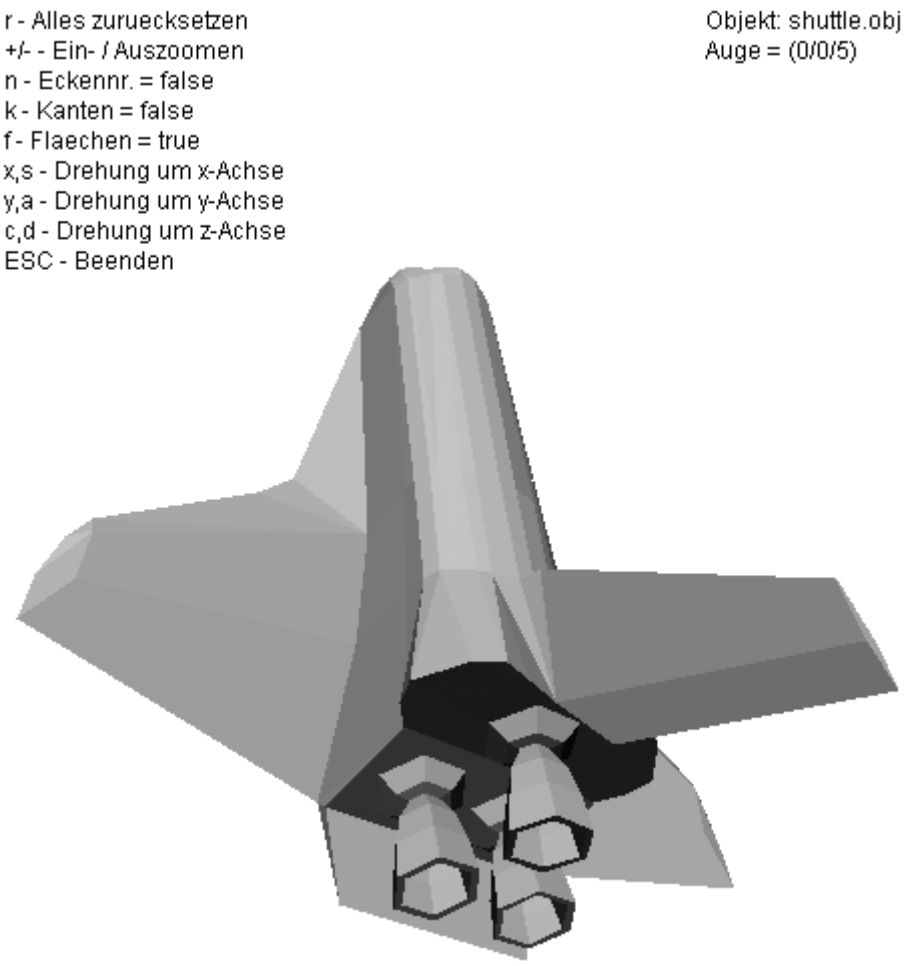
3D-Graphikprogrammierung - Ein Leitprogramm mit Java



Verfasst von Beat Trachsler, KZO Wetzikon

Version 01.04.2010

Schulbereich, Stufe

Gymnasium (letztes oder vorletztes Jahr vor der Maturitätsprüfung), Fachhochschule

Vorkenntnisse

- Grundlagen der Vektorgeometrie
- Programmiererfahrung mit Java

Bearbeitungsdauer

10 - 12 Lektionen (am besten 5 - 6 Doppellektionen)

Inhaltsverzeichnis

Einleitung

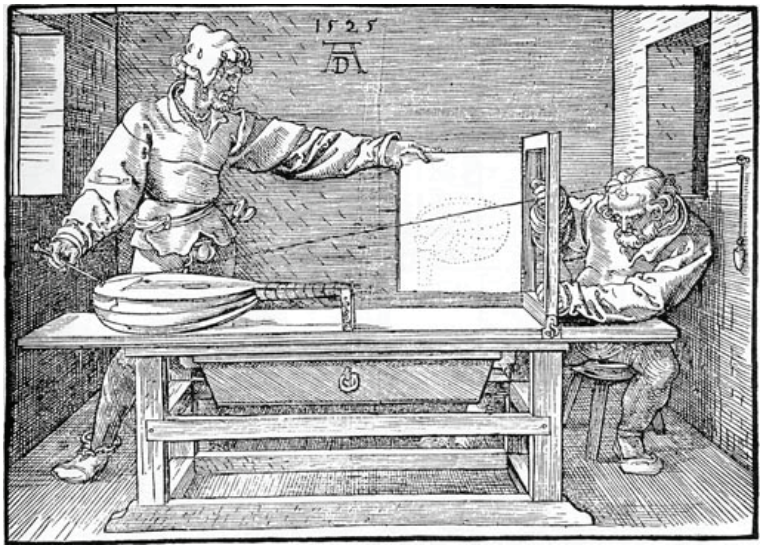


Abbildung 1: Der Zeichner der Laute, Albrecht Dürer (1525)

Das Bild von Albrecht Dürer (Abbildung 1) zeigt die Idee der perspektivischen Abbildung sehr schön: Von einem fixen Punkt aus – im Bild ein Nagel in der Wand auf der rechten Seite – wird ein Faden zum Objekt, das abgebildet werden soll, gespannt. Bei Dürer ist dies eine Laute[1]. Der Faden repräsentiert den Projektionsstrahl der perspektivischen Abbildung. Zwischen der Laute und dem Nagel in der Wand befindet sich die Bildebene in Form eines Holzrahmens, in den Dürer ein engmaschiges Netz aus Fäden gespannt hat. So kann sein Assistent den Durchstosspunkt des Projektionsstrahls durch die Bildebene genau bestimmen und hinterher auf dem Zeichenblatt festhalten.

Der grösste Vorteil von perspektivischen Bildern im Gegensatz zu Darstellungen in der Parallelprojektion ist deren realistische Wirkung auf den Betrachter. Dies liegt daran, dass der Gesichtsinn im menschlichen Auge in guter Näherung ebenfalls ein perspektivisches Abbild der Umgebung des Betrachters erzeugt. Aus diesem Grund beschränken wir uns in diesem Leitprogramm auf die perspektivische Darstellung, wobei sich die meisten der behandelten Ideen direkt auf die Parallelprojektion übertragen lassen. Die Abbildung 2 zeigt das Ziel, welches es zu erreichen gilt: Das dargestellte Ikosaeder lässt sich per Tastatur drehen und skalieren. Neben der aktuellen Darstellung kann auch nur das Drahtgitter oder die Menge der Eckpunkte dargestellt werden. Da das Programm ein gängiges 3D-Dateiformat verwendet, kann man damit auch komplexere Objekte darstellen.

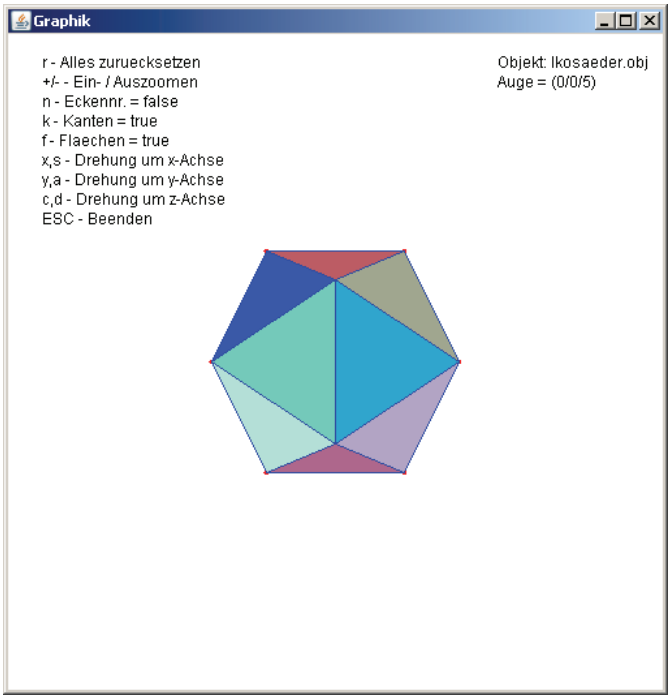


Abbildung 2: Ikosaeder dargestellt mit dem Projektor3D

[1] Die Laute ist ein Saiteninstrument, welches im Europa der Renaissance weit verbreitet war.

Abbildung 3: Virtuelle Realität im Spiel Crysis

In handelsüblichen Computerspielen wie Crysis und Collin McRae Dirt werden nahezu fotorealistische virtuelle Realitäten in Echtzeit dargestellt. Im Gegensatz zum 3D-Kino muss die Berechnung der Bilder dabei zur Laufzeit erfolgen, was die zur Verfügung stehenden Techniken stark einschränkt. Raytracer, bei denen der Strahlengang von Lichtstrahlen über mehrere Reflexionen und Brechungen hinweg verfolgt werden, sind dabei nicht einsetzbar, weil sonst die Berechnung der Bilder mehrere Tage oder gar Wochen dauern würde. Abhilfe schaffen beispielsweise so genannte Voxel. Das sind eine Art 3D-Pixel in Form von Quadern, aus denen sich Szenerien wie jene aus Abbildung 3 zusammensetzen lassen. In der Detailansicht müssen dabei die Oberflächenpolygone schnell gerendert werden. Genau darum geht es in diesem Leitprogramm.

Das Leitprogramm enthält neben der aufbauenden Theorie die folgenden, immer wiederkehrenden Bausteine, welche farblich hervorgehoben sind:

Aufgaben

Die Aufgaben dienen zur schrittweisen Erarbeitung des Lernstoffs. Zu jeder Aufgabe existiert eine detaillierte Beispiellösung auf Seite 23 ff.

Beispiele

Die Beispiele dienen zur Illustration der behandelten Theorie.

Anwendungsbeispiele

Die Anwendungsbeispiele sollen einen Bezug zu virtuellen Realitäten herstellen, wie sie beispielsweise im Computerspiel Crysis eingesetzt werden.

Tabellen

Tabellen dienen zur Erklärung von Datenstrukturen oder Hilfsklassen.

1. Kapitel: Perspektivische Abbildung von Polyedern

Modellierung der Zentralprojektion

Die perspektivische Abbildung oder Zentralprojektion wird definiert durch das Projektionszentrum, den so genannten Augpunkt, von dem die Projektionsstrahlen ausgehen, und durch die Bildebene, auf der die Bildpunkte als Durchstosspunkte der Projektionsstrahlen entstehen. Dies dient als Ausgangspunkt für unser Modell. Damit die Berechnungen möglichst einfach werden, wählen wir als Bildebene die Grundrissebene (xy-Ebene) des dreidimensionalen kartesischen Koordinatensystems. Den Augpunkt wählen wir auf der positiven z-Achse beispielsweise bei $(0 / 0 / 5)$. Daraus ergibt sich die Situation aus Abbildung 4.

