

# Ray-Tracing mit Python

Martin Reiche, Tübingen, Januar 2019

Version vom 22.11.2025

## Inhaltverzeichnis

1	Vorwort.....	2
2	Einführung .....	2
3	Konzept.....	3
4	Design .....	3
4.1	Der Sehstrahl .....	3
4.2	Reflexion an der Kugel.....	4
4.3	Die innere Berechnungsschleife .....	5
4.4	Schattenbildung.....	5
4.5	Die Aufstellung .....	5
4.6	Zwei Koordinatensysteme .....	6
5	Erweiterungen .....	6
5.1	Lichtquelle .....	6
5.2	GIF.....	6
5.3	Nicht-reflektierende Kugeln .....	6
5.4	Transparente Kugeln .....	6
5.5	Andere Objekttypen .....	7
5.6	Performance .....	7

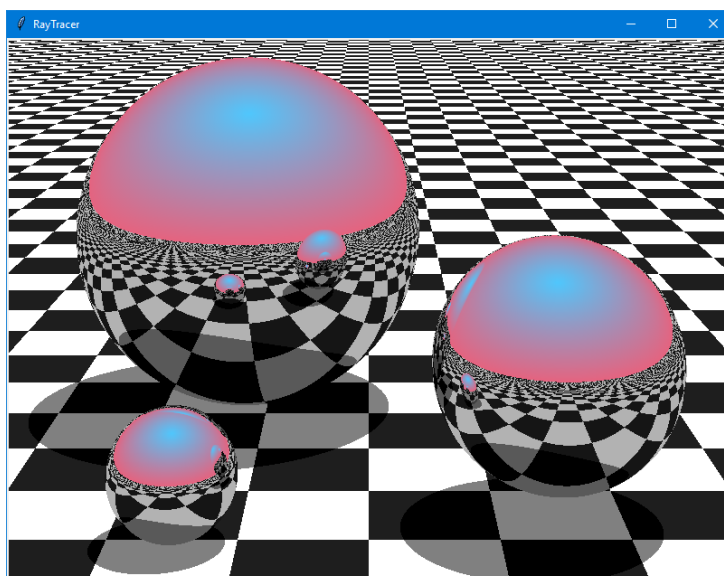
## 1 Vorwort

Zur Einordnung: Wir programmieren hier die "absoluten Basics", sprich die elementaren Grundlagen einer 3D-Darstellung. Den "state of the art", also hier den Stand der (für alle!) verfügbaren Technik kann man in vielen Videos bestaunen, wie z.B. [hier](#) mit dem Programm [Blender](#). Dort werden diese Grundlagen auch benutzt, aber sie sind unsichtbar, weil für den Benutzer des Design-Tools "Blender" uninteressant. Natürlich gibt es jede Menge solcher Programme, vor allem auch im industriellen [CAD](#) (Computer-Aided-Design). Siehe auch die Plattformen für Spiele: [Unreal](#), [Unity](#) oder [Godot](#).

Ray-Tracing für die Unterhaltungsindustrie wird heutzutage mit großem Aufwand betrieben, siehe [dieses Video](#). Während wir alle Pixel einzeln und nacheinander mit Python Basis-Funktionen berechnen, sind uns moderne Grafikkarten zillionenfach überlegen, wie [dieses Video](#) eindrücklich zeigt!

## 2 Einführung

Dieses Dokument beschreibt ein Programm zur Berechnung des folgenden Bildes (resolution = 200):

