracer/example1.py

Wenn wir dieses Skript ausführen, sehen wir anstatt einer Kugel 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.

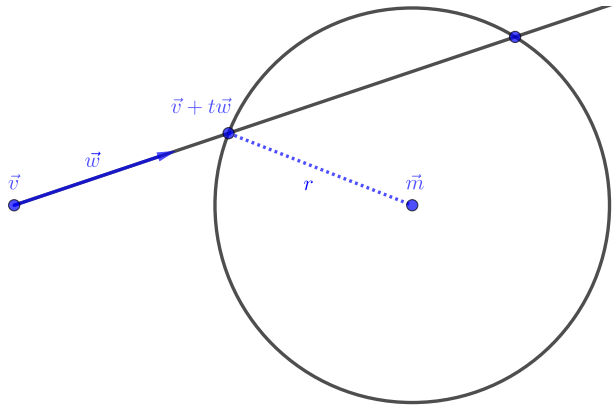

Abbildung 3: Schnittpunkt von Strahl und Kugel.

Aufgabe 1. Gegeben sei eine Kugel mit Mittelpunkt $\vec{m}$ und Radius $r > 0$, sowie ein Strahl mit Ursprung $\vec{v}$ und Richtung $\vec{w}$. Der Strahl besteht also aus der Menge aller Punkte der

Form $\vec{v} + t\vec{w}$ für ein $t > 0$. Sie dürfen dabei annehmen, dass $\vec{w}$ nicht der Nullvektor ist. Wie kann man entscheiden, ob der Strahl die Kugel schneidet? Wie erhalten Sie in diesem Fall den Parameter $s > 0$, so dass $\vec{v} + s\vec{w}$ gerade dem näherem der beiden Schnittpunkte entspricht?

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

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

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

$$at^2 + bt + c = 0$$

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 s. Der nächstgelegene Schnittpunkt ist entsprechend $\vec{v} + s\vec{w}$.

Aufgabe 2. Öffnen Sie nun das File `myobject/sphere.py` und implementieren Sie die Funktion `intersect(self, v, w)` gemäss Ihren Überlegungen aus Aufgabe 1. Die Argumente `v,w` sind die Vektoren $\vec{v}, \vec{w}$ aus Abbildung 3. Der Rückgabewert ist die Zahl $s > 0$, so dass $\vec{v} + s\vec{w}$ dem Schnittpunkt von Strahl und Kugel entspricht und $s = \infty$, falls kein Schnittpunkt existiert. Lassen Sie anschliessend das Skript `example1.py` nochmals laufen. Nun sollten Sie die Kugel sehen.

Lösung. Die Lösung könnte zum Beipiel so aussehen:

```
2     def intersect(self, v, w):
3         # Der Strahl ist beschrieben durch v+t*w mit t>0
4
5         # Berechung der Koeffizienten der quadratischen Gleichung
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
11        # Diskriminante
12        d = b**2 - 4.0 * a * c
13
14        # Fallunterscheidung: Schneidet der Strahl die Kugel?
15        if d >= 0.0:
16            t0 = (-b - sqrt(d)) / (2.0 * a)
17            if t0 > 0.0 and inner(v + t0 * w - self.m, w) < 0.0:
18                return t0
19            t1 = (-b + sqrt(d)) / (2.0 * a)
```

```
20              if t1 > 0.0 and inner(v + t1 * w - self.m, w) < 0.0:
21                  return t1
22
23          # Falls kein Schnittpunkt existert, retourniere "unendlich"
24          return inf
25
```

../raytracer/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.

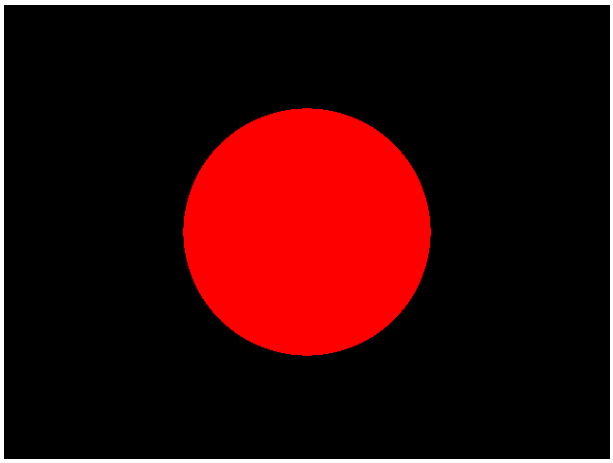

Abbildung 4: `example1.py`: Lösung der Aufgabe 2.

3.2 Licht und Beleuchtung

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 `example2.py` platziert zu diesem Zweck eine Punktförmige Lichtquelle an den Koordinaten $(0, 0, 10)$.

```python
from numpy import array, pi

from core.camera import Camera
from core.renderer import Renderer
from myobject.sphere import Sphere

camera = Camera(array([-1.0, 0.0, 1.0]), array([0.0, 0.0, 1.0]), 0.25 *
    pi, 640, 480)
renderer = Renderer(camera)

# Neu: Eine Punktförmige Lichtquelle an der Position [0, 0, 10]
lightsource = array([0.0, 0.0, 10.0])

# Die Argumente "ambient" und "diffuse" werden später erklärt
sphere = Sphere(array([5.0, 0.0, 1.0]), 1.0, ambient=0.0, diffuse=1.0)

# Neu: Dem Renderer wird nun auch die Lichtquelle übergeben
renderer([lightsource], [sphere], photo_exposure=0.0)
renderer.save_image("example2.png")
```

../raytracer/example2.py

Diese Lichtquelle 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 im Punkt $\vec{L}$. 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

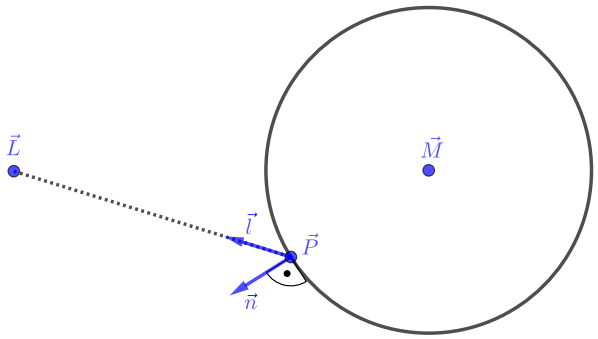

Abbildung 5: Die Kugel wird von einer Punktförmigen Lichtquelle in $\vec{v}$ beleuchtet.

Punkt $\vec{P}$ eine Zahl $I \in [0, 1]$ zu, welche die Lichtintensität („Helligkeit“) an diesem Punkt beschreibt. Dabei bedeutet $I = 1$ maximale Intensität und $I = 0$ minimale Intensität, also schwarz:

$$I = \max\left(0, \vec{n} \cdot \vec{l}\right), \quad \vec{l} = \frac{\vec{L} - \vec{P}}{\|\vec{L} - \vec{P}\|}. \tag{1}$$

Aufgabe 3. Überlegen Sie sich qualitativ, welche Punkte $\vec{P}$ auf der Kugeloberfläche welchen Wert für I zugeordnet bekommen: Welche Punkte auf der Kugeloberfläche werden als „hell“ und welche als „dunkel“ erscheinen?

Lösung. Da sowohl $\vec{n}$ als auch $\vec{l}$ Länge 1 haben, gilt

$$\cos(\alpha) = \vec{n} \cdot \vec{l},$$

wobei α der Zwischenwinkel von $\vec{n}$ und $\vec{l}$ ist. Da der Kosinus nur Werte in $[-1, 1]$ annimmt, gilt wie verlangt $I \in [0, 1]$. Die Intensität hängt also vom Einfallswinkel des Lichtes ab: Scheint das Licht rechtwinklig auf die Kugeloberfläche im Punkt $\vec{P}$, so haben wir $\alpha = 0$ und damit $I = 1$, also maximale Intensität. Je flacher der Einfallwinkel des Lichtes, desto kleiner wird I, bis schliesslich alle Punkte auf der der Lichtquelle abgewandten Seite die Intensität $I = 0$ haben.

Aufgabe 4. Öffnen Sie nun das File `myobject/sphere.py` und implementieren Sie die Funktion `get_normal(self, p)`, die den nach aussen zeigenden Normalenvektor $\vec{n}$ der Länge 1 am Punkt $\vec{P}$ zurück gibt, wobei letzterer dem Argument `p` entspricht.

