RAYTRACER
Eine Projekt–Aufgabe
BA Horb

Eckart Göhler

SS 2003

1 Das Projekt

Wie in jedem Semester soll auch diesmal ein zusammenhängendes Projekt realisiert werden, nämlich ein Programm, mit dem man aus einer *Szenenbeschreibung* photorealistische Bilder errechnen kann. Das Projekt bietet neben der Umsetzung einiger interessanter Algorithmen (rekursive Strahlverfolgung, Suche in einer verzeigerten Liste) auch ein geradezu klassisches Beispiel von objektorientierter Programmierung und Polymorphie. Darüberhinaus wird für die graphische Ausgabe eine moderne Widget-Bibliothek (QT) verwendet, die die Grundlagen von ereignisorientiertem Programmieren vermittelt.

2 Funktionsweise eines Raytracers

Ein Raytracer („Strahlverfolger“) ist ein Programm, welches versucht, anhand einer Objektbeschreibung (Position, Farbe, Reflektionsgrad etc.) die Lichtstrahlen zu verfolgen und die Farben/Helligkeiten auf einem Bild zu bestimmen. Im Prinzip genügt es, alle Lichtstrahlen der Lichtquellen zu verfolgen, die dann von den Objekten farbig gestreut oder reflektiert werden, bis sie dann auf der Bildebene ankommen. Das Problem dabei ist nur, dass es unendlich viele solcher Lichtstrahlen gibt, und nur der kleinste Teil die Bildebene erreicht.

Aus diesem Grund rechnet man rückwärts: vom Auge ausgehend verfolgt man die den Sehstrahl durch alle Punkte der Bildebene (die in Pixel unterteilt ist), und reflektiert den Sehstrahl, bis er entweder eine Lichtquelle trifft oder kein Objekt mehr treffen kann und im Hintergrund verschwindet. Auf diesem Weg werden alle Farben und Filterwirkungen aufsummiert und somit die Bildpunktfarbe bestimmt.

Diese Methode funktioniert sehr gut für reflektierte Strahlen, ist aber nur eingeschränkt für gestreutes Licht verwendbar (weil an einer Oberfläche gestreutes Licht aus allen Richtungen kommen kann). Dennoch bekommt man mit diesem Vorgehen schon sehr gute Ergebnisse. Wir werden im folgenden die verschiedenen Punkte genauer analysieren.

3 Physikalische Grundlagen

Damit ein halbwegs realistisches Bild errechnet werden kann, muss untersucht werden, was mit einem Lichtstrahl passiert, wenn es die Oberfläche eines Körpers trifft.

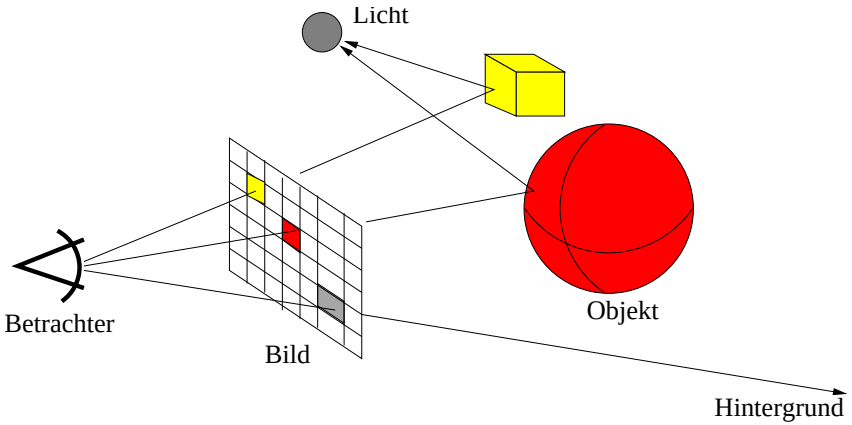

Abbildung 1: Schema eines Raytracers

3.1 Die Grundfarben

Auch wenn das Licht aus einem kontinuierlichen Spektrum an Farben (Regenbogen) besteht, genügt es für die sinnesphysiologischen Vorgänge, wenn die Intensität (Helligkeit) von nur drei Grundfarben verwendet werden, um das Spektrum abzudecken. Dies sind üblicherweise die Farben Rot, Grün und Blau[1]. Wenn ein Körper in einer bestimmen Farbe erscheint, so ist dies die Farbe, die *nicht* absorbiert (herausgefiltert) wird.

3.2 Streuung

Wenn ein Lichtstrahl auf eine nicht ganz glatte Oberfläche trifft, wird er abgeschwächt in alle Richtungen über der Oberfläche weitergeschickt. Dies wird als Streuung bezeichnet. Die Helligkeit des Strahls hängt neben der Farbe des Körpers noch von dem Auftreffwinkel des Lichtstrahls ab (weil bei schrägem Einfall des Lichts weniger Licht auf die gleiche Fläche fällt). Die Streuung wird durch folgende Formel beschrieben:

[1]Es gibt aber auch andere Grundfarben, die verwendet werden können.

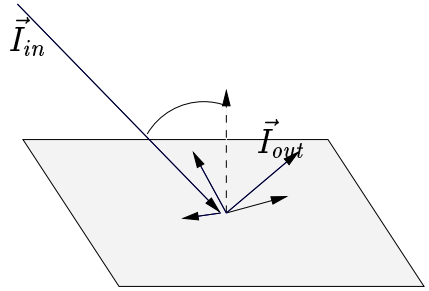

Abbildung 2: Streuung

$$I_{out} = I_{in} F(I_{in}) \cos(\vec{I_{in}}, \vec{N})$$

Dabei ist:

I_{in}	Helligkeit (Intensität) des eintreffenden Lichtstrahls für eine Farbe.
I_{out}	Helligkeit des gestreuten Lichtstrahls.
$F(I_{in})$	Farbabschwächung bei der Streuung (Filterung durch den Körper). Muss für jede Farbe separat bestimmt werden.
$\cos(\vec{I_{in}}, \vec{N})$	Kosinus des Winkels zwischen eintreffendem Lichtstrahl und dem Normalenvektor des Körpers an der Auftreffstelle.

Das Problem für einen Raytracer besteht wie schon erwähnt darin, dass man nicht alle Lichtstrahlen zu allen Körpern, die irgendwelche Lichtstrahlen weiterschicken, verfolgen kann. Deshalb beschränkt man sich bei der Streuung darauf, die Strahlen nur bis zu den sichtbaren Lichtquellen zu verfolgen. Dies ist auch die Haupquelle der gestreuten Lichtstrahlen.

3.3 Reflektion

Bei der Reflektion werden Lichtstrahlen nicht in alle Richtungen weitergeschickt, sondern nur in eine, die dem Spiegelgesetz folgt: Einfallswinkel

gleich Ausfallswinkel. Die Farbe des Körpers spielt in diesem Falle keine Rolle (man kann das z.B. durch die Spiegelung eines roten Balls auf einer glänzenden blauen Tischfolie testen).

Da nicht alle Körper gleich gut reflektieren, muss man einen Reflektionskoeffizienten einführen, welcher die Stärke der Reflektion ausdrückt. Dieser ist im allgemeinen für verschiedene Farben unterschiedlich; wir gehen aber davon aus, dass er für alle Farben gleich ist. Die Formel lautet dann:

$$I_{out} = c_r I_{in}$$

mit c_r als Reflektionskoeffizienten und

$$\omega_{in} = \omega_{out}$$

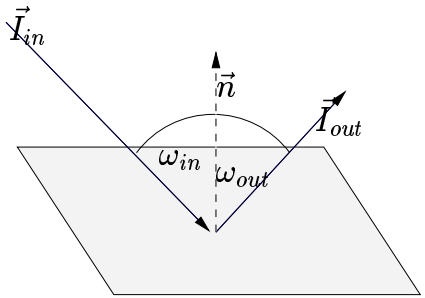

Abbildung 3: Beispiel einer Reflektion

3.4 Glanzreflektion von Lichtern

Die echte Reflektion muss bei hellen Lichtstrahlen erweitert werden, da diese (als ein Mittelding zwischen Reflektion und Streuung) auch in etwas von der echten Reflektion abweichende Richtung Licht streut. Dabei hängt die Helligkeit des reflektierten Strahls stark von dieser Abweichung ab:

$$I_{out} = c_r I_{in} cos(\omega_g)^n$$

Hier ist:

ω_g	Der Glanzwinkel zwischen der „echten“ Reflektion $I_{\text{out,refl}}$ und der Richtung von I_{out}.
c_r	Wieder der Reflektionskoeffizient.
n	Ein Reflektionsparameter, der die „Metallizität“ des Materials ausdrückt. Mit ihm kann der Glanz beeinflusst werden.

