Fachhochschule Nordwestschweiz
Hochschule für Technik

Mathematische Grundlagen der 3D Graphik

E.Gutknecht

Wer hohe Türme bauen will, muss lange beim Fundament verweilen.

—— N.N.

FS 2019

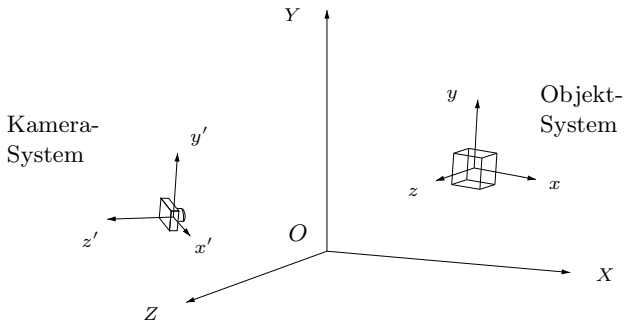
Inhaltsverzeichnis

Kapitel 1

Grundlagen von OpenGL

1.1 Koordinatensysteme

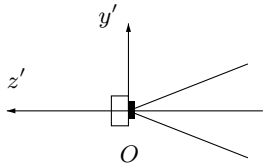
Das A und O der 3D Graphik sind Koordinatensysteme. OpenGL verwendet drei Systeme: Das absolute System *(World-System)* und zwei bewegliche Systeme, das *Kamera-System* und das *Objekt-System*.



Gemäss der Konvention der Computergraphik sind die Koordinatenachsen so gewählt sind, dass die y-Achse nach oben zeigt.

Das Kamera-System (Viewing-System, Eye-System)

Dies ist ein Koordinatensystem, welches fest mit einer virtuellen Kamera verbunden ist. In diesem System wird das Bild der Objekte erzeugt. Die Kamera befindet sich fix im Nullpunkt des Systems und ist in die negative z-Richtung gerichtet:



Die Erzeugung des Bildes eines Objektes erfolgt (wie bei einer richtigen Kamera) durch Projektion auf eine Ebene parallel zur $x'y'$-Ebene des Kamera-Systems (Filmebene).

Für die Projektion der Punkte wird die *Orthogonal-* oder *Zentralprojektion (Perspektive)* verwendet. Da die Bildebene parallel zur Aufrissebene ist, und die Kamera sich im Nullpunkt des Systems befindet, sind beide Projektionen mathematisch sehr einfach.

Das Objekt-System (Model-System, Body-Frame)

Dies ist das Koordinatensystem, auf welches sich die Koordinaten eines Objektes, z.B. eines Würfels, beziehen. Es wird so gewählt, dass die Koordinaten der Punkte des Objektes möglichst einfach sind.

Merke:

Bei der Definition eines Objektes beziehen sich die angegebenen Koordinaten der Punkte des Objektes immer auf das Objekt-System.

Homogene Koordinaten

Die Koordinaten aller Punkte sind in der 3D Graphik immer *homogene Koordinaten*. Darunter versteht man die Raumkoordinaten x, y, z des Punktes, ergänzt durch eine vierte Koordinate w, welche immer fix gleich 1 ist:

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Die homogenen Koordinaten sind erforderlich, damit auch Translationen mit Matrizen dargestellt werden können (4×4 Matrizen). Details, siehe Anhang 2, Seite 36.

Festlegung der Koordinatensysteme

Am Anfang eines Programmes fallen die drei Systeme zusammen (Ausgangslage). Sie können mit *Drehungen* und *Translationen* in eine gewünschte Lage im absoluten System bewegt werden.

Zweidimensionale Darstellungen

Für zweidimensionale Darstellungen bleibt das Kamera-System in der Ausgangslage, und man setzt alle z-Koordinaten gleich 0 (Darstellung in der xy-Ebene).

1.2 Die ModelView-Transformation

Der Grundpfeiler für 3D Darstellungen ist eine *Koordinatentransformation*, die ModelView-Transformation. Jeder Raumpunkt P hat drei Sätze von Koordinaten:

x	Koordinaten von P im Objekt-System	object coordinates
X	absolute Koordinaten von P	world coordinates
x'	Koordinaten von P im Kamera-System	eye coordinates

Beim Zeichnen einer Figur werden im ersten Schritt die Koordinaten der Punkte der Figur vom Objekt-System in das Kamera-System transformiert. Dies ist die *ModelView-Transformation*.

Alle weiteren Schritte (Projektion auf die Bildebene, Beleuchtungsrechnung, Sichtbarkeit) erfolgen mit den transformierten Koordinaten, d.h. im Kamera-System. Die anderen Koordinatensysteme werden nicht mehr benötigt.

Man beachte, dass die ModelView-Transformation eine Koordinatentransformation ist, d.h. die Raumpunkte werden *nicht* bewegt.

Transformationsmatrizen

Die ModelView-Transformation erfolgt mit Matrizen:

- Die **Model-Matrix** M transformiert die Koordinaten eines Raumpunktes vom Objekt- in das absolute System: *Model-Matrix*

$$X = M \cdot x \tag{1.1}$$

- Die **View-Matrix** V transformiert die absoluten Koordinaten des Punktes in das Kamera-System: *View-Matrix*

$$x' = V \cdot X \tag{1.2}$$

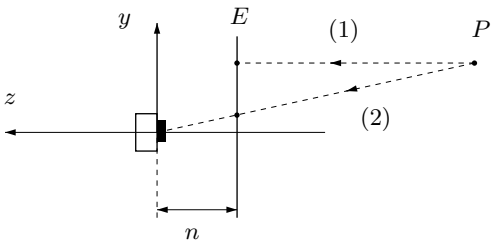
Die Koordinaten sind die homogenen Koordinaten mit 4 Komponenten als Spaltenvektoren, und die Matrizen haben infolgedessen das Format 4×4. Wie schon erwähnt, sind die homogenen Koordinaten erforderlich, damit auch Translationen der Systeme so dargestellt werden können.

1.3 Projektion auf die Bildebene

Nach der ModelView-Transformation folgt die Projektion auf eine Bildebene (Bildschirm) im Kamera-System. Dies ist (wie die Filmebene der Kamera) eine Ebene parallel zur xy-Ebene des Kamera-Systems in einem wählbaren Abstand von der Kamera.

Dazu wird die *Orthogonalprojektion* oder die *Zentralprojektion* (Perspektive) verwendet. Die Zentralprojektion entspricht einer wirklichen Betrachtung der Szene durch ein Fenster, die Orthogonalprojektion ist eine Näherung.

Kamera-System:



E Bildebene
P Raumpunkt
(1) Normalprojektion
(2) Zentralprojektion
n Abstand der Ebene E von O

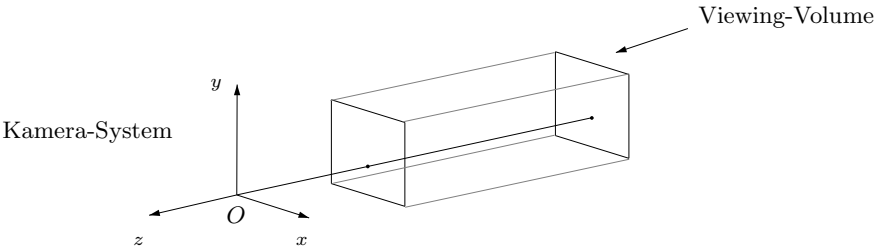
Die Projektionsmatrix

Zu den Matrizen M und V der ModelView-Transformation kommt noch eine dritte Matrix hinzu, die *Projektionsmatrix* P. Sie bestimmt die Projektionsart (Orthogonal- oder Zentralprojektion) und legt zusätzlich den *Sichtbereich* fest.

Der Sichtbereich (Viewing-Volume)

Dies ist der Bereich der Raumpunkte, die bei der Projektion in das Ausgabe-Window abgebildet werden, d.h. auf dem Bild sichtbar sind.

Bei der Orthogonalprojektion ist der Sichtbereich ein Quader mit Kanten parallel zu den Koordinatenachsen des Kamera-Systems:

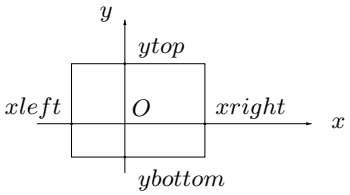


Der Sichtbereich wird mit seinen Grenzen auf den Koordinatenachsen festgelegt:

$$\mathsf{xleft} \le \mathsf{x} \le \mathsf{xright}, \quad \mathsf{ybottom} \le \mathsf{y} \le \mathsf{ytop}$$

Die Grenzen auf der z-Achse werden durch die Abstände $znear$ und $zfar$ vom Nullpunkt in Blickrichtung spezifiziert, d.h. positive Werte liegen auf der *negativen* z-Achse (wie in der obigen Figur).

Aufriss (in Blickrichtung):



Die Grenzen des Sichtbereiches sind beliebig wählbar und beziehen sich auf das Kamera-System. Sie werden so gewählt, dass die darzustellenden Figuren im Sichtbereich liegen. Teile von Figuren ausserhalb des Sichtbereiches werden abgeschnitten (clipped). Der Sichtbereich heisst daher auch *Clipping-Volume*.

ClippingVolume

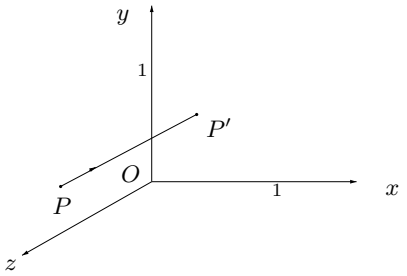
Der Sichtbereich der Zentralprojektion wird später eingeführt.

1.4 Positionierung der Koordinatensysteme

Die Default-Konfiguration

Wenn die drei Matrizen M, V und P alle gleich der Einheitsmatrix sind, fallen die drei Systeme zusammen. Dies ist die *Default-Konfiguration* von OpenGL.

Bei dieser Konfiguration werden die Punkte orthogonal auf die xy-Ebene projiziert und das Viewing-Volume ist der Würfel mit den Grenzen -1 und 1 auf jeder Koordinatenachse.



Diese Konfiguration eignet sich für 2D Darstellungen in der xy-Ebene. Dazu werden die z-Koordinaten einfach gleich 0 gesetzt.

Für andere Konfigurationen sind die Matrizen M und V zu definieren. Sie enthalten alle benötigten Informationen der Koordinatensysteme, weil sie die ModelView-Transformation vollständig festlegen.

Merke:

Für die Festlegung des Objekt- und Kamera-Systems müssen nur die Matrizen M und V definiert werden.

Positionierung Objekt- und Kamera-System

- Das Objekt-System wird normalerweise mit Bewegungen (Drehungen und Translationen) in eine gewünschte Lage gebracht. Jede Bewegung bedeutet eine Multiplikation der momentanen Model-Matrix M mit der Matrix der Bewegung. Dabei spielt es eine Rolle, ob diese Matrix links oder rechts zu M multipliziert wird. Dies werden wir später genauer untersuchen (Koordinatentransformationen).