Lösung. Die Lösung könnte zum Beispiel so aussehen:

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

../raytracer/object/sphere.py

Dabei ist `normalize` genau die Funktion, welche uns schon in der Python-Einführung in Kapitel 2 begegnet ist.

Die Funktion, welche die Intensität aufgrund der diffusen Beleuchtung berechnet, heisst `_diffuse_shader(self, l, n)` und ist im File `myobject/object.py` implementiert.

Aufgabe 5. Ergänzen Sie die Funktion `_diffuse_shader(self, l, n)`. Dabei ist `self` eine Instanz des getroffenen Objektes und `l`, `n` sind die normierten Vektoren $\vec{l}, \vec{n}$ aus Gleichung (1). Der Rückgabewert ist der RGB-Vektor gemäss diffuser Beleuchtung. Lassen Sie anschliessend das Skript `example2.py` laufen. Hinweis: Das Skalarprodukt von `n` und `l` kann berechnet werden mit `inner(n,l)`.

Lösung. Die fehlende Intensität I berechnet sich nach Gleichung (1).

```
2     @staticmethod
3     def _diffuse_shader(l, n):
4         return max(0.0, inner(n, l))
5
```

../raytracer/object/object.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.

Eine realistische Beleuchtung wäre wohl eine Mischung zwischen den Abbildungen 4 und 6. Das wollen wir nun in Angriff nehmen. Dazu müssen wir uns überlegen, wie man Farben im Python-Code darstellen kann. Wir werden Farben im sogenannten RGB-Format darstellen, das heisst als Vektor mit drei Komponenten (r, b, g), die jeweils nur

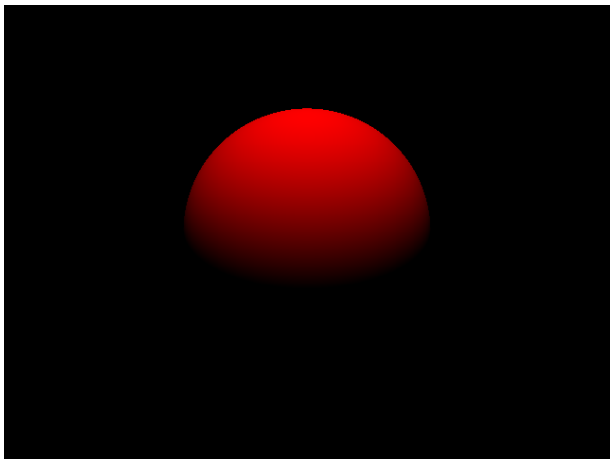

Abbildung 6: `example2.py`: Diffuse Beleuchtung aus Aufgabe 5.

Werte zwischen Null und Eins annehmen, also $r, g, b \in [0, 1]$. Dabei steht r für „rot“, g für „grün“ und b für „blau“. Der Wert der Komponente gibt das Gewicht der jeweiligen Farbe an. Letztere ergibt sich dann aus der entsprechend gewichteten Mischung dieser Farben. Zum Beispiel entspricht die Farbe rot dem RGB-Vektor $(1, 0, 0)$. Weiss hingegen wäre dargestellt als $(1, 1, 1)$. Um eine gewünschte Farbe in das RGB-Format zu übersetzen verwendet man am besten eine der vielen online-Tools.

Um ein realistisches Bild unserer Kugel zu erhalten, mischen wir einfach die RGB-Vektoren aus der *ambienten* und der *diffusen* Beleuchtung. Ambiente Beleuchtung heisst einfach, dass jedem Punkt auf dem Objekt die Farbe des Objektes zugeordnet wird. Das entspricht genau Abbildung 4. Das mischen der Farben geschieht in der Funktion `shader` im File `myobject/object.py`. Sie nimmt unter anderem den betrachteten Punkt $\vec{P}$ sowie eine Liste der Lichtquellen und Objekte entgegen. Ihr Rückgabewert ist die Farbe am Punkt $\vec{P}$ und sie berechnet sich als gewichtete Summe der RGB-Vektoren aus ambienter und diffuser Beleuchtung. Die Member-Variablen `self.ambient` und `self.diffuse` sind gerade die jeweiligen Gewichte. Wenn wir in `example2.py` die Zeile 14 ändern zu

```
14 sphere = Sphere(array([5.0, 0.0, 1.0]), 1.0, ambient=0.2, diffuse=0.8)
```

erhalten wir endlich ein realistisches Bild wie in Abbildung 7 gezeigt.

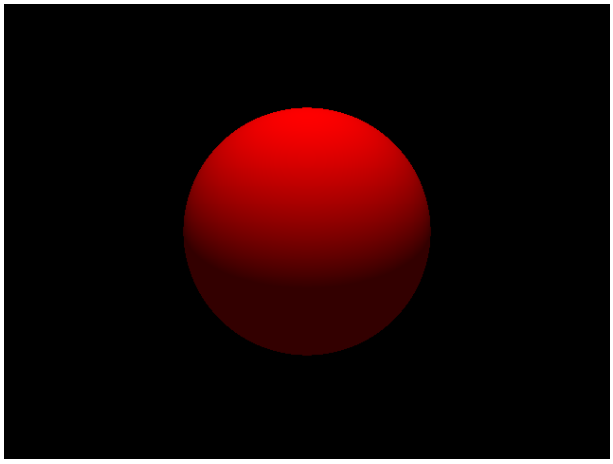

Abbildung 7: `example2.py` (geändert): Kugel mit `ambient=0.2` und `diffuse=0.8`.

```
    def shader(self, p, c, lightsource_list, object_list,
recursion_depth=5):
        c = normalize(c)       # Richtung aus der der Strahl gekommen
ist
        n = self.get_normal(p) # Normalenvektor am Punkt p

        # ambiente Beleuchtung
        color = self.ambient * self.color

        # diffuse Beleuchtung
        if self.diffuse > 0.0 and len(lightsource_list) > 0:
            # Für jede Lichtquelle berechne diffuse und spiegelnde
Reflexion
            for lightsource in lightsource_list:
                # Lichtstrahl: v + t * w
                v = lightsource
                w = p - lightsource
                obj, t = get_nearest_obstacle(v, w, object_list)
                # Wenn die Lichtquelle nicht durch ein anderes Objekt
verdeckt wird
                if t + 1.0e-10 > 1.0 and obj is self:
                    l = -normalize(w) # Richtung Lichtquelle
                    color += self.diffuse * Object._diffuse_shader(l, n)
 * self.color

        # perfekte Reflexion
        if self.reflection > 0.0 and recursion_depth > 0:
            color += self.reflection * Object._reflection_shader(c, n, p
, lightsource_list, object_list, recursion_depth - 1) * self.color

        return color

```

../raytracer/object/object.py

3.3 Die Ebene

Nach der Kugel werden wir nun ein weiteres Objekt, nämlich die Ebene implementieren. Eine Ebene im Raum ist eindeutig festgelegt durch den Normalenvektor $\vec{n}$ auf die Ebene und eine Zahl $d \in \mathbb{R}$. Dabei darf $\vec{n}$ nicht der Nullvektor sein. Die Ebene besteht dann genau aus den Punkten $\vec{x}$, welche der Gleichung

$$\vec{n} \cdot \vec{x} + d = 0 \tag{2}$$

genügen. Falls $\vec{n}$ die Länge 1 hat, so ist $|d|$ gerade der Abstand der Ebene vom Ursprung. Ähnlich wie bei der Kugel definiert der Normalenvektor $\vec{n}$ ein „Aussen“ und ein „Innen“. Der Halbraum, welcher in Richtung des Normalenvektors liegt, ist das „Aussen“. Wie schon zuvor bei der Kugel, soll ein Strahl Objekte nur von „aussen“ schneiden können. Schnittpunkte von „innen“ lassen wir nicht zu.

Beispiel. Wenn wir die xy–Ebene mit dem Normalenvektor $\vec{n} = (0, 0, 1)$ beschreiben, so ist der obere Halbraum (bestehend aus den Punkten mit positiver z–Koordinate) das „Aussen“ und der untere Halbraum entsprechend das „Innen“. Ein Strahl mit Ursprung im unteren Halbraum könnte dann niemals einen zulässigen Schnittpunkt mit der xy–Ebene haben. Würden wir die xy–Ebene mit dem Normalenvektor $\vec{n} = (0, 0, -1)$ beschreiben, so wäre es genau umgekehrt.