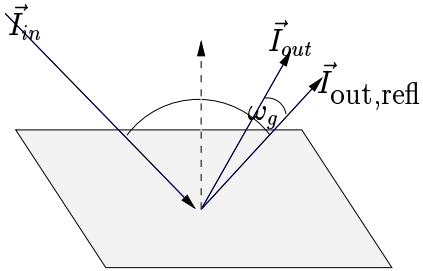

Abbildung 4: Reflektion mit Glanzwinkel von hellen Lichtquellen

4 Mathematische Grundlagen

Die Körper und die Strahlverfolgung muss in einer geeigneten Weise im Programm repräsentiert werden. Es bietet sich an, die mathematische Notation der analytischen Geometrie zu verwenden, um die Position der Körper durch Ortsvektoren, die Lichtstrahlen durch Geraden und die Strahlrichtungen durch Vektoren auszudrücken. Ebenso kann die Farbe eines Strahls durch einen Vektor mit drei Komponenten dargestellt werden.

4.1 Vektoren, Koordinatensysteme

Ein Vektor ist eine Größe, die aus mehreren Komponenten besteht und üblicherweise eine Richtung darstellt. In unserem Fall hat der Vektor drei Komponenten für die Raumrichtungen `x,y,z`. Ein Ortsvektor beschreibt

in einem Raum die Position eines Punktes, indem er dessen Koordinaten repräsentiert. Wir verwenden folgendes rechtshändige Koordinatensystem:

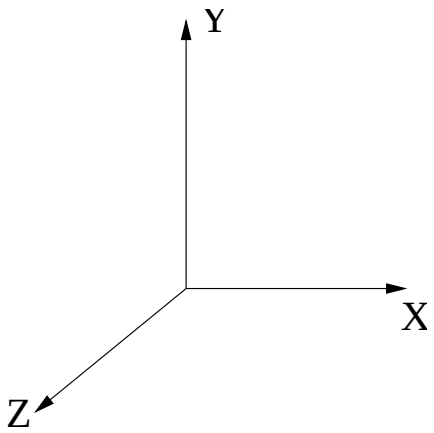

Abbildung 5: Das verwendete Koordinatensystem

Dieses hat den Vorteil, dass ein Bild in die `x/y`–Ebene gelegt werden kann (mit `z`=0), und dass der Betrachter sich auf dem positiven Z-Teil befindet.

Ein Vektor kann zu einem anderen Vektor komponentenweise addiert oder subtrahiert werden. Ebenso ist es möglich, einen Vektor komponentenweise mit einer Zahl zu multiplizieren. Auf der anderen Seite ist es nicht sinnvoll, einen Vektor komponentenweise mit einem anderen Vektor zu multiplizieren oder zu dividieren.

4.2 Betrag

Ein Vektor hat eine bestimmte Länge. Diese kann mit dem Satz des Pythagoras im dreidimensionalen bestimmt werden:

$$||\vec{a}|| = \sqrt{\sum_{i=1}^{3} a_i^2}$$

also die Wurzel aus der Quadratsumme der Komponenten.

4.3 Skalarprodukt

Man kann allerdings einen Vektor mit einem anderen Vektor komponentenweise multiplizieren, und die Summe bilden. Das Ergebnis ist eine Zahl, die als *Skalarprodukt* bezeichnet wird (Zahlen sind Skalare):

$$(\vec{a} \cdot \vec{b}) = \sum_{i=1}^{3} a_i b_i$$

Das Skalarprodukt hat eine anschauliche Bedeutung: für zwei Vektoren $\vec{a}$ $\vec{b}$ ist der eingeschlossene Winkel bestimmt durch:

$$\cos \vec{a}, \vec{b} = \frac{(\vec{a} \cdot \vec{b})}{||\vec{a}|| \cdot ||\vec{b}||}$$

Wenn also die Vektoren den Betrag eins haben, ist das Skalarprodukt gleich dem Kosinus des eingeschlossenen Winkels. Wenn zwei Vektoren die gleiche Richtung haben, ist das Skalarprodukt Null.

4.4 Projektion

Das Skalarprodukt kann auch dazu verwendet werden, die *Projektion* eines Vektors auf einen anderen zu bestimmen. Das bedeutet, dass von einem Vektor $\vec{a}$ das Lot auf den anderen Vektor $\vec{b}$ gefällt wird, und eine Zahl bestimmt wird, mit der der Vektor $\vec{b}$ multipliziert werden muss, um genau auf den Lotfußpunkt zu zeigen.

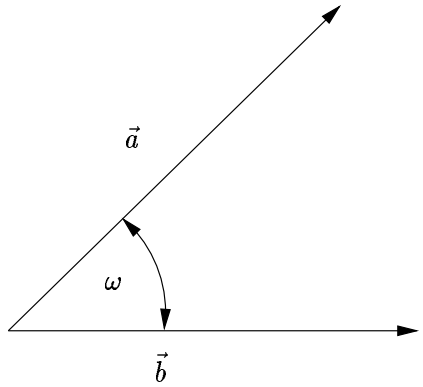

Abbildung 6: Das Skalarprodukt zweier Vektoren mit dem Winkel ω

Voraussetzung ist, dass der Vektor $\vec{b}$ die Länge eins hat. Dann ist der Verlängerungsfaktor genau:

$$\lambda = (\vec{a} \cdot \vec{b})$$

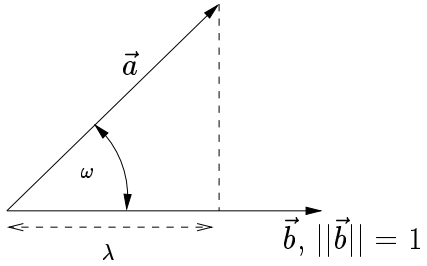

Abbildung 7: Projektion von $\vec{a}$ auf $\vec{b}$ mit dem Streckungsfaktor λ

4.5 Kreuzprodukt

Gelegentlich möchte man zu zwei Vektoren einen dritten, senkrecht darauf stehenden finden. Dies kann mit dem Kreuzprodukt erreicht werden. Dabei müssen alle Komponenten nach folgender Formel bestimmt werden:

$$\vec{c} = \vec{a} \times \vec{b}$$

$$\begin{aligned} c_x &= a_y b_z - a_z b_y & (1)\\ c_y &= a_z b_x - a_x b_z & (2)\\ c_z &= a_x b_y - a_y b_x & (3) \end{aligned}$$

4.6 Geraden

Eine Gerade kann durch zwei Vektoren beschrieben werden: Einen Ortsvektor, der einen Startpunkt festlegt, und einen zweiten Richtungsvektor, der die Richtung der Gerade bestimmt. Die Punkte einer Geraden können erreicht werden, indem der Richtungsvektor mit einem *Geradenparameter* multipliziert wird und zu dem Ortsvektor addiert wird:

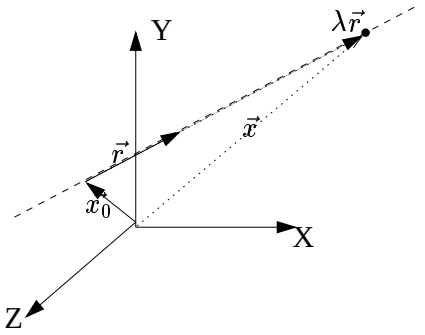

Abbildung 8: Darstellung einer Geraden

$$\vec{x} = \vec{x_0} + \lambda\vec{r}$$

$\vec{x}$	Position eines Punktes auf der Gerade
$\vec{x_0}$	Ortsvektor der Gerade
λ	Geradenparameter
$\vec{r}$	Richtungsvektor

4.7 Kugelgleichung

Eine Kugel kann dadurch beschrieben werden, dass auf einen Ortsvektor ein weiterer Vektor aufgesetzt wird, der zwar in jede beliebige Richtung zeigen darf, aber immer die gleiche Länge (den Radius) haben muss.

$$\vec{x} = \vec{x_0} + \vec{r}$$

mit $||\vec{r}|| = r$.