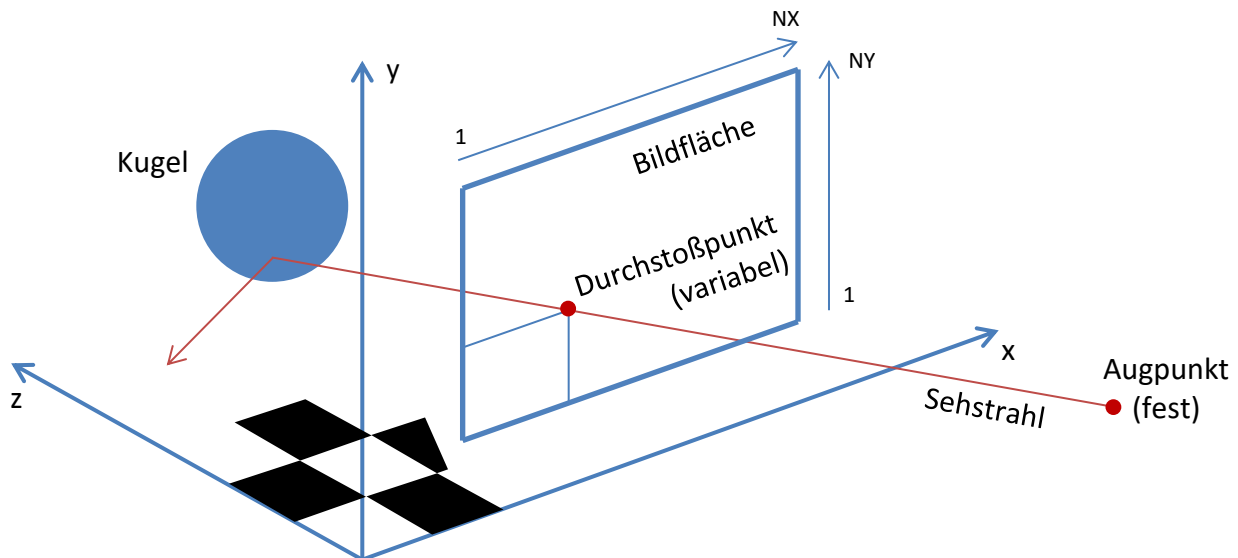


Die Auflösung lässt sich im Python Source Code leicht einstellen. Für eine (Full-HD) formatfüllende Darstellung, sollte der *resolution*-Wert auf 400 gesetzt werden.

Siehe auch <http://www.martin-reiche.de/raytracing.html>

### 3 Konzept

Folgendes Bild zeigt den geometrischen Aufbau zur Bilderzeugung:



Die Bildfläche liegt in der x-y-Ebene. Der Durchstoßpunkt wandert in kleinen Schritten über die gesamte Bildfläche. Jedes Mal wird dabei ein Sehstrahl vom Augpunkt (negative z-Koordinate) über den Durchstoßpunkt in die Szenerie (positive z-Koordinaten) konstruiert. Dort trifft er z.B. auf eine Kugel und wird reflektiert. Eventuell über ein oder mehrere Reflexionen an anderen Kugeln trifft der Strahl dann entweder auf den Boden (= x-z-Ebene) oder schießt in den Himmel. Je nachdem wird dann ein Punkt (Pixel) in der Bildfläche auf den entsprechenden Farbwert gesetzt.

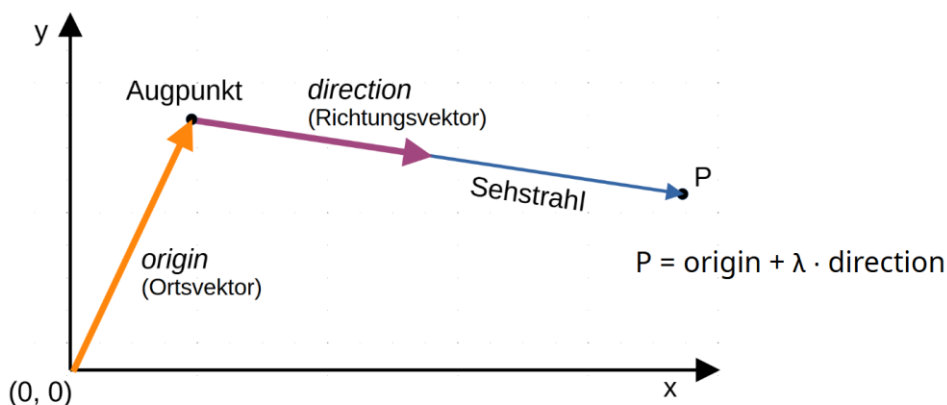
## 4 Design

### 4.1 Der Sehstrahl

Strahlen sind in der Klasse *Ray* implementiert. Der Sehstrahl besteht aus 2 Vektoren:

1. Ein Startpunkt (*origin*)
2. Eine Richtung (*direction*)

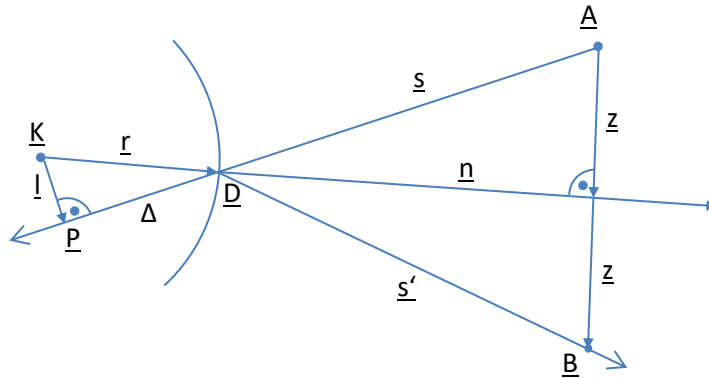
Jeder Punkt P des Strahls kann durch Wahl des Faktors  $\lambda$  erreicht werden, wobei gilt  $\lambda > 0$ .