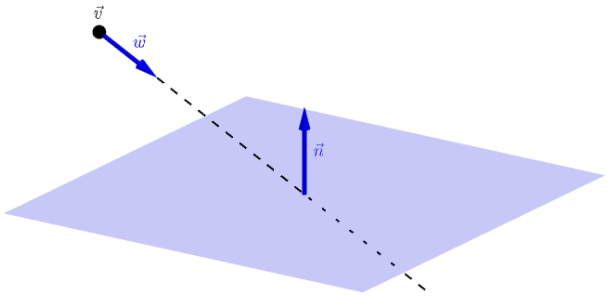

Abbildung 8: Die Ebene wird von einem Strahl von „aussen“ her geschnitten.

Aufgabe 6. Wir betrachten die Ebene welche durch Gleichung (2) beschrieben wird und den Strahl $\vec{v} + t\vec{w}$ mit $t > 0$, wobei $\vec{w}$ nicht der Nullvektor ist. Formulieren Sie die Bedingungen, unter denen genau ein Schnittpunkt zwischen Ebene und Strahl existiert. Berechnen Sie in diesem Fall die Zahl $s > 0$, so dass $\vec{v} + s\vec{w}$ dem Schnittpunkt entspricht. Berücksichtigen Sie dabei, dass er Strahl die Ebene nur von „aussen“ her scheiden darf.

Lösung. Der Strahl besteht genau aus den Punkten der Form $\vec{v} + t\vec{w}$ für ein $t > 0$. So einen beliebigen Punkt auf der Strahl setzten wir nun in die Ebenengleichung (2) ein um den Schnittpunkt zu berechnen. Wir suchen also $s > 0$, so dass

$$\vec{n} \cdot (\vec{v} + s\vec{w}) + d = 0.$$

Auflösen nach s liefert

$$s = -\frac{d + \vec{n} \cdot \vec{v}}{\vec{n} \cdot \vec{w}}.$$

Dabei können wir nur einen Schnittpunkt haben wenn $s > 0$. In diesem Fall trifft der Strahl von „aussen“ auf die Ebene genau dann wenn $\vec{n} \cdot \vec{w} < 0$, also wenn $\vec{n}$ und $\vec{w}$ einen Winkel von mehr als 90° einschliessen. Man betrachte dazu Abbildung 8.

Nun wollen wir die Ebene als neues Objekt implementieren. Das Skript `example3.py` beschreibt eine Szene bestehend aus einer roten Kugel auf einer grünen Ebene.

```python
from numpy import array, pi

from core.camera import Camera
from core.renderer import Renderer
from myobject.sphere import Sphere
from myobject.plane import Plane

camera = Camera(array([-1.0, 0.0, 1.0]), array([0.0, 0.0, 1.0]), 0.25 *
    pi, 640, 480)
renderer = Renderer(camera)

lightsource = array([0.0, 0.0, 10.0])

sphere = Sphere(array([5.0, 0.0, 1.0]), 1.0, ambient=0.2, diffuse=0.8)

# Die Farbe Grün als RGB-Vektor
green = array([0.0, 1.0, 0.0])

# Wir zeichnen hier die xy-Ebene, also die Ebene mit
# Normalenvektor n = [0.0, 0.0, 1.0] und d = 0.0 in grün.
plane = Plane(array([0.0, 0.0, 1.0]), 0.0, color=green, ambient=0.2,
    diffuse=0.8)

renderer([lightsource], [sphere, plane], photo_exposure=0.0)
renderer.save_image("example3.png")
```

../raytracer/example3.py

Aufgabe 7. Ergänzen Sie die Klasse `Plane` im File `myobject/plane.py`, welche die Ebene implementiert. Sie können Ihre Implementierung testen indem Sie das Skript `example3.py` ausführen.

Lösung. Der Code könnte zum Beispiel so aussehen.

```python
from numpy import inf, inner

from object.object import Object, normalize

class Plane(Object):
    def __init__(self, n, d, **kwargs):
        super().__init__(**kwargs)
        self.n = n # Normalenvektor n aus der Ebenengleichung
        self.d = d # Parameter d aus der Ebenengleichung

    def intersect(self, v, w):
        # Der Strahl ist beschrieben durch v+t*w mit t>0

        # Falls der Strahl von "aussen" kommt, berechne s
        nw = inner(self.n, w)
        if nw < 0.0:
            s = -1.0 * (self.d + inner(self.n, v)) / nw
            if s > 0.0:
                return s

        # Falls kein zulässiger Schnittpunkt existiert,
        # wird "unendlich" zurückgegeben.
```

```
24            return inf
25
26        def get_normal(self, p):
27            return normalize(self.n)
```

../raytracer/object/plane.py

Wenn wir mit diesem Code das Skript `example3.py` laufen lassen, erhalten wir eine rote Kugel auf einer grünen Ebene wie in Abbildung 9 gezeigt.

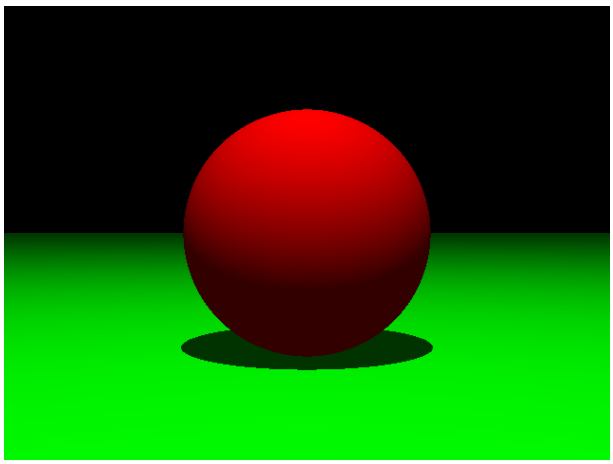

Abbildung 9: `example3.py`: Kugel auf einer Ebene.

3.4 Der Quader

Nun führen wir den Quader, im englischen auch „Box“ genannt, ein. Dabei beschränken wir uns auf den wichtigen Spezialfall des achsenparallelen Quaders. Damit ist gemeint, dass die 6 Flächen parallel zu den Koordinatenebenen sind (oder die 12 Kanten parallel zu den Koordinatenachsen). Das hat zwei Vorteile. Einerseits lassen sich die Schnittpunkte mit Strahlen einfacher und sehr effizient berechnen. Andererseits ist so ein achsenparalleler Quader durch Angabe der Position von nur zwei Ecken $\vec{p}$ und $\vec{q}$ bestimmt, wobei $\vec{p}$ komponentenweise kleiner ist als $\vec{q}$. Durch diese Bedingung sind die beiden Ecken eindeutig bestimmt. Man betrachte dazu Abbildung 10. Zur Schnittpunktberechnung stellen wir

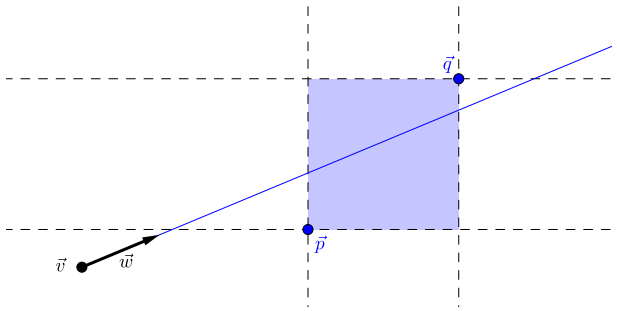

Abbildung 10: Schnittpunkt zwischen dem Strahl $\vec{v}+t\vec{w}, t > 0$ und einem achsenparallelen Quader definiert durch $\vec{p}$ und $\vec{q}$. Zur Vereinfachung wurde der Quader in der Zeichnung durch ein achsenparalleles Rechteck ersetzt, in dessen Ebene der Strahl verläuft.