  Beispiel:

  Drehung des Objekt-Systems in Bezug auf seine momentane Lage:

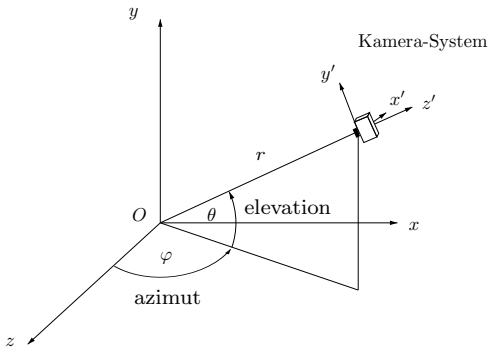
  $$M = M \cdot R \qquad R \text{ Drehmatrix}$$

  Die mathematischen Grundlagen dazu sind ein wichtiges Thema unseres Moduls.

- Für die Festlegung des Kamera-Systems werden zwei Verfahren verwendet, das *LookAt–* und das *Azimut/Elevation–Prinzip*.

  Im Moment verwenden wir nur das Azimut-Elevation-Prinzip. Das LookAt-Prinzip führen wir später ein (Vektorprodukt).

  Das Azimut-Elevation-Prinzip



  Das Lage des Kamera-Systems wird durch die View-Matrix festgelegt, und die Projektionsart (Orthogonal- oder Zentralprojektion) mit dem Viewing-Volume durch die Projektionsmatrix.

1.5 Beleuchtung

Für die Darstellung von räumlichen Körpern ist eine Beleuchtungsrechnung wichtig, d.h. die Helligkeiten der Punkte auf der Oberfläche eines Körpers werden (wie in Wirklichkeit) aufgrund des reflektierten Lichtes berechnet.

Bei der Reflexion des Lichtes sind zwei Reflexionsarten zu unterscheiden, die *diffuse* und die *spiegelnde* Reflexion. Die diffuse oder allseitige Reflexion ist bei rauhen Oberflächen aktuell (Mond, Wand), die spiegelnde bei glatten Flächen (Metall, Glas).

Die resultierende Helligkeit eines Punktes einer Oberfläche ist die Summe einer Grundhelligkeit (ambientes Licht) und dem diffus und spiegelnd reflektierten Licht.

Für die Beleuchtungsrechnung werden *Normalenvektoren* in den Punkten der beleuchteten Flächen benötigt.

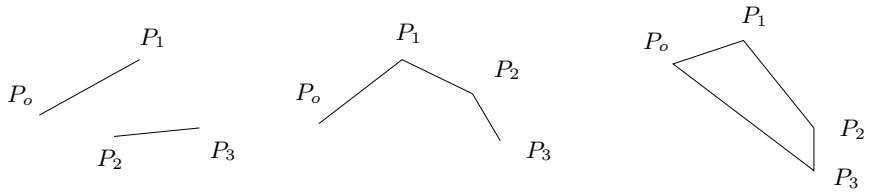
Beleuchtungsparameter:

- Position der Lichtquelle (Sonne)
- Grundhelligkeit (ambientes Licht)
- Reflexionskoeffizient für diffuse Reflexion
- Parameter für spiegelnde Reflexion

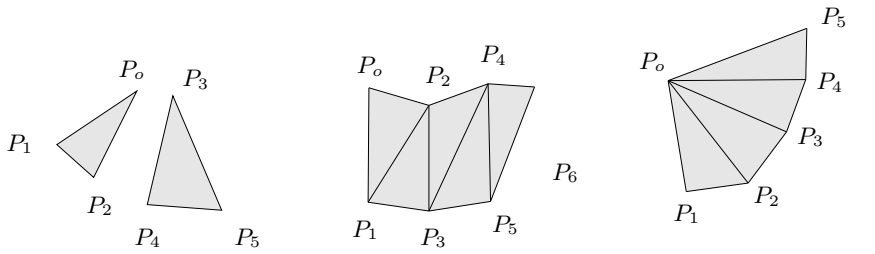
1.6 Grundfiguren und Vertices

OpenGL ist ein Lowlevel-System, d.h. es enthält nur Funktionen für die Darstellung von einfachen Grundfiguren: *Punkte*, *Strecken* und *Dreiecke*. Alle Objekte werden mit diesen Grundfiguren dargestellt.

Die folgenden Grundfiguren (Primitives) werden von OpenGL unterstützt:



GL_LINES GL_LINE_STRIP GL_LINE_LOOP



GL_TRIANGLES GL_TRIANGLE_STRIP GL_TRIANGLE_FAN

Weiter können auch auch einzelne Punkte dargestellt werden (GL_POINTS).

Vertices

Die Grundfiguren werden durch ihre Eckpunkte, die sog. Vertices definiert. Ein *Vertex* (Scheitelpunkt) ist ein Raumpunkt, versehen mit Attributen: *Vertex*

- *Koordinaten*

  Die Koordinaten eines Vertex sind die homogenen Koordinaten, d.h. die Raumkoordinaten x, y, z, ergänzt mit einer vierten Koordinate w, welche fix gleich 1 ist.

  Für zweidimensionale Darstellungen in der xy-Ebene wird einfach die z-Koordinate der Vertices auf den Wert 0 gesetzt.

- *Farbe*

  Die Farbe eines Vertex wird durch die Rot-, Grün-, Blau-Werte der Farbe und die Durchsichtigkeit spezifiziert:

  $$0 \leq r, g, b, a \leq 1$$

  Wir verwenden nur $a = 1$ (keine Durchsichtigkeit).

- *Normalenvektor*

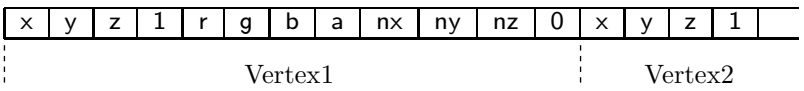
  Der Normalenvektor eines Vertex wird bei der Beleuchtungsrechnung verwendet. Er definiert die Richtung senkrecht zur Oberfläche des Körpers, zu dem der Vertex gehört und zeigt nach aussen.

  Normalenvektoren haben ebenfalls eine vierte (homogene) Komponente w, welche jedoch gleich 0 (statt 1) gesetzt wird. Dies hat mit dem Transformationskonzept zu tun.

- Weitere Attribute nach Bedarf, z.B. Textur-Koordinaten für Texturen (Oberflächenmuster).

Der Vertex-Array

Die Vertices einer Figur werden in einen Array, den *Vertex-Array (Vertex-Buffer)* eingetragen. Dies ist ein linearer Array von Float-Werten, in welchem die Attribute der Vertices sequentiell gespeichert werden. Reihenfolge der Vertices gemäss den obigen Figuren.



Diese Speicherung der Attribute heisst *interleaved*. Alternativ können sie auch in separaten Arrays gespeichert werden.

Zeichnen der Figur

Die Ausgabe der Figur auf den Bildschirm erfolgt in den folgenden beiden Schritten:

1. Der Vertex-Array wird in einen zugehörigen OpenGL-Buffer auf der Graphik-Karte kopiert.

2. Die Figur wird mit der OpenGL-Funktion glDrawArrays gezeichnet. Der gewünschte Typ der Grundfigur wird als Parameter angegeben (z.B. GL_LINES), gemäss Seite 8. *glDrawArrays*

1.7 Die Rendering-Pipeline

Unter *Rendering* versteht man die Umwandlung einer mathematisch definierten Figur (Beispiel Würfel) in das Bild auf dem Bildschirm, d.h. in ein Bitmap-Bild im Frame-Buffer des Windows.

Unter einer *Pipeline* versteht man in diesem Zusammenhang eine mehrstufige Verarbeitung, bei welcher der Output eines Schrittes als Input des nachfolgenden Schrittes verwendet wird.

Shader-Programme

Bei der Programmable Pipeline (PPP) des aktuellen OpenGL werden *Shader-Programme* aufgerufen: der *Vertex-Shader* und der *Fragment-Shader*. PPP

Shader-Programme sind Programme, die auf dem Prozessor der Graphik-Karte laufen und ebenfalls vom Benutzer geschrieben werden. Sie werden in der *OpenGL Shaderlanguage GLSL* geschrieben, welche auf *C* aufbaut. Sie werden bei jeder Ausführung des Programmes 'on the fly' compiliert und linked.

Schritte der Programmable Pipeline

Beim Aufruf der Funktion glDrawArrays zum Zeichnen einer Figur wird die folgende *OpenGL Rendering-Pipeline* gestartet.

1. *Aufruf des Vertex-Shaders*

   Im ersten Schritt der Pipeline wird für jeden Vertex der *Vertex-Shader* aufgerufen. Dieser erhält die Attribute des Vertex und weitere Daten (Transformationsmatrizen, Beleuchtungsparameter).

   Da der Vertex-Shader vom Anwender geschrieben wird, sind seine Funktionen nicht festgelegt. Wir verwenden einen Standard-Shader mit den folgenden Funktionen:

   - ModelView-Transformation

     Die Koordinaten des Vertex und des Normalen-Vektors werden mit den Matrizen M und V in das Kamera-System transformiert.

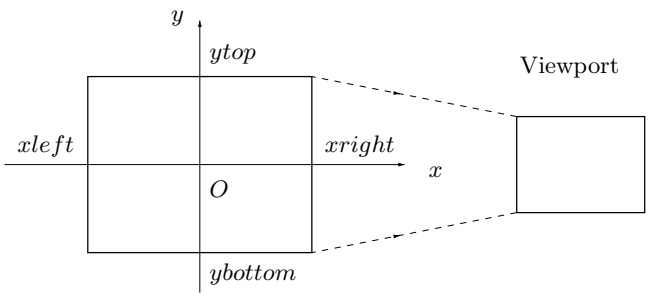
     Die weitere Bilderzeugung erfolgt ausschliesslich mit diesen Koordinaten, d.h. im Kamera-System (Kamera im Nullpunkt, Blickrichtung gegen die negative z-Achse).
   - Projektion auf die Bildebene mit der Matrix P
   - Weitergabe von Daten für den Fragment-Shader

2. *Assembly und Clipping*

   Nach dem Vertex-Shader werden die Vertices zu der gewünschten Figur zusammengesetzt, dann wird die Figur auf das Viewing-Volume zugeschnitten. Dabei werden i.a. neue Vertices (Schnittpunkte) erzeugt.

3. *Die Viewport-Transformation*

   Der *Viewport* ist das Ausgaberechteck für das Bild auf dem Bildschirm. Die *Viewport-Transformation* transformiert das Bildrechteck der Kamera in der Bildebene in die ganzzahligen Koordinaten (col, line) des ViewPorts. *Viewport*



   Die z-Koordinaten werden dabei unverändert für den Sichtbarkeitstest übernommen. Sie stellen den Abstand des Vertex von der Bildebene dar (Tiefeninformation).