*class Ray* Attribute: *Vector origin*, *Vector direction*

## 4.2 Reflexion an der Kugel

Die Reflexion an einer Kugel berechnen wir wie folgt: (Vektor-Variablen werden durch einen Unterstrich markiert.  $\underline{A}$  = Augpunkt)



Zunächst fallen wir das Lot  $\underline{l}$  vom Mittelpunkt  $\underline{K}$  der Kugel auf den einfallenden Sehstrahl  $\underline{s}$ . Dabei stellen wir fest, ob der Sehstrahl in Richtung Mittelpunkt der Kugel verläuft. Falls nicht, können wir diese Kugel ignorieren. Falls ja, prüfen wir die Länge des Lotvektors  $\underline{l}$  (=kleines L). Ist sie größer als der Kugelradius, kommt es zu keiner Reflexion und wir können diese Kugel ignorieren.

Ist der Lotvektor kleiner (kürzer) als der Radius, berechnen wir den reflektierten Strahl nach folgender Methode. Sehstrahl  $\underline{s}$  und Radius  $\underline{r}$  der Kugel liegen mit deren Mittelpunkt  $\underline{K}$ , dem Durchstoßpunkt  $\underline{D}$  und dem Lotvektor  $\underline{l}$  in einer Ebene. Der Normalenvektor  $\underline{n}$  der Kugeltangentialfläche verläuft in Richtung des Radiusvektors und damit ebenfalls in besagter Ebene. So können wir den Punkt  $\underline{B}$  als doppelte Verlängerung des Lotvektors von  $\underline{A}$  auf  $\underline{n}$  berechnen. Durch die Punkte  $\underline{D}$  und  $\underline{B}$  ist abschließend der reflektierte Strahl  $\underline{s'}$  festgelegt.

Die Rechnung in Einzelnen:

Das Lot verläuft per definitionem vom Aufpunkt  $\underline{K}$  senkrecht zum Sehstrahl und trifft ihn im Punkt  $\underline{P}$ .

$$\underline{P} = \underline{A} + \lambda \underline{s} \quad (\underline{A} \text{ ist der Startpunkt und } \underline{s} \text{ der Richtungsvektor des Strahls}) \quad (1)$$

Also muss das Skalarprodukt aus Lotvektor  $\underline{l} = \underline{P} - \underline{K}$  und Richtungsvektor der Geraden bzw. des Strahls Null sein d.h. es muss gelten:

$$\underline{s} \cdot (\underline{P} - \underline{K}) = 0 \quad (2)$$

(1) in (2) eingesetzt liefert

$$\underline{s} \cdot (\underline{A} + \lambda \underline{s} - \underline{K}) = 0 \quad (3)$$

$$\underline{s} \cdot \underline{A} + \lambda s^2 - \underline{s} \cdot \underline{K} = 0 \quad (4)$$

$$\lambda = \underline{s} \cdot (\underline{K} - \underline{A}) / s^2 \quad (5)$$

Falls  $\lambda > 0$  ist, heißt dies für uns, dass sich der Strahl in Richtung Kugel bewegt. (Den Fall, dass sich aus Sicht des Augpunktes die Kugeln überlappen d.h. hintereinander stehen, ignorieren wir zunächst.)

Um den Durchstoßpunkt  $\underline{D}$  zu berechnen, multiplizieren wir den Einheitsvektor des Sehstrahls  $\underline{s}$ , hier  $\underline{u}$  genannt, mit der Strecke  $\Delta = |\underline{D} - \underline{P}|$ .  $\Delta$  selbst ergibt sich nach Pythagoras aus  $\underline{l}$  und  $\underline{r}$ .

$$\underline{D} = \underline{K} + \underline{l} - \Delta \cdot \underline{u} \quad (6)$$

Mit bekanntem  $\underline{D}$  ist der Normvektor  $\underline{n}$  definiert, auf den wir das Lot  $\underline{z}$  von  $\underline{A}$  fällen können. Damit gelangen wir zu  $\underline{B}$  und können den reflektierten Strahl bestimmen.

$$\underline{B} = \underline{A} + 2 \cdot \underline{z} \quad (7)$$

Wir legen allerdings kein neues Objekt an, sondern „verbiegen“ nur dessen Werte. So können wir leicht zusätzliche Informationen, wie die Anzahl der Reflexionen transportieren.