Die Bestimmung eines Schnittpunktes mit einer Geraden (das werden wir brauchen) ist nicht ganz einfach und Teil der Aufgaben.

4.8 Ebenengleichung

Für die Darstellung von Ebenen gibt es mehrere Formen; entweder spannt man sie mit zwei Vektoren auf, die auf einem Ortsvektor aufsitzen, oder ein Normalenvektor (der also senkrecht zur Ebene steht) wird festgelegt, der die Richtung der Ebene beschreibt, und zudem ein Abstand zum Ursprung. Letztere Form hat den Vorteil, dass mit ihr leicht der Schnittpunkt einer Geraden bestimmt werden kann, da sie durch eine lineare Gleichung beschrieben werden kann.

Wenn nämlich der Normalenvektor (der die Länge eins haben sollte) die Komponenten [n_x,n_y, n_z] hat und die Ebene einen Abstand d zum Ursprung hat, so gilt für alle Punkte $\vec{x}$ (mit den Koordinaten x, y, z) auf der Ebene:

$$n_x x + n_y y + n_z z = d$$

Um einen Schnittpunkt mit einer Gerade zu bestimmen, genügt es, die Gerade einzusetzen und die Gleichung nach dem Geradenparameter λ aufzulösen.

Mit Hilfe des Kreuzproduktes ist es möglich, aus der oben beschriebenen ersten Form der Ebenenbeschreibung einen Normalenvektor zu errechnen, und mit Hilfe der Ebenengleichung kann man durch Einsetzen leicht den Ebenenabstand gewinnen. Offenbar sind also beide Formen gleichwertig.

5 Design des Programms

Unser Raytracer soll

- Wenigstens Ebenen und Kugeln behandeln können
- Beliebig viele Körper und Lichtquellen verarbeiten können
- Die Szenenbeschreibung (Licht/Körper) aus einer Textdatei in einem festen Format lesen können.
- Die Graphik wiederholt in einem Fenster ausgeben.

Wir werden dazu die Programmiersprache C++ verwenden, die Ausgabe soll mit dem QT-Widget-Set implementiert werden.

Die Körper und Lichter sollen durch C++ – Objekte implementiert werden, wobei die Körper von einem Grund-„Objekt“ als abstrakte Klasse abgeleitet werden sollen. An Lichtern soll es nur einen Typ geben, nämlich den der Punktquelle.

Damit die Berechnungen syntaktisch leicht umsetzbar sind, soll eine Bibliothek für Vektoren und Geraden erstellt werden, bei denen die Basisoperationen durch Operatoren implementiert sind. Ebenso soll eine Klasse für Farben implementiert werden. Alle diese Grundklassen sollen über Stream–Operatoren einlesbar und ausgebbar sein (was das einlesen einer Datei erheblich vereinfachen wird).

Das Programm soll folgende Module enthalten:

- `anageom.cc` – Implementierung der Vektor/Geradenklasse.
- `farben.cc` – Implementierung der Farbklasse.
- `main.cc` – Hauptprogrammteil mit Fensterausgabe
- `lichter.cc` – Die Lichter–Klasse

- `objekt.h` – Abstrakte Objekt-Klasse
- `kugel.cc` – Das Kugel–Objekt
- `ebene.cc` – Das Ebenen–Objekt
- `szene.cc` – Die Szenenbearbeitung; hier findet die eigentliche Strahlverfolgung statt.

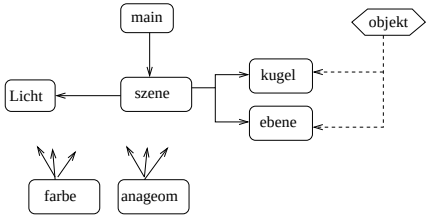

Abbildung 9: Darstellung des Designs. Gestrichelt die Vererbungsfolge.

6 Die Implementationsaufgaben

6.1 Aufgabe 1: Eine `Vektor`–Klasse

Schreiben Sie eine Klasse, die einen `Vektor`–Datentyp implementiert. Der Vektor soll drei Komponenten (x,y,z) haben, die aber nicht von außen veränderbar sein sollen.

Konstruktor

Folgende Konstruktoren sollen unterstützt werden:

- `Vektor()` – Der Default–Konstruktor. Alle Komponenten werden mit 0 initialisiert.
- `Vektor(X,Y,Z)` – Initialisiert die Komponenten.

Methoden

- `X(),Y(),Z()` – Gibt die Komponenten zurück (Inline, wegen Geschwindigkeit).

Operatoren

- `+,-` – Addieren/Subtrahieren von Vektoren.
- `*i` – Multiplikation mit einem Skalar `i`.
- `*v` – Skalarmultiplikation mit einem Vektor `v`.
- `^v` – Kreuzprodukt mit einem Vektor `v`.
- `% n` – Spiegelung eines Vektors an einer Normalen `n`.

Funktionen

Ein Teil der Vektor-Behandlung kann aus syntaktischen Gründen nicht über Methoden oder Operatoren abgearbeitet werden. So ist etwa die Betragsbestimmung eine Sache, die man besser als Funktion implementiert. Diese sollte allerdings auf die Komponenten zugreifen können. Dies kann mit Hilfe der `friend` – Konstruktion bewerkstelligt werden.

- `Betrag()` – Bestimme den Betrag eines Vektors.
- `i*v` – Multipliziere den Vektor `v` mit einer Zahl `i`.
- `ostream & operator<<(ostream &s, Vektor x )` – gib den Vektor `x` aus.
- `istream & operator>>(istream &s, Vektor x )` – lies den Vektor `x` ein. Die Eingabe soll das Format „$[x,y,z]$“ haben. (Mit x,y,z als Komponenten)

Beachte

Die größte Schwierigkeit liegt bei der Implementation des Spiegelungsoperators `%`, den wir für die Reflektion benötigen. Versuchen Sie zunächst, von Hand die notwendigen Vektoroperationen mit Hilfe des Skalarproduktes (Projektion) zu bestimmen. Es sei davon ausgegangen, dass der Normalenvektor normiert ist (also die Länge 1 hat).

Versuchen Sie, die Vektor–Klasse zu testen, indem sie mehrere Vektoren einlesen und einige der elementaren Operationen ausprobieren.

6.2 Aufgabe 2: Eine `Strahl`–Klasse

Schreiben Sie eine Klasse, die eine Gerade repräsentiert. Intern soll ein Orts– und Richtungsvektor verwendet werden.

Konstruktor

Folgende Konstruktoren sollen unterstützt werden:

- `Strahl()` – Der Default–Konstruktor.
- `Strahl(Vektor x0, Vektor r)` – Initialisierung mit einem Ortsvektor `x0` und einem Richtungsvektor `r`.

Methoden

- `Pos()` – Gib den Ortsvektor zurück.
- `Richt()` – Gib den Richtungsvektor zurück.