uns vor, dass jede der 6 Seiten zu einer Ebene im Raum erweitern wird. Die Reihenfolge in welcher der Strahl diese Ebenen durchstösst, möglicherweise ausserhalb des Quaders, sagt uns ob der Strahl den Quader trifft. In Abbildung 10 sind diese Ebenen als gestrichelte Linien dargestellt, zur Vereinfachung in 2D. Diese Schnittpunktberechnung ist bereits im File `myobject/box.py` implementiert, aber die Berechnung des Normalenvektors fehlt noch. Doch bevor wir diese implementieren, betrachten Sie das Skript `example4.py`, welches eine Szene mit drei achsenparallelen Quadern beschreibt.

```
1  from numpy import array, pi
2
3  from core.camera import Camera
4  from core.renderer import Renderer
5  from myobject.box import Box
6  from myobject.plane import Plane
7
8  camera = Camera(array([-1.0, -3.5, 3.0]), array([5.0, 0.0, 1.0]), 0.25 *
       pi, 640, 480)
9  renderer = Renderer(camera)
10
11 lightsource = array([1.0, -8.0, 10.0])
12
13 white = array([1.0, 1.0, 1.0])
14 red = array([1.0, 0.0, 0.0])
15 green = array([0.0, 1.0, 0.0])
16 blue = array([0.0, 0.0, 1.0])
17
18 pr = array([6.0, -2.0, 0.0])
19 pg = array([5.0, -0.5, 0.0])
20 pb = array([4.0, 1.0, 0.0])
21
```

```
box_red = Box(pr, pr + array([1.0, 1.0, 2.0]), color=red, ambient=0.2,
    diffuse=0.8)
box_green = Box(pg, pg + array([2.0, 1.0, 1.0]), color=green, ambient
    =0.2, diffuse=0.8)
box_blue = Box(pb, pb + array([3.0, 1.0, 0.5]), color=blue, ambient=0.2,
     diffuse=0.8)

plane = Plane(array([0.0, 0.0, 1.0]), 0.0, color=white, ambient=0.2,
    diffuse=0.8)

renderer([lightsource], [box_red, box_green, box_blue, plane],
    photo_exposure=0.0)
renderer.save_image("example4.png")
```

../raytracer/example4.py

Aufgabe 8. Ergänzen Sie im File `myobject/box.py` die Funktion `get_normal(self, p)`, welche den Normalenvektor am Punkt `p` auf dem Quader zurückgibt. Die Member-Variablen `self.lower_bound` bzw. `self.upper_bound` enthalten die Punkte $\vec{p}$ bzw. $\vec{q}$. Testen Sie Ihre Implementation mit dem Skript `example4.py`.

Lösung. Es geht darum herauszufinden, auf welcher Seite des Quaders der Schnittpunkt $\vec{P}$ liegt. Eine der vielen Möglichkeit das herauszufinden ist eine Transformation auf den Würfel $\left[-\frac{1}{2}, \frac{1}{2}\right]^3$. Man betrachte dazu Abbildung 11. Vom transformierten Schnittpunkt $\vec{P}'$

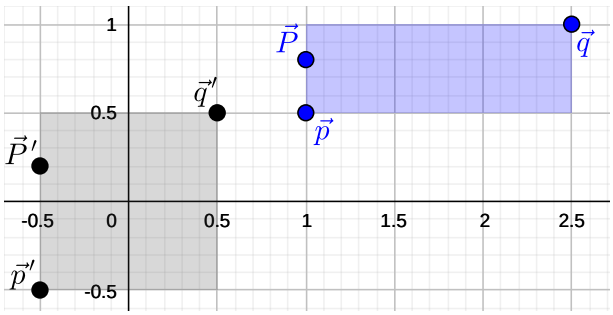

Abbildung 11: Berechnung des Normalenvektors in $\vec{P}$, zur Vereinfachung in 2D.

lässt sich die gewünschte Information an den Komponenten ablesen: Die betragsmässig grösste Komponente wird $\pm\frac{1}{2}$ sein und markiert diejenige Fläche des transformierten Quaders, auf der $\vec{P}'$ liegt. Die Fläche legt ihren zugehörigen Normalenvektor eindeutig fest und die Normalenvektoren in $\vec{P}$ und $\vec{P}'$ sind dieselben.

```
    def get_normal(self, p):
        # Zentrum des Quaders
        c = 0.5 * (self.lower_bound + self.upper_bound)
        edges = self.upper_bound - self.lower_bound

        # Transformiere den Schnittpunkt in den Würfel [-0.5, 0.5]^3
        p_rel = (p - c) / edges

        # Welcher Seite liegt der Punkt am nächsten?
        i = argmax(abs(p_rel))
        n = array([0.0, 0.0, 0.0])
        n[i] = sign(p_rel[i])
        return n

```

../raytracer/object/box.py

Abbildung 12 zeigt das Resultat.

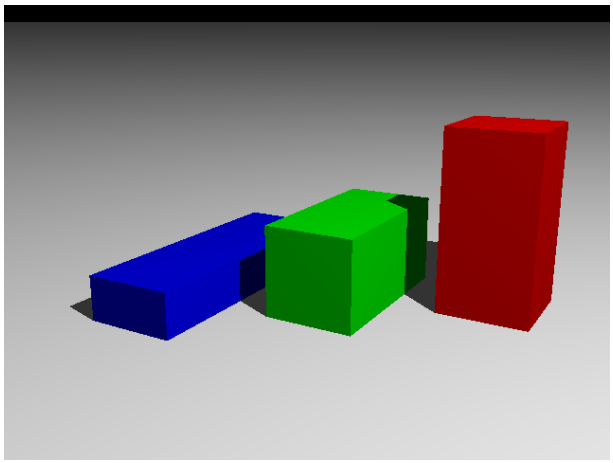

Abbildung 12: `example4.py`: Drei Quader in verschiedenen Farben.

3.5 Perfekte Reflexion

In diesem Kapitel werden wir die Funktion `shader` derart erweitern, dass wir perfekt spiegelnde Oberflächen beschreiben können. Wir können dann also eine „Spiegelkugel“ oder eine „Spiegelebene“ usw. darstellen. Dazu nutzen wir eine Stärke des Raytracing-Algorithmus: Wir können ihn rekursiv einsetzen. Um die Farbe an einem Punkt $\vec{P}$ auf einem perfekten Spiegel zu berechnen, kann einfach von $\vec{P}$ aus ein neuer, gespiegelter Strahl ausgesendet werden.

Beispiel. Wir betrachten eine Szene wie in Abbildung 13 gezeigt: Sei eine Kamera im Punkt $\vec{v}$ positioniert. Ein davon ausgesendeter Strahl $\vec{v} + t\vec{w}, t > 0$ trifft nun auf einen Spiegel. Den Schnittpunkt von Strahl und Spiegel nennen wir $\vec{P}$. Die Farbe in diesem Punkt ermitteln wir wie folgt: Wir spiegeln den Vektor

$$\vec{c} = \frac{\vec{v} - \vec{P}}{\|\vec{v} - \vec{P}\|}$$

am Normalenvektor $\vec{n}$ in $\vec{P}$. Der resultierende Vektor, wir nennen ihn $\vec{c}'$, beschreibt die Ausfallrichtung des reflektierten Strahls $\vec{P} + t\vec{c}', t > 0$. Nun wird die Farbberechnung wiederholt, diesmal aber mit dem reflektierten Strahl. Hier ist zu beachten, dass der neue Strahl ebenfalls wieder auf einen Spiegel treffen könnte. In diesem Fall muss die Prozedur wiederholt werden.

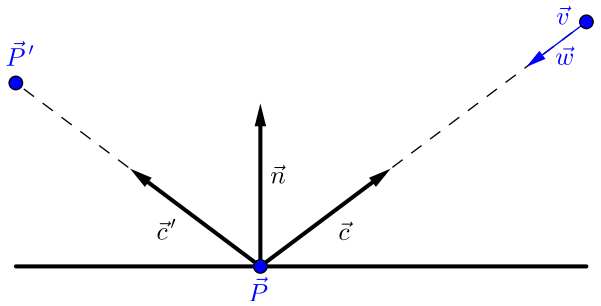

Abbildung 13: Der einfallende Strahl wir reflektiert. Um die Farbe im Punkt $\vec{P}$ zu bestimmen wird der Raytracing Algorithmus erneut auf den ausfallenden Strahl angewendet.

Aufgabe 9. Der Vektor $\vec{c}'$ einsteht durch Spiegelung von $\vec{c}$ and $\vec{n}$, betrachten Sie dazu Abbildung 13. Finden Sie eine Formel für $\vec{c}'$ in Termen von $\vec{c}$ and $\vec{n}$.

