Raytracing und Vektorgeometrie

Oliver Rietmann

8. November 2020

Inhaltsverzeichnis

Einleitung

Dieses Skript richtet sich an gymnasiale Mittelschüler die mit den grundlegenden Begriffen 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 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.

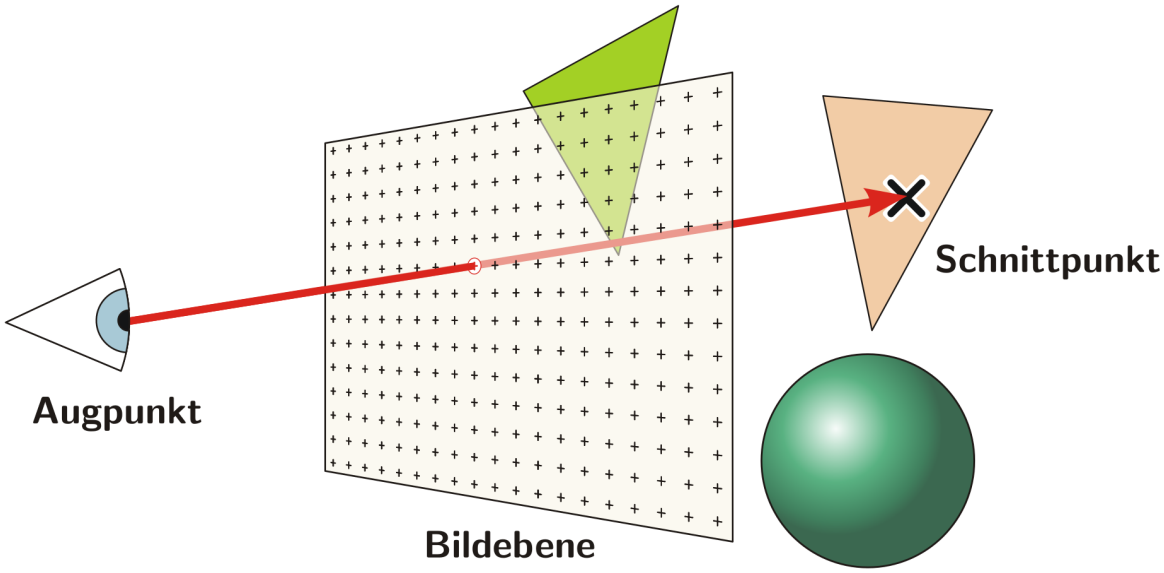

Abbildung 1: Raytracing: Es werden Strahlen ausgesendet und deren Schnittpunkte mit Objekten ermittelt. In diesem Fall wir der Durchstosspunkt auf der Bildebene in der Farbe des roten Dreiecks gefärbt.

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:

```
1 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:

```
1 l = [4, 2, 5, 9, 6, 2]
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)
print(inner(v, w)) # Skalarprodukt von v und w
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:

```
from numpy import array
from numpy.linalg import norm

```

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

Mit dem Punkt `.` kann auf die Member-Variablen und Member-Funktionen 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` defi-

niert 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 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.

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 `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 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

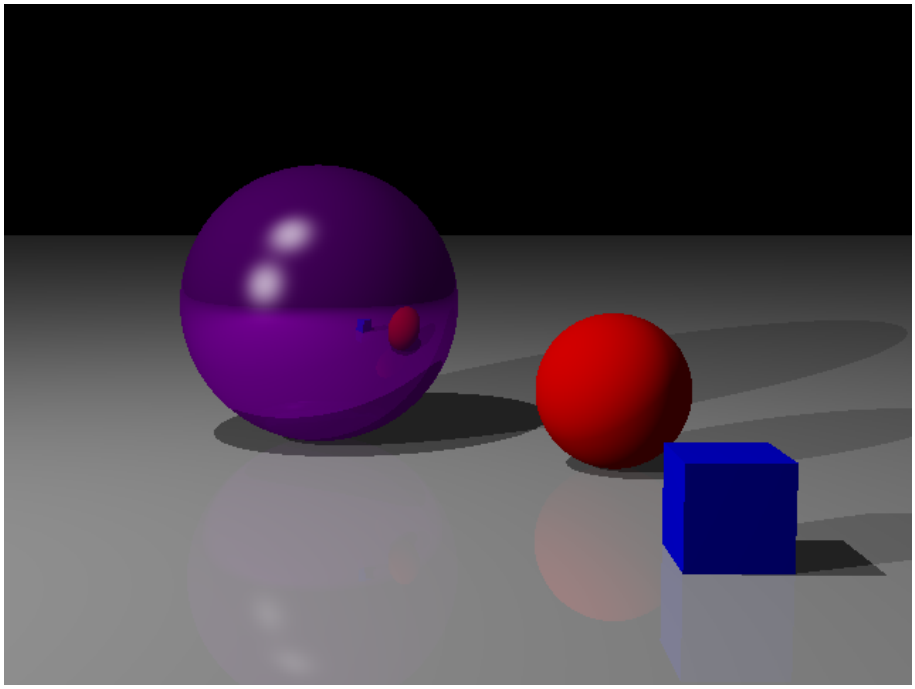

Abbildung 2: Dieses Bild wurde mit unserem Raytracing-Programm generiert.

