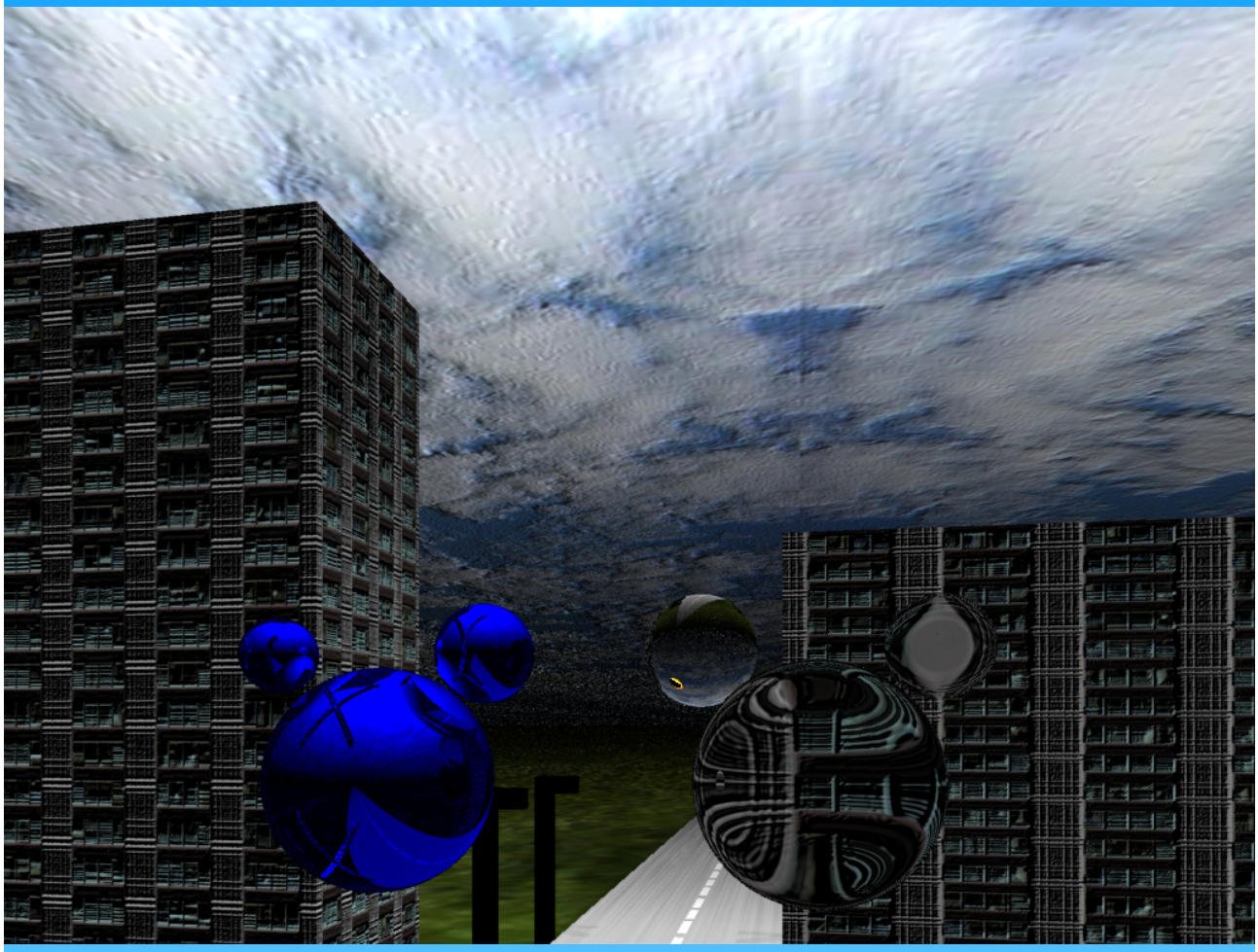
Physik Grundkurs 13. Klasse
Heinrich-Böll-Schule Hattersheim

Optimierung virtueller Realitätsumgebungen und deren 3D-Darstellung durch Einbindung physikalischer Gesetzmäßigkeiten und Effekte

Besondere Lernleistung im Fach
Physik
Abitur 2011



Marc Miltenberger

Inhaltsverzeichnis

Einleitung

Idee

Anfang 2010 wurde ich von Frau Beck gefragt, ob ich bei der hessischen Schülerakademie teilnehmen möchte, die zwei Wochen der Sommerferien in Beschlag nehmen. Nach einer Recherche über diese Veranstaltung sagte ich gerne zu. Jeder Teilnehmer musste ein Referat inklusive Präsentation vorbereiten. Mein Thema war „Darstellung 3-dimensionaler Körper und geometrische Operationen“. Daraufhin habe ich mich mit Algorithmen beschäftigt, mit deren Hilfe sich dreidimensionale Welten auf zwei Dimensionen projizieren lassen. Raytracing (engl. für „Strahlenverfolgung“) ermöglicht eine erwartungstreue Projektion.

Ich hatte die Idee, ein Programm zu schreiben, welches dieses Verfahren unterstützt.

Daraufhin suchte ich im Internet nach weiteren Informationen und entwarf nach und nach einen Designentwurf der Realisierung. Fortschritte in der Entwicklung wurden schnell sichtbar.

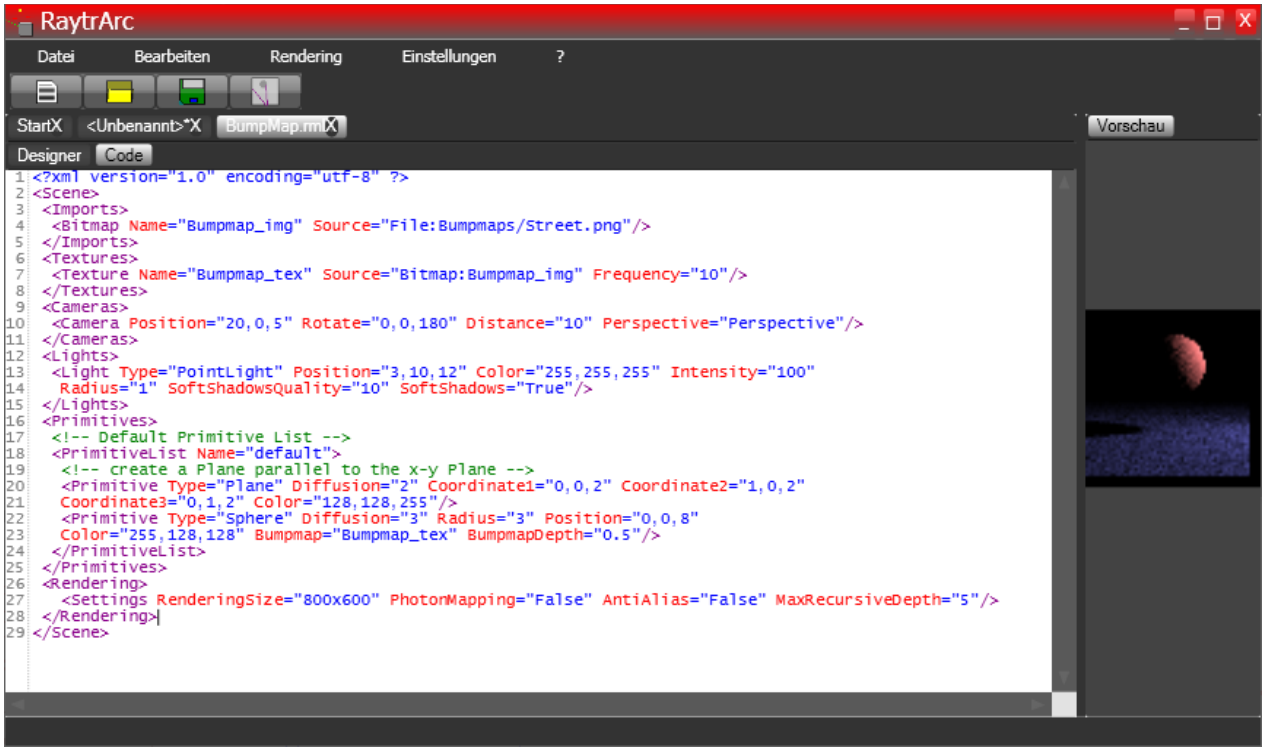


Abbildung 1: RaytrArc Markup Language

Ich entschied, dass **XML** (***E**xtensible **M**arkup **L**anguage*) die beste Wahl wäre, um Objekte in ihrer Welt (Szene) zu beschreiben. Die Szene kann durch die Objekte, die in ihr enthalten sind und deren Eigenschaften beschrieben werden.

Theoretischer Teil

Im vorliegenden Dokument werde ich zuerst Grundlegendes erläutern, insbesondere über die Wahrnehmung des Menschen. Anschließend folgt die Erklärung des Raytracing-Prinzips. Die Weiterführung besteht aus der Implementierung in Form eines Computerprogramms. Zum Abschluss ziehe ich ein Fazit über die Zukunft des Raytracings und dessen Vor- und Nachteile.

Raytracing ist ein Verfahren, um weitestgehend erwartungsgetreue Abbilder einer künstlichen Szenerie unter Verwendung der Zentralprojektion zu erhalten. Es ermöglicht die Abbildung dreidimensionaler Szenen auf zweidimensionalen Flächen.

Das Ziel besteht aus der Erzeugung einer Illusion beim Betrachter. Es werden dafür verschiedene physikalische Modelle verwendet. Raytracing ist eines der am häufigsten verwendeten Verfahren zur Darstellung (engl. „rendering“) erwartungsgetreuer Bilder.

Für die Erstellung von Graphiken mithilfe von Raytracing sind Kenntnisse im Bereich der Vektorrechnung obligatorisch. Diese werden in diesem Dokument vorausgesetzt.

Praktischer Teil

Die verwendeten Bilder entstammen alle meiner praktischen Arbeit oder wurden von mir mithilfe eines Bildbearbeitungsprogramms[1] angefertigt. Der praktische Teil besteht aus einem plattformunabhängigen Raytracer, der auf den Namen *RaytrArc* (Raytracer von Marc) getauft wurde. Er wurde von mir in mühevoller Arbeit von Grund auf programmiert und unterstützt einen Großteil der in diesem Dokument beschriebenen Algorithmen.

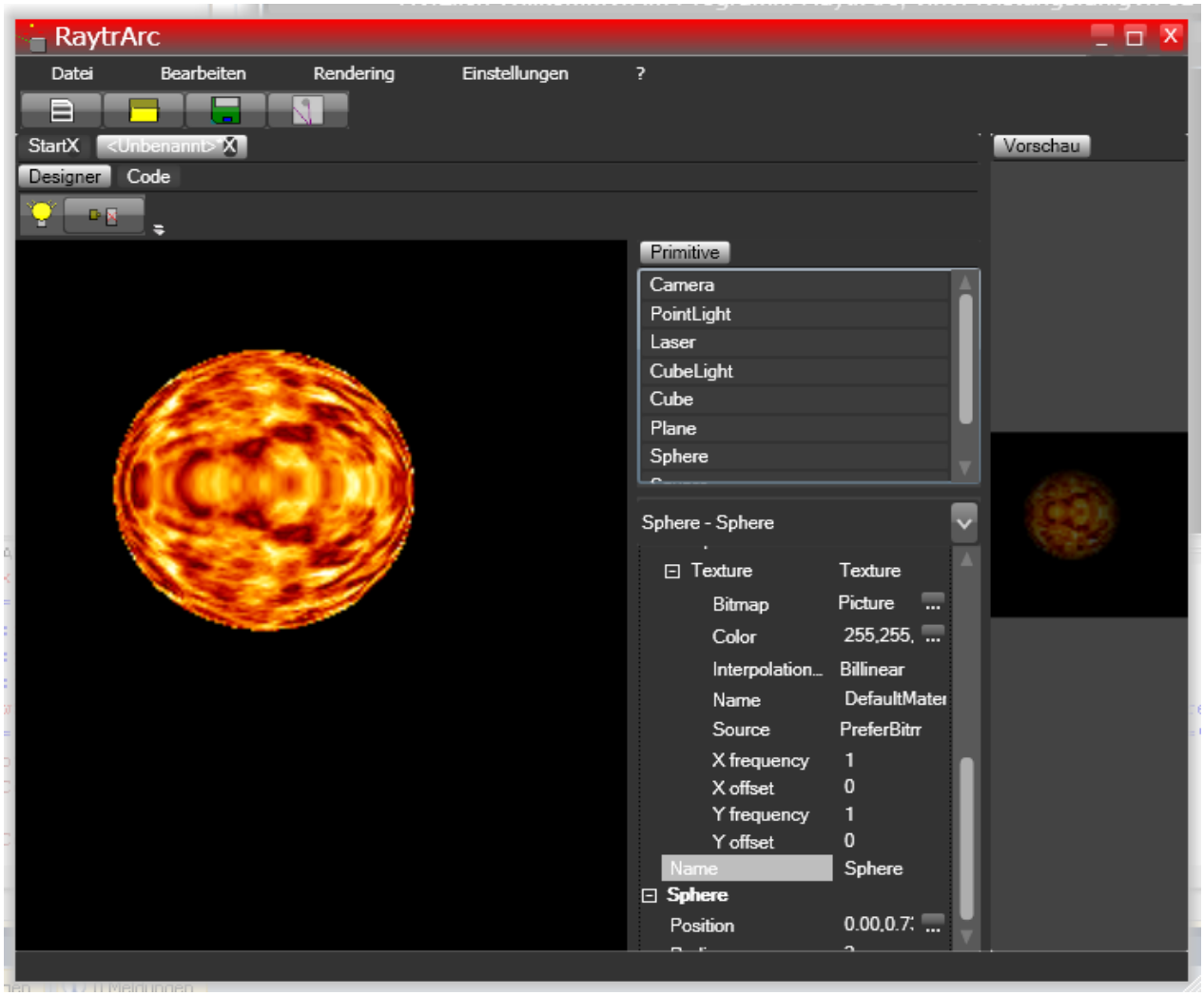


Abbildung 2: RaytrArc unter Windows basierenden Betriebssystemen

Mit ☑ gekennzeichnete Algorithmen wurden in *RaytrArc* implementiert. Bei ☐ gekennzeichneten Algorithmen wurde die Implementierung bisher nicht abgeschlossen oder nicht vorgenommen.

1 *GIMP*, the GNU Image Manipulation Program

Visuelle Wahrnehmung

Farbsehen

Die Sonne oder andere Lichtquellen strahlen Licht in Form von Photonen aus. Photonen haben sowohl Teilchen- als auch Welleneigenschaften. Licht besteht aus elektromagnetischer Strahlung. Die Photonen verfügen demnach über eine Frequenz. Die Frequenz lässt sich alternativ auch in die Einheit der Wellenlänge (Meter) umrechnen. Photonen, deren Wellenlängen sich im Bereich von ca. 380 – 780 Nanometer befinden, sind für das menschliche Auge sichtbar. Verschiedene Frequenzen werden im Gehirn zu verschiedenen Farben zusammengesetzt.[1]

Abbildung 3: Additive Farbmischung: Verschiedene Mischfarben aus Rot, Grün und Blau. Die Überschneidungen stellen jeweils Mischfarben dar.

Wenn Licht von einer Lichtquelle auf ein Objekt trifft, wird es im Regelfall in alle Richtungen reflektiert und ein Teil absorbiert.

Das Licht trifft schließlich in das menschliche Auge. Durch die Linse gelangt es anschließend auf die Netzhaut. Dort befindet sich ein auf dem Kopf stehendes Abbild der gesehenen Wirklichkeit. Dort befindliche Photorezeptoren werden von dem Licht angeregt.[2]

Es gibt zwei verschiedene Typen von Photorezeptoren: Stäbchen und Zapfen. Das menschliche Auge verfügt über mehr Stäbchen als Zapfen. Erstere sind für das Sehen in der Dämmerung relevant, letztere sind für das Farbsehen wichtig. Damit verschiedene Farben erkannt werden können, muss es verschiedene Arten von Zapfen geben, die auf Photonen unterschiedlicher Wellenlängen reagieren. Der Mensch zählt zu den Trichromaten, was bedeutet, dass sie über drei verschiedene Arten von Zapfen verfügen: rote, grüne und blaue.

Ein roter Apfel erscheint dem Mensch deshalb rot, weil der Apfel fast alle Wellenlängen absorbiert. Er reflektiert lediglich Photonen in einem flachen Spektrum im roten Spektralbereich. Diese Photonen treffen in unser Auge und reizen die roten Zapfen; der Farbeindruck „rot“ entsteht.

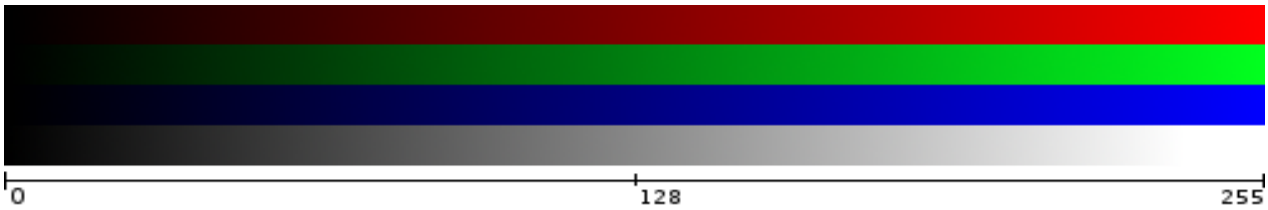


Abbildung 4:
oben: Farbverlauf von (0, 0, 0) (schwarz) nach (255, 0, 0) (Rot)
oberhalb der Mitte: Farbverlauf von (0, 0, 0) nach (0, 255, 0) (Grün)
unterhalb der Mitte: Farbverlauf von (0, 0, 0) nach (0, 0, 255) (Blau)
unten: Farbverlauf von (0, 0, 0) nach (255, 255, 255) (Weiß)

Licht, wie es die Sonne aussendet, erkennt der Mensch als Grau oder Weiß, wenn die drei Zapfen ausgeglichen gereizt werden. Dieses Prinzip wird bei Bildschirmen ausgenutzt: Um die Farben zu

1 Meyers Physiklexikon, S.527, Stichwort „Licht“, 1973
2 Lamberto Maffai & Adriana Fiorentini, Das Bild im Kopf, Birkhäuser Verlag, 1997, S. 2-3

erzeugen, werden drei Pixel (Bildpunkte), welche sich sehr nah beieinander befinden (so genannte Subpixel) und die Photonen unterschiedlicher Wellenlängen emittieren können (Rot, Grün und Blau) voneinander unabhängig angesteuert, damit in unserem Kopf durch unterschiedliche Farbintensitäten verschiedene Farben gemischt werden (siehe Abbildung 3).

Für die Berechnungen in diesem Dokument wird das RGB-Farbsystem eingesetzt, womit sich viele existierende Farben darstellen lassen. Die Farben werden als Vektoren dargestellt. Die Farbkomponenten Rot, Grün und Blau werden hierbei als X-, Y- und Z-Komponente verwendet. Am Computer wird die Intensität der Farbkomponenten als eine Zahl von *0 – 255* angegeben, wobei *0* hierbei bedeutet, dass diese Farbe nicht verwendet wird und *255 (≙ 2^8*, ein Byte) für die volle Intensität steht. Auf Abbildung 4 werden verschiedene Farbverläufe dargestellt. Auf der linken Seite ist die Intensität am geringsten, während sie auf der rechten Seite am höchsten ist.