Lösung. Die Lösung lautet

$$\vec{c}' = 2\,(\vec{n} \cdot \vec{c})\,\vec{n} - \vec{c},$$

wobei die Formel auch dann richtig ist, wenn $\vec{c}$ nicht Länge 1 hat.

Aufgabe 10. Vervollständigen Sie die Funktion `_reflection_shader(c, n, p, lightsource_list, object_list, recursion_depth)` im File `myobject/object.py`. Der Rückgabewert soll der RGB-Vektor der Farbe am Punkt $\vec{P}'$ sein. Die Argumente `c,n,p` bezeichnen die Vektoren $\vec{c}, \vec{n}, \vec{P}$ aus Abbildung 13. Nutzen Sie die Funktionen `get_nearest_obstacle` in `core/tracer.py` und `shader` in `myobject/object.py`. Schauen Sie sich deren Definition an um zu verstehen wie sie verwendet werden müssen. Die Argumente `lightsource_list, object_list, recursion_depth` können diesen Funktionen unverändert übergeben werden ohne dass Sie deren Bedeutung genau verstehen. Testen Sie Ihre Lösung mit dem Skript `example5.py`.

Lösung. Die Lösung könnte zum Beispiel so aussehen:

```
    @staticmethod
    def _reflection_shader(c, n, p, lightsource_list, object_list,
    recursion_depth):
            # Gespiegelter Strahl p + t * c_prime, t > 0
            c_prime = 2.0 * inner(n, c) * n - c
            # Ermittle das getroffene Objekt obj und den Schnittpunkt-
    Parameter t
            obj, t = get_nearest_obstacle(p, c_prime, object_list)
            if t == inf:
                # Wenn kein Objekt getroffen wird: schwarz
                return array([0.0, 0.0, 0.0])
            else:
                # Berechne die Farbe am Punkt v + t * w
                return obj.shader(p + t * c_prime, -c_prime,
    lightsource_list, object_list, recursion_depth)

```

../raytracer/object/object.py

Der Vektor $\vec{c}'$, im Code `c_prime` genannt, wurde in Aufgabe 9 berechnet. Die Rekursion passiert mit dem Aufruf der Funktion `shader`, denn diese ruft selber die Funktion `_reflection_shader` auf. Wir erhalten folgendes Bild:

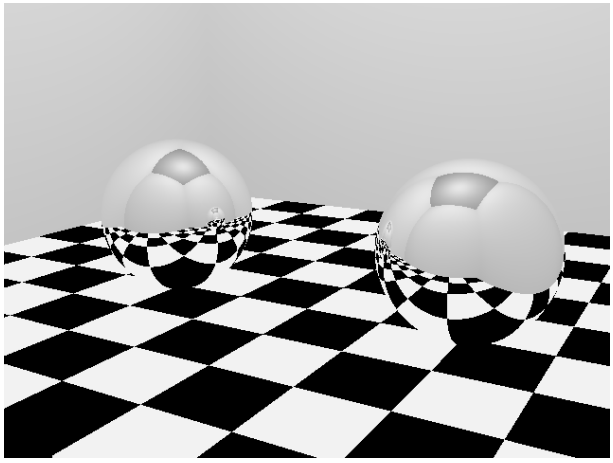

Abbildung 14: `example5.py`: Zwei perfekt reflektierende „Spiegelkugeln“.

```
from numpy import array, pi

from core.camera import Camera
from core.renderer import Renderer
from myobject.box import Box
from myobject.plane import Plane
from myobject.sphere import Sphere

camera = Camera(array([-7.5, -4.3, 3.0]), array([0.0, 0.0, 1.0]), 0.25 *
    pi, 640, 480)
renderer = Renderer(camera)

lightsource = array([0.0, 0.0, 10.0])

black = array([0.0, 0.0, 0.0])
white = array([1.0, 1.0, 1.0])
```

```
16 gray = array([0.8, 0.8, 0.8])
17
18 dx = 1.0
19 dy = 1.0
20
21 box_list = []
22 n = 10
23 for i in range(n):
24     x = dx * (n / 2 - i)
25     for j in range(n):
26         y = dy * (n / 2 - j)
27         if (i + j) % 2 == 0:
28             color = black
29         else:
30             color = white
31         p = array([x, y, -1.0])
32         q = array([x + dx, y + dy, 0.0])
33         box = Box(p, q, color=color, ambient=1.0)
34         box_list += [box]
35
36 kwargs = {'color': gray, 'ambient': 0.2, 'diffuse': 0.8}
37 plane1 = Plane(array([-1.0, 0.0, 0.0]), 6.0, **kwargs)
38 plane2 = Plane(array([0.0, -1.0, 0.0]), 6.0, **kwargs)
39 plane3 = Plane(array([1.0, 0.0, 0.0]), 5.0, **kwargs)
40 plane4 = Plane(array([0.0, 1.0, 0.0]), 5.0, **kwargs)
41 plane5 = Plane(array([0.0, 0.0, -1.0]), 12.0, **kwargs)
42 plane_list = [plane1, plane2, plane3, plane4, plane5]
43
44 kwargs = {'color': white, 'ambient': 0.0, 'reflection': 1.0}
45 sphere1 = Sphere(array([0.0, 2.0, 1.0]), 1, **kwargs)
46 sphere2 = Sphere(array([0.0, -2.0, 1.0]), 1, **kwargs)
47 sphere_list = [sphere1, sphere2]
48
49 renderer([lightsource], box_list + plane_list + sphere_list, 3.0)
50 renderer.save_image("example5.png")
```

../raytracer/example5.py

4 Erweiterungen

Sie können nun selber Szenen bauen indem Sie eigene Python-Skripts schreiben. Das aktuelle Raytracing-Programm enthält aber nur ein minimales Grundgerüst. Sie können dieses Grundgerüst erweitern. Dazu einige Vorschläge:

- Es können neue Objekttypen hinzugefügt werden wie Kegel und Zylinder.
- Objekte können verschoben, rotiert und verzerrt werden. Dazu betrachtet man invertierbare *affine* Abbildungen im Raum. Damit kann zum Beispiel ein Quader zu einem Spat transformiert werden. Anstatt Objekte selbst zu transformieren, kann man auch nur die Strahlen und die Normalenvektoren transformieren. So lassen sich diese Abbildungen unabhängig von konkreten Objekten beschreiben. Genaueres finden Sie in [4], Kapitel 20.
- Transparenz und Lichtbrechung: Mit den Fresnelschen Formeln aus der Physik lassen sich durchsichtige Objekte beschreiben. Man kann damit zum Beispiel „Glaskugeln“ sehr realistisch abbilden. Im Allgemeinen wird dabei ein in das Objekt eindringender Strahl aufgeteilt: Ein Teil wird reflektiert und der Rest wird gebrochen und dringt in das Objekt ein. Ähnlich verhält es sich, wenn der Strahl das Objekt wieder verlässt. Wie bei der Reflexion muss der Raytracing Algorithmus dazu rekursiv aufgerufen werden. Weiteres dazu finden Sie in [4], Kapitel 27.
- Effizientere Schnittpunktberechnung: Bei Szenen mit vielen Objekten kann die Bildgenerierung sehr lange dauern. Der Grund dafür ist, dass für jedes Pixel ein Strahl ausgesendet werden muss, welcher mit jedem Objekt der Szene auf Schnittpunkte getestet werden muss. Um die Anzahl Schnittpunkt-Tests zu reduzieren, kann der gesamte Raum in Quader aufgeteilt werden. Objekte in einem Quader welcher nicht vom Strahl getroffen wird, müssen dann keinem Schnittpunkt-Test unterzogen werden. (Stichworte: regular grids [4] oder octree [3])

Natürlich gibt es auch professionelle Raytracing Software. Eine der bekanntesten open source Raytracer heisst *POV-Ray* [2] und kann unter folgendem Link heruntergeladen werden: `http://www.povray.org`.