4. *Rasterung*

   Bei der Rasterung werden die Bildpunkte (Pixel) der Figur zwischen den Vertices berechnet. Für jeden dieser Punkte wird ein Datensatz, ein sogenanntes *Fragment* erstellt. Dieses enthält die Bildschirm-Koordinaten, die Tiefeninformation (z-Wert) und die Farbe des Pixels. Dabei werden die Werte der Vertices interpoliert. *Fragment*

5. *Aufruf des Fragment-Shaders*

   Vor der Ausgabe eines Pixels wird der *Fragment-Shader* aufgerufen. Dies ist der zweite obligatorische Shader eines Programmes. Er kann bei Bedarf die Attribute des Fragmentes verändern.

   Wir verwenden einen Standard-Shader, der die Beleuchtungsrechnung ausführt, wenn die Beleuchtung eingeschalten ist.

Dies ist die sog. *Programmable Pipeline (PPP)* der aktuellen Version von OpenGL. Die Bezeichnung kommt daher, dass die Shader-Programme nach Bedarf programmiert werden können. Die frühere *Fixed Function Pipeline (FFP)* von OpenGL 1.1 hatte diese Möglichkeit nicht. *PPP* *FFP*

1.8 JOGL-Programme

Das Graphik-System OpenGL besteht aus einer Library von C-Funktionen, die mit diversen Programmiersprachen aufgerufen werden können. Für den Aufruf der Funktionen in Java-Programmen steht das Java OpenGL-Binding JOGL zur Verfügung. Dieses stellt die Funktionen als Methoden eines Objektes der Klasse GL2 (OpenGL2.x) oder GL3 (OpenGL3.x) zur Verfügung. *JOGL*

JOGL verwendet das Java Native Interface (JNI) für den Aufruf der C-Funktionen von OpenGL. Installation von JOGL, siehe Anhang 1 (S.35).

Ein JOGL Programm ist eine Java Applikation mit einer 'main'-Methode und Methoden zur Verarbeitung von OpenGL-Ereignissen. Die Event-Methoden sind im Interface GLEventListener spezifiziert :

Methode	Ereignis
init	Initialisierung des Programmes
display	Bild ausgeben
reshape	Window Grössen-Veränderung
dispose	Methode nicht verwendet

Das folgende Listing stellt das kleinste JOGL-Programm dar. Es gibt ein leeres OpenGL-Window und in der Konsole die Version von OpenGL und der Shader-Language (GLSL) des Computers aus.

```
import java.awt.*;
import java.awt.event.*;
import com.jogamp.opengl.*;
import com.jogamp.opengl.awt.*;

public class MyFirstGL implements WindowListener, GLEventListener
{
   // ---------- globale Daten ----------
   String windowTitle = "JOGL-Application";
   int windowWidth = 800;
   int windowHeight = 600;
   Frame frame;
   GLCanvas canvas;                              // OpenGL Window

   // ---------- Methoden ----------
   public MyFirstGL()                            // Konstruktor

   { createFrame();
   }
   void createFrame()                            // Fenster erzeugen
   { Frame f = new Frame(windowTitle);
     f.setSize(windowWidth, windowHeight);
     f.addWindowListener(this);
     GLProfile glp = GLProfile.get(GLProfile.GL3);
     GLCapabilities glCaps = new GLCapabilities(glp);
     canvas = new GLCanvas(glCaps);
     canvas.addGLEventListener(this);
     f.add(canvas);
     f.setVisible(true);
   }
```

continued

```
    // ____________ OpenGL−Events ____________________
    @Override
    public void init(GLAutoDrawable drawable)   //  Initialisierung
    { GL3 gl = drawable.getGL().getGL3();
      System.out.println("OpenGl Version: " + gl.glGetString(gl.GL_VERSION));
      System.out.println("Shading Language: " +
                         gl.glGetString(gl.GL_SHADING_LANGUAGE_VERSION));
      System.out.println();
      gl.glClearColor(0,0,1,1);                    // Hintergrundfarbe (RGBA)
    }
    @Override
    public void display(GLAutoDrawable drawable)   // Bildausgabe
    { GL3 gl = drawable.getGL().getGL3();
      gl.glClear(gl.GL_COLOR_BUFFER_BIT);        // Bild loeschen
    }
    @Override
    public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height)
    { GL3 gl = drawable.getGL().getGL3();
      gl.glViewport(0, 0, width, height);            // Set the viewport to be the entire window
    }     @Override
    public void dispose(GLAutoDrawable drawable) { }
    // ____________ main−Methode ____________________
    public static void main(String[] args)
    { new MyFirstGL();
    }
    // ____________ Window−Events ____________________
    public void windowClosing(WindowEvent e)
    { System.out.println("closing window");
      System.exit(0);
    }
    public void windowActivated(WindowEvent e) { }
    public void windowClosed(WindowEvent e) { }
    public void windowDeactivated(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
    public void windowOpened(WindowEvent e) { }
}
```

Erklärungen

1. Das Objekt gl der Klasse GL3 enthält alle OpenGL-Methoden. Es wird vom Parameter drawable abgerufen.

2. Methode display

   Die Methode wird immer aufgerufen, wenn das Window des Programmes auszugeben ist.

3. Methode reshape

   Die Methode wird erstmalig vor display und danach immer nach Veränderungen des Formates des Windows durch den Benutzer aufgerufen. Sie erhält die Abmessungen des width, height und eignet sich daher zur Festlegung des *Viewports*.

   Viewport

   Der Viewport ist das Rechteck, in welches OpenGL zeichnet.

   Die Parameter der Methode gl.glViewport sind left, top, width, height. Die ersten beiden geben die Koordinaten der linken oberen Ecke des Viewports im OpenGL-Window an und die weiteren die Abmessungen des Viewports.

Ein vollständiges Programm mit graphischen Darstellungen benötigt zusätzlich einen Vertex-Array sowie einen Vertex- und Fragment-Shader. Zur Entlastung der Programme von OpenGL-technischen Definitionen führen wir dazu einige Java-Klassen ein. Diese sind in zwei Packages unterteilt:

- *OpenGL Hilfsmethoden*

  Package ch.fhnw.util.opengl

  Klassen:

  MyShaders
  : Die Klasse enthält unsere Standard-Shaders und eine Methode für Compilation und Link der Shaders.

  VertexArray
  : Implementation eines Vertex-Arrays und Methoden für seine Verwendung und die Definition von Vertices mit ihren Attributen

  Transform
  : Methoden für die Definition und Modifikation der Transformationsmatrizen M und V für die Festlegung des Objekt- und Kamera-Systems, sowie für die Projektionsmatrix P.

    Zusätzlich enthält die Klasse Methoden zur Festlegung der Position der Lichtquelle und der Beleuchtungsparameter.

  Die Methoden übergeben die Daten an die zugehörigen Shader-Variabeln. Sie benötigen dazu die OpenGL Programm-Identifikation und ein Objekt der Klasse GL3 mit den OpenGL-Methoden.

- *Vektor- und Matrix-Algebra*

  Package ch.fhnw.util.math

  Klassen:

  Vec3, Vec4, Mat3, Mat4

1.8.1 Die Klasse MyShaders

Die Klasse enthält zwei Vertex-Shaders vShader0, vShader1 und zwei Fragment-Shaders fShader0 und fShader1. Diese sind in der Klasse direkt als Strings definiert, da die OpenGL-Funktion für die Compilation den Shader-Code als String-Parameter erwartet.

- Die Shaders vShader0 und fShader0 sind sog. 'pass through' Shaders, d.h. sie geben die erhaltenen Daten unverändert weiter.
- Die Shaders vShader1 und fShader1 erfüllen die Standard-Funktionen der Renderig-Pipeline, d.h. die ModelView-Transformation, die Projektion und die Beleuchtungsrechnung.

Der Vertex-Shader vShader0

```
#version 330                  /* Shader Language Version */
in vec4 vPosition, vColor;    /* Vertex−Attribute */
out vec4 fColor;              /* Fragment−Farbe */
void main()
{ gl_Position = vPosition;
  fColor = vColor;
}
```

Erklärungen:

– Die *Shader-Language* GLSL basiert auf der Sprache *C* und umfasst zusätzliche Datentypen und Operationen für Vektoren und Matrizen: *GLSL*

vec3, vec4, mat3, mat4, '+', '−', '∗' (Matrix-Multiplikation)

– 'in'-Variabeln

Die Vertex-Attribute werden im Vertex-Shader als 'in'-Variablen definiert. Die Namen sind beliebig, sie werden im Hauptprogramm den Vertex-Attributen zugeordnet.

– 'out'-Variabeln

Für die Weitergabe der Farbe des Vertex wird die 'out' Variable fColor (Fragment-Color) definiert. Die out Variabeln des Vertex-Shaders können als Input-Variablen des Fragment-Shaders verwendet werden.

– Die 'main'-Funktion

In der 'main'-Funktion werden die Vertex-Koordinaten in die OpenGL-Variable gl_Position kopiert. Die Farbe wird unverändert in die Output-Parameter fColor übertragen.

Der Fragment-Shader fShader0

Der Shader überträgt die Farbe unverändert in die out Variable fragColor:

```
#version 330
in vec4 fColor;
out vec4 fragColor;
void main()
{ fragColor = fColor;
}
```

Der Vertex-Shader vShader1

Der Shader erfüllt die folgenden Funktionen:

- ModelView-Transformation der Vertex- und Normalen-Koordinaten mit den Matrizen M und V
- Projektion der Vertex-Koordinaten mit der Matrix P
- Weitergabe von Daten an den Fragment-Shader

Die Matrizen M, V und P werden dem Shader im Java-Programm übergeben. Details zum Code des Shaders, siehe später.

Der Fragment-Shader fShader1

Funktion:

- Bestimmung der definitiven Farbe des Fragmentes mit der Beleuchtungsrechnung

Die Farbe des Fragmentes, die der Fragment-Shader erhält, ist eine von OpenGL interpolierte Farbe der umgebenden Vertices des Fragmentes.

Falls die Beleuchtung aktiviert ist, wird die Helligkeit der Farbe mit der Beleuchtungsrechnung angepasst, sonst wird sie nicht verändert.

Compilation und Link der Shaders

Die Klasse enthält weiter eine Methode für Compile und Link der Shaders:

```
public static int initShaders(GL3 gl,
        String vShader,              // Vertex−Shader
        String fShader)              // Fragment−Shader
```

Das Resultat der Methode ist eine OpenGL-Programm-Identifikation, die für andere OpenGL-Aktionen benötigt wird.

Aufruf der Methode:

```
int programId = MyShaders.initShaders(gl,MyShaders.vShader1,
                                      MyShaders.fShader1);
```

1.8.2 Die Klasse VertexArray

Die Klasse implementiert einen Vertex-Array und stellt Methoden für Vertex- und Buffer-Operationen zur Verfügung.