Es folgen einige exemplarische Vektoren für verschiedene Farben.

$$\vec{C}_{\text{black}} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad \vec{C}_{\text{white}} = \begin{pmatrix} 255 \\ 255 \\ 255 \end{pmatrix} \quad \vec{C}_{\text{red}} = \begin{pmatrix} 255 \\ 0 \\ 0 \end{pmatrix} \quad \vec{C}_{\text{yellow}} = \begin{pmatrix} 255 \\ 255 \\ 0 \end{pmatrix} \quad \vec{C}_{\text{gray}} = \begin{pmatrix} 128 \\ 128 \\ 128 \end{pmatrix}$$

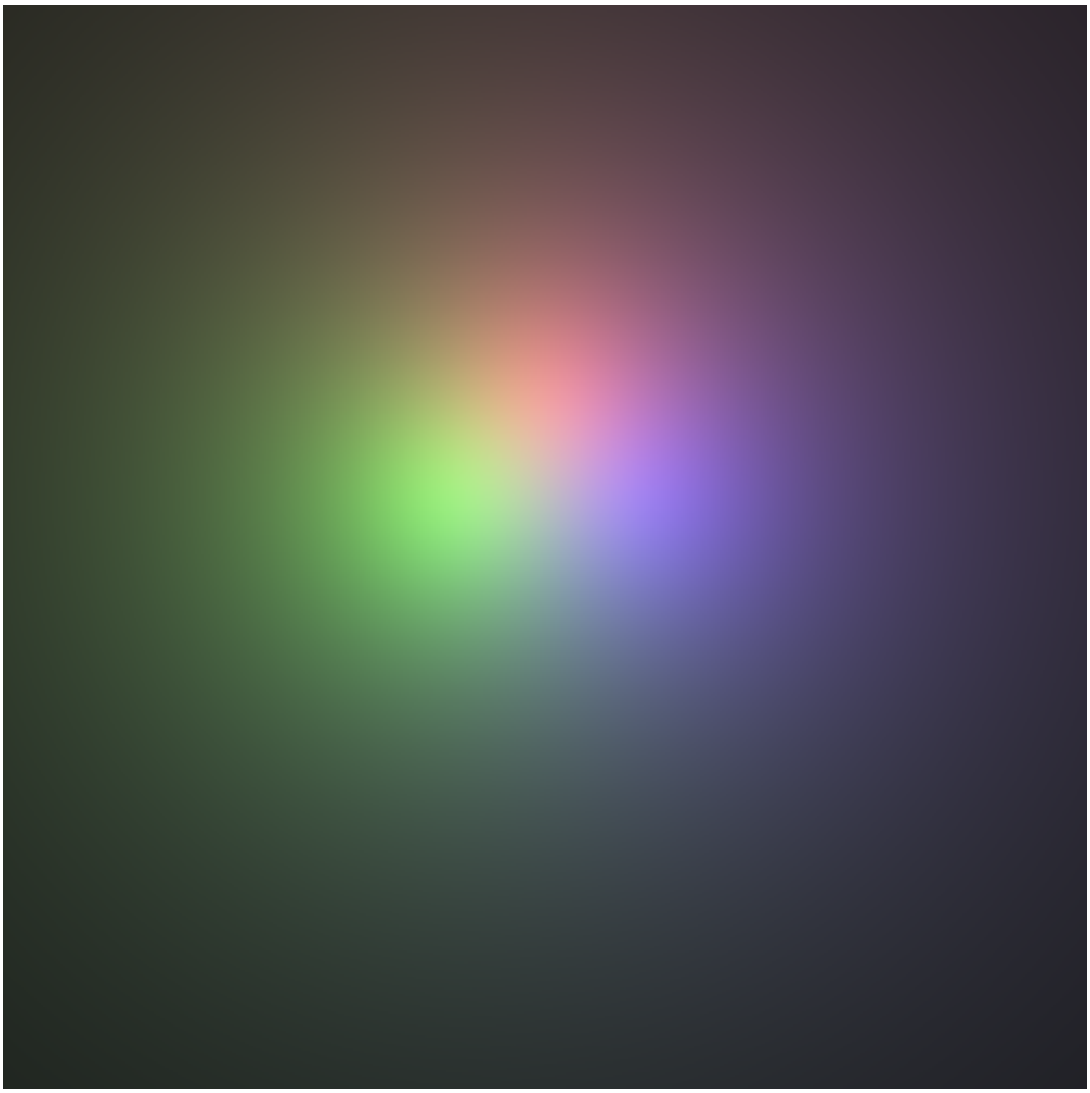


Abbildung 5: Additive Farbmischung: Gelb und Cyan sind nur mit Mühe zu erkennen.

Erzeugung einer Illusion – dreidimensionaler Raum auf zweidimensionaler Fläche

Projektionen

Zentralprojektion ☑

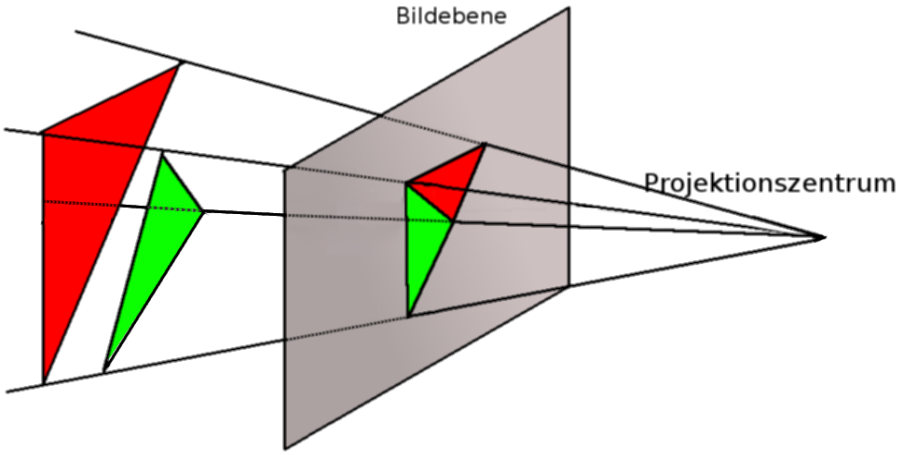


Abbildung 6: Darstellung einer Zentralprojektion

Der vorliegende dreidimensionale virtuelle Raum muss auf den Bildschirm ausgegeben werden. Da letzterer zweidimensional ist, muss eine Projektion erfolgen. Bei der Zentralprojektion schneiden sich die Projektionsstrahlen im Projektionszentrum.
Zwischen dem Projektionszentrum und den Objekten befindet sich die Bildebene, die das Resultat der Projektion darstellt. Die zu projizierenden Objekte werden kleiner dargestellt, je weiter sie von der Bildebene entfernt sind. Dies ist die zweidimensionale Abbildung eines dreidimensionalen Raumes.[1] Die Zentralprojektion wird von menschlichen Augen angewendet, mit dem Unterschied, dass die Bildebene hinter dem Projektionszentrum liegt und das Bild somit auf dem Kopf steht.

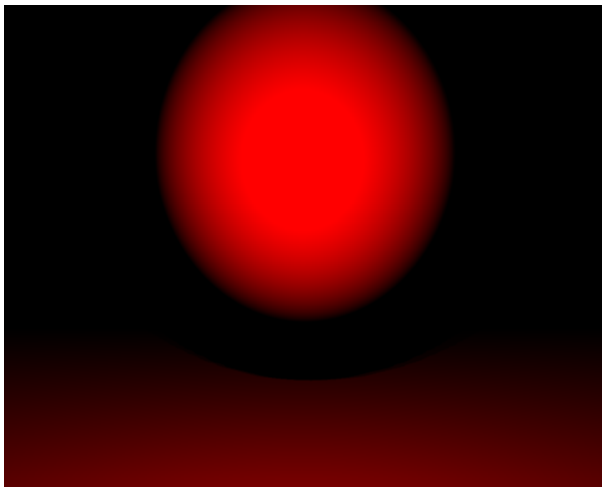


Abbildung 7: Eine Zentralprojektion - eine Kugel und ein Boden

1 Lamberto Maffai & Adriana Fiorentini, Das Bild im Kopf, Birkhäuser Verlag, 1997, S. 82

Parallelprojektion ☑

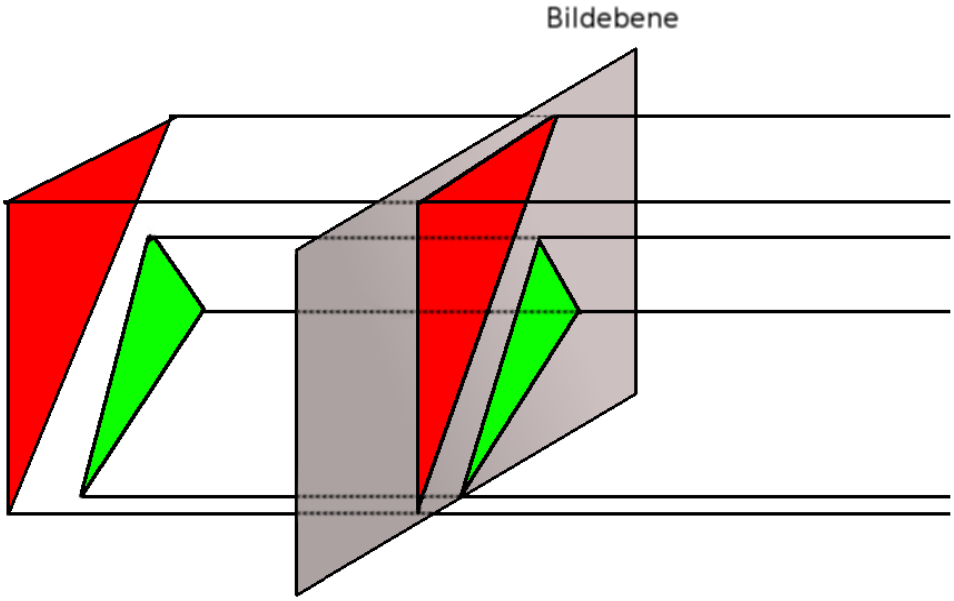


Abbildung 8: Darstellung einer Parallelprojektion

Daneben existiert die Möglichkeit einer Parallelprojektion (orthographische Projektion). Hierbei sind alle Projektionsstrahlen parallel zueinander. Somit ist kein Projektionszentrum vorhanden.

Die parallelen Projektionsstrahlen führen auf Abbildung 9 dazu, dass die Ebene nicht sichtbar ist.

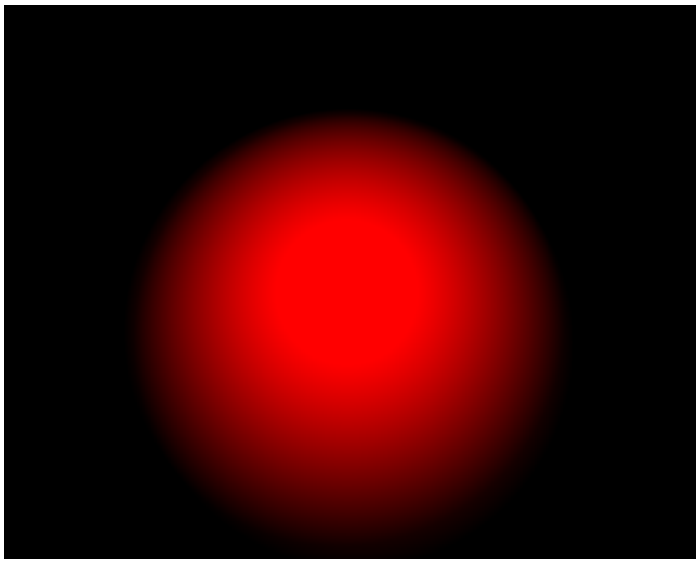


Abbildung 9: Der Boden ist unsichtbar, da er parallel zu den Projektionsstrahlen verläuft und somit nicht von diesen getroffen wird.

Stereoskopie

Trotz der Projektion ist die Darstellungsfläche lediglich zweidimensional. Dies ist dem Betrachter stets bewusst. Um dem Gehirn eine dreidimensionale Umgebung vorzutäuschen, müssen den beiden Augen jeweils unterschiedliche Bilder gezeigt werden. Da beide Augen in der Natur Bilder aus leicht unterschiedlichen Blickwinkeln wahrnehmen, kann das Gehirn daraus die Entfernung verschiedener Objekte abschätzen.

Da meist jedoch nur eine Projektionsfläche (der Bildschirm) zur Verfügung steht, muss ein Verfahren verwendet werden, welches folgende Kriterien sicherstellt.

1. Das Bild für das linke Auge und das Bild für das rechte Auge muss auf *einer* Fläche dargestellt werden.
2. Die beiden Einzelbilder müssen separiert werden, damit das linke und das rechte Auge das jeweils passende Bild zu sehen bekommt.

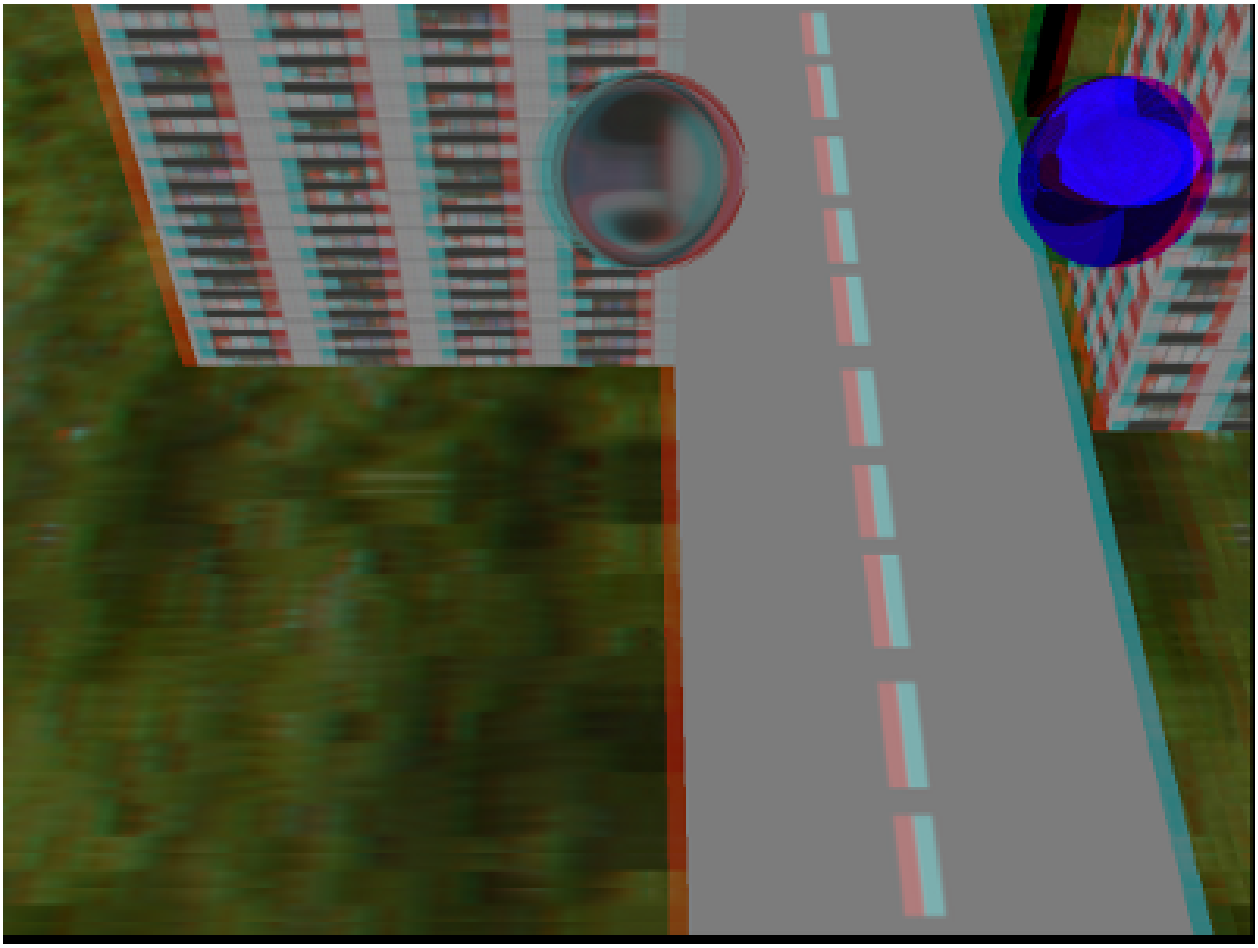


Abbildung 10: Ein gerendertes Anaglyphenbild

Farbanaglyphenbilder ☑
Eines dieser Verfahren ist das Farbanaglyphenbildverfahren. Dazu wird das Bild für das linke Auge mit Cyan und das Bild, das für das rechte Auge bestimmt ist, mit Rot eingefärbt. Beide Einzelbilder werden übereinander gelegt. Der Betrachter trägt eine Anaglyphenbrille, die farblich umgekehrt zum Einfärbungsprozess angeordnet ist. Die Cyanfolie filtert das Cyan heraus, sodass das rote Bild für das rechte Auge übrig bleibt. Die Rotfolie hingegen filtert den Rotanteil aus dem Anaglyphenbild, sodass lediglich der cyanfarbene Teil des Bildes für das linke Auge verbleibt.

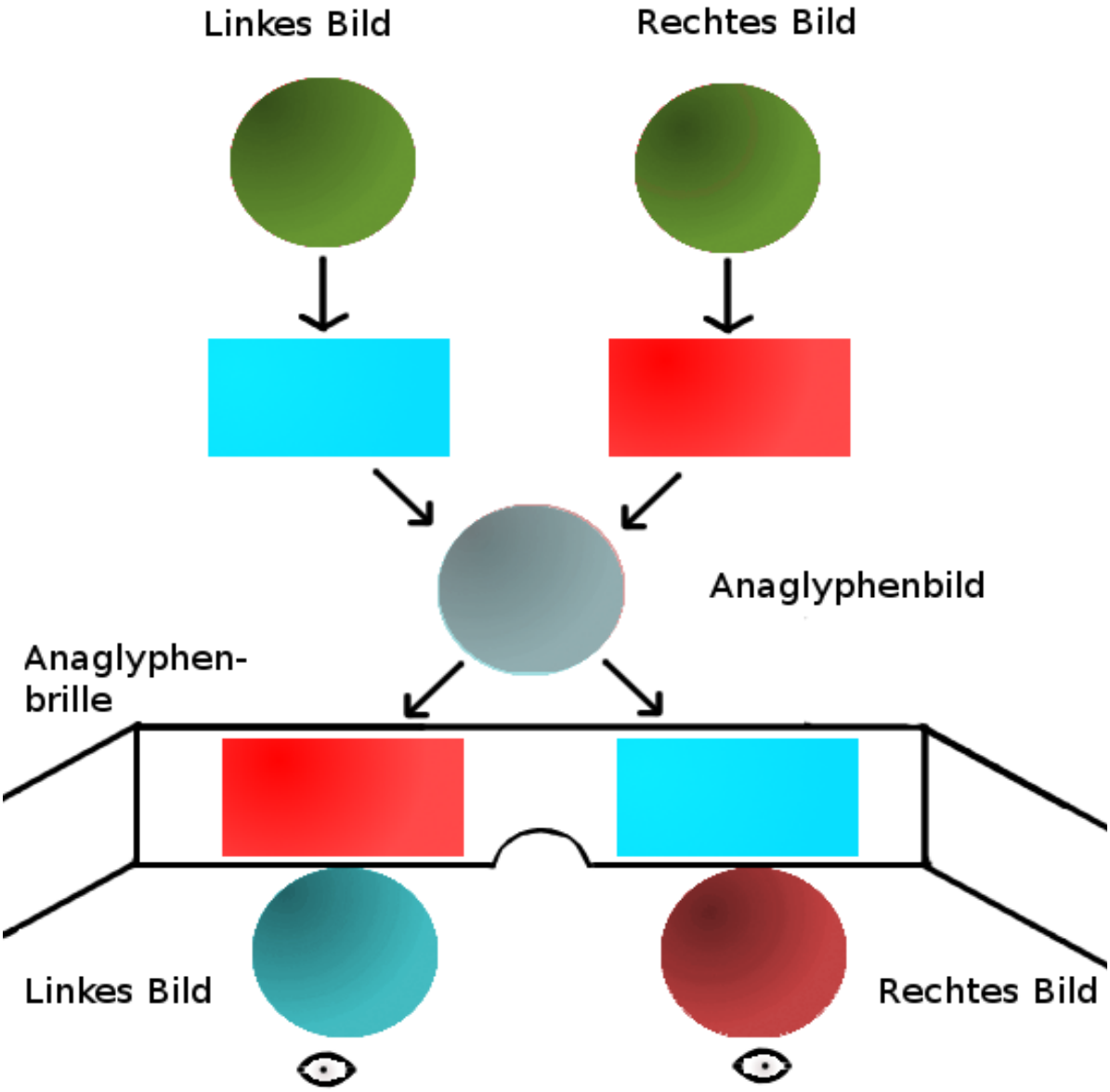


Abbildung 11: Die Funktionsweise des Anaglyphenbildverfahrens

Die entstehende Verfälschung der Farben lässt sich mit verschiedenen Algorithmen, die bei der Komposition der Einzelbilder eine Rolle spielen, begrenzen. Dabei werden die resultierenden Farbanteile unterschiedlich auf das rechte und das linke Bild verteilt.