Anhang: Codes

```
raytracer/
├── main.py
├── example1.py
├── example2.py
├── example3.py
├── example4.py
├── example5.py
├── core/
│   ├── __init__.py (empty)
│   ├── camera.py
│   ├── renderer.py
│   └── tracer.py
├── object/
│   ├── __init__.py (empty)
│   ├── object.py
│   ├── sphere.py
│   ├── plane.py
│   └── box.py
└── myobject/
    ├── __init__.py (empty)
    ├── object.py
    ├── sphere.py
    ├── plane.py
    └── box.py
```

```python
from numpy import array, cross, tan
from numpy.linalg import norm

def normalize(vector):
    return vector / norm(vector)

class Camera:
    def __init__(self, position, look_at, angle, pixels_x, pixels_y):
        self.position = position
        self.look_at = look_at
        self.angle = angle
        self.pixels_x = pixels_x
        self.pixels_y = pixels_y

    def get_ray_indices(self):
        direction = normalize(self.look_at - self.position)
        width = normalize(cross(direction, array([0.0, 0.0, 1.0])))
        height = cross(direction, width)
        pixelsize = 2.0 * tan(0.5 * self.angle) / self.pixels_x;

        for i in range(self.pixels_y):
            dy = (i - 0.5 * (self.pixels_y - 1)) * pixelsize * height
            for j in range(self.pixels_x):
                dx = (j - 0.5 * (self.pixels_x - 1)) * pixelsize * width
                v = self.position
```

```
                w = direction + dx + dy
                yield (v, w, i, j)
```

../raytracer/core/camera.py

```
from numpy import exp, ones_like, uint8, zeros

from PIL.Image import fromarray

from core.tracer import get_nearest_obstacle

class Renderer:
    def __init__(self, camera):
        self.camera = camera
        self.rgb_data = zeros((camera.pixels_y, camera.pixels_x, 3),
    dtype=float)

    def __call__(self, lightsource_list, object_list, photo_exposure
    =1.0):
        for v, w, i, j in self.camera.get_ray_indices():
            obj, t = get_nearest_obstacle(v, w, object_list)
            if obj is not None:
                self.rgb_data[i, j] = obj.shader(v + t * w, -w,
    lightsource_list, object_list)
        if photo_exposure > 0.0:
            self.rgb_data[:, :] = ones_like(self.rgb_data[:, :]) - exp(-
    self.rgb_data[:, :] * photo_exposure)

    def save_image(self, filename):
        fromarray((self.rgb_data * 255).astype(uint8), 'RGB').save(
    filename)
```

../raytracer/core/renderer.py

```
from numpy import inf

"""
Bestimmt das Objekt, welches vom Strahl
v + t * w, t > 0 getroffen wird und gibt
zudem den Schnittpunkt-Parameter t zurück
"""
def get_nearest_obstacle(v, w, object_list):
    t = inf
    nearest_obstacle = None
    for obstacle in object_list:
        s = obstacle.intersect(v, w)
        if (s < t):
            t = s
            nearest_obstacle = obstacle
    return nearest_obstacle, t
```

../raytracer/core/tracer.py

```
from numpy import array, inf, inner, pi
from numpy.linalg import norm

from core.tracer import get_nearest_obstacle

def normalize(vector):
```

```
7       return vector / norm(vector)
8
9   class Object:
10      def __init__(self, color=array([1.0, 0.0, 0.0]), ambient=1.0,
        diffuse=0.0, reflection=0.0):
11          self.color = color
12          self.ambient = ambient
13          self.diffuse = diffuse
14          self.reflection = reflection
15
16      def intersect(self, v, w):
17          pass
18
19      def get_normal(self, p):
20          pass
21
22      #---_diffuse_shader-begin---
23      @staticmethod
24      def _diffuse_shader(l, n):
25          return max(0.0, inner(n, l))
26      #---_diffuse_shader-end---
27
28      #---_reflection_shader-begin---
29      @staticmethod
30      def _reflection_shader(c, n, p, lightsource_list, object_list,
        recursion_depth):
31              # Gespiegelter Strahl p + t * c_prime, t > 0
32              c_prime = 2.0 * inner(n, c) * n - c
33              # Ermittle das getroffene Objekt obj und den Schnittpunkt-
        Parameter t
34              obj, t = get_nearest_obstacle(p, c_prime, object_list)
35              if t == inf:
36                  # Wenn kein Objekt getroffen wird: schwarz
37                  return array([0.0, 0.0, 0.0])
38              else:
39                  # Berechne die Farbe am Punkt v + t * w
40                  return obj.shader(p + t * c_prime, -c_prime,
        lightsource_list, object_list, recursion_depth)
41      #---_reflection_shader-end---
42
43      #---shader-begin---
44      def shader(self, p, c, lightsource_list, object_list,
        recursion_depth=5):
45          c = normalize(c)        # Richtung aus der der Strahl gekommen
        ist
46          n = self.get_normal(p) # Normalenvektor am Punkt p
47
48          # ambiente Beleuchtung
49          color = self.ambient * self.color
50
51          # diffuse Beleuchtung
52          if self.diffuse > 0.0 and len(lightsource_list) > 0:
53              # Für jede Lichtquelle berechne diffuse und spiegelnde
        Reflexion
54              for lightsource in lightsource_list:
55                  # Lichtstrahl: v + t * w
56                  v = lightsource
57                  w = p - lightsource
58                  obj, t = get_nearest_obstacle(v, w, object_list)
```

```
59                  # Wenn die Lichtquelle nicht durch ein anderes Objekt
    verdeckt wird
60                  if t + 1.0e-10 > 1.0 and obj is self:
61                      l = -normalize(w) # Richtung Lichtquelle
62                      color += self.diffuse * Object._diffuse_shader(l, n)
     * self.color
63
64          # perfekte Reflexion
65          if self.reflection > 0.0 and recursion_depth > 0:
66              color += self.reflection * Object._reflection_shader(c, n, p
    , lightsource_list, object_list, recursion_depth - 1) * self.color
67
68          return color
69      #---shader-end---
```

../raytracer/object/object.py

```
1  from numpy import inf, inner, sqrt
2
3  from object.object import Object, normalize
4
5  class Sphere(Object):
6      def __init__(self, m, r, **kwargs):
7          super().__init__(**kwargs)
8          self.m = m
9          self.r = r
10
11     #---intersect-begin---
12     def intersect(self, v, w):
13         # Der Strahl ist beschrieben durch v+t*w mit t>0
14
15         # Berechung der Koeffizienten der quadratischen Gleichung
16         a = inner(w, w)
17         mv = v - self.m
18         b = 2.0 * inner(w, mv)
19         c = inner(mv, mv) - self.r**2
20
21         # Diskriminante
22         d = b**2 - 4.0 * a * c
23
24         # Fallunterscheidung: Schneidet der Strahl die Kugel?
25         if d >= 0.0:
26             t0 = (-b - sqrt(d)) / (2.0 * a)
27             if t0 > 0.0 and inner(v + t0 * w - self.m, w) < 0.0:
28                 return t0
29             t1 = (-b + sqrt(d)) / (2.0 * a)
30             if t1 > 0.0 and inner(v + t1 * w - self.m, w) < 0.0:
31                 return t1
32
33         # Falls kein Schnittpunkt existert, retourniere "unendlich"
34         return inf
35     #---intersect-end---
36
37     #---get_normal-begin---
38     def get_normal(self, p):
39         return normalize(p - self.m)
40     #---get_normal-end---
```

../raytracer/object/sphere.py

```python
#---plane-begin---
from numpy import inf, inner

from object.object import Object, normalize

class Plane(Object):
    def __init__(self, n, d, **kwargs):
        super().__init__(**kwargs)
        self.n = n # Normalenvektor n aus der Ebenengleichung
        self.d = d # Parameter d aus der Ebenengleichung

    def intersect(self, v, w):
        # Der Strahl ist beschrieben durch v+t*w mit t>0

        # Falls der Strahl von "aussen" kommt, berechne s
        nw = inner(self.n, w)
        if nw < 0.0:
            s = -1.0 * (self.d + inner(self.n, v)) / nw
            if s > 0.0:
                return s

        # Falls kein zulässiger Schnittpunkt existiert,
        # wird "unendlich" zurückgegeben.
        return inf

    def get_normal(self, p):
        return normalize(self.n)
#---plane-end---
```