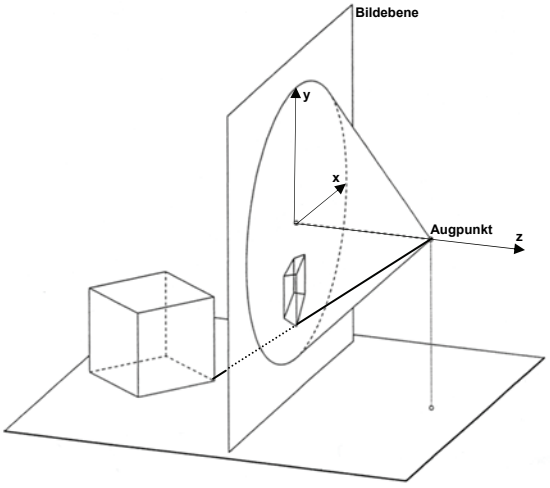


Abbildung 4: Zentralprojektion eines Würfels

Aufgabe 1

Überlegen Sie sich, wie man das Verfahren aus Abbildung 4 mit Java modellieren könnte. Welche Fragen müssen Sie klären, bevor Sie mit dem Programmieren anfangen können? Was brauchen Sie für die Implementierung in Java?

Damit wir die Koordinaten des Würfels einfach ablesen, bzw. ausrechnen können, sollten Sie bei der Wahl seiner Anfangsposition geschickt vorgehen. Im 3. Kapitel werden Sie lernen, wie man die Position des Würfels mit geometrischen Abbildungen jederzeit ändern kann. Auch dafür ist die Wahl der Anfangsposition entscheidend.

In der Computergraphik wird die Anfangsposition der Polyeder stets so gewählt, dass der Mittelpunkt des Polyeders im Nullpunkt des Koordinatensystems liegt. Dadurch wird die Beschreibung von geometrischen Abbildungen wie Drehungen oder Streckungen besonders einfach, da beispielsweise das Streckzentrum dem Nullpunkt des Koordinatensystems entspricht. Der Mittelpunkt eines Polyeders ist dabei nicht automatisch identisch mit seinem Schwerpunkt. Zur Berechnung des Mittelpunkts wählt man jeweils die grösste und die kleinste x-Koordinate, y-Koordinate und z-Koordinate aus, welche an einer Ecke des Polyeders vorkommt. Dadurch erhält man die folgenden sechs Zahlen: $x_k,\ x_g,\ y_k,\ y_g,\ z_k,\ z_g$, wobei der Index k für kleinst und der Index g für grösst steht. Die Koordinaten des Mittelpunkts ergeben sich nun als Durchschnitt der kleinsten und der grössten vorkommenden Koordinate:

$$x_M = \frac{x_k + x_g}{2}, \qquad y_M = \frac{y_k + y_g}{2}, \qquad z_M = \frac{z_k + z_g}{2}$$

Aufgabe 2

Skizzieren Sie von Hand auf einem Blatt Papier einen Würfel, dessen Mittelpunkt im Nullpunkt liegt und dessen Kanten parallel zu den drei Koordinatenachsen verlaufen. Wählen Sie die Koordinaten der Eckpunkte möglichst einfach. Schreiben Sie die Koordinaten der acht Eckpunkte heraus, so dass Sie sie später in Ihrem Programm verwenden können.

Der wichtigste Bestandteil unseres Modells ist die Projektion der Würfelpunkte in die Bildebene. Das ist die Voraussetzung für die Darstellung auf dem Computerbildschirm. Wie oben beschrieben, wählen wir als Bildebene gerade die xy-Ebene, während der Augpunkt Q die Koordinaten $(0 / 0 / 5)$ hat.

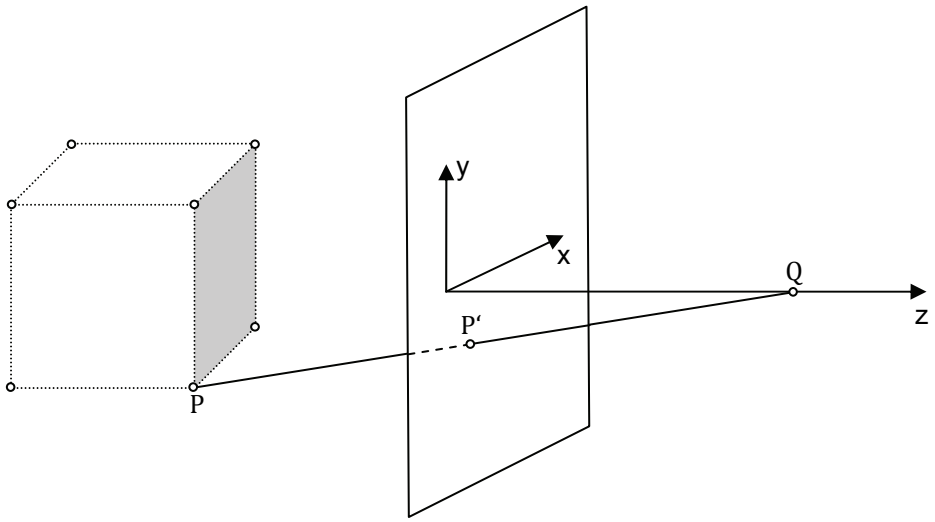


Abbildung 5: Berechnung der Koordinaten des Bildpunktes mittels Zentralprojektion

Aufgabe 3

Berechnen Sie mithilfe Ihrer Kenntnisse aus der Vektorgeometrie die Koordinaten des Bildpunktes einer beliebigen Würfelecke. Überlegen Sie sich anschliessend, wie sich diese Berechnung verallgemeinern lässt. Gesucht ist ein Algorithmus, der ausgehend von den drei Koordinaten der Würfelecke, die x-Koordinate und die y-Koordinate des Bildpunktes in der xy-Ebene berechnet.

Die Graphikumgebung

Um das perspektivische Bild darstellen zu können, brauchen Sie eine Graphikumgebung. Eine besonders intuitive und noch dazu leicht erhältliche Umgebung ist die GPanel-Graphik[1] von Aegidius Plüss (Plüss, Aplu (Aegidius Pluess) Home, 2009), die im Lehrbuch Java exemplarisch (Plüss, Java exemplarisch, 2004) detailliert beschrieben wird. Auf eine vollständige Auflistung der Methoden der Klasse GPanel wird daher verzichtet. Für unsere Zwecke reichen die nachfolgend beschriebenen Methoden aus.

[1] Eine ausführliche Dokumentation (javaDoc) dieser Hilfsklassen findet sich unter http://www.aplu.ch.

Methode	Erklärung
GPanel(double xmin, double xmax, double ymin, double ymax)	Konstruktor, der den Ausschnitt des Koordinatensystems gemäss den vier eingegebenen Parameterwerten definiert.
GPanel()	Default-Konstruktor, der den Ausschnitt des Koordinatensystems auf das Einheitsquadrat [0; 1] × [0; 1] setzt.
move(double x, double y)	Setzt die Zeichenposition auf (x / y).
draw(double x, double y)	Zeichnet eine Linie von der aktuellen Zeichenposition zum Punkt (x / y).
line(double x1, double y1, double x2, double y2)	Zeichnet eine Linie von (x1 / y1) nach (x2 / y2).
point(double x, double y)	Zeichnet den Punkt (x / y).
circle(double r)	Zeichnet an der aktuellen Zeichenposition einen Kreis mit Radius r.
fillCircle(double r)	Zeichnet an der aktuellen Zeichenposition einen gefüllten Kreis mit Radius r.
text(String s)	Schreibt die Zeichenkette s an die aktuelle Zeichenposition.

Tabelle 1: Ausgewählte Methoden der Klasse GPanel

Aufgabe 4

1. Schreiben Sie eine Klasse Projektor3D, welche die Eckpunkte des unter Aufgabe 2 skizzierten Würfel in einem GPanel zeichnet. Überlegen Sie sich vor der Implementierung, welche Methoden Ihre Klasse besitzen soll und wann diese Methoden aufgerufen werden.
2. Beschriften Sie Ihre Eckpunkte mit fortlaufenden Nummern.

Tipp: Benützen Sie für die dreidimensionalen Würfelecken die eigens für dieses Leitprogramm entwickelte Klasse Point (Vgl. Anhang A auf Seite 21).

Aufgabe 5 (Lernkontrolle)

Spielen Sie die notwendigen Schritte zur Darstellung eines Polyeders am Beispiel des Oktaeders nochmals der Reihe nach durch. Erstellen Sie dazu eine Handskizze eines Oktaeders mit Mittelpunkt im Nullpunkt. Stellen Sie daraufhin die Eckpunkte des Oktaeders mit Ihrer Projektor3D-Klasse graphisch dar. Wenn Sie mit einem der obigen Punkte Probleme haben, lesen Sie den entsprechenden Abschnitt in diesem Leitprogramm nochmals durch.

Zusatzaufgabe

Modellieren Sie weitere Polyeder und stellen Sie die Punktmengen in Ihrem Projektor3D graphisch dar.

2. Kapitel: Ein Dateiformat für 3D-Objekte

Das Wavefront Object Format

Das Grundgerüst für die Darstellung von Polyedern ist vorhanden. Wir wollen nun unser Programm so erweitern, dass damit beliebige Polyeder dargestellt werden können. Der einfachste Weg zu diesem Ziel besteht darin, die Koordinaten der Eckpunkte des Polyeders in einer Datei ausserhalb der Java-Klasse zu speichern. So kann man einfach für jedes Polyeder eine neue Datei erstellen, ohne dabei Änderungen am Java-Code vornehmen zu müssen. Grundsätzlich gibt es viele Möglichkeiten, Daten in einer Datei zu speichern. Damit Sie Ihre 3D-Modelle auch mit anderen Graphikprogrammen benützen können, werden wir hier ein bereits existierendes, weit verbreitetes Format einsetzen, das Wavefront Object Format[1]. Dieses Format hat den Vorteil, dass die Koordinaten uncodiert in einer Textdatei gespeichert werden und somit auch von Hand (ohne Spezialprogramm) bearbeitet werden können. Dabei werden die geometrischen Objekte (in unserem Fall Eckpunkte und Flächen) zeilenweise gespeichert, wobei jeweils am Anfang der Zeile mit einem Buchstabencode angegeben werden muss, um welche Art von Objekt es sich handelt. Im Folgenden die für unseren Zweck relevanten Buchstabencodes und die darauffolgenden Objektdefinitionen:

Buchstabencode und Objektdefinition	Erklärung
v *x* *y* *z*	Der Buchstabe **v** steht für **vertex**, das englische Wort für **Eckpunkt**. *x*, *y* und *z* bezeichnen die Koordinaten des Punktes.
f *v1* *v2* *v3* *v4* …	Der Buchstabe **f** steht für **face**, das englische Wort für **Fläche**. *v1*, *v2*, *v3*, *v4*, … bezeichnen die Nummern der Eckpunkte der Fläche in der Reihenfolge, in der sie im aktuellen Dokument aufgelistet sind. Die Nummerierung beginnt bei 1.
# ….	Das Zeichen **#** markiert **Kommentare**.