Operatoren

Damit die Strahl-Klasse verwendbar ist, brauchen wir eine Möglichkeit, um für einen bestimmten Geradenparameter λ einen Ortsvektor zu erhalten. Es bietet sich an, analog zur Skalierung von Vektoren dafür den Multiplikationsoperator `*` zu missbrauchen, bei dem durch die Multiplikation einer Zahl mit einer Gerade (Strahl) ein Vektor bestimmt wird:

- `Vektor operator*(double a, Strahl s)` – Bestimme den Ortsvektor für den Geradenparameter `a` mit der Geraden `s`.

Beachte

- Auch hier sollte man die `friend`–Konstruktion verwenden, um den alleinstehenden Multiplikationsoperator für die privaten Komponenten zu verwenden.
- Versuchen Sie, den größten Teil über Inline-Methoden abzuwickeln.

6.3 Aufgabe 3: Eine `Farbe`–Klasse

Schreiben Sie eine Klasse, die eine Farbe mit den Grundfarben Rot, Grün und Blau repräsentiert. Farben können addiert werden (aber nur bis zu einem bestimmten Grad – sonst wird das Bild überbelichtet). Farben können auch komponentenweise multipliziert werden; mit dieser Konstruktion kann man eine Filterwirkung erzielen. Damit die Farben sich gut in dieses Schema fügen, sollen für die Komponenten nur Werte zwischen 0..1 zugelassen sein. Fangen Sie andere Werte ab!

Konstruktor

Folgende Konstruktoren sollen unterstützt werden:

- `Farbe()` – Der Default–Konstruktor.
- `Farbe(rot,gruen,blau)` – Initialisierung mit den Grundfarben.

Methoden

- `R(),G(),B()` – Gibt die Grundfarben Rot, Grün und Blau zurück.

Operatoren

- `+` – Addiere Farben. Bei der addierten Farbe darf keine Komponente über 1 liegen.

- `*` – Multipliziere Farben. Das Ergebnis ist komponentenweise immer gleich oder kleiner 1.
- `*i` – Skaliere (multipliziere) die Farben komponentenweise mit einer Zahl `i`.

Funktionen

Auch hier muss ein Teil der Operationen in Operator–Funktionen ausgelagert werden.

- `i*f` – Multipliziere die Farbe `f` mit einer Zahl `i` (auf der linken Seite).
- `ostream & operator<<(ostream &s, Farbe f )` – gib die Farbe `x` aus.
- `istream & operator>>(istream &s, Farbe f )` – lies die Farbe `f` ein. Die Eingabe soll das Format „(r,g,b)“ haben. (Mit r,g,b als Grundfarben.)

Beachte

- Es ist nicht verkehrt, eine globale Variable namens `SCHWARZ` anzulegen.

6.4 Aufgabe 4: Eine `Licht`–Klasse

Ein Licht–Objekt benötigt nicht mehr Information als seine Position und seine Farbe (jawohl — wir können auch farbige Bühnenscheinwerfer anschalten). Allerdings sind mehrere Lichter zugelassen. Es ist also sinnvoll, eine verzeigerte Liste von Lichtern zu verwenden. Diese Liste wird beim Einlesen der Szenerie erstellt, und jedes neue Licht wird vorne angehängt. Auf das folgende Licht kann mit einer Methode zugegriffen werden.

Schreiben Sie eine Klasse, die eine punktförmige Lichtquelle repräsentiert.

Konstruktor

Folgende Konstruktoren sollen unterstützt werden:

- `Licht()` – Der Default–Konstruktor.
- `Licht(Vektor p, Farbe f)` – Initialisierung mit der Position `p` und der Farbe `f`.
- `Licht(istream &file)` – Initialisierung, die die Werte aus dem Dateistrom `file` liest. Dazu muss die Position/Farbe über den Stream–Operator `>>` gelesen werden (die ja schon definiert wurden).

Methoden

Zunächst die Methoden, um eine Liste von Lichtern zu verwalten:

- `Licht *anhaengen(Licht *naechstes)` – Hängt an das aktuelle Objekt ein weiteres Licht an, indem ein Pointer verwendet wird. Intern wird dieser Pointer abgespeichert. Es wird ein Pointer auf die aktuelle Liste (`this`) zurückgegeben.
- `Licht *weiter()` – Gibt das nächste Licht zurück, oder NULL, wenn keines vorhanden ist.

Nun die Methoden, um die Lichteigenschaften auszulesen:

- `Pos()` – Gibt die Lichtposition zurück.
- `Farb()` – Gibt die Lichtfarbe zurück.

Beachte

- Testen Sie das Einlesen eines Lichts aus einem Stream (etwa cin).
- Wie kann man eine Liste von Lichtern erstellen, und dann wieder abarbeiten?

6.5 Aufgabe 5: Die abstrakte `Objekt`–Klasse

Die Objekt–Klasse ist die „Mutter“ aller Objekte. Sie ist abstrakt, d.h. sie legt alle wichtigen Schnittstellenmethoden an, ohne sie zu implementieren. Diese müssen dann von den Objekten abgedeckt werden.

Zum einen muss die Objekt–Klasse wie bei der Licht–Klasse eine Liste von eigenen Objekten verwalten können. Zum anderen müssen die Grundeigenschaften des Raytracers unterstützt werden:

1. Bestimme, ob ein Strahl dieses Objekt schneidet, und wo (Geradenparameter wird berechnet).
2. Für die Spiegelung benötigen wir die Normale.
3. Berechne die Streuung eines Lichtstrahls.
4. Bestimme die Reflektionseigenschaften (Spiegelung eines Objektes).
5. Bestimme die Reflektionseigenschaften von Licht.

Schreiben Sie eine Klasse `Objekt`, von der alle Körper abgeleitet werden.

Konstruktor

Folgende Konstruktoren sollen unterstützt werden:

- `Objekt()` – Der Default–Konstruktor (kann auch weggelassen werden).

Methoden

Zunächst die Methoden für die Listenverwaltung (analog zur Licht–Klasse). Sie sollen bereits für das Objekt fest implementiert werden:

- `Objekt *anhaengen(Objekt *naechster)` – Hängt ein (weiteres) Objekt an das aktuelle Objekt. Gibt einen Pointer auf das aktuelle Objekt zurück.
- `Objekt *weiter()` – gibt einen Pointer auf das nächste Objekt zurück.

Wie schon erwähnt benötigen wir für die Strahlverfolgung einen weiteren Satz an Operationen, die aber alle **virtuell** sein müssen (jeder Körper hat seine eigene Art, wie Licht geschnitten/gestreut/reflektiert/glanzreflektiert wird). Es bietet sich an, diese Operationen **abstrakt** zu halten (nicht zu implementieren).

- `double Schneide(Strahl von)` – Bestimmt, ob das Objekt von der gegebenen Gerade getroffen wird. Wenn ja, wird der Geradenparameter zurückgegeben, sonst -1 (der Geradenparameter kann nicht negativ werden, wenn der Strahl in positive Richtung geht).
- `Vektor Normale(Vektor p)` – Gibt den Normalenvektor zu einem Schnittpunkt `p` zurück. Der Normalenvektor wird für die Reflektion benötigt. Er soll normiert sein.
- `Farbe Streue(Vektor R, Vektor p, Farbe f)` – Gibt die Farbe eines Strahls mit der Richtung `R` und der Farbe `f` zurück, der bei einem Schnittpunkt `p` gestreut wird.
- `Farbe Reflektiere_Objekt(Farbe f)` – Gibt die Reflektionsfarbe eines von einem Objekt reflektierten Strahls der Farbe `f` zurück.
- `Farbe Reflektiere_Licht(Vektor von, Vektor p, Vektor nach, Farbe f)` – Gibt die glanzreflektierte Farbe eines von einem Licht kommenden Strahls der Richtung `von` und der Farbe `f` zurück, der bei einem Schnittpunkt `p` in Richtung `nach` reflektiert wird.

Zum Debuggen ist es sinnvoll, noch eine Informations–Methode anzulegen:

- `Beschreibe()` – Gibt die wichtigsten Eigenschaften des aktuellen Objekts auf die Standardausgabe aus. Auch diese Methode muss virtuell und nach Möglichkeit abstrakt sein.