Andere Verfahren

Daneben existieren andere Verfahren, die das stereoskopische Sehen für Menschen bei zweidimensionalen Bildern ermöglichen. Sie nutzen unterschiedliche Methoden der Zusammenführung und der Trennung der Einzelbilder.

Beim Polfilterverfahren wird das Licht beispielsweise polarisiert. Die speziellen Monitore sind in der Lage, die Bilder für die beiden Augen unterschiedlich zu polarisieren (90° zueinander). Eine Polfilterbrille filtert das nicht für das jeweilige Auge bestimmte Bild heraus. Ein Vorteil gegenüber dem Farbanaglyphenbildverfahren ist, dass die Farben nicht verfälscht werden.

Das Shutterbrillenverfahren ist ein anderes verbreitetes Verfahren.

Die Shutterbrille ist eine Brille, die wahlweise jede der beiden Brillengläser abdunkeln kann. Ein darauf ausgerichter Monitor zeigt mit einer hohen Frequenz die beiden Bilder abwechselnd. Die Shutterbrille ist mit dem Monitor synchronisiert und verfügt über die Information, wann der Monitor welches Teilbild zeigt. Dementsprechend kann die Brille das Display für das andere Auge abdunkeln.

Wenn die Anzeige des Bildes auf dem Monitor und das Abdunkeln der Brillengläser entsprechend performant funktioniert (120 Hz oder mehr), wird das Flimmern vom Menschen nicht mehr wahrgenommen und ein stereoskopischer Effekt stellt sich beim Betrachter ein.[1]

Ein mechanisches Verfahren zur Stereoskopie lässt sich mittels eines Gitters realisieren. Der Nintendo 3DS[2] arbeitet mit diesem Verfahren. Ein feines Gitter befindet sich vor dem Display. Das Gitter sorgt dafür, dass das Licht in Abhängigkeit von der Bildschirmposition in verschiedene Richtungen abgelenkt wird. Da keine Brille benötigt wird, spricht man von einem autostereoskopischen Verfahren. Allerdings muss sich der Kopf an einer bestimmten Stelle befinden, da sonst nicht das richtige Bild zum entsprechenden Auge kommt.

1 http://www.heise.de/newsticker/meldung/Nintendo-kuendigt-Handheldkonsole-3DS-an-961085.html

2 Hessische Schülerakademie, 2010 - Referate

Grundlegendes Prinzip von Raytracing

Das Prinzip des Raytracings unterscheidet sich von der Wahrnehmung des Menschen. Vom Licht werden bewusst die Welleneigenschaften ausgelassen und nur die Strahleneigenschaften beachtet. Licht besteht in diesem Fall aus Strahlen. Man könnte zwar sehr viele Lichtstrahlen von der Lichtquelle auszusenden; die meisten dieser Strahlen würden allerdings unser Auge (im folgenden „Kamera“ genannt) verfehlen. Aus diesem Grund werden Strahlen von der Kamera und nicht von der Lichtquelle losgeschickt. Dieses Verfahren ist vergleichbar einer Theorie der alten Griechen. Diese nahmen an, dass „das Auge Strahlen aussendet, die sich wie feine Tastarme den Objekten nähern und ihre Form erfassen.“[1]

Die virtuelle Welt, in der sämtliche Objekte existieren, nennt man Szenerie. In ihr existieren Primitive, aus welchen schließlich die Objekte in der 3D-Welt aufgebaut werden. Primitive können Ebenen, Kugeln, Dreiecke oder andere Körper sein. Zum Beispiel könnte man aus zwölf Dreiecken einen Quader darstellen: Für jeden der sechs Seiten werden zwei Dreiecke benötigt, da sich aus zwei Dreiecken ein Rechteck zusammensetzen lässt. Den Computer interessiert es allerdings nicht, ob verschiedene Primitive ein für den Menschen identifizierbares Objekt bezeichnen. Deshalb werden sämtliche Berechnungen nicht mit den Objekten, sondern mit den einzelnen Primitiven, mit denen sich die verschiedenen Objekte zusammensetzen lassen, durchgeführt.

Die Kamera hat einen Ursprung in der 3D-Welt, also eine Position, an der sie sich befindet. Dies ist äquivalent mit der Position des Betrachters. Weiterhin muss die Blickrichtung festgelegt sein. Weiterhin gibt es einen „virtuellen unsichtbaren Bildschirm“ im 3D-Raum, der vor der Kamera in Blickrichtung aufgespannt ist. Für jeden Bildpunkt (Pixel) wird hierbei ein Strahl ausgehend vom Kameraursprung erzeugt. Die Strahlen gehen jeweils durch die Position des Pixels am virtuellen Bildschirm.

Diese Konstruktion ist identisch mit der vorher angesprochenen Zentralprojektion.

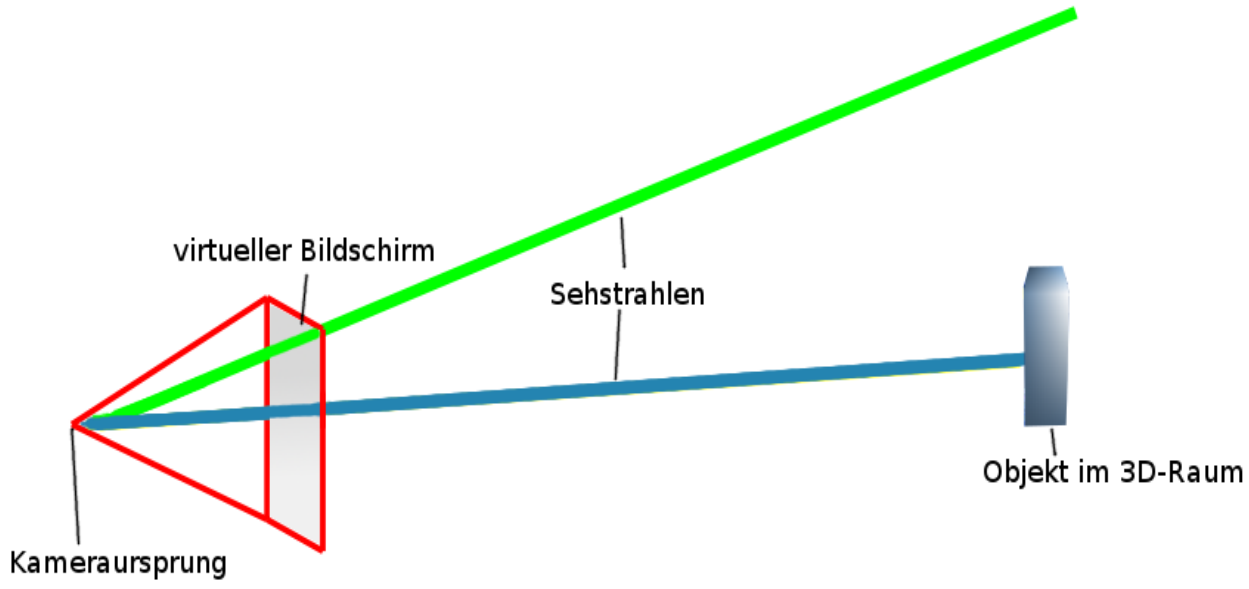


Abbildung 12: Prinzip des Raytracings

Abbildung 12 verdeutlicht dieses Prinzip. Exemplarisch wurden zwei Strahlen - ein blauer und ein

1 Lamberto Maffai & Adriana Fiorentini, Das Bild im Kopf, Birkhäuser Verlag, 1997, S. 3

grüner - eingezeichnet. Der grüne Strahl trifft kein Objekt, der blaue hingegen einen Quader.

Wenn man einen Strahl von der Kamera durch die virtuelle Welt schickt, kann es passieren, dass ein Primitiv mit diesem Strahl kollidiert. Die Farbe des Kollisionsprimitivs wird dabei zunächst einmal übernommen.

Dieses Verfahren wird nacheinander für sämtliche Bildpunkte angewendet, wobei sich nur die Richtung des Strahls ein wenig ändert.

Mathematisch gesehen besteht ein Strahl aus einem Ursprung (engl. origin) und einer Richtung (engl. direction). Beides wird durch Vektoren mit jeweils drei Koordinaten (x, y, z) in der Parameterschreibweise angegeben:

$$\vec{x}_{ray} = \vec{o}_{ray} + t * \vec{d}_{ray}, t \in \mathbb{R}$$

Die Gerade $\vec{x}_{ray}$ beschreibt den Strahl. $\vec{o}_{ray}$ ist hierbei der Ursprung des Strahls (= origin of ray), $\vec{d}_{ray}$ stellt den Richtungsvektor dar (= direction of ray). Der Ursprung könnte irgendein Punkt auf der Geraden sein, idealerweise ist er allerdings mit dem Ortsvektors des „Ausgangspunkts“ (in diesem Fall dem Kameraursprung) identisch. t sei eine reelle Zahl. Wenn der Parameter t durch einen konkreten Wert ersetzt wird, bekommt man einen Punkt auf der Gerade.

Wenn überprüft werden soll, ob ein Lichtstrahl eine bestimmte Ebene trifft, kann man wie folgt vorgehen.

Die Ebene lässt sich folgendermaßen beschreiben:

$$\vec{x}_{plane} = \vec{o}_{plane} + r * \vec{d1}_{plane} + s * \vec{d2}_{plane}, r, s \in \mathbb{R}$$

$\vec{o}_{plane}$ stellt einen Ortsvektor der Ebene dar, also einen beliebigen Punkt auf der Ebene. $\vec{d1}_{plane}$ und $\vec{d2}_{plane}$ sind zwei linear unabhängige Richtungsvektoren der Ebene. Bei drei gegebenen Punkten ($\vec{p1}, \vec{p2}, \vec{p3}$) auf der Ebene lässt sich die Ebenengleichung folgendermaßen bestimmen:

$$\vec{x}_{plane} = \vec{p1} + r * (\vec{p2} - \vec{p1}) + s * (\vec{p3} - \vec{p1}), r, s \in \mathbb{R}$$

Interessant ist nun, ob der Strahl mit der Ebene kollidiert.

Es wird ein Punkt gesucht, der sowohl auf der Geraden liegt als auch in der Ebene enthalten ist. Um diesen Punkt zu erhalten, setzt man $\vec{x}_{ray}$ und $\vec{x}_{plane}$ gleich. Von Bedeutung ist dabei lediglich der Wert von t. Mithilfe der Cramerschen Regel[1] kommt man letztlich auf eine bestimmte Lösung. Falls keine Lösung existiert, steht fest, dass der Strahl die Ebene nicht treffen kann; beide müssen somit parallel sein.

Wenn der Parameter t ausgerechnet wurde und in die Strahlengleichung eingesetzt wird, erhält man den Kollisionspunkt.

Allerdings muss dabei beachtet werden, dass für $t < 0$ kein beachtenswerter Kollisionspunkt vorliegt, da sich dieser Kollisionspunkt *hinter* dem Ursprung (der Kameraposition) befindet und somit nicht „gesehen“ werden kann. Deshalb ist es wichtig, dass nach t und nicht nach r und s aufgelöst wurde.

Letzteres würde allerdings zum selben Ergebnis führen.

Diese Berechnung wird für jede Ebene in der Szenerie angewandt. Falls es mehrere

1 Das große Tafelwerk interaktiv, Cornelsen, 2003, S. 74

Kollisionspunkte gibt, wird dieser mit der geringsten Distanz zum Kamerastandpunkt übernommen:

$$d_{Distance} = |\vec{P}_{Collisionpoint} - \vec{o}_{ray}|$$

Anders ausgedrückt: Wenn der Strahl mehrere Primitive schneidet, wird nur die Farbe des am nächsten gelegenen Primitives übernommen, da er die anderen gewissermaßen verdeckt.

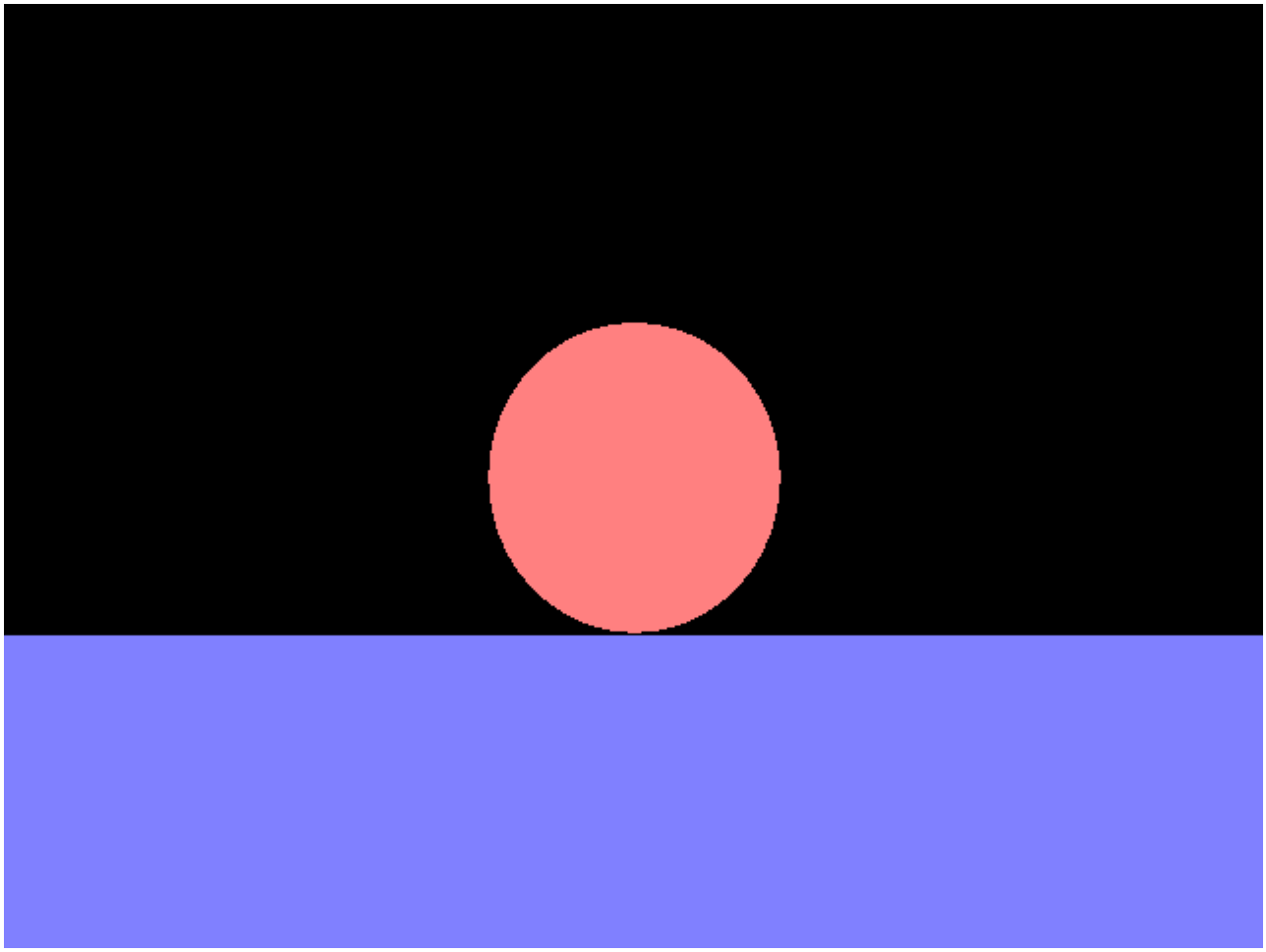


Abbildung 13: Eine Ebene und eine Kugel - keine Licht/Schatteneffekte

Abbildung 13 stellt das Resultat der bisherigen Verfahrensweise dar.

Zusätzlich habe ich noch eine Kugel in die Szenerie eingefügt.

Glücklicherweise existieren einige Erweiterungsmöglichkeiten für Raytracer, von denen viele den Realismus der resultierenden Bilder anheben, da die Kugel auf der Abbildung eher einem Kreis als einer Kugel gleicht.

Erweiterungen

Qualitätsoptimierend

Beleuchtung ☑

Das auf der vorherigen Seite gerenderte Bild sieht bisher nicht perzeptiv[1] aus. Das liegt daran, dass bisher die Farbe des getroffenen Primitivs einfach übernommen wurde. Es gibt keine Beleuchtung, sondern es ist vielmehr alles gleich hell. Wie im Kapitel „visuelle Wahrnehmung“ beschrieben, entsteht aufgrund dessen nicht einmal ein minimaler „3D-Eindruck“ beim Betrachter.

Ein Modell, welches für die Beleuchtung in 3D-Spielen und ähnlichen Anwendungen häufig eingesetzt wird, nennt sich Phong-Beleuchtungsmodell.

Eine wesentliche Rolle spielt das Skalarprodukt. Dabei wird die Normale am Kollisionspunkt benötigt. Die Normale ist ein Vektor, der senkrecht von einer Ebene absteht.

Im Falle einer Ebene ist dies relativ einfach; eine Normale lässt sich leicht über das Kreuzprodukt der beiden Richtungsvektoren ermitteln:

$$\vec{N} = \vec{d1}_{plane} \times \vec{d2}_{plane}$$

Die Normale einer Kugel ist abhängig vom Kollisionspunkt des Sehstrahls mit dem Objekt:

$$\vec{N} = \vec{P}_{Collisionpoint} - \vec{P}_{Position}$$

$\vec{P}_{Collisionpoint}$ stellt den Ortsvektor des Kollisionspunktes des Sehstrahls mit dem Objekt dar.

$\vec{P}_{Position}$ ist der Mittelpunkt der Kugel. Zwei verschiedene Normalen lassen sich auf Abbildung 14 erkennen.

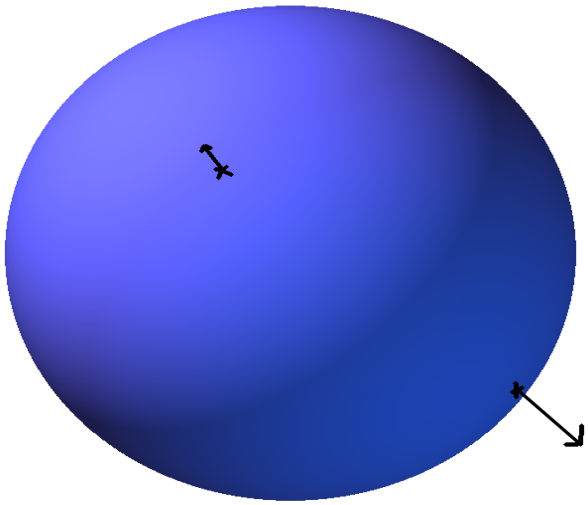


Abbildung 14: Die verschiedenen Normalen einer Kugel

1 erwartungsgetreu

Bei spitzen Winkeln scheint die Lichtquelle direkter auf die Oberfläche, d.h. das Skalarprodukt zwischen der normalisierten Normale am Kollisionspunkt und dem Vektor vom Kollisionspunkt zur Lichtquelle (normalisiert) ergibt nahezu 1.

$$\vec{L}_{CollisionToLight} = \vec{P}_{LightPosition} - \vec{P}_{Collisionpoint}$$

$\vec{N}$ und $\vec{L}_{CollisionToLight}$ müssen normalisiert werden:

$$\vec{L}_{NormalizedCollisionToLight} = \vec{L}_{CollisionToLight} * \frac{1}{|\vec{L}_{CollisionToLight}|}$$

$$\vec{N}_{Normalized} = \vec{N} * \frac{1}{|\vec{N}|}$$

Bei spitzen Winkeln hingegen scheint die Lichtquelle nicht so stark auf die Oberfläche. Am stärksten ist der Effekt bei zwei rechtwinklig zueinander stehenden Vektoren: Das Skalarprodukt beträgt 0.

$$\vec{C}_{lighting} = \vec{C}_{object} * \vec{N}_{Normalized} \cdot \vec{L}_{NormalizedCollisionToLight}$$

Das Skalarprodukt wird mit der Oberflächenfarbe $\vec{C}_{object}$ (bestehend aus einem Rot-, Grün- und Blaufarbwert) multipliziert.