Tabelle 2: Das Wavefront Object Format

Beispiel 1

Die folgenden Zeilen definieren ein Einheitsquadrat in der Grundrissebene mit Mittelpunkt im Nullpunkt:

```
v    -0.5   -0.5    0
v     0.5   -0.5    0
v     0.5    0.5    0
v    -0.5    0.5    0

f     1      2      3      4
```

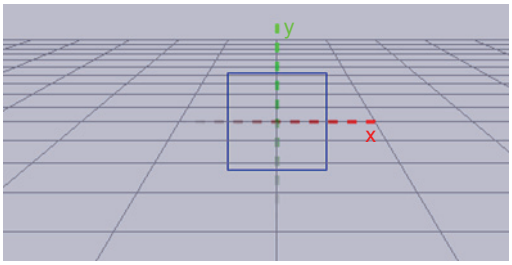


Abbildung 6: Beispielquadrat

Die Abbildung 6 zeigt das Quadrat in einer perspektivischen Darstellung.

[1] Eine Spezifikation finden Sie unter: http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/.

Dateien im Wavefront Object Format haben die Endung obj. Zum Lesen und Bearbeiten dieser Dateien reicht im Prinzip ein Texteditor. Mit einem 3D-Graphikprogramm wie Blender (Blender Foundation, 2009) können solche Dateien jedoch auch interaktiv mittels Graphikwerkzeugen bearbeitet werden.

Aufgabe 1

Erstellen Sie eine Datei Wuerfel.obj im Wavefront Object Format, die den Würfel aus dem 1. Kapitel beschreibt. Im Moment beschränken wir uns auf die Beschreibung der Eckpunkte. Die Darstellung von Kanten und Flächen wird im 4. Kapitel behandelt.

Die Hilfsklassen Polyeder und FileReader

Zum Einlesen der obj-Datei lassen sich die eigens für dieses Leitprogramm entwickelten Klassen Polyeder und FileReader einsetzen. Hinweise zum Einsatz dieser Klassen enthält die beiliegende JavaDoc. Für einen schnellen Zugriff auf die in der Datei gespeicherten Eckpunkte empfiehlt sich das folgende Vorgehen, wobei zu Beginn der Codedatei noch "import ch.beattl.projektor.*;" ergänzt werden muss.

```
FileReader myReader = new FileReader(fileName);
Polyeder polyeder = new Polyeder(myReader);
…
Point P = polyeder.getVertex(i);
```

Dabei enthält die Variable fileName den Dateinamen der .obj-Datei als String, also beispielsweise „Wuerfel.obj“. Die Anzahl der Eckpunkte kann in der Hilfsklasse Polyeder mit der Methode getNumOfVertices() ermittelt werden. Beachten Sie dabei, dass die Nummerierung der Eckpunkte wie in einem Java-Array üblich mit 0 beginnt, nicht mit 1 wie in der .obj-Datei.

Aufgabe 2

Erstellen Sie eine zweite Version der Klasse Projektor3D, in der der Würfel über Ihre obj-Datei aus Aufgabe 1 definiert wird.

Aufgabe 3 (Lernkontrolle)

Spielen Sie die notwendigen Schritte zur Darstellung eines Polyeders mittels .obj-Datei am Beispiel des Oktaeders nochmals der Reihe nach durch. Gehen Sie dabei wie folgt vor:

1. Schreiben Sie die Koordinaten der Eckpunkte in eine obj-Datei namens Oktaeder.obj.
2. Stellen Sie die Eckpunkte des Oktaeders mit Ihrer Projektor3D-Klasse graphisch dar.

Zusatzaufgaben

1. Modellieren Sie weitere Polyeder im Wavefront Object Format und stellen Sie die Punktmengen in Ihrem Projektor3D graphisch dar.
2. Suchen Sie auch auf dem Internet nach geeigneten obj-Dateien.

3. Kapitel: Geometrische Transformationen

Einführung in die Matrizenrechnung

Falls Sie bereits mit Matrizen gearbeitet haben, beispielsweise im Rahmen des Mathematikunterrichts, können Sie diesen Abschnitt überspringen und direkt mit der Skalierung beginnen. Eine Matrix ist ein rechteckiges Schema mit m Zeilen und n Spalten. Eine m × n-Matrix enthält also m · n Elemente.

Beispiel 1

3 × 3-Matrix mit 3 Zeilen und 3 Spalten: $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$.

Matrizen werden in der Informatik sehr häufig verwendet. Der Suchmaschinenanbieter Google benützt sie beispielsweise zur Berechnung des Page-Ranks von Web-Seiten[1]. In unserem Zusammenhang sollen die Matrizen geometrische Transformationen beschreiben. Dazu benötigen wir eine Operation, um Matrizen auf (Orts-)Vektoren anwenden zu können. Diese Operation heisst Matrixmultiplikation und wird für 3 × 3-Matrizen und Vektoren im dreidimensionalen Raum wie folgt definiert:

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} m_{1,1} \cdot x + m_{1,2} \cdot y + m_{1,3} \cdot z \\ m_{2,1} \cdot x + m_{2,2} \cdot y + m_{2,3} \cdot z \\ m_{3,1} \cdot x + m_{3,2} \cdot y + m_{3,3} \cdot z \end{pmatrix}$$

Jede Zeile der Matrix wird also komponentenweise mit dem Vektor malgerechnet, so dass ein resultierender Vektor im dreidimensionalen Raum entsteht[2]. Der Effekt dieser Multiplikation lässt sich am einfachsten anhand von Zahlenbeispielen verdeutlichen.

Beispiel 2

Matrixmultiplikation: $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix}$

Aufgabe 1

Berechnen Sie die folgenden Matrixmultiplikationen:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} =$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} =$$

Was fällt auf? Überlegen Sie sich, woher dieser Effekt kommt?

[1] Siehe HITS-Algorithmus (Kleinberg, 1999), http://www.cs.cornell.edu/home/kleinber/auth.pdf
[2] Abbildungen, welche sich auf diese Weise durch Matrizen beschreiben lassen, heissen linear.

Eine spezielle Bedeutung haben sogenannte Diagonalmatrizen. Bei einer Diagonalmatrix sind alle Elemente, welche nicht auf der Hauptdiagonalen liegen, gleich Null.

Beispiel 3

Diagonalmatrix: $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$

Skalierung

Da bei einer Skalierung jede Komponente des Vektors mit einem Skalar multipliziert wird, lassen sich Skalierungen mit Diagonalmatrizen beschreiben. Sie bewirken Streckungen, bzw. Stauchungen in Richtung der jeweiligen Koordinatenachse. Eine Streckung mit dem Nullpunkt als Streckzentrum wird durch eine Diagonalmatrix mit drei gleichen, von Null verschiedenen Elementen realisiert.

Beispiel 4

Diagonalmatrix für eine Streckung am Nullpunkt des Koordinatensystems mit Faktor 2:

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

Diese Matrix bewirkt eine Verdoppelung der Längen. Sie sehen jetzt auch, wieso wir im 1. Kapitel darauf geachtet haben, dass die Mittelpunkte unserer Polyeder stets im Nullpunkt liegen. Die nebenstehende Abbildung zeigt die Skalierung eines Dreiecks mit Faktor 2.

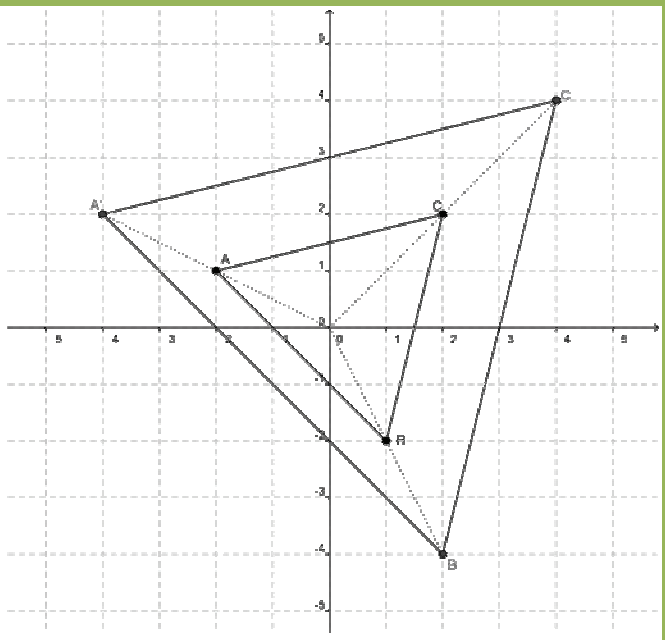


Abbildung 7: Skalierung mit Faktor 2

Aufgabe 2

Erweitern Sie Ihre Klasse Projektor3D so, dass der Benutzer mit den Tastaturtasten + und – ein- und auszoomen kann. Implementieren Sie die Zoom-Funktion im dreidimensionalen Koordinatensystem als Streckung am Nullpunkt.

Tipp: Um Tastaturtasten abzufragen, benötigen Sie die GPanel-Methode getKeyWait(), welche das Tastaturzeichen im char-Datentyp zurückgibt. Der Zugriff auf die Eckpunkte des Polyeders erfolgt mit der Methode getVertex(int) der Klasse Polyeder. Zur Implementierung der Matrizen können Sie die Klasse Matrix verwenden. Eine Beschreibung finden Sie im Anhang A auf Seite 21.

Drehungen um die Koordinatenachsen

Der Einfachheit halber beschränken wir uns in diesem Leitprogramm auf Drehungen um Koordinatenachsen. Mithilfe der Matrixmultiplikation von Matrizen können damit aber auch Drehungen um beliebige Achsen, die den Nullpunkt enthalten, realisiert werden. Die Drehung um die z-Achse lässt sich auf eine Drehung um den Nullpunkt in der xy-Ebene zurückführen, da sich die z-Koordinate bei dieser Abbildung nicht ändert. Eine solche Drehung um den Nullpunkt mit Drehwinkel φ kann mithilfe von Winkelfunktionen wie folgt beschrieben werden:

$$x' = x \cdot \cos(\varphi) - y \cdot \sin(\varphi)$$
$$y' = x \cdot \sin(\varphi) + y \cdot \cos(\varphi)$$

Diese Formeln werden in diesem Leitprogramm nicht hergeleitet. Wer eine mathematische Herleitung sucht, findet diese beispielsweise im Vektorgeometrie-Lehrmittel von Heinz Bachmann (Bachmann, 1991) auf Seite 72.

Beispiel 5

Beispiel für eine Drehung um die z-Achse mit Drehwinkel 45°:

$$\begin{pmatrix} \cos(45°) & -\sin(45°) & 0 \\ \sin(45°) & \cos(45°) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Diese Matrix bewirkt eine Drehung um den Winkel 45° um die z-Achse. Die nebenstehende Abbildung zeigt eine solche Drehung in der Grundrissebene.