Beachte

- Diese Klasse kann nicht instantiiert werden (man kann keine Variable von ihr anlegen). Testen ist insofern etwas schwierig; Sie müssen schon ein konkretes Objekt (Kugel) von der Objekt–Klasse ableiten.

6.6 Aufgabe 6: Implementierung eines `Kugel`–Objekts

Schreiben Sie eine Klasse, die eine Kugel repräsentiert. Sie muss von der Objekt–Klasse abgeleitet sein. Das Hauptproblem ist (wie sie sehen werden), den Schnittpunkt der Kugel mit einer Geraden zu bestimmen.

Konstruktor

Folgende Konstruktoren sollen unterstützt werden:

- `Kugel(Vektor pos, Farbe f, double radius, double refl_coeff, double refl_param)` – Initialisiert alle internen Parameter:
  - `pos` – Die Kugelposition (Mittelpunkt).
  - `f` – Die Kugelfarbe.
  - `radius` – Der Kugelradius.
  - refl_coeff – Der Reflektionskoeffizient der Kugel.
  - refl_param – Der Reflektionsparameter `n` der Kugel.
- `Kugel(istream &file)` – Initialisierung der Parameter aus einer Datei `file`. Verwenden Sie dazu die Stream–Operatoren für Vektoren, Farben und Zahlen.

Methoden

Es kommen zur Objekt–Klasse keine neuen Methoden hinzu; allerdings müssen alle virtuellen Methoden (die ja abstrakt=nicht implementiert sind) überschrieben werden. Hier ein paar Bemerkungen dazu:

- `Schneide(Strahl von)` – Diese Methode soll den Schnittpunkt des Strahls mit der Kugel bestimmen. Dazu überlege man sich:
  - Wann trifft der Strahl sicher nicht die Kugel? Wenn sein kleinster Abstand größer als der Radius ist. Diesen kann man durch eine Projektion des Mittelpunktes auf die Gerade finden. Wenn er nicht trifft, ist man fertig (return -1).
  - Hat man diese Projektion, muss man mit elementarer Geometrie von Projektionsabstand und Radius den Abstand zu den zwei Schnittpunkten bestimmen. Welcher von diesen der richtige ist, hängt davon ab, ob der Startpunkt in oder außerhalb der Kugel liegt. Abb. 10 verdeutlicht die Geometrie.

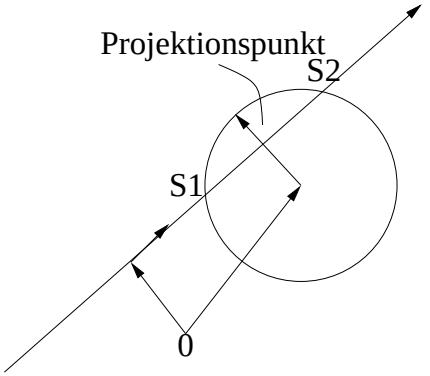

Abbildung 10: Bestimmung des Schnittpunktes einer Geraden mit einer Kugel

- `Beschreibe()` – Diese Methode sollte die Kugelposition, die Farbe und den Radius ausgeben.
- `Normale()` – Die Normale einer Kugel ist der (normierte Vektor) von dem Kugelmittelpunkt zum Schnittpunkt, der ja gegeben ist.
- `Streue()` – Hier spielt wie bei allen Körpern nur der Kosinus zwischen dem Richtungsvektor der Geraden und der Normale des Körpers eine Rolle. Dieser lässt sich leicht über das Skalarprodukt gewinnen (Sofern die Vektoren normiert sind). Man sollte dafür sorgen, dass der Kosinus nicht kleiner Null wird (auch wenn das nicht geschehen sollte, weil dies ein Winkel größer 90° entspricht).
- `Reflektiere_Objekt` – Hier wird die übergebene Farbe nur mit dem Reflektionskoeffizienten multipliziert.

- `Reflektiere_Licht()` – Das Problem ist nur, den Kosinus zwischen dem tatsächlichen Austrittsstrahl und dem reflektierten Strahl zu bestimmen. Letzteren gewinnt man mit dem Spiegeloperator `%` an der Normalen. Die Potenz mit dem Reflektionsparameter kann man mit der mathematischen Operation `pow(x,y)` berechnen.

Beachte

- Testen Sie die Kugelklasse, indem Sie einen Strahl mit einer Kugel schneiden. Lesen Sie die Parameter von der Standardeingabe (`cin`) ein. Machen Sie sich die verschiedenen Fälle plausibel.

6.7 Aufgabe 7: Die Benutzer-Oberfläche mit der QT-Bibliothek

Für die Ausgabe benötigen wir eine Bibliothek mit graphischen Möglichkeiten. Hier soll eine solche Bibliothek vorgestellt werden, die für das UNIX–Betriebssystem Linux recht populär ist (der KDE ist damit entwickelt worden), nämlich dem *QT Widget Set*[2].

Der *QT Widget-Set* ist eine Bibliothek für C++, mit der eine Applikation (ein Programm) für eine graphische Oberfläche (GUI) geschrieben werden kann.

Diese Bibliothek enthält viele Klassen, mit welchen Fensterobjekte verschiedener Art (Bild, Knopf, Schieber etc.) dargestellt werden können. Einige Klassen dienen auch speziell zur Ausgabe von Grafik.

Für ein einfaches lauffähiges GUI–Programm (GUI — Graphical User Interface) geschrieben werden kann, sind folgende Klassen nötig:

- **QApplication** – repräsentiert das Programm. Die Ausgabe wird mit Aufruf der Methode `exec gestartet`. Wichtige Methoden:
  - `QApplication(int argc, char *argv[])` - Der Konstruktor von QApplication. Ihm werden die Kommandozeilenparameter der `main()` – Funktion übergeben. Dadurch kann die Applikation mit speziellen QT Kommandozeilenoptionen gestartet werden.
  - `setMainWidget(QWidget *w)` - legt fest, dass das Fenster `w` das Hauptfenster ist (wenn dieses geschlossen wird, wird das Programm beendet).
  - `show()` - Sorgt dafür, dass das Fenster auch tatsächlich sichtbar ist.
  - `int exec()` - Startet die Applikation, d.h. öffnet alle sichtbaren Fenster und läßt auf Ereignisse (s.u.) reagieren. Wenn Fehler auftreten, wird ein von 0 verschiedener Wert zurückgegeben und die Applikation beendet (der Wert kann dann in der Main–Funktion zurückgegeben werden).
- **QWidget** – Eine Klasse, mit der ein (leeres) Fenster bzw. Unterfenster implementiert werden können. Wichtige Methoden:
  - `setGeometry(int x, int y, int w, int h)` - Setzt die Größe (Breite `w` und Höhe `h`) und Position (x,y) des Fensters.
- **QPushButton** – Stellt einen Knopf dar.
  - `QPushbutton(char *text, QWidget *F)` - Konstruktor, mit dem ein Knopf mit der Beschriftung `text` in dem Fenster `F` erscheint.
  - `setGeometry(int x, int y, int w, int h)` - Setzt die Größe (Breite `w` und Höhe `h`) und Position (x,y) innerhalb des Fensters.

QT compilieren

Damit ein Programm mit QT korrekt compiliert werden kann, müssen folgende Dinge beachtet werden:

- Die Header–Dateien der QT–Library liegen nicht in dem Standard–Verzeichnis. Der Pfad muss explizit angegeben werden:

[2] Diese Bibliothek ist für verschiedene Betriebssysteme geeignet; sie kann auch mit Windows verwendet werden.

```
-I$QTDIR/include
```

- Die QT–Library liegt nicht in dem Standard–Verzeichnis. Der Pfad muss ebenfalls explizit angegeben werden:

```
-L$QTDIR/lib
```

- Die Library muss auch verwendet werden (analog der mathematischen Library). Dies geht mit der Option `-lq`.