../raytracer/object/plane.py

```python
from numpy import argmax, array, inf, sign

from object.object import Object

class Box(Object):
    def __init__(self, lower_bound, upper_bound, **kwargs):
        if not all(l < u for l, u in zip(lower_bound, upper_bound)):
            raise ValueError("lower_bound must be component-wise smaller
    than upper_bound")
        super().__init__(**kwargs)
        self.lower_bound = lower_bound
        self.upper_bound = upper_bound

    #---intersect-begin---
    def intersect(self, v, w):
        # Der Strahl ist beschrieben durch v+t*w mit t>0

        p = self.lower_bound
        q = self.upper_bound

        tp = array([-inf, -inf, -inf])
        tq = array([inf, inf, inf])

        for i in range(3):
            if w[i] == 0.0:
                if v[i] <= p[i] or q[i] <= v[i]:
                    return inf # Kein Schnittpunkt
            else:
```

```
                # Keine division durch Null
                tp[i] = (p[i] - v[i]) / w[i]
                tq[i] = (q[i] - v[i]) / w[i]

        # Schnittpunkt existiert wenn drei orthogonale Ebenen
        # hintereinander passiert werden.
        tmin = [min(sp, sq) for sp, sq in zip(tp, tq)]
        tmax = [max(sp, sq) for sp, sq in zip(tp, tq)]
        if max(tmin) <= min(tmax) and 0.0 < max(tmin):
            return max(tmin)

        return inf # Kein Schnittpunkt
    #---intersect-end---

    #---get_normal-begin---
    def get_normal(self, p):
        # Zentrum des Quaders
        c = 0.5 * (self.lower_bound + self.upper_bound)
        edges = self.upper_bound - self.lower_bound

        # Transformiere den Schnittpunkt in den Würfel [-0.5, 0.5]^3
        p_rel = (p - c) / edges

        # Welcher Seite liegt der Punkt am nächsten?
        i = argmax(abs(p_rel))
        n = array([0.0, 0.0, 0.0])
        n[i] = sign(p_rel[i])
        return n
    #---get_normal-end---
```

../raytracer/object/box.py

```
from numpy import array, inf, inner, pi
from numpy.linalg import norm

from core.tracer import get_nearest_obstacle

def normalize(vector):
    return vector / norm(vector)

class Object:
    max_recursion_depth = 5

    def __init__(self, color=array([1.0, 0.0, 0.0]), ambient=1.0,
   diffuse=0.0, reflection=0.0):
        self.color = color
        self.ambient = ambient
        self.diffuse = diffuse
        self.reflection = reflection

    def intersect(self, v, w):
        pass

    def get_normal(self, p):
        pass

    @staticmethod
    def _diffuse_shader(self, l, n):
        """
```

```
        Ersetzen sie das "return" Statement durch den korrekten Code,
        so dass die Intensität gemäss diffuser Beleuchtung
        zurückgegeben wird.
        """
        return 0.0

    @staticmethod
    def _reflection_shader(self, c, n, p, lightsource_list, object_list,
    recursion_depth):
        """
        # Ersetzen sie das "return" Statement durch den korrekten Code,
        # so dass die Farbe am Schnittpunkt mit dem reflektierten Strahl
        # zurückgegeben wird. Trifft dieser kein Objekt, so soll schwarz
        # zurückgegeben werden.
        """
        return array([0.0, 0.0, 0.0])

    def shader(self, p, c, lightsource_list, object_list,
    recursion_depth=5):
        c = normalize(c)       # Richtung aus der der Strahl gekommen
    ist
        n = self.get_normal(p) # Normalenvektor am Punkt p

        # ambiente Beleuchtung
        color = self.ambient * self.color

        # diffuse Beleuchtung
        if self.diffuse > 0.0 and len(lightsource_list) > 0:
            # Für jede Lichtquelle berechne diffuse und spiegelnde
    Reflexion
            for lightsource in lightsource_list:
                # Lichtstrahl: v + t * w
                v = lightsource
                w = p - lightsource
                obj, t = get_nearest_obstacle(v, w, object_list)
                # Wenn die Lichtquelle nicht durch ein anderes Objekt
    verdeckt wird
                if t + 1.0e-10 > 1.0 and obj is self:
                    l = -normalize(w) # Richtung Lichtquelle
                    color += self.diffuse * Object._diffuse_shader(l, n)
     * self.color

        # perfekte Reflexion
        if self.reflection > 0.0 and recursion_depth > 0:
            color += self.reflection * Object._reflection_shader(c, n, p
    , lightsource_list, object_list, recursion_depth - 1) * self.color

        return color
```

../raytracer/myobject/object.py

```
from numpy import array, inf, inner, sqrt

from myobject.object import Object, normalize


class Sphere(Object):
    def __init__(self, m, r, **kwargs):
        super().__init__(**kwargs)
```

```
        self.m = m
        self.r = r

    def intersect(self, v, w):
        # Der Strahl ist beschrieben durch v+t*w mit t>0

        """
        Ersetzen Sie diesen Kommentar durch Ihren Code:
        Gegeben ist ein Strahl mit Ursprung v und Richtung w.
        Geben Sie den Parameter t > 0 zurück, so dass v + t * w
        gerade der an v nächstgelegene Schnittpunkt des Strahls
        mit der Kugel ist. Falls dieser nicht existiert soll
        t = inf (also Unendlich) zurückgegeben werden.
        """

        return inf

    def get_normal(self, p):
        """
        Ersetzen Sie das 'return' statement unten durch
        Ihren Code: Sei p ein Punkt auf der Kugeloberfläche.
        Geben Sie den nach aussen zeigenden, normierten
        Normalenvektor auf der Kugel am Punkt p zurück.
        """
        return array([0.0, 0.0, 0.0])
```

../raytracer/myobject/sphere.py

```
from numpy import inf, inner

from myobject.object import Object, normalize

class Plane(Object):
    def __init__(self, n, d, **kwargs):
        super().__init__(**kwargs)
        self.n = n # Normalenvektor n aus der Ebenengleichung
        self.d = d # Parameter d aus der Ebenengleichung

    def intersect(self, v, w):
        # Der Strahl ist beschrieben durch v+t*w mit t>0

        """
        Ersetzen Sie diesen Kommentar durch Ihren Code:
        Gegeben ist ein Strahl mit Ursprung v und Richtung w.
        Geben Sie den Parameter t > 0 zurück, so dass v + t * w
        gerade der an v nächstgelegene Schnittpunkt des Strahls
        mit der Ebene ist. Falls dieser nicht existiert soll
        t = inf (also Unendlich) zurückgegeben werden.
        """

        return inf

    def get_normal(self, p):
        return normalize(self.n)
```

../raytracer/myobject/plane.py

```
from numpy import argmax, array, inf, sign

```

```python
from myobject.object import Object

class Box(Object):
    def __init__(self, lower_bound, upper_bound, **kwargs):
        if not all(l < u for l, u in zip(lower_bound, upper_bound)):
            raise ValueError("lower_bound must be component-wise smaller than upper_bound")
        super().__init__(**kwargs)
        self.lower_bound = lower_bound
        self.upper_bound = upper_bound

    def intersect(self, v, w):
        # Der Strahl ist beschrieben durch v+t*w mit t>0

        p = self.lower_bound
        q = self.upper_bound

        tp = array([-inf, -inf, -inf])
        tq = array([inf, inf, inf])

        for i in range(3):
            if w[i] == 0.0:
                if v[i] <= p[i] or q[i] <= v[i]:
                    return inf # Kein Schnittpunkt
            else:
                # Keine division durch Null
                tp[i] = (p[i] - v[i]) / w[i]
                tq[i] = (q[i] - v[i]) / w[i]

        # Schnittpunkt existiert wenn drei orthogonale Ebenen
        # hintereinander passiert werden.
        tmin = [min(sp, sq) for sp, sq in zip(tp, tq)]
        tmax = [max(sp, sq) for sp, sq in zip(tp, tq)]
        if max(tmin) <= min(tmax) and 0.0 < max(tmin):
            return max(tmin)

        return inf # Kein Schnittpunkt

    def get_normal(self, p):
        """
        Ersetzen Sie das 'return' statement unten durch
        Ihren Code: Sei p ein Punkt auf dem Quader.
        Geben Sie den nach aussen zeigenden, normierten
        Normalenvektor auf dem Quader am Punkt p zurück.
        Die beiden definierenden Ecken sind durch
        self.lower_bound und self.upper_bound abrufbar.
        """
        return array([0.0, 0.0, 0.0])
```