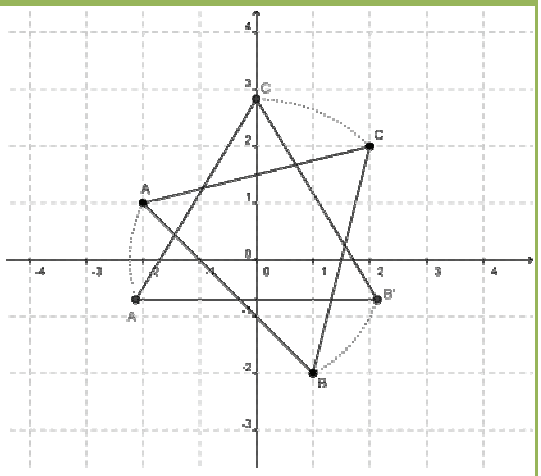


Abbildung 8: Drehung mit Drehwinkel 45°

Aufgabe 3

Geben Sie eine Drehmatrix für die Drehung um die z-Achse mit dem Drehwinkel φ an.

Für eine Drehung um die x-Achse gehen wir analog vor, wobei jetzt einfach die x-Koordinate des Punktes fix bleibt. Dies ergibt die folgende Drehmatrix:

$$R_x(\varphi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\varphi) & -\sin(\varphi) \\ 0 & \sin(\varphi) & \cos(\varphi) \end{pmatrix}$$

Aufgabe 4

Geben Sie eine Drehmatrix für die Drehung um die y-Achse mit dem Drehwinkel φ an. Achten Sie dabei auf die Lage des Koordinatensystems.

Damit sind Sie in der Lage, räumliche Drehungen in Java zu implementieren. Dies ist gerade das Ziel der nächsten Programmieraufgabe.

Aufgabe 5

Erweitern Sie Ihre Klasse Projektor3D so, dass der Benutzer das Polyeder mit Tastaturtasten um die Koordinatenachsen drehen kann. Implementieren Sie auch diese Funktionalität im dreidimensionalen Koordinatensystem mithilfe der vorgestellten Drehmatrizen.

Tipp: Winkelfunktionen werden in Java wie folgt aufgerufen: Math.sin(alpha), etc. Dabei bezeichnet alpha den Winkel im Bogenmass.

Aufgabe 6 (Lernkontrolle)

a) Geben Sie die Skalierungsmatrix für eine Stauchung mit Faktor 0.5 an.
b) Geben Sie eine Drehmatrix für eine Drehung um 30° um die y-Achse an.
c) Was geschieht mit dem Polyeder, wenn der Drehwinkel grösser als 360° ist.

Zusatzaufgaben

1. Neben den behandelten Drehungen und Skalierungen lassen sich auch Spiegelungen an den Rissebenen mit Matrizen beschreiben. Erweitern Sie die Klasse Projektor3D um die Funktionalität von Spiegelungen an den Rissebenen.
2. Eine weitere geometrische Transformation ist die Scherung. Informieren Sie sich auf dem Web, worum es dabei geht, und testen Sie verschiedene Scherungen mit dem Projektor3D.
3. Experimentieren Sie weiter mit verschiedenen Matrizen Ihrer Wahl und untersuchen Sie deren Wirkung auf das Polyeder.

Transformationen bei der Darstellung virtueller Realitäten

Professionelle Programmbibliotheken (engl. APIs) für 3D-Graphik wie DirectX oder OpenGL bieten allesamt schnelle Implementierungen für geometrische Transformationen an. Diese basieren auf der in diesem Kapitel vorgestellten Matrizenrechnung. Ohne diese Funktionalität wäre es unmöglich, die Bewegungen der Protagonisten in einem Spiel wie Crysis in Echtzeit darzustellen. Damit auch Verschiebungen mit Matrizen dargestellt werden können, verwendet man dort allerdings normalerweise vierdimensionale Matrizen und homogene Koordinaten[1].

[1] Mehr zu diesem Thema erfährst du beispielsweise im Buch Grundkurs Computergrafik mit Java von Frank Klawonn (Vgl. Literaturverzeichnis).

4. Kapitel: Darstellung von Oberflächenpolygonen

Oberflächenpolygone

Wie wir im 2. Kapitel gesehen haben, werden im Wavefront Object Format Eckpunkte (vertices) und Flächen (faces) gespeichert, wobei wir die Flächen bisher ausgelassen haben. Dies holen wir jetzt nach. Dabei wird sich zeigen, dass für eine korrekte Darstellung der Flächen die Normalenvektoren der zugehörigen Ebenen bestimmt werden müssen. Aus der Vektorgeometrie ist bekannt, dass der Normalenvektor stets senkrecht auf der Ebene steht. Um die Richtung des Normalenvektors eindeutig festlegen zu können, treffen wir eine *Abmachung*, die für eine korrekte Darstellung der Flächen entscheidend sein wird:

Achten Sie bei der Eingabe der Oberflächenpolygone (faces) darauf,
dass die Normalenvektoren stets nach aussen zeigen.

Im Folgenden wird anhand des Würfelbeispiels aus dem 2. Kapitel erklärt, in welcher Reihenfolge die Eckpunkte aufgelistet werden müssen, damit der zugehörige Normalenvektor nach aussen zeigt.

Beispiel 1

Wir betrachten dazu das Bodenquadrat des Würfels, das im Wavefront Object Format wie folgt definiert werden kann:

```
v    -0.5   -0.5   -0.5
v     0.5   -0.5   -0.5
v     0.5    0.5   -0.5
v    -0.5    0.5   -0.5
…
f     1      2      3      4
```

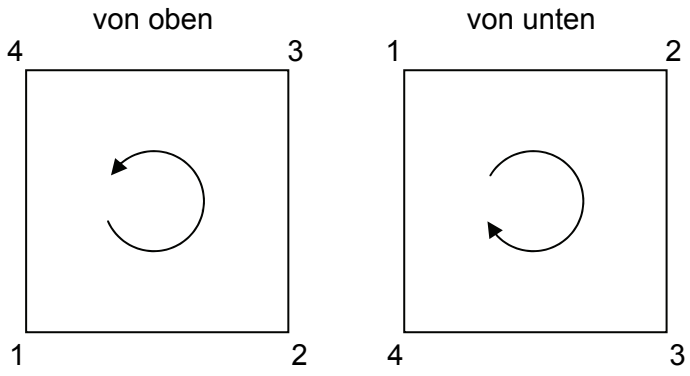


Abbildung 9: Bodenquadrat des Würfels

Diese Definition führt jedoch dazu, dass der Normalenvektor in Richtung des Augpunktes zeigt und somit ins Innere des Polyeders, da es sich ja um das Bodenquadrat handelt und weil wir von oben auf den Würfel hinunterschauen. Beachten Sie dazu auch die Abbildung 9: Die Quadratecken sind im Gegenuhrzeigersinn beschriftet, wenn der Betrachter von oben auf das Quadrat schaut. Schaut der Betrachter hingegen von unten auf das Quadrat verläuft die Beschriftung im Uhrzeigersinn. Wir müssen also die Reihenfolge der Eckpunkte bei der Definition der Fläche auf der letzten Zeile in Beispiel 1 anpassen. Dazu listen wir die Eckpunkte in umgekehrter Reihenfolge auf:

```
f     4      3      2      1
```

Schaut jetzt ein Betrachter von aussen, d.h. von unten, auf den Würfelboden, so erscheinen die Eckpunkte im Gegenuhrzeigersinn.

Aufgabe 1

Vervollständigen Sie die Datei Wuerfel.obj, indem Sie die Definitionen für die sechs Oberflächenquadrate ergänzen.

Tipp: Achten Sie darauf, dass die Normalenvektoren stets nach aussen zeigen. Benützen Sie dazu eine Variante der so genannten Rechte-Hand-Regel. Halten Sie Ihre rechte Hand wie in Abbildung 10 gezeigt. Zeigen nun die gekrümmten Finger in Richtung des Umlaufsinns gemäss Ihrer Nummerierung, so gibt Ihnen der Daumen die Richtung des Normalenvektors an.

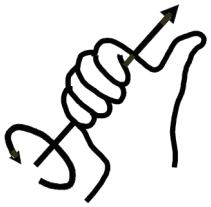


Abbildung 10: Rechter-Daumen-Regel

Das Drahtgittermodell

Bevor wir mit der Darstellung von Flächen beginnen, wollen wir uns zunächst einmal den Kanten annehmen. Diese werden im Wavefront Object Format nicht direkt gespeichert. Wenn wir jedoch alle Oberflächenpolygone zeichnen, erhalten wir automatisch auch alle Kanten, wobei genau genommen jede Kante zweimal gezeichnet wird. Das ist vielleicht etwas unschön, lässt sich jedoch nur mit grösserem Aufwand vermeiden, beispielsweise durch das Erstellen einer so genannten Adjazenzmatrix, welche für jedes Paar von Eckpunkten angibt, ob die entsprechende Kante existiert. Da dies für unsere Zwecke wenig bringt, gehen wir in diesem Leitprogramm nicht weiter darauf ein.

Bevor Sie mit der Implementierung anfangen können, wird hier kurz erklärt, wie die Hilfsklasse Polyeder die Flächen abspeichert. Diese sind nämlich nicht wie die Eckpunkte in einem Array abgelegt. Da die Flächen später für die Sichtbarkeitsbestimmung gemäss ihrem Abstand zum Augpunkt sortiert sein sollen, wurde als Datentyp ein sortierter Suchbaum gewählt. Was es damit im Detail auf sich hat, erfahren Sie gegen Ende des Kapitels. Hier nur so viel, dass Sie die Flächen aus dem Suchbaum extrahieren können. Betrachten Sie dazu die folgenden Zeilen:

```
Iterator<Face> it = polyeder.getFaces();
while (it.hasNext())
{
     Face flaeche = it.next();
     …
}
```

polyeder ist eine Instanz der Hilfsklasse Polyeder, genau wie im 2. Kapitel beschrieben. Mit der Methode getFaces() wird ein Iterator[1] für die Oberflächenpolygone übergeben. Dieser Iterator wird unter der Variable it abgelegt. In der while-Schleife wird als Bedingung gefragt, ob der Iterator noch eine weitere Fläche findet. Wenn ja, wird diese Fläche im Schleifeninne-

[1] Ein Iterator erlaubt den Zugriff auf die Elemente einer Liste, vgl. auch: http://de.wikibooks.org/wiki/Java_Standard:_Muster_Iterator

ren mit der Methode it.next() übergeben. Die Instanz flaeche der Hilfsklasse Face enthält ein Array mit den Eckennummern der aktuellen Fläche. Die Anzahl der Eckpunkte der Fläche erhalten Sie mit der Methode getNumOfVertices(). Die Nummer des i-ten Eckpunktes erhalten Sie mit getVertexN(i). Zusammengefasst können Sie also wie folgt auf den i-ten Eckpunkt der Fläche flaeche zugreifen:

```
int n = flaeche.getVertexN(i);
polyeder.getVertex(n);
```

Weitere Informationen zu den Hilfsklassen Polyeder und Face erhalten Sie in der JavaDoc zum package ch.beattl.projektor. Zum Zeichnen von Polygonen brauchen Sie die GPanel-Methode polygon(Point2D.Double[] corner). Diese Methode zeichnet ein Polygon, definiert durch die Eckpunkte aus dem Point2D-Array corner. Damit sind wir bereit, die Kanten des Polyeders zu zeichnen.