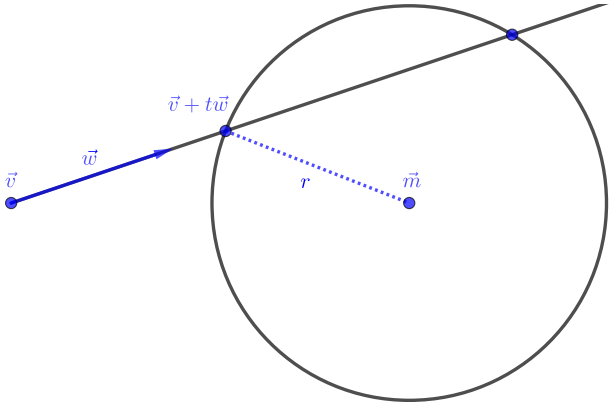

Abbildung 3: Schnittpunkt von Strahl und Kugel.

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

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

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.

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 $\vec{l}$. 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 Punkt $\vec{p}$ eine Zahl $c \in [0, 1]$

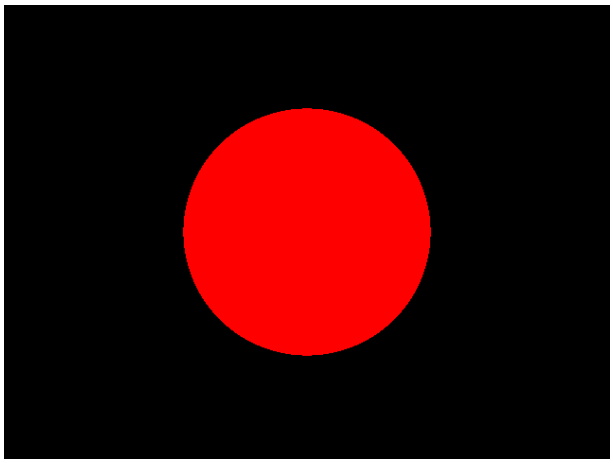

Abbildung 4: Lösung der Aufgabe 2.

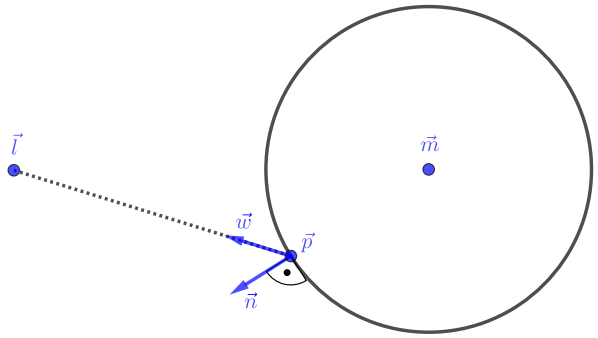

Abbildung 5: Die Kugel wird von einer Punktförmigen Lichtquelle in $\vec{v}$ beleuchtet.

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

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

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

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

$$\cos\left(\alpha\right) = \vec{n} \cdot \vec{w},$$

wobei α der Zwischenwinkel von $\vec{n}$ und $\vec{w}$ ist. Da der Kosinus nur Werte in $[-1, 1]$ annimmt, gilt wie verlangt $c \in [0, 1]$. Die Helligkeit 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 $c = 1$, also maximale Helligkeit. Je flacher 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.

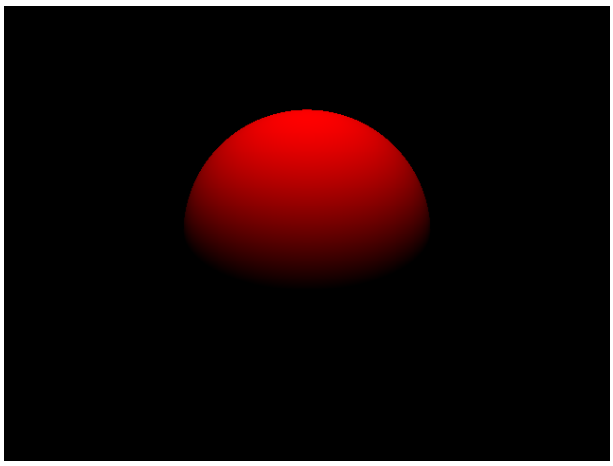

Abbildung 6: Lösung der Aufgabe 4.