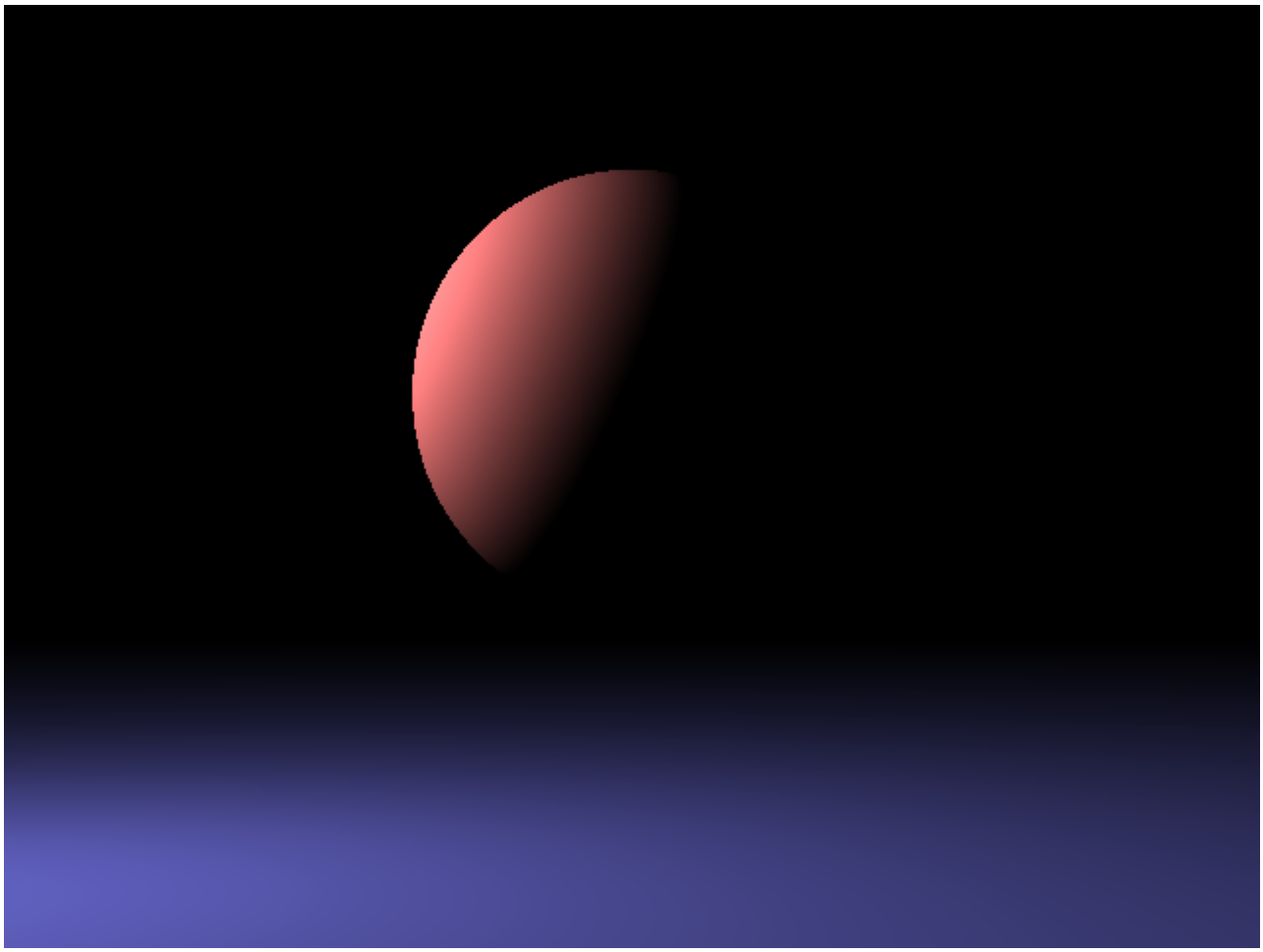


Abbildung 15: Die gleiche Szene wie in Abbildung 13*, diesmal mit Licht*

Einfache Schatten ☑

Ein Schatten entsteht, wenn das Licht einer Lichtquelle den Kollisionspunkt nicht erreichen kann, weil sich ein anderes Primitiv dazwischen befindet. Der Vereinfachung halber geht man davon aus, dass das Licht punktförmig ist, was zu harten Schatten führt.

Es wird also ein neuer Lichtstrahl vom Kollisionspunkt zu jeder Lichtquelle berechnet und geprüft, ob das am nächsten gelegene Primitiv die Lichtquelle ist. Ist das der Fall, gibt es keinen Schatten durch diese Lichtquelle, andernfalls wird die Lichtquelle für das Beleuchtungsmodell nicht beachtet und mit der nächsten Lichtquelle weitergemacht (für diese Lichtquelle kommt also kein Licht an).

Diese Technik ist ein stark vereinfachtes Modell. Das Licht wird als ein Strahl betrachtet und ist unendlich schnell. Außerdem breitet es sich vollständig geradlinig aus. Aber es ist ein recht guter Kompromiss aus Geschwindigkeit und Ästhetik: Pro Kollisionspunkt muss nur ein zusätzlicher Strahl (pro Lichtquelle) berechnet werden.

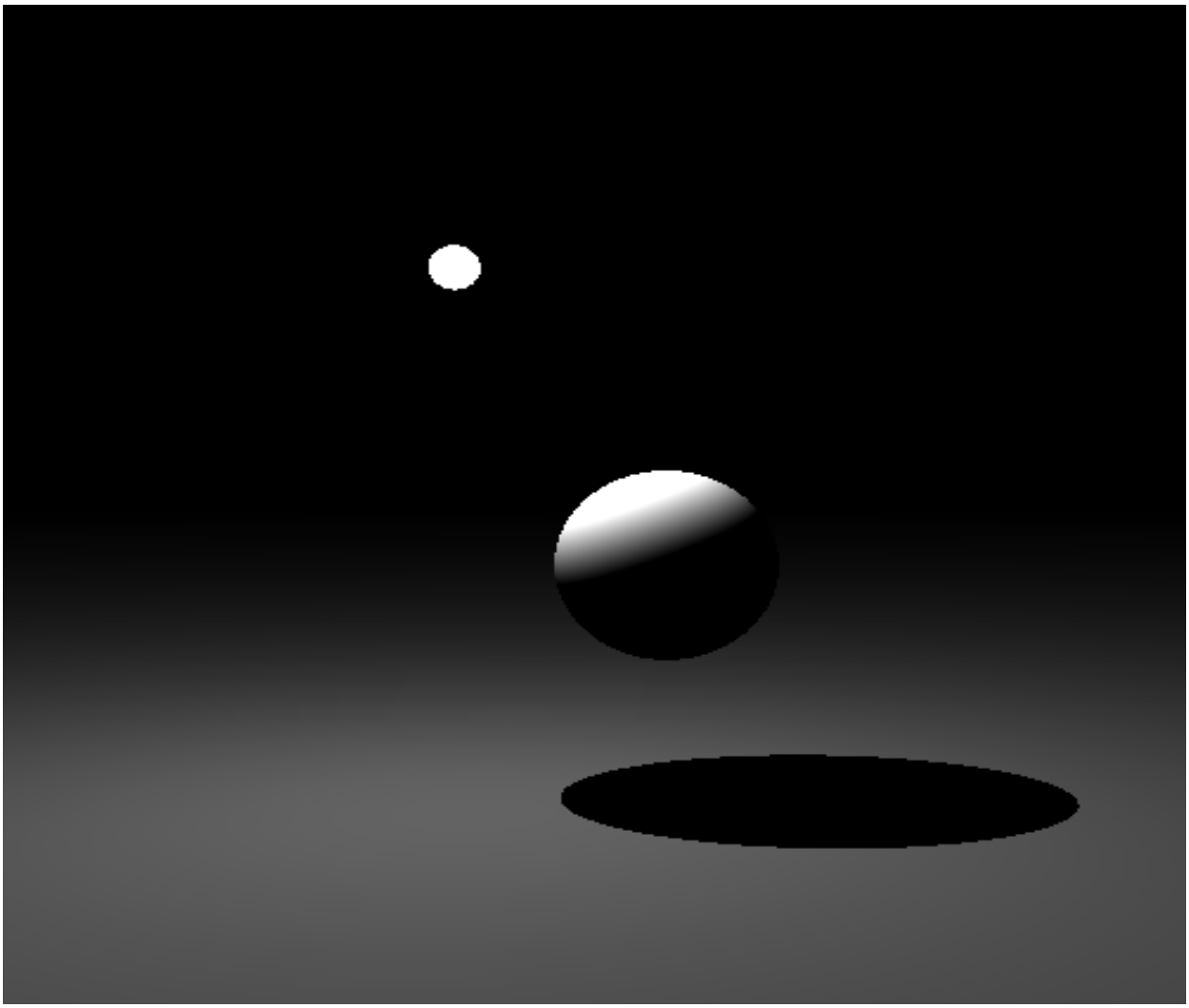


Abbildung 14: Eine Szene mit einfachen (harten) Schatten

Weiche Schatten ☑

Der Grund, warum die Schatten „harte Kanten“ haben, ist wie bereits erwähnt, dass Lichtquellen als punktförmig betrachtet wurden. In der Realität gibt es jedoch keine punktförmigen Lichtquellen, auch wenn die Glühbirne recht klein ist. Sie verfügt stets über einen Radius.

Der Einfachheit halber gehe ich davon aus, dass es sich um eine kreisrunde Glühbirne handelt. In der Realität wird das Licht von jeder Stelle im Glühdraht ausgesendet und somit existieren praktisch unendlich viele Teilschatten, die sich alle mehr oder weniger überlappen, weswegen „weiche Schatten“ entstehen.

Da sich unendlich viele Schatten nicht abbilden lassen (der Computer würde folglich unendlich viel Rechenzeit in Anspruch nehmen), nimmt man exemplarisch einige Strahlen heraus. Statt unendlich vieler Strahlen dienen beispielsweise nur noch 9 Strahlen, deren Strahlenursprung irgendwo zufällig auf dem Glühdraht liegt, als Basis. In der Praxis werden 9 zufällige Punkte innerhalb der Position der Lichtquelle und dessen Radius (innerhalb der Kugel) ausgewählt.

Jeder dieser 9 Teilschatten verfügt nur über 1/9 der „Verdeckungsstärke“. Diese neun Teilschatten werden unabhängig voneinander betrachtet, als wären anstelle eines Schattens neun schwächere Teilschatten vorhanden, die geringfügig variieren.

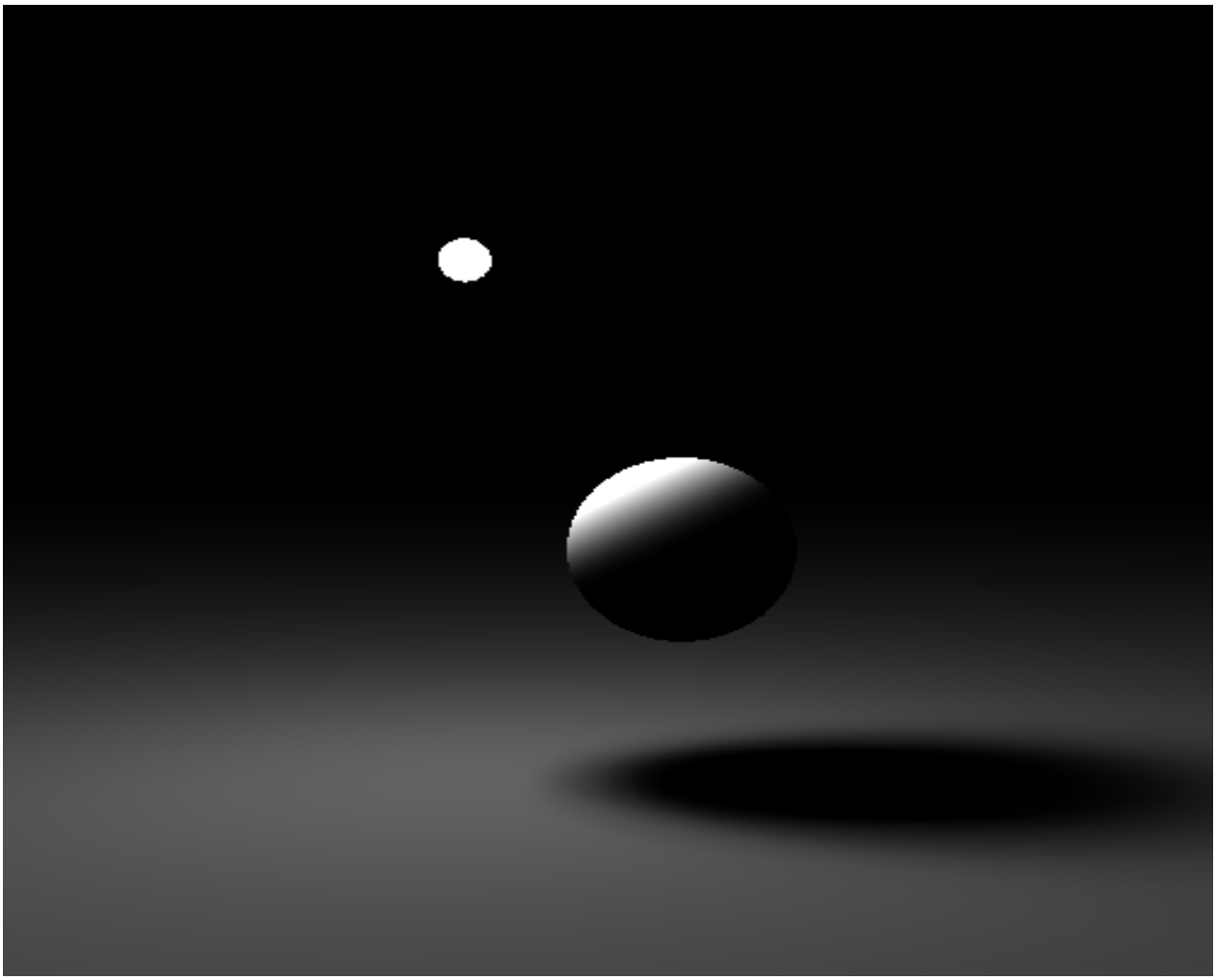


Abbildung 15: Weiche Schatten mit 1000 Teilschatten

Reflexionen ☑

Mit den bisher gezeigten Erweiterungen lassen sich bereits einige ansehnliche Ergebnisse produzieren. Allerdings können mit den bisher gezeigten Methoden keine Spiegel oder Christbaumkugeln dargestellt werden, da diese aus reflektierenden Materialien bestehen und solche bisher keine Erwähnung fanden.

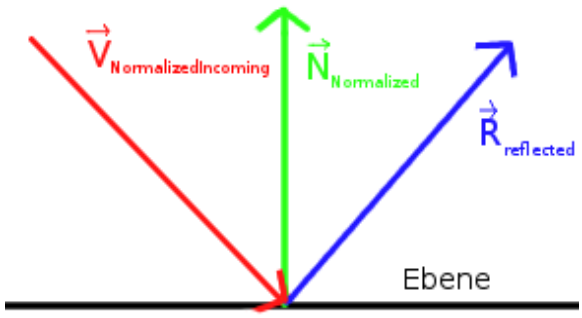


Abbildung 16: Reflexion an einer Ebene

$$\vec{R}_{reflected} = \vec{V}_{NormalizedIncoming} - 2 * \left(\vec{N}_{Normalized} \cdot \vec{V}_{NormalizedIncoming} \right) * \vec{N}_{Normalized}$$ [1]

$\vec{R}_{reflected}$ ist der resultierende Richtungsvektor. $\vec{V}_{NormalizedIncoming}$ ist der Eingangsvektor, der dem (normalisierten) Richtungsvektor des Sehstrahls entspricht. $\vec{N}_{Normalized}$ ist der normalisierte Normalvektor an der Stelle des Kollisionspunkts.

Nachdem der neue Richtungsvektor berechnet wurde, kann die Renderfunktion erneut rekursiv aufgerufen werden. Diesmal entspricht der Strahlenursprung dem jetzigen Kollisionspunkt und der Richtungsvektor des Strahls dem ausgerechneten Richtungsvektor. Der resultierende Farbwert wird mit der Objektfarbe verrechnet (siehe Kapitel *Beleuchtung*).

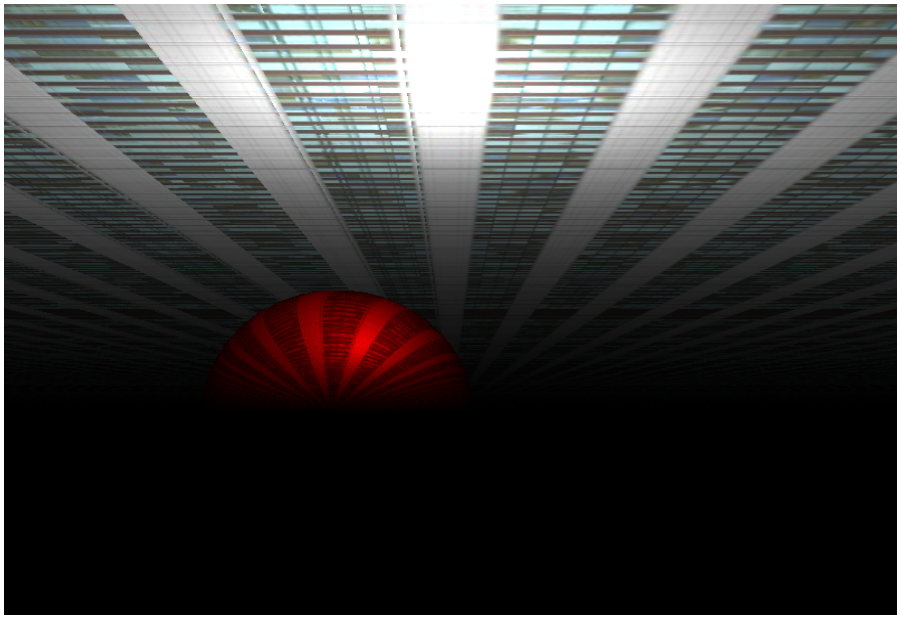


Abbildung 17: Reflexion an einer roten Kugel

1 http://keepcoding.bplaced.com/kcdev/tutorials/HLSL%20Introduction.pdf, S. 4

Lichtbrechung ☑

Nun lassen sich zwar reflektierende Primitive darzustellen, allerdings keine Materialien wie Glas oder Flüssigkeiten, die nur bis zu einem gewissen Maße reflektierend sind.

Um solche Materialien darstellen zu können, wird das Licht in Richtung des Lots (der Normalen) gebrochen. Die „Stärke“ der Brechung bezeichnet man als Brechungsindex.

Den Brechungsindex berechnet man, indem man den Quotienten von der Lichtgeschwindigkeit im Ausgangsmedium und der Lichtgeschwindigkeit im Übergangsmedium bildet:

$$n = \frac{c_{Ausgangsmedium}}{c_{Übergangsmedium}}$$

Dies bezeichnet man als das snelliussche Brechungsgesetz.[1]

Verschiedene Materialien haben unterschiedliche Brechungsindizes: Wasser hat beispielsweise einen Brechungsindex von 1.33 und Vakuum einen von ungefähr 1 (keine Änderung der Richtung).

$$\vec{R}_{Refractioned} = \frac{n_1}{n_2} * \vec{V}_{incoming} - \vec{N} * \left(\frac{n_1}{n_2} + \sqrt{1 - \left(\left(\frac{n_1}{n_2} \right)^2 * \left(1 - \left(\vec{V}_{incoming} \cdot \vec{N} \right)^2 \right) \right)} \right)$$ [2]

$\vec{N}$ und $\vec{V}_{incoming}$ müssen normalisiert sein

Anschließend wird ähnlich wie bei der Reflexion ein weiterer Strahl verfolgt. Dieser trifft auf einen anderen Kollisionspunkt (Refraktionkollisionspunkt). Die Farbe an diesem Punkt wird schließlich mit der resultierenden Farbe verrechnet.

Lambert-Beersches Gesetz ☑

In der Wirklichkeit sind Flüssigkeiten nicht vollständig klar, vielmehr werden Objekte in einer weiten Entfernung zunehmend unschärfer und schwächer erscheinen, da das Licht in gewisser Weise in Abhängigkeit von der Entfernung absorbiert wird. Mit der Entfernung steigt exponentiell die Absorptionsintensität.

$$I = I_0 * e^{-\alpha * c * d}$$ [3]

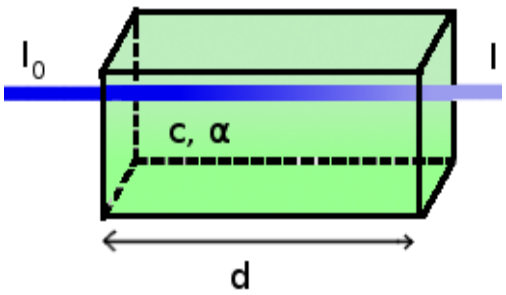


Abbildung 18: Auswirkungen des Lambert-Beersches Gesetz

α und c sind materialabhängige Konstanten; α ist der spezifische Extinktionskoeffizient, während c die Konzentration in $\frac{mol}{l}$ darstellt.