Beispiel:

```
g++ -o bla -I$QTDIR/include -lqt -L$QTDIR/lib bla.cc
```

Einfaches Beispielprogramm

```
1  //QT - Beispielprogramm
2
3  /*QT Library Einbinden:*/
4  #include <qapplication.h>
5  #include <qpushbutton.h>
6
7
8  int main(int argc, char * argv[])
9  {
10
11   // Objekt für die Applikation
12   QApplication beispiel (argc, argv);
13
14
15   // Ausgabefenster
16   // Dieses Objekt ist auf dem Heap angelegt.
17   QWidget *Ausgabe = new QWidget();
18
19   // Ausgabegröße setzen:
20   Ausgabe -> setGeometry(400,400,300,300);
21
22   // Ein Knopf, um das Programm zu beenden:
23   QPushButton *quitKnopf =
24     new QPushButton("Knopf", Ausgabe);
25
26   // Größe setzen:
27   quitKnopf -> setGeometry(0,0,50,30);
28
29   // Hauptfenster festlegen:
30   beispiel.setMainWidget(Ausgabe);
31
32   // Hauptfenster zeigen:
33   Ausgabe -> show();
34
35   // Programm starten
36   return beispiel.exec();
37 }
```

Ereignisbehandlung

Anders als bei textbasierten Programmen (einfache C/C++ – Programme), welche schrittweise Daten einlesen, bearbeiten und dann wieder ausgeben können, ist der Umgang mit einer graphischen Applikation **Ereignis-Orientiert**. Das bedeutet, dass bei einer solchen Applikation jederzeit ein Knopf gedrückt werden kann oder eine Taste betätigt wurde, worauf eine Aktion gestartet werden muss. Damit die Applikation nicht nur damit beschäftigt ist, etwa den Zustand der Knöpfe (gedrückt/nicht gedrückt) abzufragen, können spezielle Funktionen mit den Ereignissen (etwa: Knopf wurde gedrückt) verknüpft werden.

In QT geschieht dies dadurch, dass eine Methode in einer Klasse als „Steckdose“ (englisch `slot`) deklariert wird. Beim Start des Programmes wird festgelegt, welches Knopf–Ereignis mit dieser Steckdose verbunden wird. Die Syntax für diese Verbindung ist:

```
QObject::connect(Knopf, SIGNAL(clicked()),
                 Widget, SLOT(Slot-Methode));
```

Dabei wird das Ereignis (Signal) `clicked()` (Knopf wurde gedrückt) eines Knopfes *Knopf* mit einer Methode *Slot-Methode* eines bestimmten graphischen Objektes (*Widget*) verbunden.

Wenn der Knopf gedrückt wird, wird die entsprechende *Slot-Methode* ausgeführt.

Damit dieser Mechanismus funktioniert, müssen folgende Dinge beachtet werden:

- Die Klasse, in der ein *Slot* definiert wurde, muß von der Klasse `QWidget` abgeleitet sein.
- In dieser Klasse muss als erstes das Wort `Q_OBJECT` stehen (ohne Semikolon).
- Die Datei, in der diese Klasse definiert wird, muß durch einen speziellen QT-Compiler namens **moc** (in `$QTDIR/bin`) geschickt werden. (Erst dadurch werden die Funktionen mit Knopf-Ereignissen verknüpfbar). Dessen Produkt muss ebenfalls mit dem C++ – Compiler übersetzt und zum Programm gelinkt werden.

Das Rahmenprogramm

1. Kopieren Sie sich die Dateien `beispiel.cc` und `beispiel.h` aus dem Verzeichnis
   `~goeh/public/qt/` in ihr Arbeitsverzeichnis. Dies geht mit dem UNIX-Befehl
   `cp` *Datei* ..
2. Übersetzen Sie das Programm und starten Sie es.
3. Kopieren Sie das `makefile` in ihr Arbeitsverzeichnis, und übersetzen Sie das Programm mit Hilfe von `make`.
4. Ändern Sie die Größe der Applikation.
5. Ändern Sie die Knopfbeschriftung.
6. Fügen Sie einen neuen Knopf zur Applikation, welcher das Fenster verkleinert.

6.8 Aufgabe 8: Die graphische Ausgabe

Um unter QT geometrische Figuren zeichnen zu können, gibt es die Klasse `QPainter`. Mit ihr können Punkte, Linien, Rechtecke, Kreise und kompliziertere Figuren auf ein Fenster gezeichnet werden. Die Klasse ist mit einbinden von `#include <qpainter.h>` verfügbar. Der Ablauf beim Zeichnen ist wie folgt:

1. Starte den Zeichenvorgang mit der Methode `begin(F)` der Klasse `QPainter` auf dem Fenster `F`. (`F` ist ein Pointer auf die Instanz des Ausgabefensters, kann also in unserem Falle mit **this** für die eigene Klasseninstanz dargestellt werden.)
2. Führe die gewünschten Zeichenmethoden aus.
3. Rufe die Methode `end()` der Klasse `QPainter` auf. Erst dann erscheinen die Figuren auf dem Fenster.

Wichtige Zeichenmethoden sind:

- `void drawPoint ( int x, int y )` - Zeichnet einen Punkt bei x,y
- `void drawLine ( int x1, int y1, int x2, int y2 )` - zieht eine Linie von (x1,y1) nach (x2,y2)
- `void drawRect ( int x, int y, int w, int h )` - Zeichnet ein Rechteck mit der Weite w, Höhe h an die Stelle (x,y).

Wir benötigen eigentlich nur die `drawPoint()` – Methode, da ja unser Bild aus einzelnen Punkten zusammengesetzt wird.

Die Strich/Füllfarbe kann mit den folgenden Methoden gesetzt werden:

- `setPen(QColor c)` - Setzt die Farbe der Striche (Punkte) auf c.
- `setBrush(QColor c)` - Setzt die Farbe der Füllung auf c.

Die Klasse `QColor` hat einen Konstruktor, dem `int`–Werte für Rot,Grün und Blau übergeben werden können. Diese dürfen den Wert 255 nicht überschreiten (!). Es gibt aber auch Konstanten als Farbe: `black, white, darkGray, gray, lightGray, red, green, blue, cyan, magenta, yellow, darkRed, darkGreen, darkBlue, darkCyan, darkMagenta, darkYellow`.

Hier ein Beispiel, wie eine Linie in der eigenen Klasse (die von QWidget abgeleitet sein muss) ausgegeben wird:

```
void Fenster::zeichne() // Zeichenmethode der Klasse Fenster

   QPainter Stift;
   Stift.begin(this);                // zeichne auf dem eigenen Fenster
   Stift.setPen(QPainter::red);      // Roter Stift,
   Stift.setBrush(QPainter::white); // Weisser Hintergrund
   Stift.drawLine(5,5,100,100);      // Ziehe eine rote Linie
   Stift.end();                      // Und beende damit die Ausgabe-Operation
```

Aufgabe

Leiten Sie von der Klasse `QWidget` eine Klasse `Bild` ab, die die Bildausgabe übernimmt. Binden Sie diese Klasse in das Beispiel–Programm der vorigen Aufgabe ein. Die Klasse soll als Konstruktor einen Dateinamen übergeben bekommen, der festlegt, von welcher Datei die Informationen über das zu berechnende Bild kommen.

Konstruktor

Folgende Konstruktoren sollen unterstützt werden:

- `Bild()` – Der Default–Konstruktor. Die Datei heiße dann `input.dat`.
- `Bild(char *filename)` – Initialisierung den Dateinamen mit `filename`.

Methoden

Folgende Methode soll als *Slot* eingebunden werden. Sie soll im Beispielprogramm auf Knopfdruck ausgeführt werden:

- `zeichne()` – Zeichne auf das eigene Widget ein Bild. Das Bild soll im fertigen Programm das über Raytracing bestimmte Bild sein. Da hierzu noch einige Operationen fehlen, soll die Bildausgabe einfarbig sein. Zeichnen Sie hierzu Punkt für Punkt das Bild mit einer Farbe voll, die aus der Datei gelesen wird, deren Namen mit dem Konstruktor übergeben wurde (Farben können mit dem überladenen `<<`–Operator eingelesen werden). Die Farbkomponenten, die Werte zwischen 0..1 haben, müssen dazu mit 255 multipliziert werden und an den Konstruktor von `QColor` übergeben werden.