Aufgabe 2

Ergänzen Sie Ihre Klasse Projektor3D so, dass auch die Kanten des Polyeders gezeichnet werden. Testen Sie Ihre Klasse mit der Datei Wuerfel.obj, die sie bei Aufgabe 1 um Flächeninformationen ergänzt haben.

Rückseitenentfernung

Zur Darstellung der Oberflächenpolygone fehlt uns jetzt nicht mehr viel. Sie haben schon in der letzten Aufgabe ein Programm geschrieben, das auf alle Oberflächenpolygone zugreift und ein Drahtgittermodell des Polyeders zeichnet. Sie können diese Polygone mit der GPanel-Methode fillPolygon(Point2D.Double[] corner) mit Farben Ihrer Wahl füllen. Als Füllfarbe wird die aktuelle Farbe benützt, welche mit der Methode color(java.awt.Color color) geändert werden kann. Dabei bezeichnet der Parameter color eine Instanz der AWT-Klasse Color.

Aufgabe 3

Ergänzen Sie Ihre Klasse Projektor3D so, dass die Oberflächenpolygone mit Farben Ihrer Wahl gefüllt werden. Testen Sie Ihre Klasse mit der Datei Wuerfel.obj. Drehen Sie insbesondere den Würfel auch um verschiedene Achsen. Was fällt auf?

Zur Lösung des Sichtbarkeitsproblems aus Aufgabe 3 gehen wir von der folgenden Beobachtung aus: Offenbar zeichnet unser Algorithmus zahlreiche Flächen vergebens, diejenigen nämlich, welche sich vom Augpunkt aus gesehen auf der Rückseite des Polyeders befinden. Diese Flächen können eliminiert werden, indem eine Instanzvariable visible auf false gesetzt wird. Dies erreichen Sie mit der Methode setVisible(Boolean) der Klasse Face. In der Methode draw() kann daraufhin bei jeder Fläche mit der Methode isVisible() überprüft werden, ob sie sichtbar ist. Dieses Verfahren heisst *Rückseitenentfernung*.

Aber wie kann man bestimmen, ob eine Fläche sichtbar ist oder nicht? Bereits zu Beginn dieses Kapitels haben wir den Normalenvektor eines Oberflächenpolygons angeschaut, um

die Nummerierung der Eckpunkte so zu wählen, dass sie von aussen betrachtet im Gegenuhrzeigersinn angeordnet sind. Die Normalenvektoren aller Oberflächenpolygone zeigen daher nach aussen. Das können wir jetzt ausnützen. Wenn das Skalarprodukt des Normalenvektors einer Fläche mit dem Verbindungsvektor vom Schwerpunkt S des Oberflächenpolygons zum Augpunkt Q positiv ist, so ist die Fläche sichtbar. Bei negativem Skalarprodukt befindet sich die Fläche auf der Rückseite des Polyeders. Ist das Skalarprodukt genau 0, so erscheint die Fläche als Strecke. Entscheidend für die *Rückseitenentfernung* ist also das Vorzeichen des Skalarproduktes $\vec{n} \cdot \overrightarrow{SQ}$, wobei $\vec{n}$ den Normalenvektor, S den Schwerpunkt und Q den Augpunkt bezeichnet (Vgl. Abbildung 11). Durch die Rückseitenentfernung ändert sich nicht viel. Es genügt, wenn wir am Anfang der draw()-Methode jeweils die Rückseitenentfernung durchführen. Der Rest bleibt im Wesentlichen gleich. Damit wir allerdings die Schwerpunkte der Oberflächenpolygone nicht jedes Mal neu berechnen müssen, wird dies in der Hilfsklasse Polyeder gleich am Anfang beim Einlesen der Daten aus der .obj-Datei gemacht. Dies bedeutet, dass wir bei Transformationen des Polyeders, also bei Skalierungen und Drehungen, die Schwerpunkte jeweils auch skalieren, bzw. drehen müssen. Dies kann wie folgt implementiert werden:

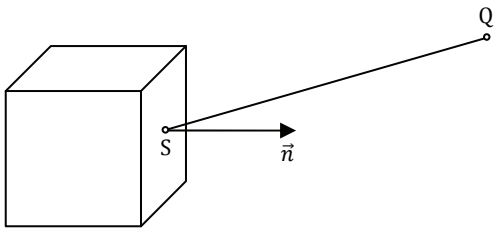


Abbildung 11: Rückseitenentfernung

```
Iterator<Face> it = polyeder.getFaces();
while (it.hasNext())
{
    Face flaeche = it.next();
    Point S = polyeder.getCenter(flaeche);
    m.transform(S);  // transformiert den Schwerpunkt S mit m
}
```

Die Methode getCenter(Face) liefert den Schwerpunkt des Oberflächenpolygons. Dieser wird in der Klasse Polyeder gemäss der folgenden Formel berechnet:

$$\overrightarrow{OS} = \frac{1}{n} \cdot \left(\overrightarrow{OP_1} + \overrightarrow{OP_2} + \cdots + \overrightarrow{OP_n}\right)$$

Dabei bezeichnet S den Schwerpunkt und P_1, P_2, … die Eckpunkte des Polygons. Damit lässt sich der Prioritätsalgorithmus in unserer Klasse Projektor3D implementieren.

Aufgabe 4

Implementieren Sie eine Methode removeHiddenFaces(), welche für Oberflächenpolygone, welche gemäss dem oben beschriebenen Kriterium auf der Vorderseite liegen, die Instanzvariable visible auf true setzt, für alle übrigen Polygone hingegen auf false. Berechnen Sie dazu die Normalenvektoren der Oberflächenpolygone und überprüfen Sie das Skalarprodukt mit dem Verbindungsvektor vom Schwerpunkt zum Augpunkt. Passen Sie die draw-Methode so an, dass nur sichtbare Flächen gezeichnet werden.

Ein Objektraumverfahren für Polyeder mit Einbuchtungen

Aufgabe 5

Modellieren Sie das Polyeder aus Abbildung 12. Es handelt sich dabei um einen Würfel mit einer pyramidenförmigen Einbuchtung auf der Vorderseite und auf der Rückseite. Die beiden pyramidenförmigen Einbuchtungen treffen sich im Mittelpunkt des Würfels. Testen Sie Ihr Modell hinterher, indem Sie es um verschiedene Achsen drehen. Was fällt auf?

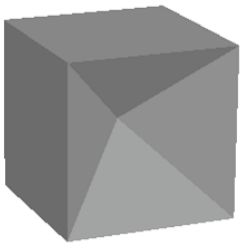


Abbildung 12: Würfel mit Einbuchtung

Das Sichtbarkeitsproblem, das Sie bei Aufgabe 5 kennengelernt haben, kann auf viele Arten gelöst werden. Die entsprechenden Algorithmen heissen Visibilitätsverfahren. Eine Idee besteht darin, die Projektionsebene mit einem Pixelraster zu überziehen. Durch jedes Pixel des Rasters wird in der Folge ein vom Augpunkt ausgehender Strahl gelegt (Vgl. Abbildung 13). Sichtbar ist folglich dasjenige Objekt, welches vom Strahl als erstes getroffen wird. Verfahren, die auf dieser Idee beruhen, heissen *Bildraumverfahren*. Sie benötigen bei p Pixeln und n Objekten in der Regel n · p Bearbeitungsschritte als Rechenaufwand.

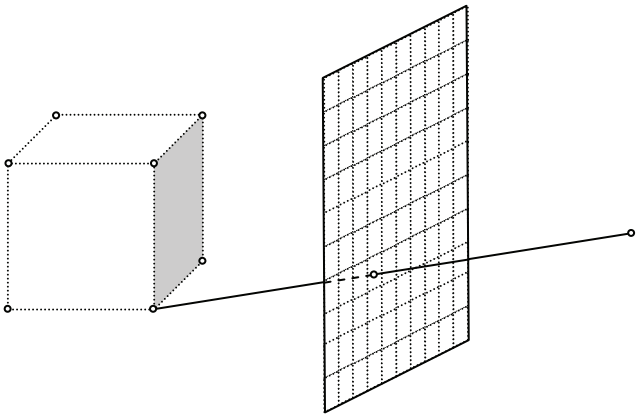


Abbildung 13: Bildraumverfahren mit Pixelraster

Da wir mit unseren Polyedern Objekte vor uns haben, die sehr spezielle Eigenschaften aufweisen, bietet sich uns jedoch ein effizienteres Verfahren an. Dazu gehen wir von den Oberflächenpolygonen aus. Da die Oberflächenpolygone ein Polyeder mit allfälligen Einbuchtungen bilden, wissen wir, dass sie sich nicht überschneiden. Es lässt sich also immer bestimmen, welches Polygon im Bezug auf den Augpunkt weiter vorne und welches weiter hinten liegt. Es sei denn, die beiden Polygone liegen exakt auf der gleichen Höhe. Dann spielt es natürlich keine Rolle, welches der beiden zuerst gezeichnet wird. Diese Tatsache nutzen wir aus, indem wir für jedes Oberflächenpolygon den Abstand des Schwerpunktes vom Augpunkt ausrechnen. Ordnet man jedem Oberflächenpolygon diesen Abstand zu, lassen sich die Oberflächenpolygone gemäss ihrem Abstand vom Augpunkt sortieren. Dann muss nur noch dafür gesorgt werden, dass die Polygone in der Reihenfolge der absteigenden Abstandswerte gezeichnet werden, d.h. das Polygon mit dem grössten Abstandswert zuerst usw. Verfahren, die wie der eben beschriebene *Prioritätsalgorithmus* die Eigenschaften der darzustellenden Objekte berücksichtigen, heissen *Objektraumverfahren*. Sie sind unabhängig von der Auflösung des projizierten Bildes.

Aufgabe 6

Erweitern Sie in Ihrer Klasse Projektor3D die Methode removeHiddenFaces() so, dass die Oberflächenpolygone gemäss ihrem Abstand zum Augpunkt sortiert werden. Dabei genügt es, wenn Sie für jedes Oberflächenpolygon zunächst den Abstand zum Augpunkt berechnen und diesen mit der Methode setDistance(double) zu den Flächeneigenschaften hinzufügen[1]. Anschliessend wird das Polygon in einen neuen sortierten Suchbaum namens ts eingefügt. Dadurch werden die Polygone automatisch sortiert. Benützen Sie dazu das folgende Programmgerüst:

```
TreeSet<Face> ts = new TreeSet<Face>();
Iterator<Face> it = polyeder.getFaces();
while (it.hasNext())
{
      Face flaeche = it.next();
      …
      it.remove();
      ts.add(flaeche);
}
polyeder.setFaces(ts);
```

Aufgabe 7 (Lernkontrolle)

a) Wozu dient die Rückseitenentfernung?

b) Erklären Sie den Unterschied zwischen einem Bildraum- und einem Objektraumverfahren. Gehen Sie insbesondere auf die Vor- und Nachteile der beiden Verfahren ein.

c) Erklären Sie in eigenen Worten das Sichtbarkeitsproblem bei der Darstellung von Polyedern mit Einbuchtungen. Wann tritt das Problem auf?