1 Meyers Physiklexikon, S.134, Stichwort „Brechung“, 1973
2 http://www.devmaster.net/articles/raytracing_series/Reflections%20and%20Refractions%20in%20Raytracing.pdf, S. 7
3 http://www2.fh-swf.de/fbtbw/downloads/bangert/06-Lambert-Beer_sches_Gesetz.pdf, S. 64

I_0 entspricht der anfänglichen Lichtintensität (in diesem Fall 1 $\frac{W}{m^2}$). I ist die resultierende Lichtintensität. Diese wird mit der Farbe, die durch die Lichtbrechung errechnet wurde, multipliziert. d ist die Distanz/Schichtdicke. In diesem Fall entspricht d dem Betrag der Differenz zwischen dem Kollisionspunkt und dem Refraktionkollisionspunkt; d nimmt also zu, je weiter weg der Refraktionkollisionspunkt vom Kollisionspunkt weg ist. Somit nimmt die resultierende Lichtintensität ab.

Texturing ☑

Als „Texturing“ bezeichnet man das Umwickeln eines Bildes um ein Primitiv. Dieses Prinzip wird bei Computerspielen sehr häufig eingesetzt. Auf Flächen (z. B. virtuelle Wände) wiederholen sich die Texturen nach einiger Entfernung (siehe Abbildung 17).

Bisher hatten Primitive einfache Farben, was recht schnell beim Benutzer ein Gefühl der Langeweile auslösen konnte. Eine virtuelle Grasfläche sollte nicht einfach nur grün sein, sondern an verschiedenen Stellen verschiedene Grüntöne zeigen.

Ausgangspunkt ist eine zweidimensionale Graphik, die letztlich über ein dreidimensionales Objekt gelegt werden muss. Je nach Primitivform variiert der Komplexitätsgrad dieser Aufgabe. Als einfachstes Beispiel dient eine Ebene: Da es sich bei meiner Implementierung des Primitivs „Ebene“ um eine mathematische, also unendlich große Ebene handelt, muss sich die Textur, die eine endliche Größe hat, in gewissen Abständen wiederholen.

Das Primitiv muss in der Lage sein, aus einem Kollisionspunkt einen Bildpunkt mit (x, y)-Koordinaten einer Textur anzugeben. Bei der Ebene könnte man die r- und s-Werte aus der Kollisionsberechnung als Grundlage für die x und y-Werte nehmen:

$$x_{Texture} = (f * r_{Plane}) \bmod Texture_{Width}$$

$$y_{Texture} = (f * s_{Plane}) \bmod Texture_{Height}$$

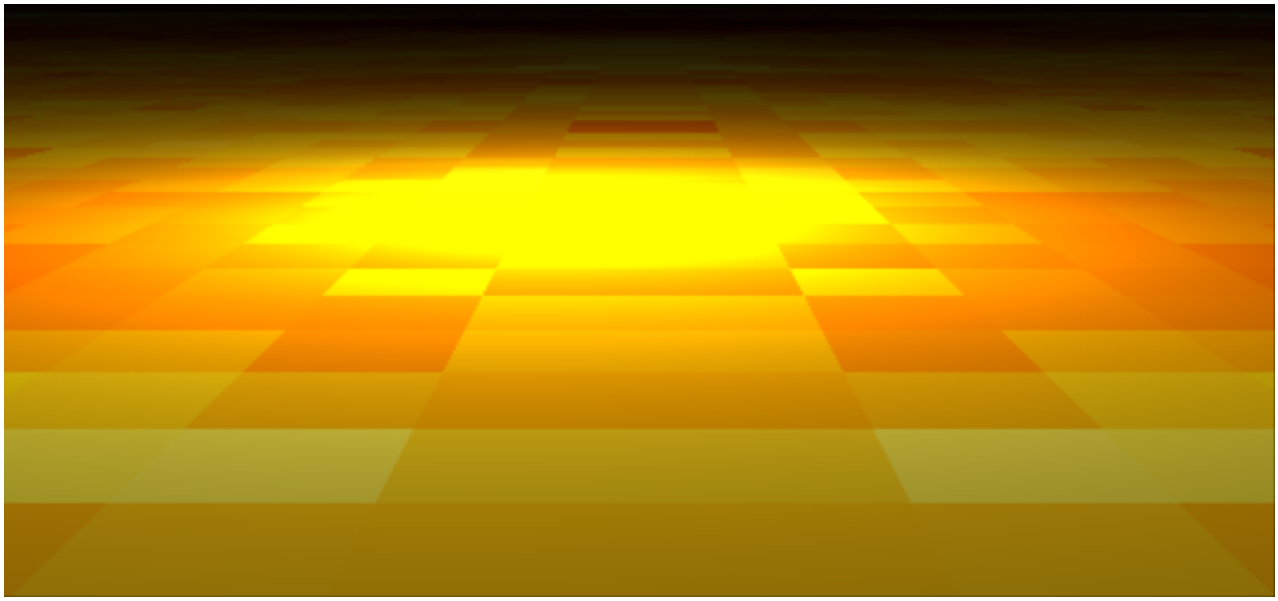
f ist hierbei die Frequenz der Wiederholung. Je größer die Frequenz ist, desto öfter wiederholt sich das Bild.

Die x-Koordinate der Textur beträgt 0, wenn $\frac{r_{Plane}}{Texture_{Width}}$ sich ohne Rest teilen lässt.

Ansonsten entspricht die x-Koordinate immer den Rest, der zwischen 0 und $Texture_{Width}$ liegt.

Die Koordinaten für die Textur sind natürliche Zahlen, während es sich bei den r und s-Werte meist um rationale Zahlen handelt. Bei der Modulo-Operationen werden hingegen zwei natürliche Zahlen benötigt, weswegen an dieser Stelle gerundet wird.

Das führt dazu, dass zwei unterschiedliche Stellen auf der Ebene, die sehr nah beieinander liegen, den gleichen Pixel auf der Textur erhalten:



Das führt dazu, dass die Textur „verpixelt“ erscheint.

Das Problem lässt sich lösen, indem zwischen den Pixeln interpoliert wird. Es werden mehrere Pixel aus der Textur übernommen und unterschiedlich gewichtet: Beträgt der r-Wert beispielsweise 5,75, so ergibt sich bei einer Frequenz von 1 und einer Texturengröße von 200 x 200 Pixel nach bisheriger Form die X-Koordinate

$$x1_{Texture} = (1 * 5{,}75) \, mod \, 200 = 5 \, mod \, 200 = 5$$

Nun möchte man allerdings nicht den Pixel an der Stelle von $r = 5$, sondern von $r = 5{,}75$. Den Wert von $r = 6$ lässt sich ohne Probleme berechnen:

$$x2_{Texture} = (1 * 5{,}75) \, mod \, 200 = 6 \, mod \, 200 = 6$$

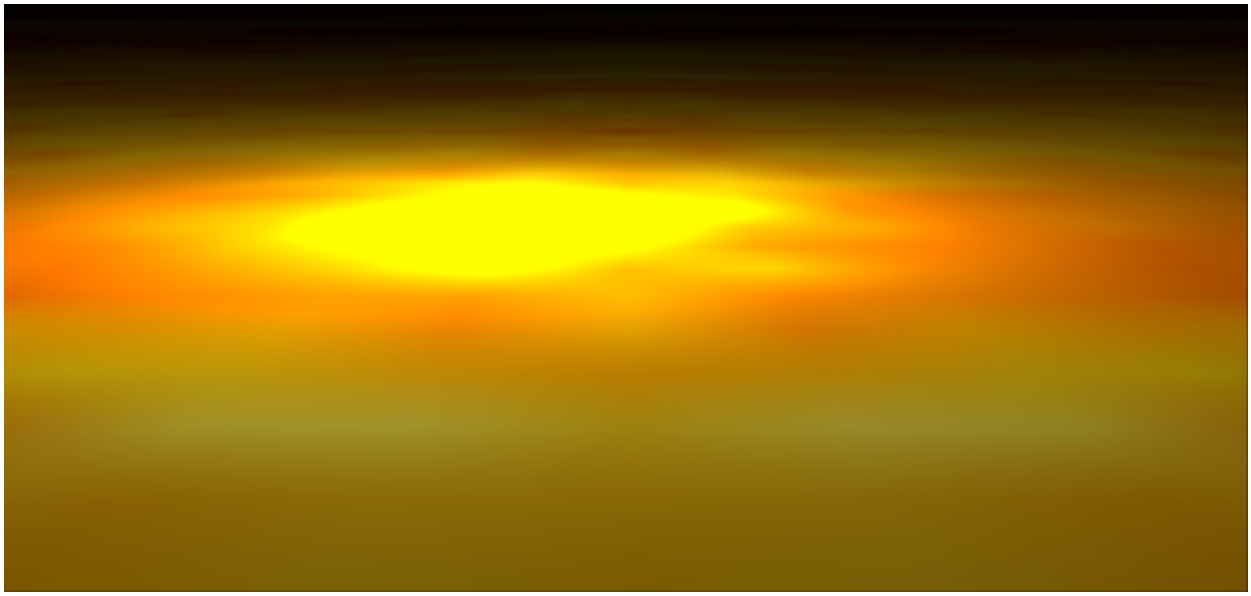
Die Farbwerte für $x1_{Texture}$ und $x2_{Texture}$ werden (nach dazugehöriger Berechnung der Y-Koordinate) ermittelt und anschließend wird zwischen ihnen linear interpoliert.

X-Wert	**Farbwerte (R, G, B)**
5	**128**, **128**, **128**
6	*196*, *196*, *0*
5,75	**128** * 0.25 + *196* * 0.75, **128** * 0.25 + *196* * 0.75, **128** * 0.25 + *0* * 0.75 --- 179, 179, 32

Tabelle 1: Beispiel für eine lineare Interpolation

Bei diesem Beispiel werden die Farben unterschiedlich stark gewichtet: 5,75 befindet sich näher an 6 als an 5, dementsprechend sind die Farbkomponenten von 6 stärker gewichtet (75%) als die von 5 (25%).

Andere Interpretationsmodelle sind beispielsweise die Bilinear- oder Trilinear-Interpolation. Beide werden häufig in Spielen eingesetzt.



Bump Mapping

Bisherige Oberflächen sind alle matt und glatt. In der Realität gibt es allerdings selten solche Flächen. Raue Oberflächen sind interessanter als glatte und verändern Bilder wesentlich. Raue Oberflächen (inklusive Wellen) lassen sich mithilfe von Bump Mapping erzeugen.

Das Verfahren von Blinn (die Illusion) ☑
Das Verfahren des Bump Mappings wurde erstmals 1978 von Blinn verwendet.[1] Hierbei werden die Tiefeninformationen in Form von Farben in einem Bild abgespeichert. Diese wird Heightmap (Höhenkarte) genannt. Diese wird ähnlich wie beim Texturing um das Primitiv herum gelegt, anders als beim Texturing bleibt die Heightmap allerdings unsichtbar.

Beim Bump Mapping nach Blinn wird die Normale an den Stellen, die tiefer erscheinen soll, verändert. Das „Bump Mapping“-Verfahren erhöht auch in den Realitätsgrad, da in der Realität keine perfekten Objekte (beispielsweise mathematisch glatte Kugeln) existieren.

Displacement Mapping ☐

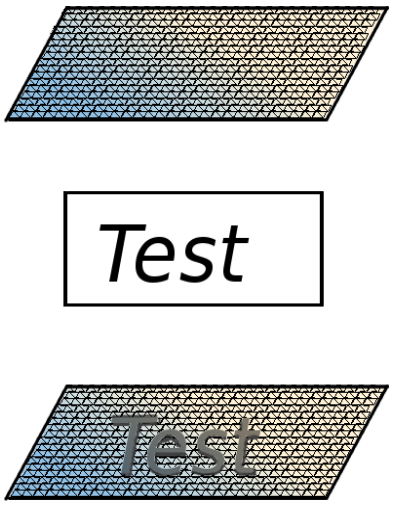


Abbildung 20: Displacement Mapping

1 http://www.siggraph.org/conferences/reports/s2001/interview/blinn.html

Der Komplexitätsgrad der Implementierung des Displacement Mappings liegt deutlich höher als bei Blinns Verfahren. Beim Displacement Mapping wird tatsächlich die Objektgeometrie verändert.Das obere Bilddrittel stellt eine Ebene dar, aufgeteilt in sehr viele kleine Dreiecke. Das Bild darunter ist die zu verwendende Heightmap. Die Koordinaten der betroffenen Dreiecke wird tatsächlich verändert, sodass die Ebene nun mathematisch gesehen keine mehr ist. Sie besteht aus vielen kleinen Dreiecken. Der Vorteil besteht darin, dass das Objekt Schatten auf sich selbst werfen kann. Aus einem anderen Blickwinkel werden die unterschiedlichen Tiefen noch deutlicher sichtbar.

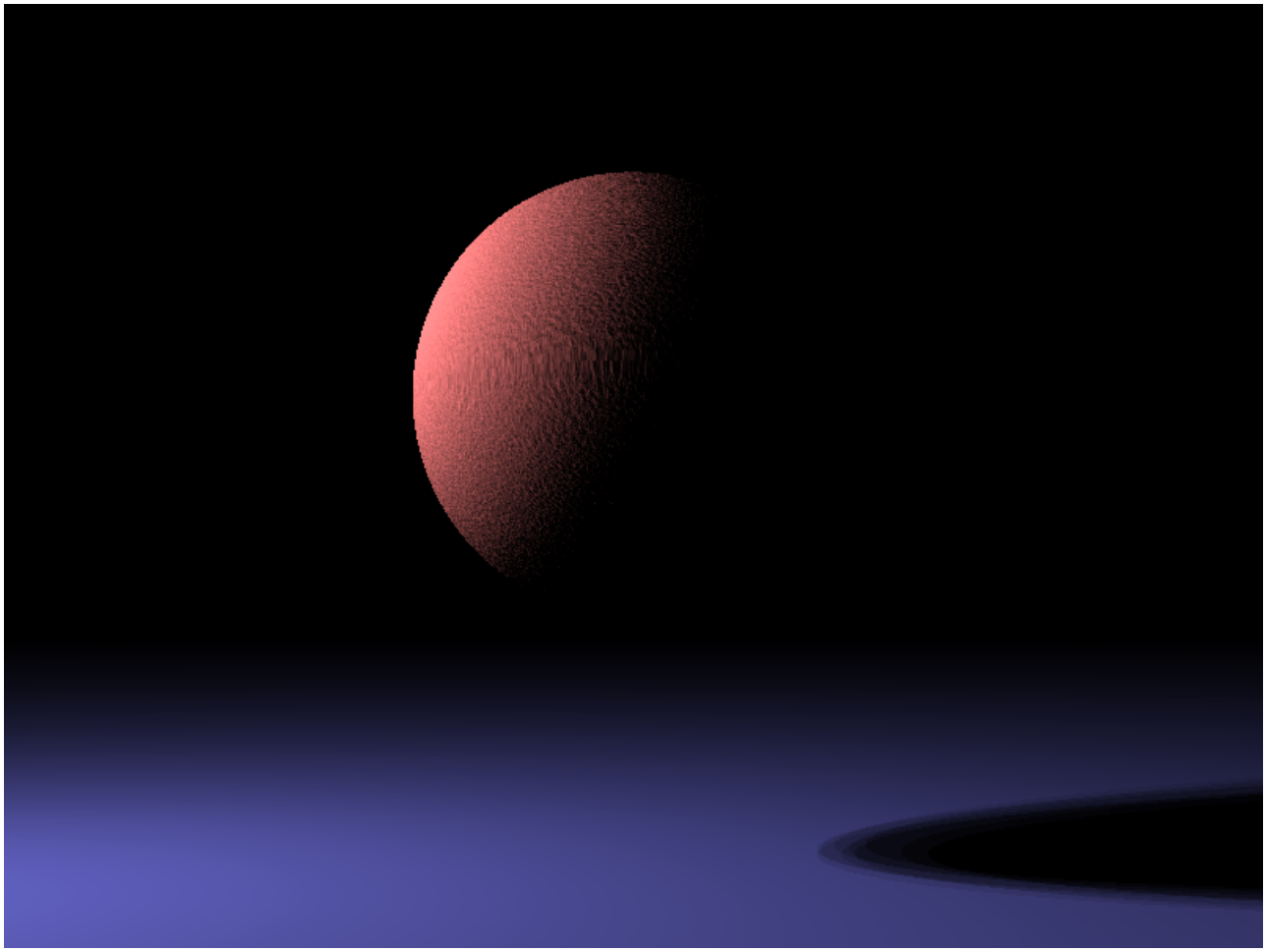


Abbildung 21: Eine Kugel mit einer Bumpmap – Die Kugel scheint eine raue Oberfläche zu besitzen. In Wirklichkeit ist sie rund – die Wirkung kommt durch Licht/Schatteneffekte zustande.

Bump Mapping (nach Blinn)		**Displacement Mapping**	
Pro	**Contra**	**Pro**	**Contra**
- Einfache Implementierung - Vernachlässigbarer Rechenaufwand	- Nicht realistisch bei tieferen Löchern - Keine Selbstverdeckung - Schatten unrealistisch	- Sehr realistisch - Selbstverdeckung - Schatten realistischer	- Schwere Implementierung - Großer Rechenaufwand (Geometrie des Objekts muss geändert werden)

Tabelle 2: Vergleich der Bump Mapping-Verfahren

Beide Verfahren haben Vor- und Nachteile. Es gäbe auch die Möglichkeit eines „Hybrid-Bump-Mappers“: Für kleinere Höhenunterschiede ließe sich Bump Mapping nach Blinn realisieren, für größere Höhenunterschiede kann immer das Displacement Mapping-Verfahren verwendet werden.

Photon Mapping ☑

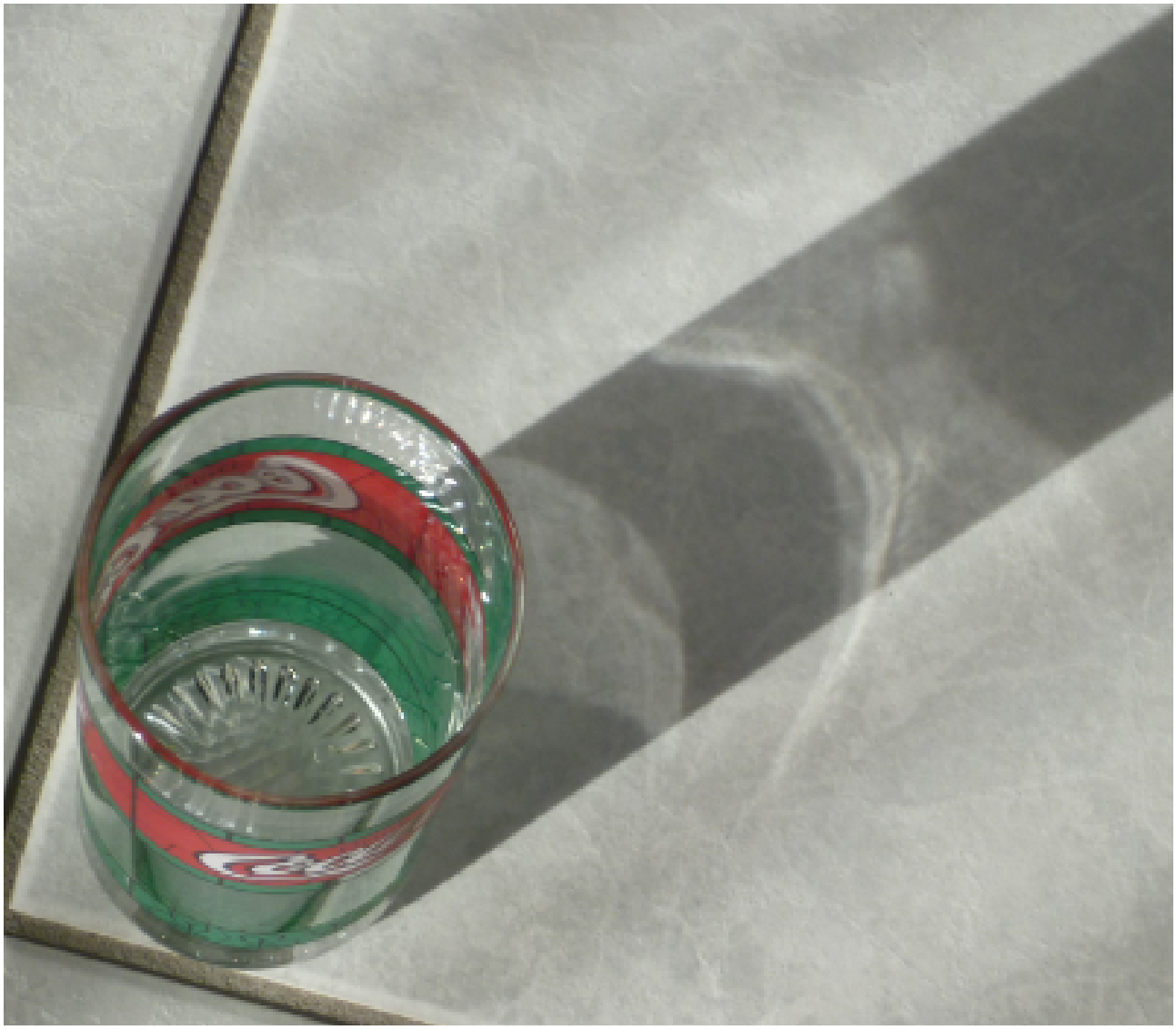


Abbildung 22: Ein Glas mit Kaustik – ein Photo

Auf der obigen Abbildung lässt sich die Kaustik im Schatten erkennen. Kaustiken entstehen durch eine unterschiedliche Brechung von Strahlen in Strahlenbündeln, die sich in einzelnen Bereiche konzentrieren und somit verstärken.