../raytracer/myobject/box.py

```python
from sys import argv

from numpy import array, pi

from core.camera import Camera
from core.renderer import Renderer

from object.box import Box
```

```
from object.plane import Plane
from object.sphere import Sphere

if len(argv) == 2:
    filename = str(argv[1])
else:
    filename = "image.png"

camera = Camera(array([-10.0, 0.0, 2.0]), array([0.0, 0.0, 1.0]), 0.25 *
    pi, 640, 480)
renderer = Renderer(camera)

red = array([1.0, 0.0, 0.0])
sphere_red = Sphere(array([4.0, -2.0, 0.0]), 1.0, color=red, ambient
   =0.2, diffuse=0.8)

purple = array([148.0 / 255.0, 0.0, 211.0 / 255.0])
sphere_purple = Sphere(array([6.0, 2.0, 1.0]), 2.0,
                color=purple, ambient=0.2, diffuse=0.2, reflection=0.6)

blue = array([0.0, 0.0, 1.0])
box_blue = Box(array([-0.5, -3.0, -1.0]), array([0.5, -2.0, 0.0]),
                color=blue, ambient=0.2, diffuse=0.8)

gray = array([0.7, 0.7, 0.7])
plane_gray = Plane(array([0.0, 0.0, 1.0]), 1.0, color=gray, ambient=0.2,
    diffuse=0.6,
                reflection=0.2)

lightsource_list = [array([0.0, 5.0, 10.0]), array([-5.0, 10.0, 5.0])]
object_list = [sphere_red, sphere_purple, box_blue, plane_gray]
renderer(lightsource_list, object_list)
renderer.save_image(filename)
```

../raytracer/main.py

```
from numpy import array, pi

from core.camera import Camera
from core.renderer import Renderer
from myobject.sphere import Sphere

# Platziere eine Kamera im Punkt [-1.0, 0.0, 1.0] welche in Richtung
# des Punktes [0.0, 0.0, 1.0] schaut.
camera = Camera(array([-1.0, 0.0, 1.0]), array([0.0, 0.0, 1.0]), 0.25 *
   pi, 640, 480)

# Renderer ist eine Klasse zum Generieren der Bilder
renderer = Renderer(camera)

# Eine Kugel um [5.0, 0.0, 1.0] mit Radius 1 (in der Farbe rot)
sphere = Sphere(array([5.0, 0.0, 1.0]), 1.0)

# Man übergiebt dem Renderer eine Liste von Objekten (die Kugel)
renderer([], [sphere], photo_exposure=0.0)

# Hier wird das Bild generiert und abgespeichert
renderer.save_image("example1.png")
```

../raytracer/example1.py

```
from numpy import array, pi

from core.camera import Camera
from core.renderer import Renderer
from myobject.sphere import Sphere

camera = Camera(array([-1.0, 0.0, 1.0]), array([0.0, 0.0, 1.0]), 0.25 *
    pi, 640, 480)
renderer = Renderer(camera)

# Neu: Eine Punktförmige Lichtquelle an der Position [0, 0, 10]
lightsource = array([0.0, 0.0, 10.0])

# Die Argumente "ambient" und "diffuse" werden später erklärt
sphere = Sphere(array([5.0, 0.0, 1.0]), 1.0, ambient=0.0, diffuse=1.0)

# Neu: Dem Renderer wird nun auch die Lichtquelle übergeben
renderer([lightsource], [sphere], photo_exposure=0.0)
renderer.save_image("example2.png")
```

../raytracer/example2.py

```
from numpy import array, pi

from core.camera import Camera
from core.renderer import Renderer
from myobject.sphere import Sphere
from myobject.plane import Plane

camera = Camera(array([-1.0, 0.0, 1.0]), array([0.0, 0.0, 1.0]), 0.25 *
    pi, 640, 480)
renderer = Renderer(camera)

lightsource = array([0.0, 0.0, 10.0])

sphere = Sphere(array([5.0, 0.0, 1.0]), 1.0, ambient=0.2, diffuse=0.8)

# Die Farbe Grün als RGB-Vektor
green = array([0.0, 1.0, 0.0])

# Wir zeichnen hier die xy-Ebene, also die Ebene mit
# Normalenvektor n = [0.0, 0.0, 1.0] und d = 0.0 in grün.
plane = Plane(array([0.0, 0.0, 1.0]), 0.0, color=green, ambient=0.2,
    diffuse=0.8)

renderer([lightsource], [sphere, plane], photo_exposure=0.0)
renderer.save_image("example3.png")
```

../raytracer/example3.py

```
from numpy import array, pi

from core.camera import Camera
from core.renderer import Renderer
from myobject.box import Box
from myobject.plane import Plane

camera = Camera(array([-1.0, -3.5, 3.0]), array([5.0, 0.0, 1.0]), 0.25 *
    pi, 640, 480)
```

```
renderer = Renderer(camera)

lightsource = array([1.0, -8.0, 10.0])

white = array([1.0, 1.0, 1.0])
red = array([1.0, 0.0, 0.0])
green = array([0.0, 1.0, 0.0])
blue = array([0.0, 0.0, 1.0])

pr = array([6.0, -2.0, 0.0])
pg = array([5.0, -0.5, 0.0])
pb = array([4.0, 1.0, 0.0])

box_red = Box(pr, pr + array([1.0, 1.0, 2.0]), color=red, ambient=0.2,
    diffuse=0.8)
box_green = Box(pg, pg + array([2.0, 1.0, 1.0]), color=green, ambient
    =0.2, diffuse=0.8)
box_blue = Box(pb, pb + array([3.0, 1.0, 0.5]), color=blue, ambient=0.2,
     diffuse=0.8)

plane = Plane(array([0.0, 0.0, 1.0]), 0.0, color=white, ambient=0.2,
    diffuse=0.8)

renderer([lightsource], [box_red, box_green, box_blue, plane],
    photo_exposure=0.0)
renderer.save_image("example4.png")
```

../raytracer/example4.py

```
from numpy import array, pi

from core.camera import Camera
from core.renderer import Renderer
from myobject.box import Box
from myobject.plane import Plane
from myobject.sphere import Sphere

camera = Camera(array([-7.5, -4.3, 3.0]), array([0.0, 0.0, 1.0]), 0.25 *
     pi, 640, 480)
renderer = Renderer(camera)

lightsource = array([0.0, 0.0, 10.0])

black = array([0.0, 0.0, 0.0])
white = array([1.0, 1.0, 1.0])
gray = array([0.8, 0.8, 0.8])

dx = 1.0
dy = 1.0

box_list = []
n = 10
for i in range(n):
    x = dx * (n / 2 - i)
    for j in range(n):
        y = dy * (n / 2 - j)
        if (i + j) % 2 == 0:
            color = black
        else:
```

```
            color = white
        p = array([x, y, -1.0])
        q = array([x + dx, y + dy, 0.0])
        box = Box(p, q, color=color, ambient=1.0)
        box_list += [box]

kwargs = {'color': gray, 'ambient': 0.2, 'diffuse': 0.8}
plane1 = Plane(array([-1.0, 0.0, 0.0]), 6.0, **kwargs)
plane2 = Plane(array([0.0, -1.0, 0.0]), 6.0, **kwargs)
plane3 = Plane(array([1.0, 0.0, 0.0]), 5.0, **kwargs)
plane4 = Plane(array([0.0, 1.0, 0.0]), 5.0, **kwargs)
plane5 = Plane(array([0.0, 0.0, -1.0]), 12.0, **kwargs)
plane_list = [plane1, plane2, plane3, plane4, plane5]

kwargs = {'color': white, 'ambient': 0.0, 'reflection': 1.0}
sphere1 = Sphere(array([0.0, 2.0, 1.0]), 1, **kwargs)
sphere2 = Sphere(array([0.0, -2.0, 1.0]), 1, **kwargs)
sphere_list = [sphere1, sphere2]

renderer([lightsource], box_list + plane_list + sphere_list, 3.0)
renderer.save_image("example5.png")
```

../raytracer/example5.py

Literatur

[1] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press Ltd., GBR, 1989.

[2] Persistence of Vision Pty. Ltd. Persistence of vision (TM) raytracer. `http://www.povray.org/`, 2004. Persistence of Vision Pty. Ltd., Williamstown, Victoria, Australia.

[3] Hanan Samet. Implementing ray tracing with octrees and neighbor finding. *Comput. Graph.*, 13:445–460, 1989.

[4] Kevin Suffern. *Ray Tracing from the Ground Up*. A. K. Peters, Ltd., USA, 2007.