Zusatzaufgaben

1. Suchen Sie auf dem Internet die Datei shuttle.obj, mit der die Titelseite dieses Leitprogramms erstellt wurde. Rendern[2] Sie das Bild mit Ihrem Prioritätsalgorithmus.
2. Überlegen Sie sich, welche Eigenschaften der Polyeder der hier vorgestellte Prioritätsalgorithmus ausnützt. Suchen Sie daraufhin auf dem Internet nach .obj-Dateien mit Objekten, welche sich mit dem Prioritätsalgorithmus nicht mehr korrekt darstellen lassen. Welche Eigenschaften verursachen die Probleme? Lassen sich diese Probleme allenfalls, durch geringfügige Anpassung des Algorithmus beheben?
3. Suchen Sie weitere Objekte auf dem Internet, die Sie rendern können.

[1] Aus Effizienzgründen wird oft das Quadrat des Abstandes verwendet.

[2] Rendern bedeutet, ein Bild aus einer 3D-Szene berechnen.

5. Kapitel: Oberflächenschattierung – ein Ausblick

Beleuchtungsmodelle

Die Polyeder aus dem 4. Kapitel waren bereits recht gelungen. Allerdings fehlte für einen realistischen Eindruck noch immer die Schattierung der Oberfläche (engl. shading). Diese hängt natürlich von der Art der Beleuchtung, vom so genannten Beleuchtungsmodell ab. An dieser Stelle beschränken wir uns auf die Beleuchtung mit parallelem Licht. Bei der Schattierung konzentrieren wir uns auf die diffuse Reflexion gemäss dem Lambertschen Reflexionsgesetz, welche relativ leicht in unser Modell integriert werden kann. Daneben gibt es in der Computergraphik noch zahlreiche weitere Modelle, beispielsweise die Spiegelreflexion oder das so genannte Radiosity-Modell, welches dem Verlauf der Lichtstrahlen mittels Raytracing folgt. Eine leicht verständliche Übersicht findet der interessierte Leser beispielsweise im Lehrbuch Grundkurs Computergrafik mit Java (Klawonn, 2009).

Diffuse Reflexion nach Lambert

Das Lambertsche Reflexionsgesetz basiert auf der Beobachtung, dass die Lichtenergie am grössten ist, wenn das Licht senkrecht auf die beleuchtete Fläche auftrifft. Bei flacher Einstrahlung verteilt sich die Energie auf eine grössere Fläche. Die Energie an einem Punkt ist daher kleiner. Damit ergibt sich die folgende Formel für die Intensität I des reflektierten Lichts:

$$I = I_L \cdot k \cdot \cos(\varphi),$$

wobei I_L die Intensität des auftreffenden Lichts, k ein materialabhängiger Reflexionskoeffizient und φ der Winkel des einfallenden Strahls zum Normalenvektor ist (Vgl. Abbildung 14). Daraus folgt wie beabsichtigt, dass die Intensität $I = I_L \cdot k$ ist, falls das Licht senkrecht auftrifft ($\cos(0) = 1$). Ausserdem gilt: Je grösser der Winkel φ, umso kleiner wird die Intensität.

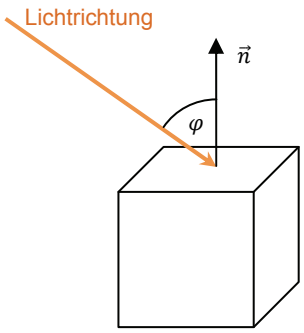


Abbildung 14: Diffuse Reflexion

Aufgabe 1

Schreiben Sie das Lambertsche Reflexionsgesetz als Skalarprodukt mithilfe des Vektors $\vec{l}$, der die Lichtrichtung angibt. Erstellen Sie dazu auch eine passende Skizze.

Wir können unser Modell nun so erweitern, dass die Oberflächenpolygone das eintreffende Licht gemäss dem Lambertschen Reflexionsgesetz diffus reflektieren. Da wir auch dafür wieder den Normalenvektor brauchen, macht es Sinn, diesen in der Hilfsklasse Polyeder zu speichern. Dies ist bereits vorgesehen. Mit der Methode generateNormals() können die Normalenvektoren sämtlicher Oberflächenpolygone berechnet werden. Die so berechneten Normalenvektoren sind ausserdem normiert auf die Länge 1. Mit der Methode getNormal(Face) können Sie jederzeit auf den Normalenvektor einer Fläche zugreifen.

Aufgabe 2

Erweitern Sie die Klasse Projektor3D so, dass die Szene mit parallelem Licht beleuchtet wird. Die sichtbaren Oberflächenpolygone sollen das Licht gemäss dem Lambertschen Reflexionsgesetz reflektieren. Als Reflexionskoeffizient können Sie dabei 1 wählen. Probieren sie die so modifizierte Klasse mit verschiedenen .obj-Dateien aus.

Tipp: Aus Effizienzgründen empfiehlt es sich, die Normalenvektoren bei Drehungen mitzudrehen, statt sie mit generateNormals() neu zu berechnen.

Zusatzaufgabe

Informieren Sie sich auf dem Web über die Spiegelreflexion. Versuchen Sie, die Spiegelreflexion in Ihre Klasse zu integrieren. Wie wirken die beiden Reflexionsarten zusammen.

Shading virtueller Realitäten

Ein gutes Shading trägt entscheidend zur Qualität einer virtuellen Realität bei. So benützt beispielsweise die Cry Engine von Crysis ein globales Beleuchtungsmodell analog zum oben erwähnten Radiosity-Modell, bei dem Reflektionen und Lichtbrechung berücksichtigt werden. Damit die Berechnungen nicht zu zeitaufwändig werden, wird das so genannte Deferred Shading eingesetzt, bei dem verdeckte Polygone vor dem Rendern eliminiert werden. Ausserdem wird Oberflächentesselierung verwendet, eine Technik, welche es erlaubt, je nach Distanz zu einer Oberfläche diese mit mehr oder weniger Details darzustellen. Auch dadurch lässt sich Zeit sparen. Das Fernziel bleibt natürlich das Echtzeit-Raytracing, da nur ein vollständiges Strahlenmodell eine fotorealistische Darstellung erlaubt. Bis anhin konnten aber auf Echtzeit-Raytracing basierende Spiele wegen fehlender Rechenleistung mit der hardwareoptimierten Graphik von Spielen wie Crysis nicht mithalten. Es bleibt zu vermuten, dass auch in Zukunft eine Kombination von Objektraumverfahren und Raytracing-Effekten aus Effizienzgründen bei der Programmierung von 3D-Spielen bevorzugt wird.

Abbildungsverzeichnis

Literaturverzeichnis

1. Bachmann, H. (1991). *Vektorgeometrie* (Ausgabe a Ausg.). Zürich: sabe.
2. Blender Foundation. (21. Juni 2009). Blender 2.49a. http://www.blender.org.
3. Klawonn, F. (2009). *Grundkurs Computergrafik mit Java.* Wiesbaden: Vieweg+Teubner.
4. Kleinberg, J. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM , 36* (5), S. 604-632.
5. Plüss, Ä. (2009). *Aplu (Aegidius Pluess) Home*. Abgerufen am 2. 11 2009 von http://www.aplu.ch
6. Plüss, Ä. (2004). *Java exemplarisch.* Oldenbourg.

Anhang A: Hilfsklassen zum Leitprogramm

Begleitend zum Leitprogramm wurden Hilfsklassen erstellt, die die Verwaltung der Daten (Punkte, Vektoren, Matrizen) vereinfachen sollen. Ausserdem sollen diese Klassen dafür sorgen, dass Sie sich auf das Wesentliche konzentrieren können und nicht unnötig Zeit verlieren, beispielsweise beim Einlesen der Daten aus einer .obj-Datei.

Installation der Hilfsklassen

Die Hilfsklassen sind in einer Java-Archivdatei namens projektor.jar abgelegt. Diese Archivdatei müssen Sie bei der Compilierung Ihres Codes in den Pfad einbinden, wenn Sie die Hilfsklassen verwenden wollen.

Eclipse

Mit der Entwicklungsumgebung eclipse geht dies beispielsweise wie folgt:

1. Kopieren Sie die Datei projektor.jar in Ihren Projektordner.
2. Wenn die Datei im Package-Explorer von eclipse erscheint, eröffnen Sie mit der rechten Maustaste ein Kontextmenü und wählen den folgenden Menü-Befehl aus:

   Build Path → Add to Build Path

Dokumentation der Hilfsklassen

Alle verfügbaren Hilfsklassen und die darin angebotenen Methoden sind in einer JavaDoc dokumentiert. Daher wird auf eine vollständige Auflistung der Methoden verzichtet. Im Folgenden eine Liste mit den für dieses Leitprogramm relevanten Klassen und Methoden aus dem package ch.beattl.gm3d. Ergänzen Sie die folgende Zeile zu Beginn Ihrer Klasse:

```
import ch.beattl.gm3d.*;
```

Klasse Point

Methode	Erklärung
Point(double x, double y, double z)	Konstruktor, der das Objekt Point instanziert, mit dem Punkte im 3D-Raum verwaltet werden.
Point()	Default-Konstruktor für den Nullpunkt.

Auf die Koordinaten kann man neben den üblichen getter- und setter-Methoden auch direkt über öffentliche Instanzvariablen zugreifen, beispielsweise P.x für die x-Koordinate von P.

Klasse Vector

Methode	Erklärung
Vector(Tuple t)	Konstruktor, der das Objekt Vector anhand des Tupels t (Point oder Vector) instanziert.
Vector ()	Default-Konstruktor für den Nullvektor.

scale(double k)	Skaliert den aktuellen Vektor mit k.
add(Vector v)	Addiert zum aktuellen Vektor den Vektor v.
sub(Vector v)	Subtrahiert vom aktuellen Vektor den Vektor v.
length()	Gibt die Länge des aktuellen Vektors als double zurück.
lengthSquared()	Gibt das Quadrat der Länge des aktuellen Vektors als double zurück.
cross(Vector u, Vector v)	Berechnet das Vektorprodukt der Vektoren a und b und speichert es im aktuellen Vektor.
dot(Vector v)	Berechnet das Skalarprodukt des aktuellen Vektors mit dem Vektor v und gibt das Resultat als double zurück.

Auf die Komponenten kann man neben den üblichen getter- und setter-Methoden auch direkt über öffentliche Instanzvariablen zugreifen, beispielsweise v.x für die x-Komponente von v.