Beachte

- Der Nullpunkt der Graphikausgabe befindet sich links oben; wenn wir Bilder berechnen wollen, muss dies berücksichtigt werden, damit das Ergebnis nicht auf dem Kopf steht.
- Ändern Sie auf jeden Fall den Namen der Beispieldatei, und passen Sie das `makefile` an, so dass es alle relevanten Programmteile abdeckt.

6.9 Aufgabe 9: Strahlberechnungen in einer `Szene`–Klasse

Nun haben wir fast alles, was zu einem Raytracer gehört – die Ausgabe, die Objekte als Klassen, die Lichter, eine Basisbibliothek; was fehlt, ist die eigentliche Strahlberechnung. Diese soll in einer eigenen Klasse durchgeführt werden — die `Szene`–Klasse.

Diese Klasse muss viel leisten: Sie muss die Datei einlesen, die Objekte anlegen und für einen bestimmten Punkt die Strahlberechnung durchführen. Dennoch sieht ihre Schnittstelle vergleichsweise harmlos aus:

Konstruktor

Folgende Konstruktoren sollen unterstützt werden:

- `Szene(char *filename)` – Lies die Szeneninformation aus der gegebenen Datei. Diese muss in geeigneter Weise gespeichert werden. Glücklicherweise haben wir dafür schon gute Vorarbeit geleistet:
  - `Objekte` werden in einer verzeigerten Liste gespeichert, deren Startpunkt in der `Szene`–Klasse als Attribut gespeichert wird. Verwenden Sie die `anhaengen()`–Methode, um an neue Objekte bereits bestehende Objekte anzuhängen (Die Reihenfolge ist nicht relevant). Wie lesen wir die Objekte ein? Da ja die Kugel–Klasse einen Konstruktor hat, der eine Kugel aus einer Datei initialisieren kann, muss vorher nur noch festgestellt werden, dass jetzt eine Kugel eingelesen werden soll. Dazu soll der Buchstabe **k** verwendet werden.
  - `Lichter` werden ebenfalls in einer verzeigerten Liste abgelegt, die auch als Attribut in der `Szene`–Klasse enthalten sein muss. Auch Lichter können mit dem Konstruktor komplett aus einer Datei gelesen werden. Hier soll der Buchstabe **l** als Kennung dienen.

  Damit sind die Listen der Objekte und Lichter erstellt. Vorher sollte noch eine Hintergrundfarbe und die Position des Betrachters eingelesen werden, also eine Farbe und ein Vektor. Dies ist mit den Stream–Operatoren dieser Klassen leicht möglich.

  Mit der Betrachterposition kann leicht die Perspektive geändert werden, und mit dem Hintergrund kann ein Himmel oder sonstige Schattierung simuliert werden. Default seien die Positionen `[0,0,200]` und die Farbe Schwarz.

Methoden

Es gibt nur eine nach außen sichtbare Methode, um ein Bild zu berechnen:

- `Farbe berechne(int x, int y, bool perspektivisch=1)` – Berechnet für den Bildpunkt `(x,y)` (`z=0`) für eine Szene die Farbe. Wenn die Szene perspektivisch sein soll, wird von einem Betrachter–Standpunkt aus die Strahlverfolgung vorgenommen, sonst für alle Punkte in die $-z$–Richtung.

Private Methoden

Die eigentliche Strahlverfolgung sollte in der `berechne()`–Methode stattfinden. Weil diese Aufgabe recht umfangreich ist, teilen wir sie in mehrere private Methoden auf, von denen eine rekursiv aufgerufen wird:

- `Objekt *suche_schnitt(Strahl s, double &a)` – Suche für eine Gerade `s` das am nächsten liegende Objekt, das geschnitten wird. Dieses Objekt soll zurückgegeben werden. `a` ist der Geradenparameter, für den das Objekt geschnitten wurde.

  Das Vorgehen dabei ist einfach: man gehe durch alle Objekte (mit der Methode `weiter()` kommt man an ein nachfolgendes Objekt), rufe die Methode `Schneide()` und merke sich das Objekt, für das der mit `Schneide()` bestimmte Geradenparameter am kleinsten ist.

  Wenn kein Objekt gefunden wurde, soll die Methode `NULL` und den Geradenparameter `a=-1` zurückgeben (man beachte die Referenz für `a`).

- `Farbe folge_strahl(Strahl s)` – Diese Methode folgt nun (endlich) dem Strahl `s` und bestimmt dessen Farbe. Dazu muss sie:
  1. Das nächste geschnittene Objekt bestimmen. Wenn keines gefunden wurde, sind wir fertig, und die Farbe ist die Hintergrundfarbe.
  2. Mit Hilfe des beim Schneiden gefundenen Geradenparameters den Schnittpunkt bestimmen.
  3. Den Strahl `s` reflektieren. Den dazu nötigen Normalenvektor bekommen wir von dem geschnittenen Objekt unter Angabe des Schnittpunktes. Die neue Richtung kann mit dem `%`–Operator für Vektoren bestimmt werden.
  4. Bestimme die Farbe des reflektierten Strahls. Dazu wird diese Methode `folge_strahl()` erneut mit dem neuen Strahl aufgerufen.
  5. Zusätzlich muss noch die Farbe des von den Lichtern gestreute und glanzreflektierte Licht bestimmt werden. Dazu wird eine weitere Methode `folge_licht()` verwendet.
  6. Die Farben der letzten beiden Schritte wird addiert und zurückgegeben.
- `Farbe folge_licht(Vektor nach, Vektor bei, Objekt *obj)` – Diese Methode bestimmt, welche Farbe der Lichter von einem Objekt `obj` bei einem Schnittpunkt `bei` in die Richtung `nach` gestreut/glanzreflektiert wird. Es werden dabei nur Strahlen von Lichtern berücksichtigt; also keine reine Reflektion. Die Schritte sind wie folgt:
  1. Gehe durch alle Lichter und addieren die Farb–Ergebnisse.
  2. Bestimme die Richtung des Objekt–Schnittpunktes zu dem jeweiligen Licht.
  3. Untersuche, ob ein Objekt dazwischen liegt. Wenn ja, dann ist das Licht von dem Objekt abgeschattet und muss nicht berücksichtigt werden.
  4. Wenn das Licht sichtbar ist, bestimme die Streuung und die Glanzreflektion (die dafür nötigen Richtungen sind schon bestimmt) unter Zuhilfenahme des Objektes `obj`, von dem ausgegangen wurde.
  5. Addiere diese Farben zum Ergebnis und gib dieses zurück.

Damit ist alles beieinander, was wir benötigen. Die Methode `berechne()` ruft die Methode `folge_strahl()` für eine Gerade vom Beobachter (wenn perspektivisch) oder vom Bildpunkt aus (wenn nicht perspektivisch) zum Bildpunkt `(x,y)`, und gibt den Farbwert zurück. In der `Bild`–Klasse wird dann die Methode `berechne()` aufgerufen.

Beachte

- Machen Sie sich den Ablauf der obigen Algorithmen mit einer Skizze auf einem Blatt Papier klar!
- Achten Sie darauf, dass beim Schneiden der Geradenparameter auch in der Nähe von Null liegen kann. Dies bedeutet, dass sie gerade das eigene Objekt geschnitten haben, von dem Sie ausgegangen sind. Vermeiden Sie dies durch eine geeignete Abfrage.
- Bauen sie eine Instanz (Variable) der Szene–Klasse in die `Bild`–Klasse ein, die die Methode `berechne()` aufruft. Nun können Sie Bilder von Kugeln malen lassen. Testen Sie das Ergebnis vorsichtig mit verschiedenen Kugeln und verzweifeln Sie nicht, wenn das Ergebnis nicht auf Anhieb befriedigt.