### 4.3 Die innere Berechnungsschleife

Wir rastern die Bildfläche gemäß der gewünschten Auflösung, indem aus den Strahlen für jeden Rasterpunkt genau ein Pixel berechnet wird. Dazu prüfen wir alle möglichen Reflexionen solange, bis keine Kugel dem Strahl mehr im Wege steht. Danach entscheiden wir anhand der y-Komponente der Strahlrichtung, ob dieser über dem Horizont (=farbiger Himmel) oder darunter (=Schachbrett) liegt und nehmen eine entsprechende Färbung vor.

### 4.4 Schattenbildung

Bevor wir den Strahl auf den Boden treffen lassen, prüfen wir noch, ob die Auftreffstelle im Schatten liegt. Dazu erzeugen wir von dort einen Strahl in Richtung „Sonne“ und prüfen, ob irgendeine Kugel im Wege steht. Konfiguration

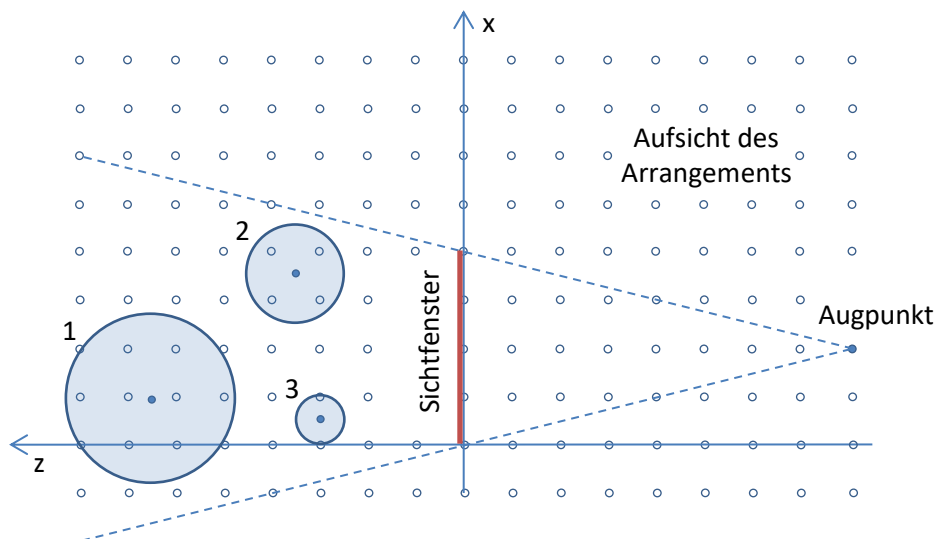
### 4.5 Die Aufstellung

(siehe den Quellcode!) Das Sichtfenster hat eine Höhe von 3 und eine Breite von 4.

Der Augpunkt liegt bei den Koordinaten (2,0, 4,5, -8,0)

Die 3 Kugeln sind wie folgt positioniert:

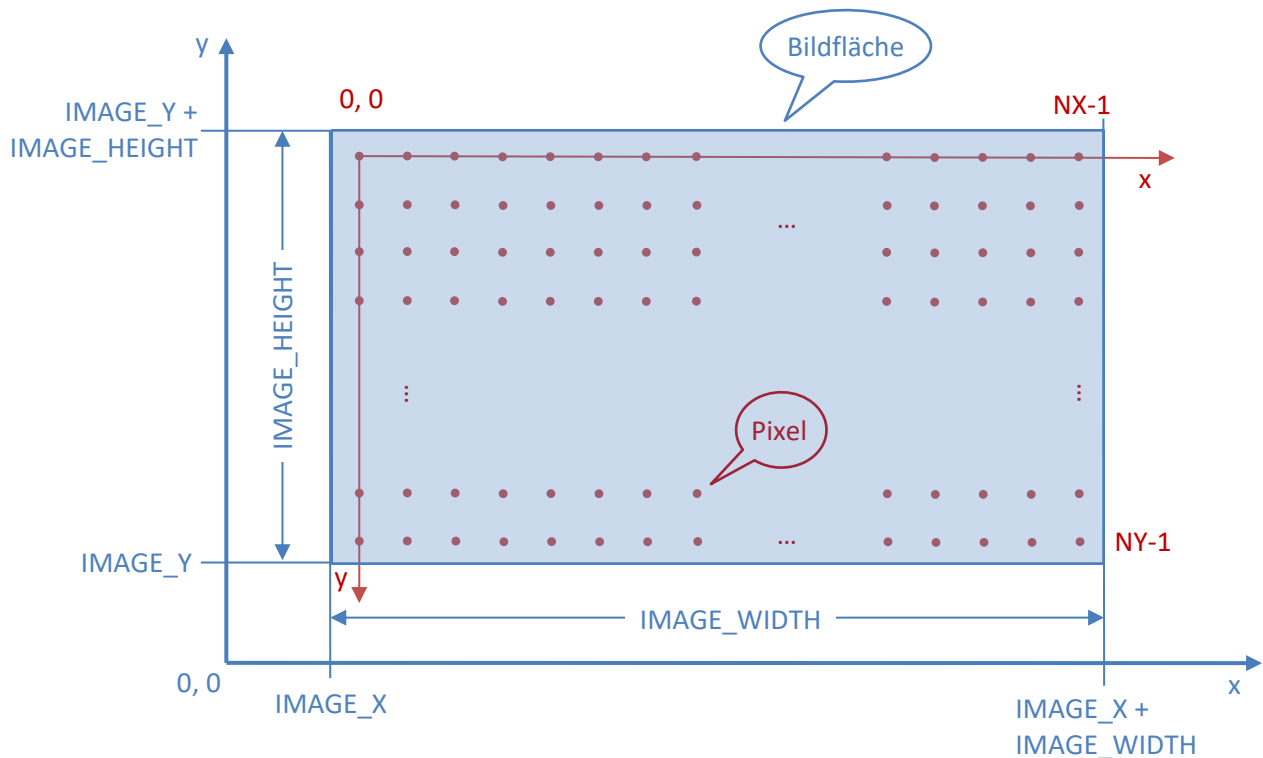
X	Y	Z	Radius
0,8	1,7	6,5	1,7
3,5	1,2	3,5	1,0
0,5	0,5	3,0	0,5



Die gestrichelten Linien begrenzen das Sichtfeld in der Horizontalen.

## 4.6 Zwei Koordinatensysteme

Weltkoordinaten (float, ohne Einheit) vs. **Bildkoordinaten** (int, Pixel)



## 5 Erweiterungen

### 5.1 Lichtquelle

Aktuell wird mit der Schattenbildung der Kugeln implizit eine punktförmige Lichtquelle angenommen, die auf den Kugeloberseiten aber nicht zu sehen ist. Dieser Defekt sollte behoben werden.

### 5.2 GIF

Mit einer Art Stop-Motion-Technik kann man Bewegungen einer oder mehrerer Kugeln simulieren und aus den Einzelbildern eine GIF-Datei erstellen. Optimalerweise schließt das letzte Bild an das erste so an, dass der Eindruck einer kontinuierlichen Bewegung entsteht. (Kein „Ruckeln“ am Ende der GIF-Bildersequenz).

Ob sich die Kugeln gleichmäßig zu bewegen scheinen oder diese z.B. umeinander kreisen, sei der Kreativität überlassen.

### 5.3 Nicht-reflektierende Kugeln

Undurchsichtige, nicht reflektierende Kugeln können realistisch dargestellt werden, wenn die Helligkeit der Oberfläche entsprechend moduliert ist – oben heller, unten dunkler.

### 5.4 Transparente Kugeln

Ganz oder teilweise durchsichtige Kugeln, die auch etwas Licht reflektieren können auch interessante Objekte darstellen.

## 5.5 Andere Objekttypen

Auch zylindrische Objekte sind mit vertretbarem Aufwand noch programmierbar. Mehraufwand entsteht, wenn die kreisförmigen Begrenzungsflächen ins Blickfeld geraten.

Ähnliches gilt für Würfel und Quader.

## 5.6 Performance

Das vorliegende Programm ist (vermutlich) recht langsam in der Ausführung z.B. gegenüber einer Implementierung in C. Doch die Tatsache, dass sich jedes Pixel unabhängig vom anderen berechnen lässt, eröffnet die Möglichkeit der zeitlich parallelen Ausführung. Heutige Gaming-Laptops für 1300 € bieten z.B. 16 Kerne, die zeitgleich verschiedene Dinge tun können. Das multiprocessing-Modul in Python ermöglicht die parallele Ausführung von Prozessen. Ist unser Bild z.B. 1600 Pixel breit, könnte man jeweils 100 Spalten einem der 16 Kerne zuweisen. Das Aufspalten der Arbeit und das Zusammenführen der Ergebnisse erfordert sicherlich auch Rechenzeit, aber eine Verzehnfachung der Performance sollte so erreichbar sein. Für Prozessoren mit nur 2 oder 4 Kernen sähe der Gewinn entsprechend magerer aus.