Klasse Matrix

Methode	Erklärung
Matrix(double m00, double m01, double m02, double m10, double m11, double m12, double m20, double m21, double m22)	Konstruktor, der das Objekt Matrix instanziert, mit dem 3 × 3-Matrizen verwaltet werden.
Matrix ()	Default-Konstruktor für die Nullmatrix.
mul(double scalar)	Skaliert die aktuelle Matrix mit dem Skalar scalar.
mul(Matrix m1)	Multipliziert die aktuelle Matrix mit der Matrix m1 und setzt die aktuelle Matrix auf das Resultat.
mul(Matrix m1, Matrix m2)	Setzt die aktuelle Matrix auf das Resultat der Matrixmultiplikation der Matrix m1 mit der Matrix m2.
rotX(double alpha)	Setzt die aktuelle Matrix auf eine Drehmatrix um die x-Achse mit Winkel alpha im Gegenuhrzeigersinn.
rotY(double alpha)	Setzt die aktuelle Matrix auf eine Drehmatrix um die y-Achse mit Winkel alpha im Gegenuhrzeigersinn.
rotZ(double alpha)	Setzt die aktuelle Matrix auf eine Drehmatrix um die z-Achse mit Winkel alpha im Gegenuhrzeigersinn.
set(double scale)	Setzt die aktuelle Matrix auf eine Skalierungsmatrix mit Faktor scale.
transform(Tuple t)	Multipliziert das Tupel t (Point oder Vector) von links mit der Matrix. Das Resultat wird direkt im Tupel t abgespeichert.

Anhang B: Lösungen

Lösungen zum 1. Kapitel

S. 3

Aufgabe 1

Das wichtigste ist eine Graphikumgebung, um das perspektivische Bild darstellen zu können. Ausserdem muss ich mir überlegen, wie ich rechnerisch von den Koordinaten des Würfels zu den Koordinaten der Bildpunkte komme. Schliesslich sollte ich die Lage des Würfels so wählen, dass ich die Koordinaten im dreidimensionalen, kartesischen Koordinatensystem einfach angeben kann.

S. 4

Aufgabe 2

A(-0.5 / -0.5 / -0.5), B(0.5 / -0.5 / -0.5), C(0.5 / 0.5 / -0.5), D(-0.5 / 0.5 / -0.5),
E(-0.5 / -0.5 / 0.5), F(0.5 / -0.5 / 0.5), G(0.5 / 0.5 / 0.5), H(-0.5 / 0.5 / 0.5)

Die Wahl von 0.5 als Koordinate (anstelle von beispielsweise 1) ist nicht entscheidend. Mit dieser Wahl ergibt sich als Kantenlänge gerade 1. Die Beispielprogramme in diesem Leitprogramm gehen von der Kantenlänge 1 aus.

Aufgabe 3

Für die Berechnung des Bildpunktes benützen wir die folgende Vektorgleichung:

$$\overrightarrow{OP'} = \overrightarrow{OQ} + \overrightarrow{QP'} = \overrightarrow{OQ} + t \cdot \overrightarrow{QP},$$

wobei Q den Augpunkt, P die Würfelecke und P‘ den Bildpunkt bezeichnen. *t* ist ein Parameter für die Streckung des Vektors $\overrightarrow{QP}$ auf die gewünschte Länge. Da P‘ in der xy-Ebene liegt, ist die z-Koordinate 0. Damit lassen sich der Parameter *t* und damit auch die Bildpunkte leicht berechnen:

$$t_1 = \frac{11}{10}\text{:}\quad A'(-0.45 / -0.45),\ B'(0.45 / -0.45),\ C'(0.45 / 0.45),\ D'(-0.45 / 0.45)$$

$$t_2 = \frac{10}{9}\text{:}\quad E'(-0.56 / -0.56),\ F'(0.56 / -0.56),\ G'(0.56 / 0.56),\ H'(-0.56 / 0.56)$$

Im Allgemeinen gelten die folgenden Formeln zur Berechnung des Parameters *t* und der Koordinaten der Bildpunkte:

$$t = -\frac{z_Q}{z_P - z_Q}$$

$$x_{P'} = x_Q + t \cdot (x_P - x_Q), \qquad y_{P'} = y_Q + t \cdot (y_P - y_Q)$$

Da der Augpunkt auf der z-Achse liegt, lassen sich die Formeln wie folgt vereinfachen:

$$x_{P'} = t \cdot x_P, \qquad y_{P'} = t \cdot y_P$$

S. 5

Aufgabe 4

Es werden die folgenden Methoden implementiert:

Methode	Erklärung
Point2D.Double projectOnPi(Point P)	Projiziert den Punkt vom dreidimensionalen Datentyp Point auf einen zweidimensionalen Punkt vom Datentyp Point2D.Double.
void draw()	Zeichnet den Würfel durch wiederholten Aufruf der Methode projectOnPi für jede Würfelecke. Damit die Punkte gut sichtbar sind, werden sie als gefüllte Kreise mit Radius 0.025 dargestellt.

Sie finden den folgenden Beispielcode auch elektronisch als Beilage zum Leitprogramm in der Datei Projektor3D_v01.java.

```
1.    import java.awt.geom.Point2D;
2.
3.    import ch.aplu.util.*;
4.    import ch.beattl.gm3d.*;
5.
6.    //Projiziert 3D-Objekte auf den Bildschirm
7.    public class Projektor3D_v01 extends GPanel
8.    {
9.      static final double Radius = 0.025; // Radius des Eckpunktes
10.
11.     Point[] wuerfel; // Variable fuer den Wuerfel
12.     Point Auge; // Position des Auges
13.
14.     /**
15.      * Projiziert den Punkt P auf die Bildebene Pi (xy-Ebene) und
16.      * gibt den Bildpunkt zurueck.
17.      * @param P Punkt P
18.      * @return Bildpunkt
19.      */
20.     Point2D.Double projectOnPi(Point P)
21.     {
22.       // Parameterwert bis zur Projektionsebene Pi
23.       double t = -Auge.z / (P.z - Auge.z);
24.       // R = Durchstosspunkt des Strahls (Auge,P) durch Pi
25.       double x = Auge.x + t * (P.x - Auge.x);
26.       double y = Auge.y + t * (P.y - Auge.y);
27.       Point2D.Double R = new Point2D.Double(x, y);
28.
29.       // Rueckgabewert
30.       return R;
31.     }
32.
```

```
33.   /**
34.      * Zeichnet die Eckpunkte des Wuerfels in einer GPanel-Graphik.
35.      */
36.     void draw()
37.     {
38.       // Alle Ecken zeichnen
39.       for (int i = 0; i < wuerfel.length; i++)
40.       {
41.         // Durchstosspunkt <Auge,Ecke> mit Pi berechnen
42.         Point2D.Double R = projectOnPi(wuerfel[i]);
43.
44.         // Punkt R zeichnen und beschriften
45.         move(R);
46.         fillCircle(Radius);
47.         move(R.x + .05, R.y);
48.         text("" + (i + 1));
49.       }
50.     }
51.
52.   Projektor3D_v01()
53.     {
54.       // Graphikfenster definieren
55.       super("Graphik", -2.5, 2.5, -2.5, 2.5);
56.
57.       // Augposition festlegen
58.       Auge = new Point(0, 0, 5);
59.
60.       // Definition des Wuerfels
61.       wuerfel = new Point[]{new Point(-0.5, -0.5, -0.5),
62.            new Point(0.5, -0.5, -0.5),
63.            new Point(0.5, 0.5, -0.5),
64.            new Point(-0.5, 0.5, -0.5),
65.            new Point(-0.5, -0.5, 0.5),
66.            new Point(0.5, -0.5, 0.5),
67.            new Point(0.5, 0.5, 0.5),
68.            new Point(-0.5, 0.5, 0.5)};
69.     }
70.
71.     public static void main(String[] args)
72.     {
73.       Projektor3D_v01 proj = new Projektor3D_v01();
74.
75.       // Wuerfel zeichnen
76.       proj.draw();
77.     }
78.   }
```

Aufgabe 5 (Lernkontrolle)

Die Eckpunkte des Oktaeders können beispielsweise wie folgt definiert werden:

```
oktaeder = new Point[]{new Point(1.0, 0.0, 0.0),
           new Point(0.0, 1.0, 0.0),
           new Point(-1.0, 0.0, 0.0),
           new Point(0.0, -1.0, 0.0),
           new Point(0.0, 0.0, 1.0),
           new Point(0.0, 0.0, -1.0)};
```

Der Java-Code ist bis auf den Variablennamen identisch mit der Lösung von Aufgabe 4.

Lösungen zum 2. Kapitel

S. 7

Aufgabe 1

Die Datei Wuerfel.obj enthält die folgenden Definitionen:

```
v  -0.5  -0.5  -0.5
v   0.5  -0.5  -0.5
v   0.5   0.5  -0.5
v  -0.5   0.5  -0.5
v  -0.5  -0.5   0.5
v   0.5  -0.5   0.5
v   0.5   0.5   0.5
v  -0.5   0.5   0.5
```

Aufgabe 2

Sie finden den folgenden Beispielcode auch elektronisch als Beilage zum Leitprogramm in der Datei Projektor3D_v02.java.

```
1.    import java.awt.geom.Point2D;
2.
3.    import ch.aplu.util.*;
4.    import ch.beattl.gm3d.*;
5.    import ch.beattl.projektor.*;
6.
7.    //Projiziert 3D-Objekte auf den Bildschirm
8.    public class Projektor3D_v02 extends GPanel
9.    {
10.     static final double Radius = 0.025; // Radius des Eckpunktes
11.     static final double Epsylon = 0.0001; // Ab wann ist x=0?
12.     // static final String fileName = "Tetraeder.obj";
13.     static final String fileName = "Wuerfel.obj";
14.     // static final String fileName = "Oktaeder.obj";
15.     // static final String fileName = "Ikosaeder.obj";
16.     // static final String fileName = "Dodekaeder.obj";
17.
18.     Polyeder polyeder; // Variable fuer das Polyeder
19.     Point Auge; // Position des Auges
20.
21.     /**
22.      * Projiziert den Punkt P auf die Bildebene Pi (xy-Ebene) und
23.      * gibt den Bildpunkt zurueck.
24.      * @param P Punkt P
25.      * @return Bildpunkt
26.      */
27.     Point2D.Double projectOnPi(Point P)
28.     {
29.       // Parameterwert bis zur Projektionsebene Pi
30.       double t = P.z - Auge.z;
31.       if (Math.abs(t) > Epsylon)
32.         t = -Auge.z / t;
33.       // R = Durchstosspunkt des Strahls (Auge,P) durch Pi
34.       double x = Auge.x + t * (P.x - Auge.x);
35.       double y = Auge.y + t * (P.y - Auge.y);
36.       Point2D.Double R = new Point2D.Double(x, y);
```

```
37.
38.         // Rueckgabewert
39.         return R;
40.       }
41.
42.       /**
43.        * Zeichnet die Eckpunkte des Wuerfels in einer GPanel-Graphik.
44.        */
45.       void draw()
46.       {
47.         // Alle Ecken zeichnen
48.         for (int i = 0; i < polyeder.getNumOfVertices(); i++)
49.         {
50.           // Durchstosspunkt <Auge,Ecke> mit Pi berechnen
51.           Point2D.Double R = projectOnPi(polyeder.getVertex(i));
52.
53.           // Punkt R zeichnen und beschriften
54.           move(R);
55.           fillCircle(Radius);
56.           move(R.x + .05, R.y);
57.           text("" + (i + 1));
58.         }
59.       }
60.
61.       Projektor3D_v02()
62.       {
63.         // Graphikfenster definieren
64.         super("Graphik", -2.5, 2.5, -2.5, 2.5);
65.
66.         // Augposition festlegen
67.         Auge = new Point(0, 0, 5);
68.
69.         // Objektdefinition
70.         FileReader myReader = new FileReader(fileName);
71.         polyeder = new Polyeder(myReader);
72.       }
73.
74.       public static void main(String[] args)
75.       {
76.         Projektor3D_v02 proj = new Projektor3D_v02();
77.
78.         proj.draw();
79.       }
80.   }
```

Aufgabe 3 (Lernkontrolle)

Die Datei Oktaeder.obj enthält die folgenden Definitionen:

```
v  1.0  0.0  0.0
v  0.0  1.0  0.0
v -1.0  0.0  0.0
v  0.0 -1.0  0.0
v  0.0  0.0  1.0
v  0.0  0.0 -1.0
```

Der Java-Code ist bis auf den Dateinamen identisch mit der Lösung von Aufgabe 2.