VertexArray(GL3 gl, int programId, int maxVerts) // Konstruktor

Der Parameter programId ist die OpenGL-Identifikation des Programmes (Resultat von initShaders).

maxVerts max. Anzahl Vertices im Array

void putVertex(float x, float y, float z)

Speichere einen Vertex mit seinen Attributen in den Vertex-Array. Die Koordinaten werden von den Parametern entnommen, die Farbe und die Normale von globalen Variabeln, welche mit den nachfolgenden Methoden gesetzt werden können.

void setColor(float r, float g, float b)

Setze die Vertexfarbe für nachfolgend definierte Vertices. Die Farbkomponenten r, g, b sind Werte im Intervall $[0, 1]$.

void setNormal(float x, float y, float z)

Setze die Normale für nachfolgend definierte Vertices.

void rewindBuffer(GL3 gl)

Rückstellung des Vertex-Arrays (setzt die Einfügeposition auf 0)

void copyBuffer(GL3 gl)

Kopiere den Vertex-Array in den zugehörigen OpenGL-Buffer

void drawArrays(GL3 gl, int figureType)

Zeichne die Figur mit den Vertices im Vertex-Array als Grundfigur des Typs figureType. Dabei ist figureType eine der Konstanten gl.GL_LINES, gl.GL_LINE_STRIP usw., gemäss der Figur auf Seite 8. Beim Aufruf dieser Methode wird die Rendering-Pipeline gestartet.

void drawArrays2(GL3 gl, int figureType, int startVertex, int nVertices)

Wie die vorangehende Methode, aber nur mit den Vertices ab Index startVertex bis startVertex+nVertices−1.

void drawAxis(GL3 gl, float a, float b, float c);

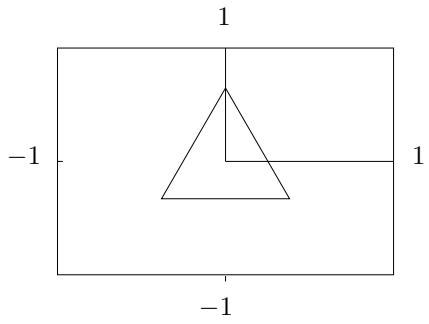
Zeichne die Koordinatenachsen mit den Längen a, b und c.

Beispiel: Strecke AB im Raum zeichnen

```
public void zeichneStrecke(GL3 gl, VertexArray vArray,
                           Vec3 A, Vec3 B, Vec3 color)
{ vArray.rewindBuffer(gl);
  vArray.setColor(color.x,color.y,color.z); // Farbe setzen
  vArray.putVertex(A.x,A.y,A.z);
  vArray.putVertex(B.x,B.y,B.z);
  vArray.copyBuffer(gl);
  vArray.drawArrays(gl, GL3.GL_LINES);
}
```

1.8.3 Vollständiges 2D-Beispiel

Das folgende Programm zeichnet ein gleichseitiges Dreieck in der xy-Ebene (z-Koordinaten null). Das Programm verwendet die Default-Konfiguration (ohne Vertex-Transformationen). Dazu werden nur die 'pass through' Shaders vShader0 und fShader0 benötigt.



Die Ecken des Dreiecks erhalten die Farben Rot, Grün bzw. Blau. Dies bewirkt, dass OpenGL die Farben der inneren Punkte interpoliert, sodass kontinuierliche Farbübergänge entstehen.

```
1  // ———— JOGL 2D Beispiel−Programm (Gleichseitiges Dreieck) ————————
2  import java.awt.*;
3  import java.awt.event.*;
4  import com.jogamp.opengl.*;
5  import com.jogamp.opengl.awt.*;
6  import ch.fhnw.util.opengl.MyShaders;
7  import ch.fhnw.util.opengl.VertexArray;
8
9  public class MyFirst2D
10       implements WindowListener, GLEventListener
11 {
12    // ———— globale Daten ————————————
13    String windowTitle = "JOGL−Application";
14    int windowWidth = 600;                  // Window−Groesse
15    int windowHeight = 600;
16    String vShader = MyShaders.vShader0;    // Vertex−Shader
17    String fShader = MyShaders.fShader0;    // Fragment−Shader
18    Frame frame;                            // Java−Frame
19    GLCanvas canvas;                        // OpenGL Window
20    int programId;                          // OpenGL Program−Ident.
21    VertexArray vArray;
22    int maxVerts = 2048;                    // max. Anzahl Vertices im Vertex−Array
23    float s = 1.2f;                         // Dreiecksseite
24    float h = 0.5f*s*(float)Math.sqrt(3);   // Hoehe
25
26    // ———— Methoden ————————————————
27    public MyFirst2D()                      // Konstruktor
28    { createFrame();
29    }
30
31    void createFrame()                      // Fenster erzeugen
32    { Frame f = new Frame(windowTitle);
33      f.setSize(windowWidth, windowHeight);
34      f.addWindowListener(this);
35      GLProfile glp = GLProfile.get(GLProfile.GL3);
36      GLCapabilities glCaps = new GLCapabilities(glp);
37      canvas = new GLCanvas(glCaps);
38      canvas.addGLEventListener(this);
39      f.add(canvas);
40      f.setVisible(true);
41    }
```

```
42
43     public void zeichneDreieck(GL3 gl, VertexArray vArray,
44                                float x1, float y1,
45                                float x2, float y2,
46                                float x3, float y3)
47     { vArray.rewindBuffer(gl);
48       vArray.setColor(1,0,0);                    // Rot
49       vArray.putVertex(x1,y1,0);
50       vArray.setColor(0,1,0);                    // Gruen
51       vArray.putVertex(x2,y2,0);
52       vArray.setColor(0,0,1);                    // Blau
53       vArray.putVertex(x3,y3,0);
54       vArray.copyBuffer(gl);
55       vArray.drawArrays(gl, gl.GL_TRIANGLES);
56     }
57
58     // ____________ OpenGL-Events ____________________
59
60     @Override
61     public void init(GLAutoDrawable drawable) //  Initialisierung
62     { GL3 gl = drawable.getGL().getGL3();
63       System.out.println("OpenGl Version: " + gl.glGetString(gl.GL_VERSION));
64       System.out.println("Shading Language: " + gl.glGetString(gl.GL_SHADING_LANGUAGE_VERSION));
65       System.out.println();
66       gl.glClearColor(0,0,0,1);                  // Hintergrundfarbe
67       // Compile and Link Shaders
68       programId = MyShaders.initShaders(gl,vShader,fShader);
69       vArray = new VertexArray(gl, programId, maxVerts);
70    }
71
72     @Override
73     public void display(GLAutoDrawable drawable)
74     { GL3 gl = drawable.getGL().getGL3();
75       gl.glClear(GL3.GL_COLOR_BUFFER_BIT); // Frame-Buffer loeschen
76       vArray.setColor(0.7f,0.7f,0.7f);
77       vArray.drawAxis(gl,5,5,5);                 // Koordinatenachsen zeichnen
78       float s2 = 0.5f*s;
79       zeichneDreieck(gl,vArray,-s2,-h/3,s2,-h/3,0,2*h/3);
80     }
81
82     @Override
83     public void reshape(GLAutoDrawable drawable, int x, int y,
84                         int width, int height)
85     { GL3 gl = drawable.getGL().getGL3();
86       // ----Set the viewport to the entire window
87       gl.glViewport(0, 0, width, height);
88     }
89
90     @Override
91     public void dispose(GLAutoDrawable drawable) { } // not needed
92
93     // ____________ main-Methode ____________________
94
95     public static void main(String[] args)
96     { new MyFirst2D();
97     }
98
99     // ____________ Window-Events ____________________
100
101    // wie im Programm MyFirstGL 102
103 }
```

Erklärungen:

1. Zeile 68: Compilation und Link der Shaders
2. Zeile 69: Erzeugung eines Objektes der Klasse VertexArray
3. Zeile 79: Gleichseitiges Dreieck mit Ecken

$$A(-\tfrac{s}{2},-\tfrac{h}{3},0), \quad B(\tfrac{s}{2},-\tfrac{h}{3},0), \quad C(0,\tfrac{2h}{3},0)$$

1.8.4 Die Klasse Transform

Die Klasse enthält Methoden für die Definition und Bearbeitung der Transformations-Matrizen M und V für das Objekt- und Kamera-System, sowie für die Projektionsmatrix P.

Zusätzlich werden die Position der Lichtquelle und die Beleuchtungsparameter mit Methoden der Klasse spezifiziert.

Die Methoden übergeben die Daten an den Vertex- und Fragment-Shader, wo sie zum Einsatz kommen.

Die Klasse Transform ist eine Java-Adaption eines entsprechenden Frameworks in C aus dem Manual "OpenGL ES 3.0 Programming Guide", Second Edition 2014, Dan Ginsburg et a.

Konstruktor

ObjectSystem(GL3 gl, int programId)

Der Parameter programId ist die OpenGL-Identifikation des Programmes (Resultat von initShaders.

Methoden für die Model-Matrix M (Objekt-System)

resetM(GL3 gl)

Rücksetzung der Model-Matrix auf die Einheitsmatrix (Ausgangsposition des Objekt-Systems)

void rotateM(GL3 gl, float phi, float x, float y, float z)

Multipliziere M von rechts mit der Matrix der Drehung mit Drehwinkel *phi* in Grad und Drehachse mit Komponenten x, y, z. Dies entspricht geometrisch einer Drehung des momentanen Objekt-Systems relativ zur momentanen Lage.

void translateM(GL3 gl, float x, float y, float z)

Analog rotateM mit der Translation anstelle der Drehung.

void scaleM(GL3 gl, float sx, float sy, float sz)

Multipliziere M von rechts mit der Matrix der Skalierung (Streckung) mit den Streckungsfaktoren sx, sy, sz. Dies bedeutet geometrisch eine Skalierung der Koordinatenachsen.

void multM(GL3 gl, Mat4 A)

Multipliziere M von rechts mit A : $M = M \cdot A$

void pushM()

Speichere M auf einen Stack.

Mat4 popM(GL3 gl)

Hole die oberste Matrix vom Stack und setze sie als aktuelle Matrix M.

Methoden für die View-Matrix V (Kamera-System)

resetV(GL3 gl)

Rücksetzung der View-Matrix auf die Einheitsmatrix (Ausgangsposition des Kamera-Systems)

void lookAt(GL3 gl,Vec3 eye, Vec3 target, Vec3 up)

Setze die ViewMatrix für ein Kamera-System, welches gemäss dem LookAt-Prinzip positioniert ist. Dieses Prinzip wird später eingeführt (Vektorprodukt). Parameter:

eye Kamera-Position, target Zielpunkt, up Aufwärtsrichtung

void lookAt2(GL3 gl, float dist, float azimut, float elevation)

Setze die ViewMatrix für ein Kamera-System in der Lage gemäss dem Azimut-Elevation-Prinzip (Seite 6). Die Winkel azimut und elevation werden in Grad angegeben.

void setV(GL3 gl, Mat4 ViewMatrix)

Direkte Festlegung der View-Matrix

Mat4 getV()

Methoden für Projektions-Matrix P

void ortho(GL3 gl, float xleft, float xright, float ybottom, float ytop, float znear, float zfar)

Berechne die Projektionsmatrix für die Orthogonalprojektion mit dem Viewing-Volume mit den angegebenen Koordinaten-Grenzen

void perspective(GL3 gl, float xleft, float xright, float ybottom, float ytop, float znear, float zfar)

Berechne die Projektionsmatrix für die Zentralprojektion (Perspektive) mit dem Viewing-Volume mit den angegebenen Koordinaten-Grenzen. Details, siehe später.

void setP(GL3 gl, Mat4 ProjectionMatrix)

Direkte Festlegung der ProjektionsMatrix

Mat4 getP()

Methoden für Beleuchtungsparameter

`void setLightPosition(GL3 gl, float x, float y, float z);`

Setze die Koordinaten der Lichtquelle. Die Koordinaten werden mit den aktuellen Matrizen M und V in das Kamera-System transformiert. Im Vertex-Shader werden sie nicht mehr transformiert, da sie nicht zu den Vertices gehören.