Mit der bisher beschriebenen Techniken waren solche Bilder nicht möglich. Um solche Effekte simulieren zu können, kann man das Photon Mapping-Verfahren implementieren.

Dort werden vor dem eigentlichen Rendervorgang viele virtuelle Photonen von den Lichtquellen in zufällige Richtungen abgeschossen, die an anderen Objekten reflektiert, gebrochen, gestreut oder absorbiert werden. Die Photonen sind physikalisch gesehen keine „echten“ Photonen, da sie keine Wellenlänge haben und auch Farben repräsentieren können, die keine Spektralfarben sind.

Die Entscheidung, ob ein Photon an einer Oberfläche reflektiert, gebrochen oder absorbiert wird sind, kann mithilfe der „Russischen Roulette“-Technik getroffen werden.[1] Die Materialeigenschaften werden in Form von Wahrscheinlichkeiten angegeben.

1 http://www.iwr.uni-heidelberg.de/groups/ngg/CG2008/Txt/Kapitel4.pdf

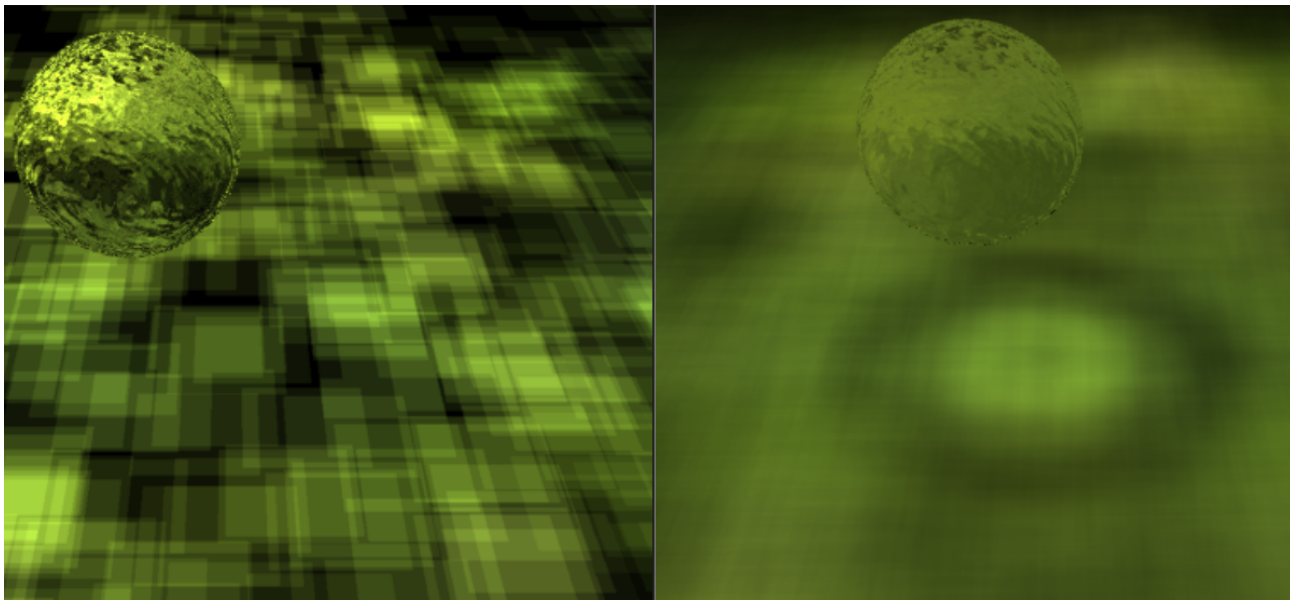
$$p_{diffusion} + p_{reflection} + p_{refraction} + p_{absorption} = 1$$

Wenn beispielsweise $p_{reflection}$ = 1 ist, ist die Oberfläche dieses Materials voll reflektierend.

Nun wird eine pseudozufällige Zahl $Z \in [0,1]$ errechnet.

Wenn	Dann
$Z \in [0, p_{diffusion}]$	Das auftreffende Photon wird in einer so genannten Photon Map („Photonenkarte“) eingetragen. Dort wird die Position und die Intensität/Farbe des Photons gespeichert.
$Z \in [p_{diffusion}, p_{diffusion} + p_{reflection}]$	Das Photon wird reflektiert.
$Z \in [p_{diffusion} + p_{reflection}, p_{diffusion} + p_{reflection} + p_{refraction}]$	Das Photon wird gemäß des Brechungsindex gebrochen.
$Z \in [p_{diffusion} + p_{reflection} + p_{refraction}, 1]$	Das Photon wird absorbiert.

Beim Rendering wird auf die Photon Map zurückgegriffen, um herauszufinden, wie stark eine bestimmte Position mit Licht bestrahlt wird.



Durch die Lichtbrechung können die Wege einiger Photonen so verändert werden, sodass sie mit vielen anderen an einer Stelle auftreffen (siehe Abbildung). Diesen Effekt nennt man Kaustik.

Photon Mapping ist in der Lage, die so genannte globale Beleuchtung zu simulieren. In der Realität werden Photonen von einer diffusen Fläche gestreut. Die gestreuten Photonen können wiederum andere diffuse Flächen erreichen. Dadurch erscheint die gesamte Szene ein wenig ausgeleuchteter. Beim reinen Raytracing hingegen leuchten sich diffuse Flächen gegenseitig nicht an, da nach dem Auftreffen eines Strahls auf einer diffusen Fläche kein weiterer Strahl berechnet wird.

Dispersion □

Als Dispersion wird die Abhängigkeit der Brechzahl von der Wellenlänge des Lichts bezeichnet. Es gibt verschiedene Algorithmen, um die Brechzahl für eine bestimmte Wellenlänge zu ermitteln. Für die Simulation dieses Effekts wird Photon Mapping benötigt.

Cauchy-Gleichung

1830 hat Augustin Lous Cauchy einen empirischen Zusammenhang zwischen der Brechzahl und der Wellenlänge ermittelt.

[1]

$$n(\lambda) = \sum_{j=0}^{i} \frac{B_j}{\lambda^{2*j}}$$

Die Formel ist nur in einem engen Spektralbereich sehr genau. Die verschiedenen B's mit Indizes von 0 bis i werden als Cauchy-Parameter bezeichnet und sind materialabhängig. Für verschiedene Glassorten haben sie unterschiedliche Werte. Je mehr Koeffizienten experimentell ermittelt worden sind, desto exakter wird das Ergebnis.

Sellmeier-Gleichung

Bei der Sellmeier-Gleichung handelt es sich um eine Erweiterung des Modells von Cauchy, die auch in größeren Spektralbereichen noch vergleichsweise genau ist. Auch sie benötigt materialabhängige Koeffizienten. Diese werden wie bei der Cauchy-Gleichung ebenfalls experimentell ermittelt.

[2]

$$n(\lambda) = 1 + \sum_{j=1}^{i} \frac{B_j * \lambda^2}{\lambda^2 - C_j}$$

Es gibt zwei Typen von Koeffizienten: Die dimensionslosen B-Koeffizienten und die C-Koeffizienten, dessen Einheit in m^2 angegeben wird.

Aufteilung in die Spektralfarben

Damit das Licht je nach Wellenlänge unterschiedlich gebrochen werden kann, müssen die Wellenlängen gefunden werden, die das Licht repräsentieren.

Gemäß des Farbspektrums ergibt eine Addition aller Wellenlängen des sichtbaren Lichts näherungsweise das weiße Licht wieder. Da in einem Programm nicht unendlich viele verschiedene Wellenlängen berechnet werden können, muss hier Sampling (eine Stichprobe) durchgeführt werden.

Dies geschieht ähnlich wie bei den weichen Schatten. Die herausgesuchten Stichproben ergeben näherungsweise das weiße Licht.

Nun wird für jede dieser Wellenlängen der dazugehörige Brechungsindex berechnet. Die Photonen werden gemäß der Brechungsindizes gebrochen und deren Weg weiterverfolgt. Anschließend werden die Photonen in der *Photon Map* gespeichert.

---

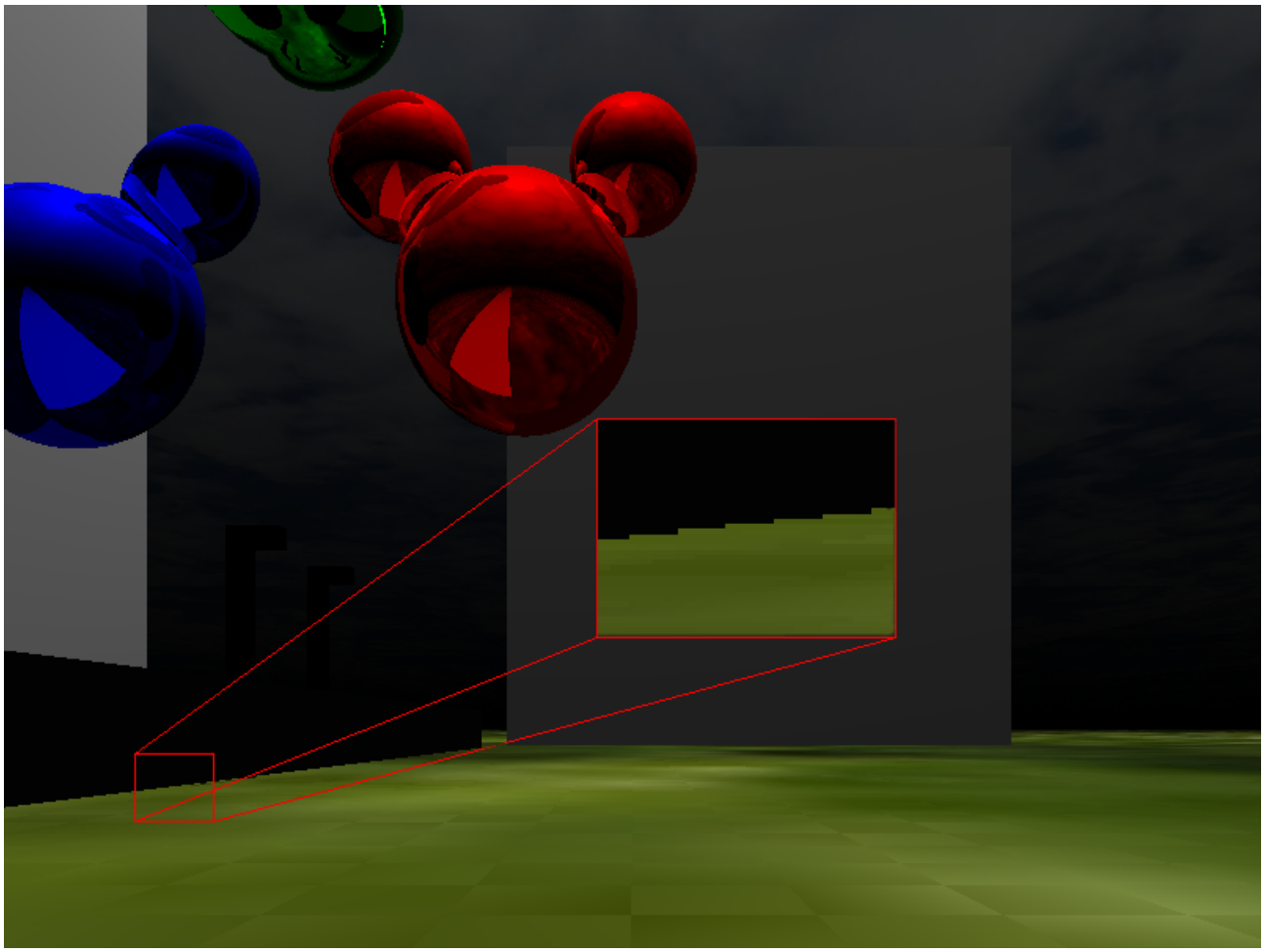
1 Kühlke, Dieter, Optik: Grundlagen und Anwendungen, 2007, S. 52

2 http://www.uni-koeln.de/math-nat-fak/kristall/forschung/kristallphysik/optik/node6.html

Antialiasing/Supersampling ☑

Die Bildqualität lässt sich noch weiter erhöhen, indem man nicht für jeden Pixel einen Strahl durchrechnet, sondern fünf. Den einen, der genau auf den Pixel passt und jeweils vier Strahlen, die durch einen Pixel zwischen den umliegenden gehen:

Das Resultat zeigt nicht mehr so starke „Treppeneffekte“ bei ungeraden Linien und das Bild wirkt weicher sowie weniger kantig:



Auf dieser Abbildung sind deutlich die Treppeneffekte zu erkennen.

In der Vergrößerung sieht man, wie die Ebene und das schwarze Objekt (eine Straße, die von diesem Winkel nicht sichtbar ist) abrupt ineinander übergehen.

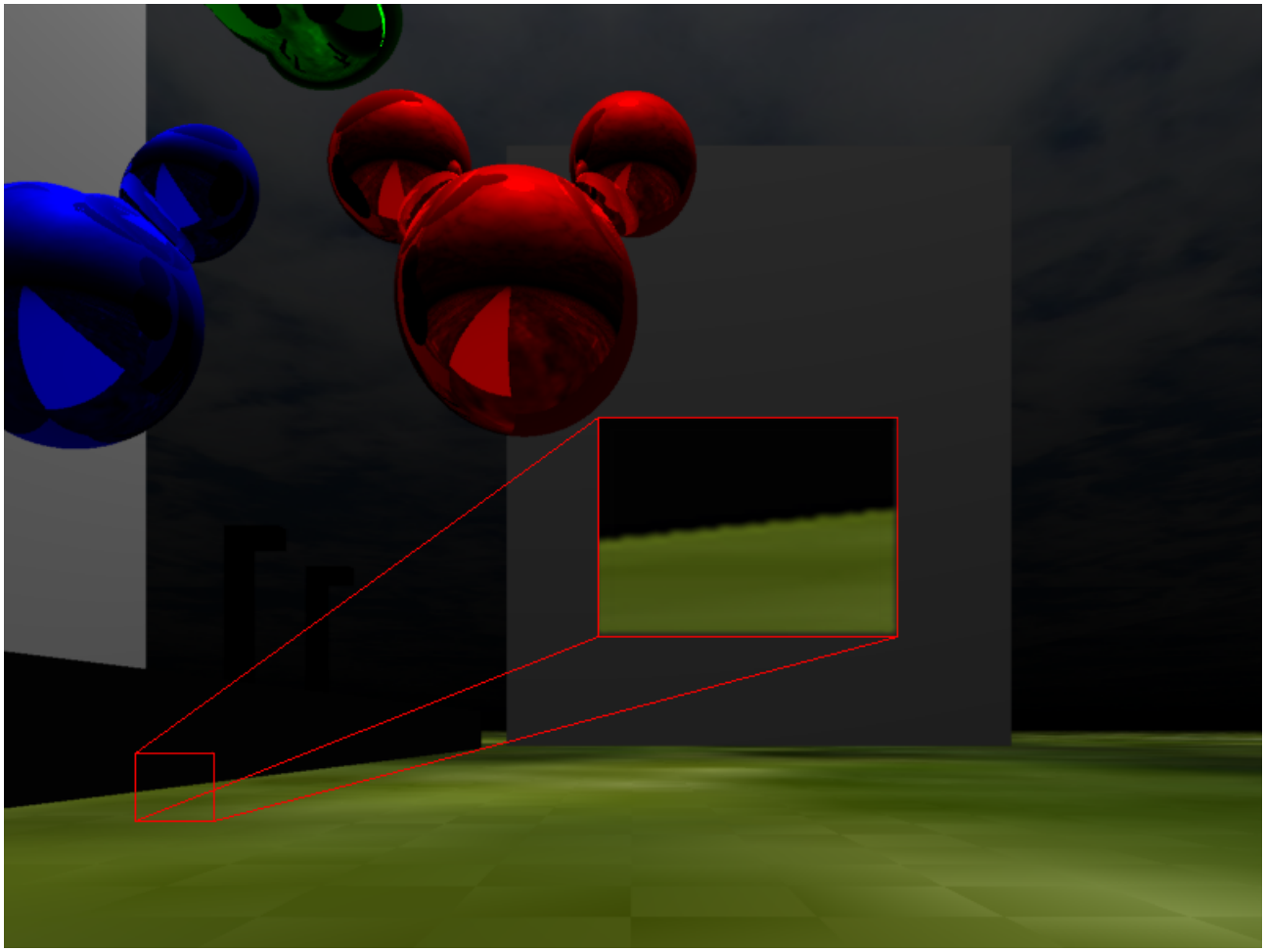


Abbildung 25: Mit Antialiasing

Hier gehen die beiden Objekte auch in der Vergrößerung quasi nahtlos ineinander über. Das Bild wirkt dadurch „weicher“.

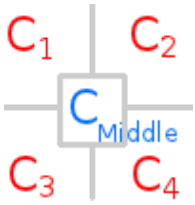


Abbildung 26: C_1, C_2, C_3, C_4 *stellen die umliegenden Farben dar.* C_{Middle} *ist die Farbe des Pixels in der Mitte.*

$$C_{ResultingColor} = \frac{1}{6} * (C_1 + C_2 + C_3 + C_4) + \frac{1}{3} * C_{Middle}$$

Geschwindigkeitsoptimierend

Bounding Boxes ☑

Das Problem beim bisherigen Verfahren ist, dass jeder Strahl mit jedem Primitiv überprüft werden muss, selbst wenn dieses überhaupt nicht getroffen werden kann. Mit jedem weiteren Primitiv muss für jede Strahlenberechnung immer ein Primitiv mehr überprüft werden, weshalb der Rechenaufwand linear steigt. Es gibt glücklicherweise Verfahren, die es ermöglichen, Unterteilungen vorzunehmen, weswegen weniger Kollisionstest erforderlich sind.

Bounding Boxes ist eine der einfachsten Möglichkeiten, um die Zeit, die für die Berechnungen benötigt wird, zu verringern. Man unterteilt die Szenerie in relativ große so genannte „Axis Aligned Bounding Boxes“ (AABB) ein, also umgebende Boxen, die sich an den Koordinatenachsen orientieren.

Der Einfachheit halber sind die Bounding Boxes genauso groß und liegen direkt aneinander. Anschließend wird für jede Box geprüft, ob und welche Primitive sich in ihr befinden:

Name	Position	Größe	Enthaltene Primitive
Box1	(0, 0, 0)	(1, 1, 1)	Kugel
Box2	(0, 0, 1)	(1, 1, 1)	Kugel, Quader
Box3	(0, 1, 0)	(1, 1, 1)	-

Tabelle 3: Beispiel: Bounding Boxes

Wie der Tabelle zu entnehmen ist, ist jede Box ein (unsichtbarer) Würfel mit der Kantenlänge 1. Jeder dieser Boxen liegt direkt an einer anderen an und enthält verschiedene Primitive.

Die Bounding Boxes können nun mehrere dieser Primitive umschließen. Trifft der Strahl schon die Bounding Box nicht, brauchen die Primitive in ihr gar nicht geprüft zu werden, da sie ebenfalls nicht getroffen werden können.