Lösungen zum 3. Kapitel

Aufgabe 1

S. 8

$$\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{pmatrix}\cdot\begin{pmatrix}0\\ 1\\ 0\end{pmatrix}=\begin{pmatrix}2\\ 5\\ 8\end{pmatrix}$$

$$\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{pmatrix}\cdot\begin{pmatrix}0\\ 0\\ 1\end{pmatrix}=\begin{pmatrix}3\\ 6\\ 9\end{pmatrix}$$

Die Matrixmultiplikation in den drei Beispielen mit diesen speziellen Vektoren, welche auch Basisvektoren des Koordinatensystems heissen, liefert gerade die Spalten(-vektoren) der Matrix als resultierende Vektoren.

Aufgabe 2

S. 9

Der folgende Java-Code realisiert eine Skalierung sämtlicher Polyederecken mit dem Skalierungsfaktor factor:

```
void scale(double factor)
{
     for (int i = 0; i < polyeder.getNumOfVertices(); i++)
     {
          Point P = polyeder.getVertex(i);
          Matrix m = new Matrix(
                    factor, 0, 0,
                    0, factor, 0,
                    0, 0, factor);
          m.transform(P);
     }
}
```

Die vollständige Lösung finden Sie in der Datei Polyeder3D_v03.java.

Aufgabe 3

S. 10

$$R_z(\varphi)=\begin{pmatrix}\cos(\varphi) & -\sin(\varphi) & 0\\ \sin(\varphi) & \cos(\varphi) & 0\\ 0 & 0 & 1\end{pmatrix}$$

Aufgabe 4

$$R_y(\varphi)=\begin{pmatrix}\cos(\varphi) & 0 & \sin(\varphi)\\ 0 & 1 & 0\\ -\sin(\varphi) & 0 & \cos(\varphi)\end{pmatrix}$$

S. 11

Aufgabe 5

Der folgende Java-Code realisiert eine Drehung sämtlicher Polyederecken mit Drehwinkel phi um die angegebene Koordinatenachse:

```
public void rotate(double phi, char axis)
{
     Matrix m;
     switch (axis)
     {
          case 'x':
               m = new Matrix(
                         1, 0, 0,
                         0, Math.cos(phi), -Math.sin(phi),
                         0, Math.sin(phi), Math.cos(phi));
               break;
            case 'y':
               m = new Matrix(
                         Math.cos(phi), 0, Math.sin(phi),
                         0, 1, 0,
                         -Math.sin(phi), 0, Math.cos(phi));
               break;
          case 'z':
               m = new Matrix(
                         Math.cos(phi), -Math.sin(phi), 0,
                         Math.sin(phi), Math.cos(phi), 0,
                         0, 0, 1);
               break;
          default:
               m = new Matrix();
     }
     for (int i = 0; i < polyeder.getNumOfVertices(); i++)
     {
          Point P = polyeder.getVertex(i);
          m.transform(P);
     }
}
```

Die vollständige Lösung finden Sie in der Datei Polyeder3D_v04.java.

Aufgabe 6 (Lernkontrolle)

a) $\begin{pmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{pmatrix}$

b) $R_y\left(\frac{\pi}{6}\right) = \begin{pmatrix} \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \end{pmatrix}$

c) Der Drehwinkel wird modulo 360° bestimmt, das heisst, es gilt der Divisionsrest beim Teilen durch 360°.

Lösungen zum 4. Kapitel

S. 13

Aufgabe 1

Hier also den vollständigen Inhalt der Datei Wuerfel.obj:

```
#Wuerfel

v  -0.5  -0.5  -0.5
v   0.5  -0.5  -0.5
v   0.5   0.5  -0.5
v  -0.5   0.5  -0.5
v  -0.5  -0.5   0.5
v   0.5  -0.5   0.5
v   0.5   0.5   0.5
v  -0.5   0.5   0.5

f  4  3  2  1
f  1  2  6  5
f  2  3  7  6
f  3  4  8  7
f  1  5  8  4
f  5  6  7  8
```

S. 14

Aufgabe 2

Der folgende Code zeichnet die Kanten aus der Instanz polyeder der Hilfsklasse Polyeder:

```
Iterator<Face> it = polyeder.getFaces();
while (it.hasNext())
{
     Face flaeche = it.next();
     // Polygon zusammensetzen
     int k = flaeche.getNumOfVertices();
     Point2D.Double[] polygon = new Point2D.Double[k];
     for (int j = 0; j < k; j++)
     {
          // Nummer der Ecke finden
          int n = flaeche.getVertexN(j);
          // Ecke auf Bildebene projizieren
          polygon[j] = projectOnPi(polyeder.getVertex(n));
     }
     color(Color.blue);
     polygon(polygon);
}
```

Die vollständige Lösung finden Sie in der Datei Polyeder3D_v05.java.

S. 14

Aufgabe 3

Den Code zu dieser Aufgabe finden Sie in der Datei Polyeder3D_v06.java. Das Problem ist, dass die Oberflächenpolygone immer in der Reihenfolge gezeichnet werden, in der sie in der Datei Wuerfel.obj aufgelistet sind. Spätestens nach einigen Drehungen kommt es daher zu Situationen, in denen Flächen von der Vorderseite des Polyeders durch Flächen von der Rückseite des Polyeders verdeckt werden (*Sichtbarkeitsproblem*).

Aufgabe 4 S. 15

Der folgende Code zeigt die Rückseitenentfernung:

```
private void removeHiddenFaces()
{
     Iterator<Face> it = polyeder.getFaces();
     while (it.hasNext())
     {
          Face flaeche = it.next();
          Vector OS = new Vector(polyeder.getCenter(flaeche));
          Vector SQ = new Vector(Auge);
          SQ.sub(OS);
          // Rueckseitenentfernung
          int k = flaeche.getVertexN(0);
          Vector OP = new Vector(polyeder.getVertex(k));
          int l = flaeche.getVertexN(1);
          Vector a = new Vector(polyeder.getVertex(l));
          a.sub(OP);
          int m = flaeche.getVertexN(2);
          Vector b = new Vector(polyeder.getVertex(m));
          b.sub(OP);
          Vector n = new Vector();
          n.cross(a, b);
          flaeche.setVisible(n.dot(SQ) >= 0);
     }
}
```

Diese Methode muss am Anfang der Methode draw() aufgerufen werden. Ausserdem müssen die Transformationsmethoden scale und rotate wie auf Seite 15 beschrieben angepasst werden. Die vollständige Lösung finden Sie in der Datei Polyeder3D_v07a.java.

Aufgabe 5 S. 16

Hier also den Inhalt der Datei Wuerfel_mit_Einbuchtung.obj:

```
#Wuerfel mit Einbuchtung
v  -0.5  -0.5  -0.5
v   0.5  -0.5  -0.5
v   0.5   0.5  -0.5
v  -0.5   0.5  -0.5
v  -0.5  -0.5   0.5
v   0.5  -0.5   0.5
v   0.5   0.5   0.5
v  -0.5   0.5   0.5
v   0.    0.    0.
f  1  2  6  5
f  2  3  7  6
f  3  4  8  7
f  1  5  8  4
f  9  2  1
f  9  3  2
f  9  4  3
f  9  1  4
f  5  6  9
```

```
f  6  7  9
f  7  8  9
f  8  5  9
```

Dies ist ein weiteres Beispiel für ein Sichtbarkeitsproblem: Spätestens nach einigen Drehungen kommt es zu Situationen, in denen näher gelegene Flächen durch weiter entfernte Flächen überdeckt werden.

Aufgabe 6

S. 17

Der folgende Code zeigt die Methode removeHiddenFaces() aus Aufgabe 4 erweitert um die verlangte Sortierung mittels TreeSet (Vgl. auch Polyeder3D_v08a.java).

```
private void removeHiddenFaces()
{
     TreeSet<Face> ts = new TreeSet<Face>();
     Iterator<Face> it = polyeder.getFaces();
     while (it.hasNext())
     {
          Face flaeche = it.next();
          Vector OS = new Vector(polyeder.getCenter(flaeche));
          Vector SQ = new Vector(Auge);
          SQ.sub(OS);
          // Rueckseitenentfernung
          int k = flaeche.getVertexN(0);
          Vector OP = new Vector(polyeder.getVertex(k));
          int l = flaeche.getVertexN(1);
          Vector a = new Vector(polyeder.getVertex(l));
          a.sub(OP);
          int m = flaeche.getVertexN(2);
          Vector b = new Vector(polyeder.getVertex(m));
          b.sub(OP);
          Vector n = new Vector();
          n.cross(a, b);
          flaeche.setVisible(n.dot(SQ) >= 0);
          // Berechnet den Abstand der Flaeche zum Betrachter
          flaeche.setDistance(SQ.lengthSquared());
          it.remove();
          ts.add(flaeche);
     }
     polyeder.setFaces(ts);
}
```

Aufgabe 7 (Lernkontrolle)

a) Vgl. Theorie S. 14

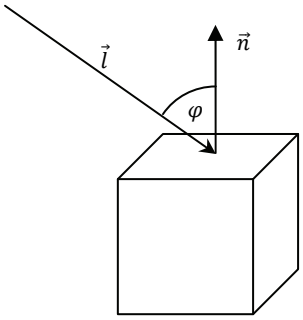
b) Bildraumverfahren sind universell einsetzbar, Objektraumverfahren basieren auf den Eigenschaften spezieller Körper. Dafür sind Objektraumverfahren effizienter, da jedes Objekt nur einmal behandelt wird.

c) Vgl. Theorie S. 16.

Lösungen zum 5. Kapitel

S. 18

Aufgabe 1



$$I = I_L \cdot k \cdot \cos(\varphi) = I_L \cdot k \cdot \left(-\vec{l} \cdot \vec{n}\right), \text{ wobei } \left|\vec{l}\right| = |\vec{n}| = 1$$

S. 19

Aufgabe 2

Hier also die Methode zur Bestimmung der Lichtintensität nach Lambert:

```
private double computeLambert(Face flaeche, Vector l)
{
     Vector n = polyeder.getNormal(flaeche);
     double lambert = -n.dot(l);
     if(lambert < 0)
          lambert = 0;
     return lambert;
}
```

Die vollständige Lösung finden Sie in der Datei Polyeder3D_v09.java.