`void setShadingLevel(GL3 gl, int level);`

Setze die Beleuchtungsstufe. 0: ohne Beleuchtung, 1: Beleuchtung

`void setShadingParam(GL3 gl, float ambient, float diffuse);`

Setze die Parameter für ambientes Licht und diffuse Reflexion.

Defaultwerte: $ambient = 0.2$, $diffuse = 0.4$

`void setShadingParam2(GL3 gl, float specular, float specExp);`

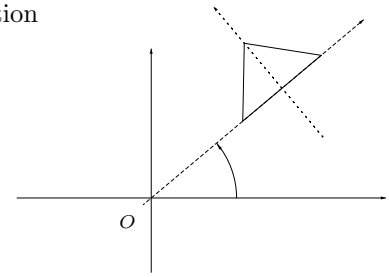
Setze die Parameter für spiegelnde Reflexion.

Defaultwerte: $specular = 0.4$, $specExp = 20$.

Beispiele zu Bewegungen des Objekt-Systems

1. Drehung gefolgt von Translation

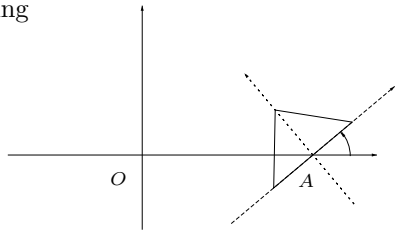
```
objsys.resetM(gl);
objsys.rotateM(phi,0,0,1);
objsys.translateM(d,0,0);
zeichneDreieck(gl);
```



Die Drehachse der Drehung ist die z-Achse, welche senkrecht auf der xy-Ebene steht. Die Translation wird *relativ* zum momentanen System interpretiert, d.h. sie verschiebt in Richtung der gedrehten x-Achse.

2. Translation gefolgt von Drehung

```
objsys.resetM(gl);
objsys.translateM(d,0,0);
objsys.rotateM(phi,0,0,1);
zeichneDreieck(gl);
```

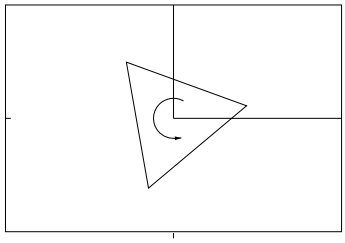


Man beachte, dass die Drehung um den Punkt A dreht, da die Transformationen im momentanen Objekt-System interpretiert werden.

Wie man sieht, spielt die Reihenfolge der Transformationen eine Rolle für das Resultat.

1.8.5 Erweitertes 2D-Beispiel

Wir erweitern das obige Dreiecksprogramm so, dass es eine Animation wird, welche das Dreieck um seinen Mittelpunkt rotieren lässt.



Eine *Animation* ist ein Programm, bei dem die Display-Methode periodisch aufgerufen wird und bei jedem Aufruf das neue Bild der Bewegung zeichnet (wie in einem Film). *Animation*

Animationen können in JOGL sehr einfach realisiert werden. Dazu wird ein Objekt der Klasse FPSAnimator (Package com.jogamp.opengl.util) erzeugt und gestartet. Der Praefix 'FPS' steht für 'frames per second' und bezieht sich auf den zweiten Parameter des Konstruktors, in welcem die gewünschte Anzahl Aufrufe der Display-Methode pro Sekunde angegeben werden kann.

Implementation der Drehung des Dreiecks

1. Bei jedem Frame wird das Objekt-System zunächst in die Ausgangslage zurückgesetzt (resetM).
2. Dann wird das absolute Koordinatensystem gezeichnet.
3. Anschliessend wird das Objekt-System mit der Methode rotateM um einen Winkel *phi* gedreht, und das Dreieck wird gezeichnet.
4. Schliesslich wird der Drehwinkel *phi* für das nächste Frame erhöht.

```
1  // ---------- JOGL 2D Beispiel-Programm (Rotierendes Dreieck) ----------
2  import java.awt.*;
3  import java.awt.event.*;
4  import com.jogamp.opengl.*;
5  import com.jogamp.opengl.awt.*;
6  import com.jogamp.opengl.util.FPSAnimator;
7  import ch.fhnw.util.opengl.MyShaders;
8  import ch.fhnw.util.opengl.VertexArray;
9  import ch.fhnw.util.opengl.Transform;
10 import ch.fhnw.util.math.Vec3;
11 import ch.fhnw.util.math.Mat4;
12
13 public class MyFirst2Da
14       implements WindowListener, GLEventListener, KeyListener
15 {
16
17    // ---------- globale Daten ----------
18
19    String windowTitle = "JOGL-Application";
20    int windowWidth = 600;                  // Window-Groesse
21    int windowHeight = 600;
22    String vShader = MyShaders.vShader1;  // Vertex-Shader
23    String fShader = MyShaders.fShader1;  // Fragment-Shader
```

```
24    Frame frame;                          // Java-Frame
25    GLCanvas canvas;                      // OpenGL Window
26    int programId;                        // OpenGL Program-Ident.
27    VertexArray vArray;
28    Transform transform;
29    int maxVerts = 2048;                  // max. Anzahl Vertices im Vertex-Array
30    float s = 1.2f;                       // Dreiecksseite
31    float h = 0.5f*s*(float)Math.sqrt(3); // Hoehe
32    float phi = 0;                        // Drehwinkel
33    float dphi = 1.0f;                    // Zunahme Drehwinkel
34
35    // ---------- Methoden ----------
36
37    public MyFirst2Da()                   // Konstruktor
38    { createFrame();
39    }
40
41    void createFrame()                    // Fenster erzeugen
42    { Frame f = new Frame(windowTitle);
43      f.setSize(windowWidth, windowHeight);
44      f.addWindowListener(this);
45      GLProfile glp = GLProfile.get(GLProfile.GL3);
46      GLCapabilities glCaps = new GLCapabilities(glp);
47      canvas = new GLCanvas(glCaps);
48      canvas.addGLEventListener(this);
49      f.add(canvas);
50      f.setVisible(true);
51      f.addKeyListener(this);
52      canvas.addKeyListener(this);
53    }
54
55
56    public void zeichneDreieck(GL3 gl, VertexArray vArray,
57                               float x1, float y1,
58                               float x2, float y2,
59                               float x3, float y3)
60    { vArray.rewindBuffer(gl);
61      vArray.setColor(1,0,0);             // Rot
62      vArray.putVertex(x1,y1,0);
63      vArray.setColor(0,1,0);             // Gruen
64      vArray.putVertex(x2,y2,0);
65      vArray.setColor(0,0,1);             // Blau
66      vArray.putVertex(x3,y3,0);
67      vArray.copyBuffer(gl);
68      vArray.drawArrays(gl, gl.GL_TRIANGLES);
69    }
70
71    // ---------- OpenGL-Events ----------
72
73    @Override
74    public void init(GLAutoDrawable drawable) //  Initialisierung
75    { GL3 gl = drawable.getGL().getGL3();
76      System.out.println("OpenGl Version: " + gl.glGetString(gl.GL_VERSION));
77      System.out.println("Shading Language: " + gl.glGetString(gl.GL_SHADING_LANGUAGE_VERSION));
78      System.out.println();
79      gl.glClearColor(0,0,0,1);           // Hintergrundfarbe
80      // Compile and Link Shaders
81      programId = MyShaders.initShaders(gl,vShader,fShader);
82      vArray = new VertexArray(gl, programId, maxVerts);
83      transform = new Transform(gl, programId);
84      FPSAnimator anim = new FPSAnimator(canvas, 60, true); // Animations-Thread, 60 Frames/sek
85      anim.start();
86    }
87
88    @Override
89    public void display(GLAutoDrawable drawable)
90    { GL3 gl = drawable.getGL().getGL3();
91      gl.glClear(GL3.GL_COLOR_BUFFER_BIT); // Frame-Buffer loeschen
92      vArray.setColor(0.7f,0.7f,0.7f);
93      transform.resetM(gl);               // Objekt-System zuruecksetzen
94      vArray.drawAxis(gl,5,5,5);          // Koordinatenachsen zeichnen
95      float s2 = 0.5f*s;
96      transform.rotateM(gl,phi,0,0,1);    // Objekt-System drehen
```

```
97        zeichneDreieck(gl,vArray,−s2,−h/3,s2,−h/3,0,2*h/3);
98        phi += dphi;
99      }
100
101     @Override
102     public void reshape(GLAutoDrawable drawable, int x, int y,
103                         int width, int height)
104     { GL3 gl = drawable.getGL().getGL3();
105       // −−−−−−− Set the viewport to the entire window
106       gl.glViewport(0, 0, width, height);
107     }
108
109     @Override
110     public void dispose(GLAutoDrawable drawable) { } // not needed
111
112     // ___________ main−Methode ___________________
113
114     public static void main(String[] args)
115     { new MyFirst2Da();
116     }
117
118     // ___________ Window−Events ___________________
119
120     public void windowClosing(WindowEvent e)
121     { System.out.println("closing window");
122         System.exit(0);
123     }
124     public void windowActivated(WindowEvent e) { }
125     public void windowClosed(WindowEvent e) { }
126     public void windowDeactivated(WindowEvent e) { }
127     public void windowDeiconified(WindowEvent e) { }
128     public void windowIconified(WindowEvent e) { }
129     public void windowOpened(WindowEvent e) { }
130
131     // ___________ Keyboard−Events ________________
132
133     public void keyPressed(KeyEvent e)
134     { int code = e.getKeyCode();
135       switch (code)
136       {
137         case KeyEvent.VK_UP: dphi += 0.05f; // Rotationsgeschwindigkeit
138                              break;
139         case KeyEvent.VK_DOWN: dphi −= 0.05f;
140                              break;
141       }
142       canvas.repaint();
143     }
144     public void keyReleased(KeyEvent e) { }
145     public void keyTyped(KeyEvent e) { }
146
147 }
```

Erklärungen

1. Zeilen 84-85

   Hier wird ein Animator-Objekt erzeugt und gestartet. Dieses ruft die Methode display in einem separaten Thread periodisch auf.

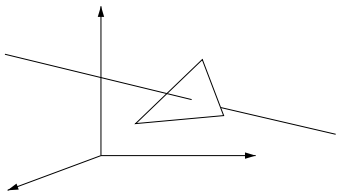
   Der Parameter '60' ist die Anzahl Frames pro Sekunde.

2. Zeile 96

   Die Drehachse der Drehung ist die z-Achse, welche senkrecht auf der xy-Ebene steht.