Das ist vergleichbar mit folgender Situation: Man stellt beispielsweise ein Objekt in eine Box und schießt auf diesen. Wenn die Box nicht getroffen wird, braucht man nicht die Box zu öffnen, um zu schauen, ob das Objekt im Inneren getroffen wurde.

Wenn der Strahl allerdings beispielsweise *Box2* trifft, muss überprüft werden, ob die Kugel oder der Quader getroffen wurde.

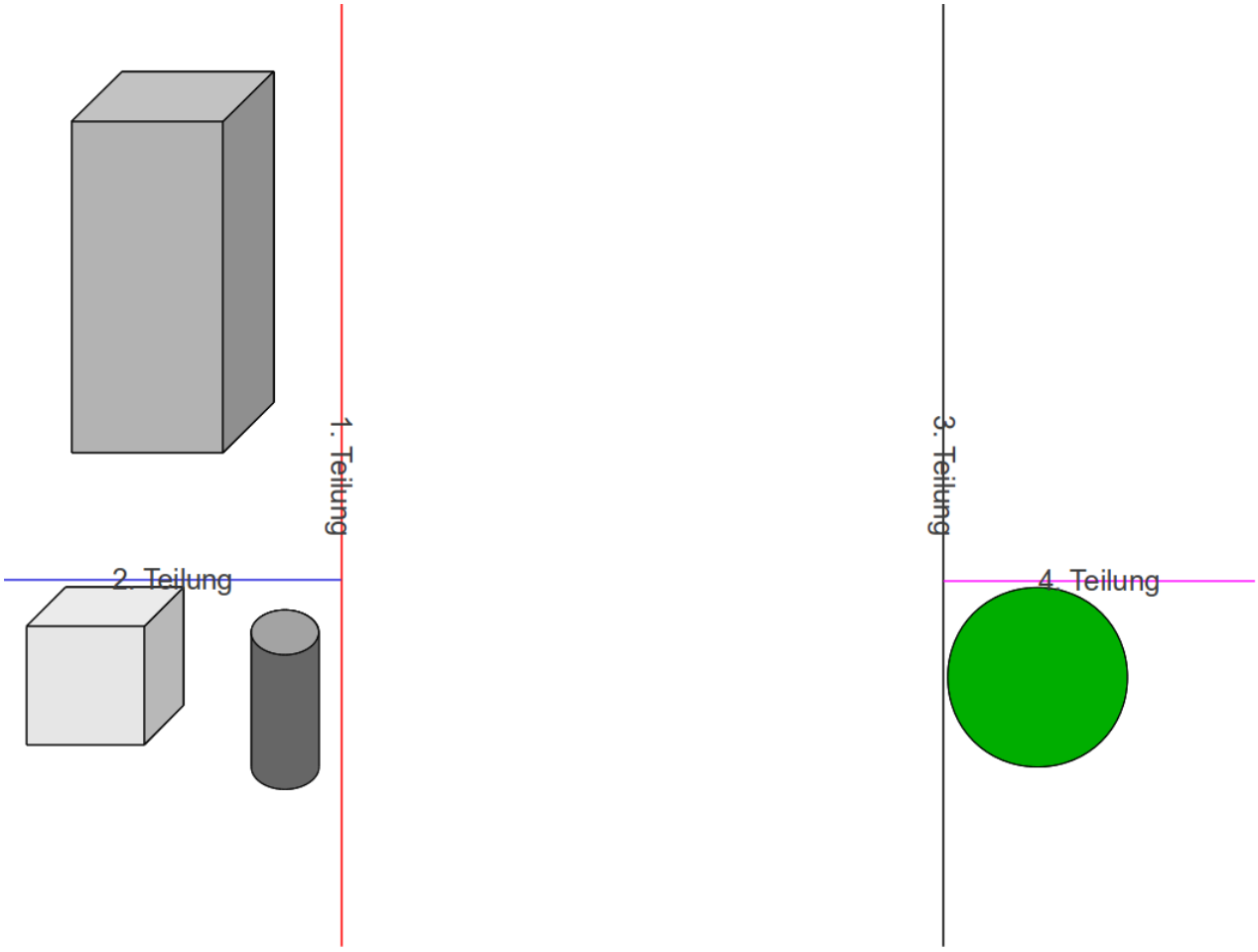
Das Verfahren funktioniert noch besser, wenn die relativ großen Bereiche in noch kleinere Bounding Boxes eingeteilt werden. Bounding Boxes können also auch Bounding Boxes enthalten und sind somit rekursiv. „Leere“ Bounding Boxes (wie *Box3*), die keinerlei Objekte (Primitive und Bounding Boxes) enthalten, können entfernt werden – sie werden nicht benötigt.

Je nach der Einteilung der Bounding Boxes variiert die Effektivität des Verfahrens.

K-dimensionaler Baum ☑

Ein effektiveres Verfahren zur Optimierung stellt der k-dimensionale Baum, der so genannten kd-tree, dar. Auch dies ist ein Verfahren zur Einteilung von Primitiven in eine äußere Hülle. Es ist in einer gewissen Hinsicht eine Erweiterung des Bounding Boxes-Verfahrens.

Es handelt sich um einen binären Baum, jeder Knoten hat also maximal zwei Unterknoten. Am Anfang wird eine große Bounding Box (wie schon bei den Axis Aligned Bounding Boxes) erstellt, die alle Primitive einer Szene enthält. Diese Bounding Box wird in zwei weitere eingeteilt, die dessen Kindknoten darstellen. Auch die beiden Kindknoten werden jeweils erneut geteilt. Dieser Prozess setzt sich bis zu einer gewissen Rekursionstiefe fort:



Von zwei Kindknoten eines Knotens kann der eine als linker und der andere als ein rechter Knoten bezeichnet werden. Welcher der beiden als links und welcher als rechts bezeichnet wird, ist dabei egal. Es geht dabei lediglich um eine Unterscheidung.

Dennoch gibt es einen entscheidenden Unterschied zum reinen „Bounding Boxes“-Verfahren: Die Feststellung der Trennachse. Es wird versucht, möglichst große Bereiche freizulassen und damit Bounding Boxes zu sparen.

Es wird eine Kostenfunktion gebraucht, die eine Schätzung abgibt, wie lange eine durchschnittliche Überprüfung mit allen nötigen Rechenoperationen dauern würde. Die Kostenfunktion kann folgendermaßen aussehen:

$$C_{Result} = 2 * C_{\text{AABB intersection}} + V_{\text{Left}} * n_{\text{L primitives costs}} + V_{\text{Right}} * n_{\text{R primitives costs}}$$

Die Kosten könnte beispielsweise die durchschnittliche benötigte Rechenzeit in Millisekunden sein, die mithilfe eines Benchmarks ermittelt werden könnte. Unter Benchmarking versteht man die Messung der Dauer, in die der Rechenaufwand bewältigt werden kann. Dazu führt man für jeden Primitivtyp beispielsweise 100.000 Kollisionstests durch und misst die Zeit, die dieser Vorgang in Anspruch nimmt.

Das beschriebene Verfahren ist heuristisch, das heißt, dass ein guter Schätzwert gegenüber dem reellen Zeitaufwand berechnet wird. Für jedes Primitivtyp (Ebene, Dreieck, Kugel, etc.) oder eben auch eine AABB ergibt sich meist ein unterschiedlicher Kostenwert, da der Rechenaufwand unterschiedlich ist. $C_{\text{AABB intersection}}$ entspricht den Kosten für einen Test mit einem Strahl und einer Bounding Box. Da die beiden zukünftigen Kindknoten jeweils über eine umgebende Bounding Box verfügen, fallen die Kosten zwei Mal an: Einmal für jeden der beiden Kindknoten.

V_{Left} entspricht dem Volumen der linken Bounding Box und $n_{\text{L primitive costs}}$ der Summe der einzelnen Primitivkosten in der linken Box; da für jeden Strahl, der die Bounding Box berührt, alle Primitive in ihr getestet werden müssen, steigt der Aufwand bei größeren Bounding Boxen, weil größere Boxen häufiger getroffen werden als kleinere. Analog wird der Aufwand für den rechten Kindknoten berechnet und dazuaddiert.

Die Summe $n_{\text{L primitive costs}}$ lässt sich folgendermaßen definieren:

$$n_{\text{L primitive costs}} = \sum_{i=1}^{c_L} n_i$$

c_L steht für die Anzahl der Primitive im linken Kindknoten (c = count, engl. Anzahl).

n_i bezeichnet die Kosten eines Primitives in Abhängigkeit vom Index i, der durch alle Primitive des linken Kindknotens iteriert. Analog lässt sich auch die mathematische Definition für den rechten Kindknoten finden.

Es wurde nun eine Möglichkeit gefunden, abzuschätzen, wie hoch der Rechenaufwand ist, wenn die Bounding Box eines Knotens an einer bestimmten Stelle in die beiden Kindknoten aufgeteilt wurde. Nun gilt es, verschiedene mögliche Teilungspositionen gegeneinander abzuwägen. Im Klartext heißt das: Man probiert verschiedene Teilungspositionen aus, berechnet ihre Kosten und wählt am Ende die Möglichkeit mit den geringsten Kosten, da diese den geringsten Rechenaufwand bedeutet. Falls alle Unterteilungen höhere Kosten verursachen als keine, wird die Unterteilung nicht weiter durchgeführt. Es kommt somit zu einem Ende der Rekursion.

Es werden nicht alle möglichen Positionen wahllos durchprobiert. Da möglichst große Bereiche möglichst leer sein sollen, sollte dafür gesorgt werden, dass sich dasselbe Primitiv nicht gleichzeitig in beiden zukünftigen Kindknoten befindet; sonst müsste es doppelt geprüft werden, wenn der Strahl beide Bounding Boxes berührt.

Die sinnvollen Trennpunkte befinden sich genau an den Rändern der Primitive, da sich das Primitiv bei der Trennung am Rand nur in einer Bounding Box gleichzeitig befinden kann, da es *nicht* in der Mitte getrennt wurde.

Zusammenfassend könnte ein Algorithmus für die Erstellung eines KdTrees folgendermaßen aussehen:

1. Führe einen Benchmark durch, der zeigt, wie hoch der Rechenaufwand für einzelne Primitivtypen (Dreieck, Würfel, Kugel, etc.) und für AABB ist.
2. Finde eine AABB, die groß genug ist, um alle Primitive aufzunehmen.
3. Berechne den Kostenaufwand, falls nicht geteilt wird. Dies soll die Ausgangssituation sein und ist die günstigste Situation (bisher wurde keine andere Lösung gefunden).
4. Für jede Bounding Box (genannt *Elternknoten*):
   I. Suche alle Primitive, die sich innerhalb vom *Elternknoten* befinden.
   II. Finde die Ränder der Primitive (anhand der Bounding Boxes der Primitive) und teile die Box provisorisch ein.
      1. Berechne die „Kosten“ (Rechenaufwand) für diese provisorische Aufteilung.
      2. Falls die Kosten für diese Aufteilung geringer ist als die bisher beste Lösung, ist nun diese Aufteilung die bisher beste. Diese sollte gesichert werden.
   III. Nehme die optimale Trennung vor (falls eine Trennung notwendig ist) und springe rekursiv jeweils für den linken und den rechten Kindknoten zu Punkt 3 (werden beide jeweils neue *Elternknoten*).

Bemerkungen über Erweiterungen

Wenn man eine Erweiterung für einen Raytracer einbaut, werden (durch die rekursive Natur des Verfahrens) die Erweiterungen auch für alle anderen Erweiterungen übernommen:

Angenommen, man hat einen Raytracer, der sowohl Reflexionen als auch Schatten unterstützt. Wenn sich nun eine reflektierende Kugel in der Szene befindet, reflektiert die Kugel automatisch auch ihren Schatten, da bei der Berechnung der Farbe des reflektierenden Strahls die Schattenberechnung miteinbezogen wird.

Meist lassen sich Erweiterungen in drei Kategorien unterteilen:

1. Erweiterungen, die die Qualität verbessern.
2. Erweiterungen, die den Aufwand minimieren und somit eine schnellere Berechnung ermöglichen.
3. Weitere Primitivtypen.

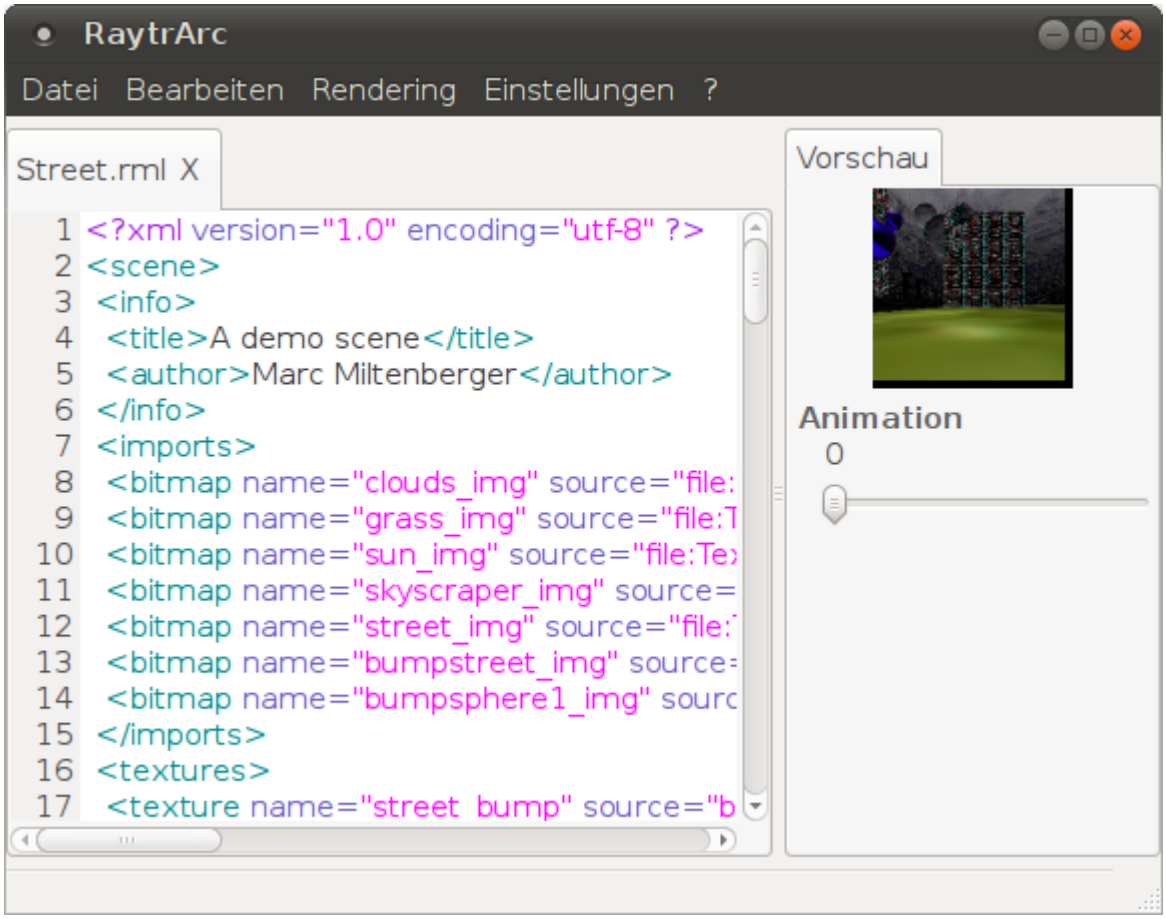
Der Nachteil von Erweiterungen, die den Realitätsgrad erhöhen, ist, dass der Aufwand des Prozesses zur Berechnung eines Strahls und somit auch des gesamten Bildes durch jede Erweiterungen signifikant ansteigt. Die langsamere Geschwindigkeit können geschwindigkeitsoptimierende Verfahren wie beispielsweise Bounding Boxes oder kd-Trees weitestgehend kompensieren.

Schließlich können recht einfach weitere Primitive implementiert werden: Ein Primitiv muss nur berechnen können, wann bei ihm ein Strahl wo kollidiert und welche Normale er an der Stelle hat. Wenn Texturing verwendet werden soll, muss noch eine Zuordnung existieren, die den Kollisionspunkten auf dem Primitiv Pixelpositionen auf der Textur zuordnet (siehe Kapitel *Texturing*).

Die Implementierung

Die Umsetzung des vorigen Theorieteils erfolgte in C# („C Sharp“), einer .NET[1]-Programmiersprache. Der Kern, der sich um die Grafik kümmert, verfügt (ohne Leerzeilen) über knapp 7.000 Codezeilen. Die graphische Benutzeroberfläche benötigt 500 Codezeilen.

Das Programm ist portabel und somit auch auf anderen Betriebssystemen wie beispielsweise *Linux* oder *Mac OS X* einsetzbar.



Bis auf wenige optische Verschönerungen (zum Beispiel die Einfärbung des Codes (Syntax Highlighting) oder Teile des dunklen Designs) wurde alles von mir programmiert und gestaltet.

Aufgrund des Komplexitätsgrads des Codes kann ich nur kleine Ausschnitte zeigen. Wie heutzutage üblich, wurde das Programm konsequent objektorientiert gestaltet.

Da die Vektorrechnung eine große Rolle spielt, wurde hierfür eine eigene Vektorklasse mit dem Namen `Engine.Graphics.Math.Vector` erstellt. Sie stellt Methoden für die verschiedenen Vektoroperationen wie beispielsweise für das Skalarprodukt oder das Kreuzprodukt bereit. Sie kann drei Komponenten speichern.

In folgendem Codeausschnitt finden Sie die Implementierung des Minusoperators, der dafür sorgt, dass `new Vector(1, 2, 3) - new Vector(1, 2, 2)` einen Vektor mit den Koordinaten (0, 0, 1) zurückgibt.

1 Eine von Microsoft entwickelte Softwareplattform, die eine Laufzeitumgebung und eine Standardbibliothek mit einigen Funktionen umfasst.

```
/// <summary>
/// Implements the operator -. Subtracts two vectors.
/// </summary>
/// <param name="v1">The first vector.</param>
/// <param name="v2">The second vector.</param>
/// <returns>The result of the operation.</returns>
public static Vector operator -(Vector v1, Vector v2)
{
    return new Vector(v1.X - v2.X, v1.Y - v2.Y, v1.Z - v2.Z);
}
```

Code 1: Subtraktion von Vektoren

Ein Strahl (`Engine.Graphics.Math.Ray`) verfügt über einen Ursprungsvektor `Origin` und einen normalisierten Richtungsvektor `Direction`.
Für die Bilderzeugung muss für jeden Pixel auf dem Bildschirm zuerst der Richtungsvektor des Strahls berechnet werden. Der Ursprungsvektor entspricht dem Ortsvektor der Position der Kamera.

```
/// <summary>
/// Calculates the ray using a given screen position.
/// </summary>
/// <param name="camera">The camera which is being used.</param>
/// <param name="pointOnScreen">The point on screen.</param>
/// <returns>The resulting ray.</returns>
public Ray CalcRayFromScreenPosition(Camera camera, PointF pointOnScreen)
…
```

Code 2: Berechnung des Strahls – die verwendete Kamera sowie der Punkt auf dem Bildschirm werden der Funktion übergeben. Der Strahl ist der Rückgabewert der Funktion.

Anschließend wird die Methode `Colliding` jedes Primitives aufgerufen. Diese prüft, ob eine Kollision des Strahls mit dem Primitiv vorliegt. Abhängig vom Primitivtyp erfolgt die Kollisionsüberprüfung unterschiedlich.

```
/// <summary>
/// Tests whether a ray collides with an specific element.
/// </summary>
/// <param name="rayTest">The ray which may or may not have an intersection point
with the element.</param>
/// <param name="normal">the normal vector at the collision point (if any).</param>
/// <param name="intersectionPoint">The intersection point.</param>
void Colliding(Ray rayTest, ref Vector normal, ref Vector intersectionPoint)
…
```

Code 3: Prüft, ob ein Strahl ein Primitiv trifft. `rayTest` *entspricht dem zu prüfenden Strahl. Falls es einen Kollisionspunkt gibt, wird die Normale* `normal` *und der Kollisionspunkt* `intersectionPoint` *zurückgegeben.*

Nachdem ein Kollisionspunkt gefunden wurde, der möglichst nah am Strahlenursprung liegt, muss die Farbe des Strahls ermittelt werden.

Dafür ist der *Shader* (Schattierer) zuständig. Dieser nimmt die Normale und den Kollisionspunkt entgegen und berechnet die Farbe. Der Algorithmus wird im Unterkapitel „Beleuchtung“ beschrieben.