6.10 Aufgabe 10: Implementierung eines `Ebenen`–Objektes

Schreiben Sie eine Klasse `Ebene`, die eine Ebene repräsentiert. Sie muss von der Objekt–Klasse abgeleitet sein.

Konstruktor

Folgende Konstruktoren sollen unterstützt werden:

- `Ebene(Vektor n, Farbe f, double dist, double refl_coeff, double refl_param)` – Initialisiert alle internen Parameter:
  - `n` – Der Normalenvektor der Ebene.
  - `f` – Die Ebenenfarbe
  - `dist`– Der Abstand der Ebene zum Ursprung
  - refl_coeff – Der Reflektionskoeffizient der Ebene.
  - refl_param – Der Reflektionsparameter der Ebene.
- `Ebene(istream &file)` – Initialisierung der Parameter aus einer Datei `file`.

Methoden

- `Schneide(Strahl von)` – Diese Methode soll den Schnittpunkt des Strahls mit der Ebene bestimmen. Dies ist mit Hilfe der Ebenengleichung durch Einsetzen der Geradengleichung leicht zu bestimmen (wie?). Wenn der Geradenparameter negativ ist, wird die Ebene nicht geschnitten.
- `Beschreibe()` – Diese Methode sollte den Normalenvektor, die Farbe und den Abstand ausgeben.
- `Normale()` – Die Normale der Ebene, die als Attribut gespeichert wurde.
- `Streue()` – Auch hier spielt wie bei allen Körpern nur der Kosinus zwischen dem Richtungsvektor der Geraden und der Normale des Körpers eine Rolle, der über das Skalarprodukt bestimmt werden kann.
- `Reflektiere_Objekt` – Hier wird die übergebene Farbe nur mit dem Reflektionskoeffizienten multipliziert.
- `Reflektiere_Licht()` – Das Problem ist nur, den Kosinus zwischen dem tatsächlichen Austrittsstrahl und dem reflektierten Strahl zu bestimmen. Letzteren gewinnt man mit dem Spiegeloperator `%` an der Normalen. Die Potenz mit dem Reflektionsparameter kann man mit der mathematischen Operation `pow(x,y)` berechnen[3].

Beachte

- Fügen Sie die Ebenen-Klasse zur Szenenbestimmung hinzu. Verwenden Sie als Kennung den Buchstaben **e**.
- Achten Sie auf Fließkommaprobleme, wenn Sie den Schnittpunkt berechnen. Ein Aufpunkt kann auch leicht etwas unter der Ebene sein und dennoch nicht diese Ebene schneiden.
- Möglicherweise sehen Sie unerwünschte Muster auf der Ebene. Diese können von dem diskreten Farbschema kommen. Abhilfe verspricht möglicherweise ein Zufallsgenerator bei der Streuungsberechnung.

7 Ausblick und Erweiterungsmöglichkeiten

Der entwickelte Raytracer bietet schon einen guten Einblick, wie Bilder errechnet werden können und kann auch komplexe Sachverhalte (Spiegel im Spiegel etc.) berücksichtigen. Er hat genügend Potential, um erweitert werden zu können. Hauptmanko dürfte das geringe Repertoire an Objekten sein. Es fehlen auch einige Features, die bei jedem etwas anspruchsvolleren Raytracer zu finden sind, etwa das Bearbeiten von transparenten Objekten, bei denen neben einem reflektierten Strahl noch der gebrochene Strahl verfolgt wird.

7.1 Bilder abspeichern

Zudem wäre es gut, wenn die berechneten Bilder auch abgespeichert werden können oder wenigstens etwas persistenter sind. Dies ist mit Hilfe des

[3] Es wäre vielleicht geschickt gewesen, diese Berechnung aus den Objekten herauszuhalten.

QT-Widget-Sets leicht zu bewerkstelligen (man zeichnet in eine Bitmap statt in das Fenster), geht aber über das Projekt hinaus. Bei Interesse möge man die QT-Referenzen verwenden.

7.2 Weitere Objekte

Wie schon erwähnt sind Kugel und Ebene etwas dürftig. Allerdings ist es nun relativ einfach, eine Palette weiterer Objekte zu definieren, vielfach, indem die Ebene erweitert wird[4]. Beispiele:

- `Schachbrett` – Man untersucht den Schnitt einer Ebene; wenn der Schnittpunkt modulo einer bestimmten Größe in einem bestimmten Bereich ist, wird eine Farbe gewählt; sonst die andere.
- `Dreiecke` – Man untersucht auf Schnitt mit der Ebene, die von dem Dreieck aufgespannt wird (der Normalenvektor lässt sich mit dem Kreuzprodukt bestimmen). Es genügt dann, mit einfachen geometrischen Betrachtungen zu bestimmen, ob sich der Schnittpunkt **in** oder **außerhalb** der Ebene befindet.
- `Polygone` – Beliebige gerade Flächen lassen sich aus Dreiecken zusammensetzen, die wir gerade behandelt haben. Es gibt aber auch die Möglichkeit, bei Polygonen auf einer Ebene direkt zu bestimmen, ob sich ein Punkt innerhalb oder außerhalb der Fläche befindet (Schnitt nach außen, Schnittkanten zählen).
- `Würfel` – Würfel lassen sich aus Dreiecken/Polygonen zusammensetzen.
- `Zylinder` – Werden wie Kugeln behandelt, aber mit einem Abstand zu einer Geraden.
- `Kegel, Splines, etc.`

Weiterhin wäre es sinnvoll, komplexe Körper drehen zu können. Dazu müssen die Koordinaten eines Körpers in geeigneter Weise (Matrize) in neue Koordinaten umgerechnet werden. Günstig wäre hierfür eine Matrizen-Klasse, die dies durchführt.

[4]Um Sie zu beruhigen: Dies müssen Sie **nicht** programmieren.

7.3 Textur

Die Oberfläche unserer Körper ist unrealistisch einförmig; wir können tatsächlich ja nur die Farbe und einige grundlegende Eigenschaften festlegen. Es wäre günstig, wenn man ein Bild um die Körper „wickeln“ könnte, so dass eine realistischere Oberfläche entsteht. Diese Möglichkeit wird *Textur* genannt. Dabei wird für jeden Körper eine Abbildung von dem Körper auf eine rechteckiges Bild festgelegt. Dann kann für jeden Aufpunkt des Körpers ein anderer (Bild)-Punkt verwendet werden.

Beispielsweise könnte man Kugelkoordinaten leicht auf ein rechteckiges Bild umrechnen, um eine Weltkarte auf die Kugel zu projizieren.

7.4 Antialiasing

Die Kanten eines Körpers haben in den Bildern häufig eine gezackte Struktur; dieses Problem wird als *Aliasing* bezeichnet. Um es zu umgehen, kann man jedes Pixel im Bild in Unterpixel unterteilen, um dann dem fertigen Pixel einen Mittelwert mitzugeben. Der Vorteil ist, dass die Zacken ausgewaschen werden; der Nachteil ist die natürlich vielfach höhere Rechenzeit.

7.5 Effizienz

Dies ist nämlich ein Hauptproblem der Bildberechnung: dass die Berechnungen sehr viel Rechenzeit benötigen (vor ca. 7 Jahren wäre ein solches Projekt unrealistisch, da die Bildberechnung 12 Stunden dauern würde). Man kann einige Dinge betrachten, die die Bildberechnung beschleunigen können:

- Untersuche durch „einkasteln“, ob ein Körper überhaupt von einem Strahl getroffen werden kann.
- Beschränkte die maximale Strahlverfolgung.
- Untersuche anhand bereits berechneten Strahlen, welcher Körper als Schnittpunkt überhaupt in Frage kommt (das ist allerdings eine komplexe Wissenschaft für sich).

Inhaltsverzeichnis