1.8.6 Vollständiges 3D-Beispiel

Das folgende Programm zeichnet ein Dreieck, welches von einer Strecke durchstossen wird:



Für 3D Darstellungen sind die folgenden Erweiterungen erforderlich:

- Aktivierung des Sichtbarkeits-Tests (OpenGL-Funktion glEnable)
- Definition des Kamera-Systems durch die View-Matrix V und die Projektionsmatrix P (Methoden lookAt2 und ortho der Klasse Transform)
- Definition einer Position der Lichtquelle mit der Methode setLightPosition der Klasse Transform.

Weitere Erklärungen folgen anschliessend an das Programm-Listing.

Das Programm erlaubt die interaktive Veränderung der Betrachtungswinkel Azimut und Elevation mit den Pfeil-Tasten (left, right bzw. up, down).

```
1  // ___________ JOGL 3D Beispiel-Programm (Lichtstrahl durch Dreieck) ___________
2  import java.awt.*;
3  import java.awt.event.*;
4  import com.jogamp.opengl.*;
5  import com.jogamp.opengl.awt.*;
6  import com.jogamp.opengl.util.*;
7
8  import ch.fhnw.util.opengl.MyShaders;
9  import ch.fhnw.util.opengl.VertexArray;
10 import ch.fhnw.util.opengl.Transform;
11 import ch.fhnw.util.math.Vec3;
12 import ch.fhnw.util.math.Mat4;
13
14 public class MyFirst3D
15       implements WindowListener, GLEventListener, KeyListener
16 {
17    // ___________ globale Daten ____________________
18    String windowTitle = "JOGL-Application";
19    int windowWidth = 800;
20    int windowHeight = 600;
21    String vShader = MyShaders.vShader1;  // Vertex-Shader
22    String fShader = MyShaders.fShader1;  // Fragment-Shader
23    Frame frame;
24    GLCanvas canvas;                      // OpenGL Window
25    int programId;                        // OpenGL-Id
26    VertexArray vArray;
27    Transform transform;
28    int maxVerts = 2048;                  // max. Anzahl Vertices im Vertex-Array
29    float elevation = 10;                 // Betrachtungswinkel (Kamera)
30    float azimut = 30;
31    float dist = 8;                       // Abstand Kamera vom Ursprung
32    float xleft=-3.0f, xright=3.0f;       // Viewing-Volume
33    float ybottom, ytop;
34    float znear=-10, zfar=100;
35
36    // ___________ Methoden ____________________
37
```

```
38    public MyFirst3D()                          // Konstruktor
39    { createFrame();
40    }
41
42    void createFrame()                          // Fenster erzeugen
43    { Frame f = new Frame(windowTitle);
44      f.setSize(windowWidth, windowHeight);
45      f.addWindowListener(this);
46      GLProfile glp = GLProfile.get(GLProfile.GL3);
47      GLCapabilities glCaps = new GLCapabilities(glp);
48      canvas = new GLCanvas(glCaps);
49      canvas.addGLEventListener(this);
50      f.add(canvas);
51      f.setVisible(true);
52      f.addKeyListener(this);
53      canvas.addKeyListener(this);
54    }
55
56    public void zeichneStrecke(GL3 gl, VertexArray va,
57                               Vec3 A, Vec3 B, Vec3 color)
58    { vArray.rewindBuffer(gl);
59      vArray.setColor(color.x,color.y,color.z); // Farbe setzen
60      vArray.putVertex(A.x,A.y,A.z);
61      vArray.putVertex(B.x,B.y,B.z);
62      vArray.copyBuffer(gl);
63      vArray.drawArrays(gl, GL3.GL_LINES);
64    }
65
66    public void zeichneDreieck(GL3 gl, VertexArray va,
67                               Vec3 A, Vec3 B, Vec3 C, Vec3 color)
68    { vArray.rewindBuffer(gl);                  // Vertex-Array leeren
69      vArray.setColor(color.x,color.y,color.z); // Farbe setzen
70      Vec3 u = B.subtract(A);
71      Vec3 v = C.subtract(A);
72      Vec3 n = u.cross(v);                      // Dreiecks-Normale
73      vArray.setNormal(n.x,n.y,n.z);
74      vArray.putVertex(A.x,A.y,A.z);
75      vArray.putVertex(B.x,B.y,B.z);
76      vArray.putVertex(C.x,C.y,C.z);
77      vArray.copyBuffer(gl);
78      vArray.drawArrays(gl, gl.GL_TRIANGLES);
79    }
80
81    // ___________ OpenGL-Events ___________________
82    @Override
83    public void init(GLAutoDrawable drawable) //  Initialisierung
84    { GL3 gl = drawable.getGL().getGL3();
85      System.out.println("OpenGl Version: " + gl.glGetString(gl.GL_VERSION));
86      System.out.println("Shading Language: " + gl.glGetString(gl.GL_SHADING_LANGUAGE_VERSION));
87      System.out.println();
88      gl.glClearColor(0,0,1,1);                 // Hintergrundfarbe
89      // Compile/Link Shaders
90      programId = MyShaders.initShaders(gl,vShader,fShader);
91      vArray = new VertexArray(gl, programId, maxVerts);
92      transform = new Transform(gl, programId);
93    }
94
95    @Override
96    public void display(GLAutoDrawable drawable)
97    { GL3 gl = drawable.getGL().getGL3();
98      // ---- Sichtbarkeitstest
99      gl.glEnable(GL3.GL_DEPTH_TEST);
100     // ---- Color- und Depth-Buffer loeschen
101     gl.glClear(GL3.GL_COLOR_BUFFER_BIT | GL3.GL_DEPTH_BUFFER_BIT);
102     // ---- Kamera-System und Lichtquelle festlegen
103     transform.lookAt2(gl,dist,azimut, elevation);
104     transform.setLightPosition(gl,-1.0f,2.7f,-1.0f);
105     // ---- Koordinatenachsen zeichnen
106     vArray.setColor(0.7f,0.7f,0.7f);
107     vArray.drawAxis(gl,5,5,5);
108     // ---- Figuren zeichnen
109     Vec3 A = new Vec3(0,0.3f,0.3f);       // Eckpunkte Dreieck
```

```
110       Vec3 B = new Vec3(2.5f,0.8f,1);
111       Vec3 C = new Vec3(0.5f,1.5f,−1);
112       transform.setShadingLevel(gl,1);        // Beleuchtung aktivieren
113       Vec3 color = new Vec3(0.8f,0.8f,0.8f);
114       zeichneDreieck(gl,vArray,A,B,C,color);
115       transform.setShadingLevel(gl,0);        // ohne Beleuchtung
116       color = new Vec3(1,0,0);
117       zeichneStrecke(gl,vArray,new Vec3(−1,3,5),
118                      new Vec3(2,−0.5f,−4),color);
119    }
120
121    @Override
122    public void reshape(GLAutoDrawable drawable, int x, int y,
123                        int width, int height)
124    { GL3 gl = drawable.getGL().getGL3();
125       // −−−−−− Set the viewport to the entire window
126       gl.glViewport(0, 0, width, height);
127       float aspect = (float)height/width;
128       ybottom=aspect*xleft;
129       ytop=aspect*xright;
130       // −−−−−− Projektionsmatrix (Orthogonalprojektion)
131       transform.ortho(gl,xleft,xright,ybottom,ytop,znear,zfar);
132    }
133
134    @Override
135    public void dispose(GLAutoDrawable drawable) { } // not needed
136
137    // ———————— main−Methode ————————————————
138    public static void main(String[] args)
139    { new MyFirst3D();
140    }
141
142    // ———————— Window−Events ————————————————
143    public void windowClosing(WindowEvent e)
144    { System.out.println("closing window");
145       System.exit(0);
146    }
147    public void windowActivated(WindowEvent e) { }
148    public void windowClosed(WindowEvent e) { }
149    public void windowDeactivated(WindowEvent e) { }
150    public void windowDeiconified(WindowEvent e) { }
151    public void windowIconified(WindowEvent e) { }
152    public void windowOpened(WindowEvent e) { }
153
154    // ———————— Keyboard−Events ———————————————
155    public void keyPressed(KeyEvent e)
156    { int code = e.getKeyCode();
157       switch (code)
158       { case KeyEvent.VK_LEFT: azimut−−;
159                                break;
160         case KeyEvent.VK_RIGHT: azimut++;
161                                break;
162         case KeyEvent.VK_UP: elevation++;
163                                break;
164         case KeyEvent.VK_DOWN: elevation−−;
165                                break;
166       }
167       canvas.repaint();
168    }
169    public void keyReleased(KeyEvent e) { }
170    public void keyTyped(KeyEvent e) { }
171 }
```

Erklärungen

1. Zeilen 21-22

   Für dieses Programm werden die Shaders vShader1 und fShader1 benötigt, welche die Standard-Funktionen ausführen (Vertex-Transformationen, Beleuchtung).

2. Zeilen 70-73

   Hier wird die Normale des Dreiecks berechnet, welche für die Beleuchtungsrechnung benötigt wird.

3. Zeile 99

   Hier wird der Sichtbarkeitstest (Depthtest) aktiviert, der für 3D Darstellungen essentiell ist. Er funktioniert mit dem *Depth-Buffer* (z-Buffer). Details, siehe später.

4. Zeile 101

   Für den Depthtest muss bei jedem Frame neben dem Color-Buffer auch der Depth-Buffer gelöscht werden.

5. Zeile 103

   Die Methode lookAt2 setzt die View-Matrix des Kamera-Systems nach dem Azimut-Elevation-Prinzip (Seite 6).

6. Zeile 104

   Die Methode setLightPosition definiert die Position der Lichtquelle für die Beleuchtungsrechnung. Die angegebenen Koordinaten werden mit den aktuellen Matrizen M und V in das Kamera-System transformiert.

7. Zeile 115

   Hier wird die Beleuchtung deaktiviert, sodass die Strecke (Lichtstrahl) mit konstanter Helligkeit erscheint.

8. Zeile 131

   Die Methode ortho setzt die Projektionsmatrix für die Orthogonalprojektion und das ViewingVolume mit den angegebenen Grenzen. Die Grenzen ytop und ybottom werden vorgängig so berechnet, dass das Verhältnis

   ```
   (ytop−ybottom) / (xright−xleft)
   ```

   gleich dem Verhältnis

   ```
   height / width
   ```

   des Windows ist (aspect). Dadurch werden Verzerrungen vermieden (Kreise erscheinen nicht als Ellipsen, Quadrate nicht als Rechtecken).

1.8.7 Vektor- und Matrix-Algebra

Java und OpenGL enthalten keine Datentypen und Operatoren für Vektoren und Matrizen. Wir verwenden dazu das Package

```
ch.fhnw.util.math
```

von Stefan Arizona und Simon Schubiger (ehemalige Dozenten Computergraphik, FHNW) Das Package enthält Klassen und Methoden für Vektor- und Matrix-Algebra, u.a. die Klassen

```
Vec3, Vec4, Mat3, Mat4
```

Die Objekte der Klassen sind *immutable* (wie Java Strings), was sich sehr bewährt. Die Komponenten der Vektoren und Matrizen haben den Datentyp `float` (nicht `double`), weil dies für OpenGL erforderlich ist.