Nachdem die Farbe ermittelt wurde, kann sie für den entsprechenden Pixel im Bild (`pointOnScreen`) übernommen werden. So entsteht nach und nach ein vollständiges Abbild.

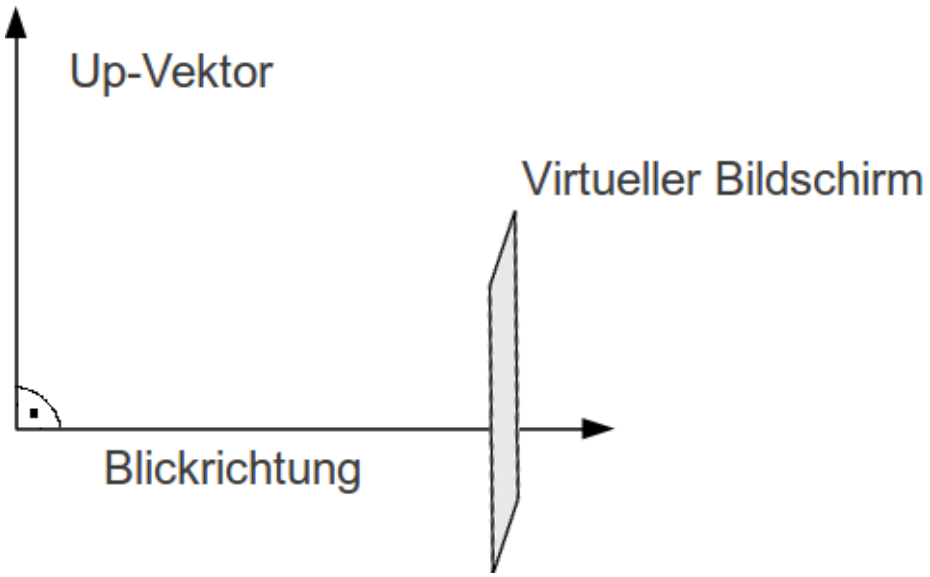
```
//Iteriere durch alle Lichtquellen
for (int i = 0; i < World.LightsColors.Length; i++)
{
    //Die Position der aktuellen Lichtquelle
    Vector LightPoint = World.LightsPosition[i];

    //Die Farbe der aktuellen Lichtquelle (X = rot, Y = grün, Z = blau)
    Vector LightColor = World.LightsColors[i];

    //Der Vektor, der von der Lichtposition auf den Kollisionspunkt zeigt.
    Vector LightToIntersection = LightPoint – intersectionPoint;
    float shade = 1;


    //Normalisiert den Vektor LightToIntersection
    LightToIntersection = LightToIntersection.Normalize();

    //Wird der Lichtstrahl von einem anderen Objekt verdeckt?
    if (!MaterialInfo.IsLight)
    {
        if (World.IsOccluding(new Ray(LightPoint, LightToIntersection), Distance,
            CollisionPrimitive))
            //Das Objekt wird verdeckt; es befindet sich Schatten an dieser Stelle.
            shade = 0;
        else
            //Die Beleuchtungsstärke entspricht der Beleuchtungsstärke der
            //Lichtquelle.
            shade = World.LightsShade[i];

    }

    //diffusion
    if (shade != 0)
    {
        float dot = LightToIntersection.ScalarProduct(Normal);

        //Wenn das Skalarprodukt kleiner als 0 ist, ist die Normale vom Punkt
        //abgewandt – die Rückseite soll unbeleuchtet sein, also wird nicht weiter
        //verfolgt.
        if (dot > 0)
        {
            //Das Objekt ist an dieser Stelle dem Licht zugewandt.
            float diff = dot * Diffusion * shade;

            //Berechne die resultierende Farbe.
            ResultingColor.X += PrimitiveColorDevidedX * LightColor.X * diff;
            ResultingColor.Y += PrimitiveColorDevidedY * LightColor.Y * diff;
            ResultingColor.Z += PrimitiveColorDevidedZ * LightColor.Z * diff;
        }
    }
}
```

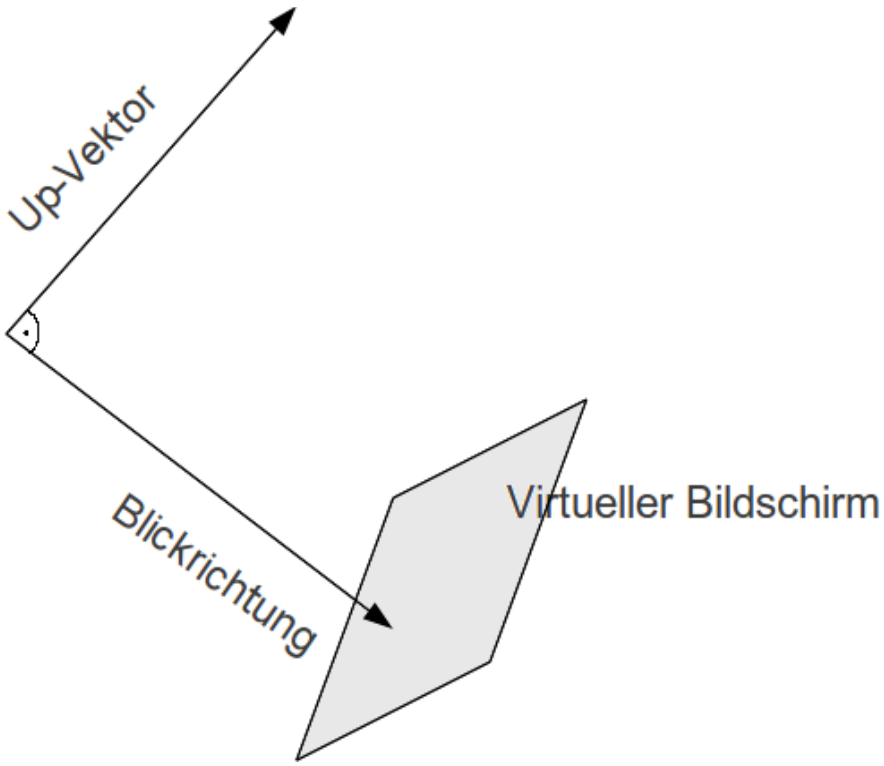
Code 4: Determiniert die resultierende Farbe eines Punktes. (Ausschnitt aus dem Shader)

Probleme bei der Implementierung

Bei der Implementierung stieß ich auf einige Probleme. Das größte Problem hing mit der Perspektive zusammen.

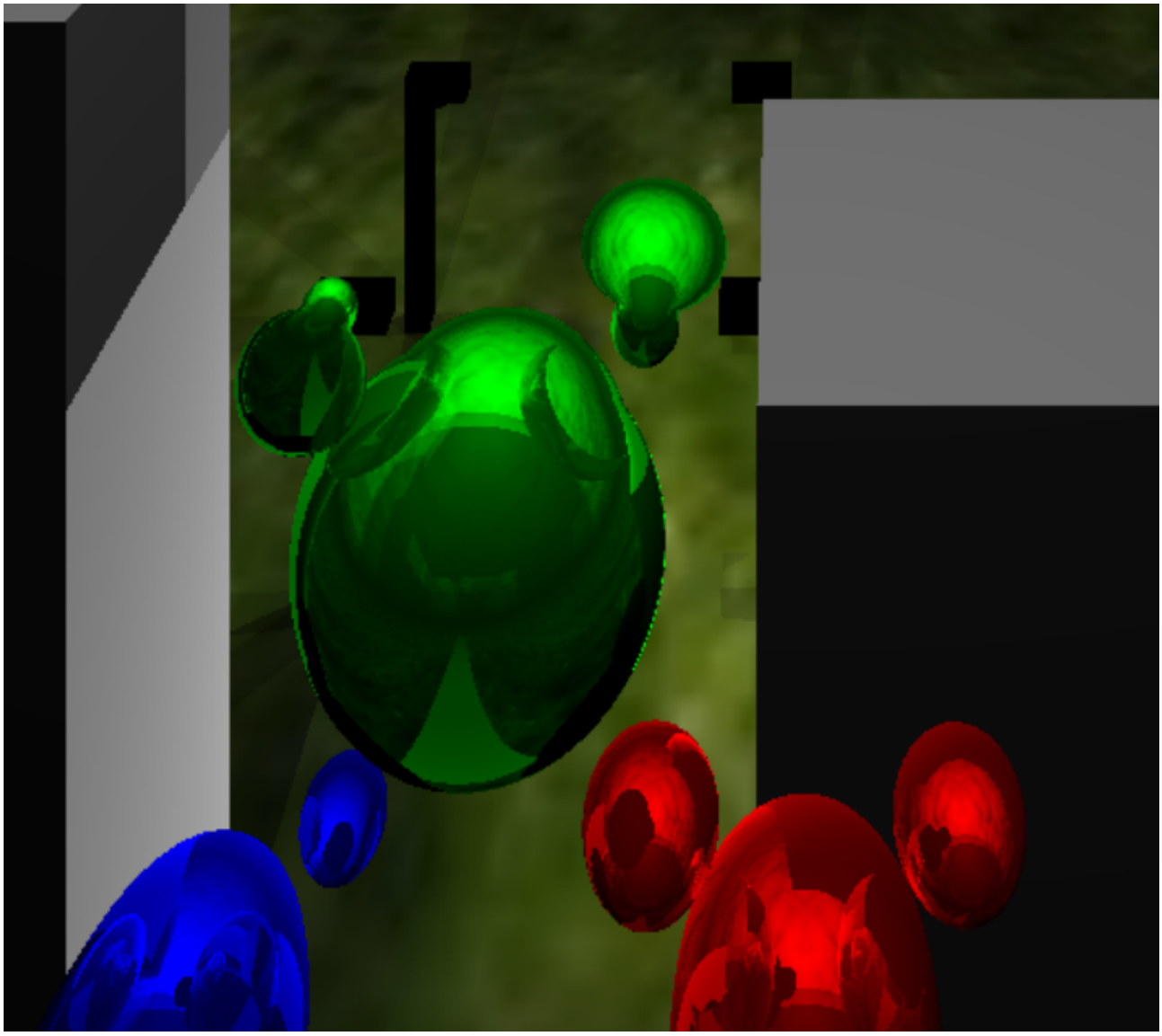
Der Up-Vektor legt fest, wo das obere Ende des virtuellen Bildschirms ist. Er bildet stets einen rechten Winkel mit der Blickrichtung.





Ich beachtete anfangs den Up-Vektor nicht, der sich bei einer Veränderung der Blickrichtung ebenfalls ändern musste. Es entstanden verzerrte Bilder, bei denen die Kugeln die merkwürdigsten Formen hatten.

Der Up-Vektor der Kamera wird benötigt, um für jeden Pixel auf dem virtuellen Bildschirm den dazugehörigen Strahl zu berechnen.

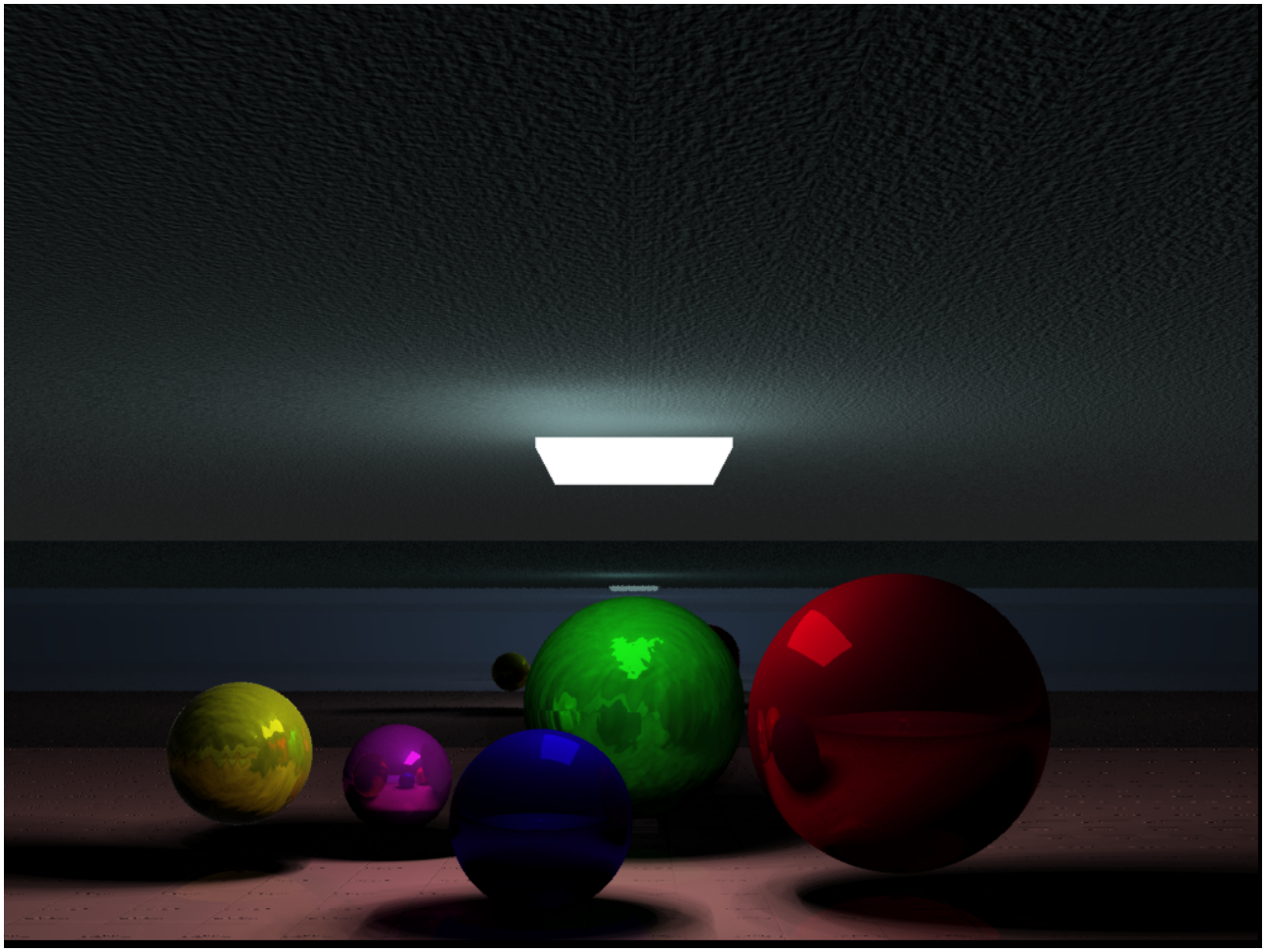


Anwendungsgebiete

Für Raytracing existieren bereits einige Anwendungsgebiete. Es wird aktuell größtenteils dort eingesetzt, wo es mehr auf die Bildqualität als auf die Renderingzeit ankommt.

Die Filmindustrie verwendet dieses Verfahren bereits jetzt, weswegen viele Kinoeffekte sehr realistisch aussehen.

Durch die fortschreitende Digitalisierung von Bibliotheken könnte man sich vorstellen, dass man in einigen Jahren ähnlich wie in einem Spiel eine virtuelle Bibliothek (im Rahmen einer „Virtual Reality“, einer virtuellen Realität) durchwandern kann und dabei Bücher aus dem Regal greifen und lesen kann.
Architekten benutzen Raytracing, um den Kunden einen möglichst genauen Eindruck vom zukünftigen Heim darbieten zu können.
Für Simulationen ist der Grad der Realität bei der Darstellung häufig der entscheidende Faktor.



Beurteilung über das Raytracing

Das größte Problem von Raytracing ist die Performance. Wenn die Grafikkarte mitrechnen würde, die sehr gut mit Vektorrechnungen umgehen kann, könnte man die Geschwindigkeit drastisch erhöhen. Das ist bereits heutzutage mittels NVidias *CUDA*[1] oder AMDs *Stream*[2] möglich, die allerdings zueinander inkompatibel sind.

Bis dahin wird die „klassische“ Renderingtechnik mit Z-Buffern insbesondere den Spielemarkt beherrschen. Allerdings gab es schon verschiedene Projekte, die sich um die Umsetzung eines Raytracers in Form eines Spiels bemüht haben, welche von unterschiedlichen Erfolgsaussichten geprägt waren.

Durch steigende Rechnerkapazitäten wird die Bedeutung von Raytracing wahrscheinlich noch weiter steigen, obwohl es eigentlich ein altes Verfahren ist, welches schon beim Spiel „Doom“ (1993) in einer sehr einfachen Art und Weise eingesetzt wurde.

Ein weiteres Problem ist der Realitätsgrad: Oftmals muss zwischen empirischen Modellen und exakten physikalischen Berechnungen gewählt werden, wobei letztere meist aufwändiger (im Sinne des Rechenaufwandes) und schwieriger zu implementieren sind.

Deshalb wird meist ein Kompromiss zwischen beidem eingegangen. Doch da das Gehirn sich leicht austricksen lässt, ist dies zweitrangig. Das visuell Wahrgenommene ist stark subjektiv. Das Hirn hat das Bestreben, stets normalisiert zu sehen, also die empfangenen Sehinformationen den Erfahrungen anzupassen.

Auch bei anderen Naturphänomenen müssen Tricks verwendet werden, um die gewünschten Effekte zu erreichen. Ein Beispiel dafür wäre der Himmel. Der Himmel erscheint uns blau, weil blaues Licht ca. 16 mal stärker gestreut wird als rotes Licht. Die Streuung entsteht durch die Moleküle in der Erdatmosphäre. Die Sonne erscheint vom Weltraum aus weiß und wird auf der Erde aufgrund von der Lichtstreuung nicht in dieser Farbe wahrgenommen.

Um dieses Phänomenen in angemessenem Zeitaufwand zu simulieren, wird eine Wolkentextur, also ein einfaches Bild, auf eine Ebene gelegt, die den Himmel repräsentieren soll. Der Mensch ist trotzdem bereit, ihn mehr oder weniger als „Himmel“ zu akzeptieren.

Quantenmechanische Effekte lassen sich leider nicht oder nur sehr schwer darstellen, da Raytracing auf der geometrischen Optik basiert und nur auf makroskopische Phänomene ausgelegt worden ist.

So lässt sich beispielsweise das Doppelspaltexperiment, bei dem die Welleneigenschaften des Lichts durch Interferenzmuster sichtbar gemacht werden können, nicht einfach durchführen.

1 http://www.nvidia.com/object/cuda_home_new.html
2 http://www.amd.com/stream

Quellenverzeichnis

Internetquellen:
http://mathworld.wolfram.com/Reflection.html, 14.09.2010
http://www.informatik.uni-frankfurt.de/~sschaef/Diplomarbeit_schaefer.pdf, 25.08.2010
http://www.devmaster.net/articles/raytracing_series/part7.php, 28.09.2010
http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html, 30.09.2010
http://www.springerlink.com/content/p636k7p5443528r5/, 17.10.2010
http://www.easyrgb.com/index.php?X=MATH&H=02#text2, 17.10.2010
http://www.flipcode.com/archives/reflection_transmission.pdf, 30.09.2010
http://keepcoding.bplaced.com/kcdev/tutorials/HLSL%20Introduction.pdf, 30.09.2010
http://www.devmaster.net/articles/raytracing_series/Reflections%20and%20Refractions%20in%20Raytracing.pdf, 19.01.2011
http://iwr.uni-heidelberg.de/groups/ngg/CG2008/Txt/Kapitel4.pdf, 19.01.2011
http://www.uni-koeln.de/math-nat-fak/kristall/forschung/kristallphysik/optik/node6.html, 17.10.2010
http://www.heise.de/newsticker/meldung/Nintendo-kuendigt-Handheldkonsole-3DS-an-961085.html, 21.01.2011

Literaturquellen:
Kühlke, Dieter, Optik: Grundlagen und Anwendungen, Wissenschaftlicher Verlag Harri Deutsch GmbH, 2007
Lengyel, E., Mathematics for 3D game programming and computer graphics, 2001
Maffai, L. & Fiorentini, A., Das Bild im Kopf, Birkhäuser Verlag, 1997
Meiers Physiklexikon, hg. v. der Fachredaktion für Naturwissenschaften und Technik des Bibliographisches Instituts; Bibliographisches Institut AG, 1973
Das große Tafelwerk interaktiv, Cornelsen, 2003

Bildquellen:
Alle Bilder wurden von mir angefertigt.

Ich danke
Frau Beck und Herrn Pfannebecker
für die Unterstützung.