Die Klasse Vec3 (Auszug)

- Daten

  Wegen der Immutability können die Komponenten `public` definiert werden:

  ```
  public final float x;
  public final float y;
  public final float z;
  ```

- Konstruktoren

  ```
  Vec3(float x, float y, float z)
  Vec3(double x, double y, double z)
  Vec3(float[] vec)
  ```

- Oeffentliche Methoden

  Da die Objekte immutable sind, muss für jede Aenderung ein neues Objekt als Resultat zurückgegeben werden.

  ```
  float length()          // Norm des Vektors
  Vec3 add(Vec3 v)        // result = this + v
  Vec3 subtract(Vec3 v)   // result = this − v
  Vec3 scale(float s)     // result = s * this
  Vec3 normalize()        // Normierung des Vektors
  float dot(Vec3 a)       // Skalarprodukt
  Vec3 cross(Vec3 a)      // Vektorprodukt
  float[] toArray()       // Konversion
  String toString()       // Konversion
  ```

Beispiel:

```
Vec3 v = new Vec3(0.4f,0.3f,0.5f);
v = v.scale(0.5f);          // Multiplikation mit 0.5
```

Die Klasse `Vec4` ist analog aufgebaut.

Die Klasse Mat4 (Auszug)

- Matrix-Algebra

```
Mat4 multiply(Mat4 a, Mat4 b)      // result = a * b
Mat4 postMultiply(Mat4 mat)        // result = this * mat
Mat4 preMultiply(Mat4 mat)         // result = mat * this
Vec4 transform(Vec4 vec)           // result = m * vec
Vec3 transform(Vec3 vec)           // result = m * vec
Mat4 transpose()                   // Transponierte
float determinant()                // Determinante
Mat4 inverse()                     // Inverse
```

- Transformationsmatrizen

  Die Anwendung der folgenden Methoden wird später nach Bedarf eingeführt. Die mathematischen Grundlagen der Transformationsmatrizen ist ein Hauptthema unseres Moduls.

  `static Mat4 translate(float tx, float ty, float tz)`
  Return the translation matrix

  `static Mat4 translate(Vec3 t)`
  Return the translation matrix

  `static Mat4 rotate(float angle, float x, float y, float z)`
  Return the rotation matrix (see figure below)

  `static Mat4 rotate(float angle, Vec3 axis)`
  Return the rotation matrix

  `static Mat4 scale(float sx, float sy, float sz)`
  Return the scale matrix

  `static Mat4 lookAt(Vec3 position, Vec3 target, Vec3 up)`
  Return the view matrix for camera system at `position`, looking to `target` with up-direction `up`.

  `static Mat4 lookAt2(double r, double azimut, double elevation)`
  Return the view matrix for the camera system at the distance `r` from the origin in the direction given by the angles `azimut` and `elevation`, looking to the origin. The angles are in degrees.

  `static Mat4 ortho(float left, float right, float bottom, float top, float near, float far)`
  Return orthographic projection matrix

  `static Mat4 perspective(float left, float right, float bottom, float top, float near, float far)`
  Return perspective projection matrix

- Konversionen

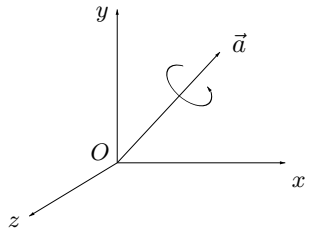
```
float[] toArray()                  // Umwandlung in OpenGL−Format
String toString()
```

Drehungen

Die Methode rotate berechnet die Matrix der Drehung mit den folgenden Parametern:

Drehwinkel: angle in Grad (nicht rad)
Drehachse: Vektor $\vec{a}$ mit Komponenten x,y,z.
Die Drehachse geht immer durch den Nullpunkt

Der Drehsinn der Drehung ist gemäss der rechten Hand Regel definiert: Zeigt der Daumen der rechten Hand in die Richtung des Vektors (x, y, z), so geben die Finger den Drehsinn für positive Drehwinkel an.



rechte Hand Regel

Anders ausgedrückt: Blickt man in Richtung der Drehachse, so dreht die Drehung im Uhrzeigersinn.

Drehungen in der xy-Ebene werden mit Drehungen in $\mathbb{R}^3$ um die z-Achse (d.h. $x = 0$, $y = 0$, $z = 1$) realisisert.

1.9 Die Shader-Language GLSL

Die Grundlagen der Shader-Language wurden schon bei den minimalen Shaders vShader0 und fShader0 eingeführt.

Der Vertex-Shader vShader1

Unser Standard-Vertex-Shader vShader1 erfüllt die folgenden Funktionen:

- ModelView-Transformation und Projektion der Vertex-Koordinaten mit den Matrizen M (Model-Matrix), V (View-Matrix) und P (Projektionsmatrix)
- ModelView-Transformation der Vertex-Normalen (ohne Projektion)
- Initialisierung der out-Variabeln für die Beleuchtungsrechnung im Fragment-Shader

Die Matrizen M, V und P sind uniform-Variabeln. Das sind Input-Variabeln, die im Hauptprogramm vor dem Zeichnen einer Figur (drawArray) dem Vertex-Shader übergeben werden (OpenGL-Funktion gl.glUniform).

Eine uniform-Variable hat (im Gegensatz zu den Vertex-Attributen) einen festen Wert für alle Vertices einer Figur.

```
1   #version 330
2   uniform mat4 V, M, P;                /* Transformations-Matrizen */
3   in vec4 vPosition, vColor, vNormal;  /* Vertex-Attribute */
4   out vec4 fPosition, fNormal, fColor; /* fuer Fragment-Shader */
5
6   /* ------- main-function --------- */
7   void main()
8   { vec4 vWorldCoord = M * vPosition;  /* Vertex-Transf. */
9     vec4 vEyeCoord = V * vWorldCoord;
10    fPosition = vEyeCoord;
11    gl_Position = P * vEyeCoord;       /* Projektion */
12    vec4 nWorldCoord = M * vNormal;    /* Transf. der Normalen */
13    fNormal = V * nWorldCoord;
14    fColor = vColor;
15  }
```

Der Fragment-Shader fShader1

Der Standard-Shader fShader1 berechnet die definitive Farbe des Fragmentes mit der richtigen Helligkeit aufgrund der Beleuchtung. Dazu wird ein Beleuchtungsmodell benötigt, welches im Kapitel 'Vektoren' eingeführt wird.

```
1   #version 330
2   in vec4 fPosition, fColor, fNormal;
3   uniform vec4 lightPosition; /* Position Lichtquelle (im Cam.System) */
4   uniform int shadingLevel; /* 0 ohne Beleucht, 1 mit Beleucht. */
5   uniform float ambient; /* ambientes Licht */
6   uniform float diffuse; /* diffuse Reflexion */
7   uniform float specular; /* spiegelnde Reflexion */
8   uniform float specExp; /* Shininess (Exponent) */
```

```
9     vec3 whiteColor = vec3(1,1,1);
10    out vec4 fragColor; /* Output-Farbe */
11    void main()
12    { if (shadingLevel < 1.0)
13      { fragColor = fColor; /* ohne Beleuchtung */
14        return;
15      }
16      /* ----- Beleuchtungsrechnung ---------- */
17      vec3 toEye = -normalize(fPosition.xyz);
18      vec3 normal = normalize(fNormal.xyz);
19      vec3 toLight = normalize(lightPosition.xyz - fPosition.xyz);
20      /* ----- diffuse Reflexion -------- */
21      float ndotl = dot(normal, toLight); /* Skalarprod */
22      float ndote = dot(normal, toEye);
23      if (ndotl < 0.0 || ndotl*ndote < 0.0)
24      { fragColor = vec4(ambient*fColor.x,ambient*fColor.y,ambient*fColor.z,1);
25        return;
26      }
27      float diffuseIntens = diffuse * ndotl; /* diffuse Reflexion */
28      vec3 computedColor = (ambient + diffuseIntens)*fColor.xyz;
29      vec3 halfBetween = normalize(toLight + toEye);
30      /* ------- spiegelnde Reflexion -------- */
31      float ndoth = dot(normal,halfBetween); /* Skalarprod */
32      if ( ndoth > 0.0 )
33      { float specularIntens = specular*pow( ndoth, specExp);
34        computedColor += specularIntens * whiteColor;
35      }
36      computedColor = min(computedColor, whiteColor);
37      fragColor = vec4(computedColor.r, computedColor.g, computedColor.b,1.0);
38    }
```

Phong- und Gouraud-Shading

Wenn die Beleuchtungsrechnung im Fragment-Shader erfolgt, spricht man von *Phong-Shading*. Die Berechnung könnte auch im Vertex-Shader erfolgen. Dies ist das *Gouraud-Shading*. Das Gouraud-Shading ist schneller, gibt aber i.a. etwas weniger gute Resultate als das Phong-Shading.

Grund:

Beim Gouraud-Shading wird die Beleuchtungsrechnung nur für jeden Vertex gemacht, und die berechneten Farben werden für die dazwischen liegenden Fragmente von OpenGL interpoliert.

Beim Phong-Shading werden die Vertex-Normalen für die Fragmente interpoliert, und die Beleuchtungsrechnung wird für jedes Fragment gemacht.

1.10 Anhang 1: Installation von JOGL

JOGL besteht aus *Java-Archiven* (.jar), welche die Java-Klassen und die systemspezifischen DLL (Dynamik Link Libraries) enthalten.

Installations-File:

jogamp−all−platforms.7z

Dieses kann von der folgenden Adresse bezogen werden:

http://jogamp.org/deployment/jogamp−current/archive/

Das Installationsfile muss mit einem geeigneten Programm (7-Zip von der Website www.7-zip.org) in ein Directory (c:\jogl) entpackt werden.

Benötigte Java-Archive (Win64):

(a)	gluegen−rt.jar	(Java-Klassen)
(b)	jogl−all.jar	(Java-Klassen)
(c)	gluegen−rt−natives−windows−amd64.jar	(DLL)
(d)	jogl−all−natives−windows−amd64.jar	(DLL)

Diese befinden sich im Unterverzeichnis 'jar' des Installationsverzeichnisses.

Für die Verwendung dieser Archive müssen die ersten beiden (a) und (b) in der Windows-Umgebungsvariablen classpath angegeben werden:

.;c:\jogl\jar\gluegen−rt.jar;c:\jogl\jar\jogl−all.jar

(Der erste Pfad '.' steht für das momentane Verzeichnis).

Eclipse

Bei Verwendung der Entwicklungsumgebung Eclipse müssen die beiden Files nicht in der Systemvariabeln classpath angegeben werden, sondern im classpath des Eclipse-Projektes. Dazu wird in Eclipse das Menu

Project/Properties/Java Build Path

geöffnet. Dann werden die beiden Files mit 'Add Extrenal JARs' zum classpath hinzugefügt.

Dokumentation (jogamp.org und andere Quellen):

OpenGL:	OpenGL Programming Guide, D. Shreiner et a.
JOGL:	Downloading and installing JOGL (HTML-Format)
	Java Dokumentationen der Klassen

1.11 Anhang 2: Homogene Koordinaten

Für eine einheitliche Darstellung von Translationen, Drehungen und Skalierungen mit Matrizen werden (wegen den Translationen) homogene Koordinaten und 4×4 Matrizen benötigt. Die Koordinaten der Raumpunkte werden zu diesem Zweck mit einer vierten Koordinate erweitert, welche gleich 1 ist (homogene Koordinaten).

Euklidische Koordinaten	Homogene Koordinaten
$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$	$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$

Beispiele von Transformations-Matrizen:

1. Translation

$$\begin{pmatrix} 1 & 0 & 0 & a_1 \\ 0 & 1 & 0 & a_2 \\ 0 & 0 & 1 & a_1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + a_1 \\ y + a_2 \\ z + a_3 \\ 1 \end{pmatrix}$$

2. Drehung um z-Achse

$$\begin{pmatrix} \cos(\varphi) & -\sin(\varphi) & 0 & 0 \\ \sin(\varphi) & \cos(\varphi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\varphi)\, x - \sin(\varphi)\, y \\ \sin(\varphi)\, x + \cos(\varphi)\, y \\ z \\ 1 \end{pmatrix}$$

Kapitel 2

Vektoren

> The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.
>
> "Begin at the beginning," the king said, gravely, "and go on till you come to the end: then stop."
>
> – Lewis Carroll, Alice in Wonderland

2.1 Der Vektorraum $\mathbf{R}^n$

Ein *Vektor* in $\mathbb{R}^n$ ist ein n-Tupel (Array) von reellen Zahlen:

$$\vec{x} = (x_1, \ldots, x_n) \qquad x_i \in \mathbb{R}$$

Die Zahlen x_i heissen *Komponenten* des Vektors. Je nach Situation werden die Vektoren als Zeilen- oder Spaltenvektoren geschrieben:

$$\vec{x} = (2.5,\ -4.2,\ 3.1) \qquad \vec{x} = \begin{pmatrix} 2.5 \\ -4.2 \\ 3.1 \end{pmatrix}$$

Im Zusammenhang mit der Matrizenmultiplikation sind die beiden Darstellungen nicht mehr äquivalent.

Vektor-Algebra

Die algebraischen Grundoperationen sind komponentenweise definiert:

Addition/Subtraktion

$$\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} \pm \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_1 \pm b_1 \\ \vdots \\ a_n \pm b_n \end{pmatrix}$$

Multiplikation mit $t \in \mathbb{R}$

$$t \cdot \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} t \cdot a_1 \\ \vdots \\ t \cdot a_n \end{pmatrix}$$

Reelle Zahlen t nennt man zur Unterscheidung von Vektoren *Skalare*. Die bekannten Gesetze der reellen Zahlen (Klammerregeln) übertragen sich auf die Vektor-Operationen. Dank diesen Gesetzen kann man mit Vektoren wie mit Zahlen rechnen:

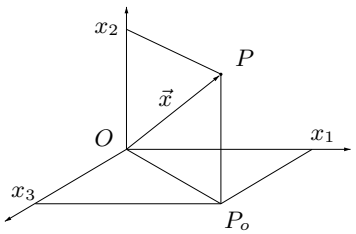
$$3 \cdot (\vec{a} - 4\,\vec{b}) + 5 \cdot (\vec{a} + \vec{c}) = 3\,\vec{a} - 12\,\vec{b} + 5\,\vec{a} + 5\,\vec{c} = 8\,\vec{a} - 12\,\vec{b} + 5\,\vec{c}$$

Allgemeine Vektorräume

Allgemein ist ein *Vektorrraum* über den rellen Zahlen eine Menge von beliebigen Elementen (Vektoren genannt), für welche eine Addition '+' für je zwei Vektoren und eine Multiplikation '$\cdot$' eines Vektors mit einer reellen Zahl gegeben sind. Dabei müssen diese Operationen die gewohnten Gesetze der reellen Zahlen erfüllen. *Vektorraum*

Ortsvektoren und freie Vektoren

Für die Dimensionen $n = 2$ und 3 können Vektoren geometrisch interpretiert werden. Nach der Einführung eines Koordinatensystems in der Ebene oder im Raum kann jedem Punkt P sein *Ortsvektor* zugeordnet werden. Dies ist der Vektor $\vec{x}$, bestehend aus den Koordinaten x_i des Punktes: *Ortsvektor*



Ortsvektor von P:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Damit können Punkte mit ihren Ortsvektoren identifiziert werden.

Definition der *Norm* eines Vektors $\vec{x}$:

$$|\vec{x}\,| = \sqrt{x_1^2 + x_2^2 + x_3^2}$$

Aus dem Satz von Pythagoras folgt, dass dies die *Länge* des Ortsvektors, d.h. der Abstand des zugehörigen Punktes vom Nullpunkt ist:

Die Strecke $d = \overline{OP_o}$ ist nach Pythagoras

$$d = \sqrt{x_1^2 + x_3^2}$$

Im rechtwinkligen Dreieck OP_oP mit rechtem Winkel bei P_o gilt damit wieder nach Pythagoras:

$$|\vec{x}\,|^2 = d^2 + x_2^2 = x_1^2 + x_3^2 + x_2^2$$

Analog ist die Norm eines Vektors $\vec{x} \in \mathbb{R}^n$ definiert.

Ein Vektor mit Norm 1 heisst *Einheitsvektor*. *Einheitsvektor*

Normierung eines Vektors

Sei $\vec{x} \in \mathbb{R}^n$, $\vec{x} \neq 0$. Die Normierung von $\vec{x}$ ist der Vektor

$$\hat{\vec{x}} = \frac{1}{|\vec{x}\,|} \cdot \vec{x} \tag{2.1}$$

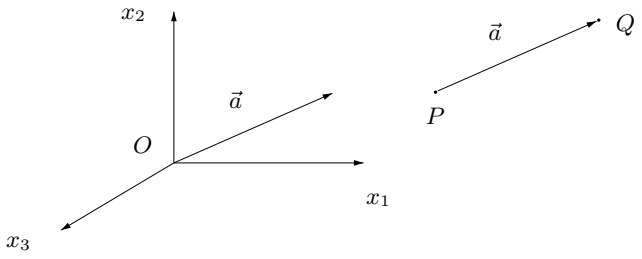
Dies ist ein Einheitsvektor mit derselben Richtung wie $\vec{x}$.

Freie Vektoren

Für viele Anwendungen eignen sich frei verschiebbare Vektoren ohne festen Anfangspunkt, sogenannte *freie Vektoren.*

Beispiele: Geschwindigkeitsvektoren, Richtungsvektor einer Geraden

Unter dem *freien Vektor* eines Ortsvektors $\vec{a}$ versteht man die Gesamtheit (Menge) aller Pfeile, die durch eine Parallelverschiebung aus $\vec{a}$ hervorgehen. Jedes Element dieser Menge heisst *Repräsentant* des freien Vektors.

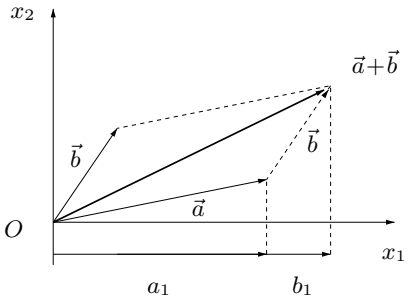


Praktisch heisst dies, dass ein freier Vektor durch eine *Länge* und eine *Richtung*, d.h. durch eine gerichtete Strecke (Pfeil), gegeben durch ein Punktepaar (P, Q), gegeben ist.

Bezeichnung: $\vec{a} = \overrightarrow{PQ}$

Da zu jedem freien Vektor ein eindeutig bestimmter Ortsvektor (Anfangspunkt O) gehört, können alle Definitionen und Operationen von Ortsvektoren auf freie Vektoren übertragen werden.

Geometrische Interpretation der Addition



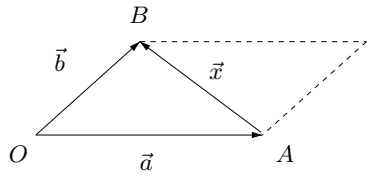
Die Summe ist die Diagonale des aufgespannten Parallelogrammes (Parallelogramm-Regel).

oder

Die beiden Vektoren werden aneinander gehängt.

Dies folgt aus der Figur unter Beachtung, dass die Komponenten gemäss der Addition von reellen Zahlen durch Verschiebung addiert werden.

Subtraktion



In der Figur gilt gemäss Definition der Addition $\vec{b} = \vec{a} + \vec{x}$, d.h. $\vec{x} = \vec{b} - \vec{a}$

$$\boxed{\overrightarrow{AB} = \vec{b} - \vec{a}} \qquad (2.2)$$

Multiplikation mit Skalar

Die *Multiplikation* eines Vektors mit einer reellen Zahl t bedeutet einfach eine *Streckung* des Vektors mit dem Faktor t, da jede Komponente mit diesem Faktor gestreckt wird.

Parallele Vektoren

Ein Vektoren $\vec{a}$ ist parallel oder antiparallel zu einem Vektor $\vec{b}$, wenn er ein Vielfaches von $\vec{b}$ ist:

$$\vec{a} = t \cdot \vec{b} \qquad t \in \mathbb{R}$$

Die Vektoren heissen in diesem Fall *kollinear*.

Notations-Konvention

Wir bezeichnen Punkte mit grossen Buchstaben A, B, P, Q, X, ... und ihre Ortsvektoren mit den zugehörigen Kleinbuchstaben $\vec{a}$, $\vec{b}$, $\vec{p}$, $\vec{q}$, $\vec{x}$, ...

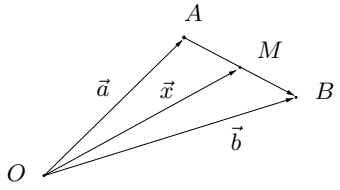
Beispiele:

1. Mittelpunkt M einer Strecke AB

   $A = (2, -1, 5)$, $B = (5, 2, -1)$

   Gemäss Figur gilt für den Ortsvektor $\vec{x}$ von M:

$$\begin{aligned} \vec{x} &= \vec{a} + 0.5 \cdot \overrightarrow{AB} \\ &= \vec{a} + 0.5 \cdot (\vec{b} - \vec{a}) \\ &= \vec{a} + 0.5 \cdot \vec{b} - 0.5 \cdot \vec{a} \\ &= 0.5 \cdot \vec{a} + 0.5 \cdot \vec{b} \\ &= 0.5 \cdot (\vec{a} + \vec{b}) \end{aligned}$$



   Resultat:

$$\vec{x} = \frac{\vec{a} + \vec{b}}{2} \qquad M(3.5, 0.5, 2)$$

   Der Ortsvektor $\vec{x}$ ist das *arithmetische Mittel* der Ortsvektoren $\vec{a}$ und $\vec{b}$ der Punkte A und B.

2. Abstand zweier Punkte A und B

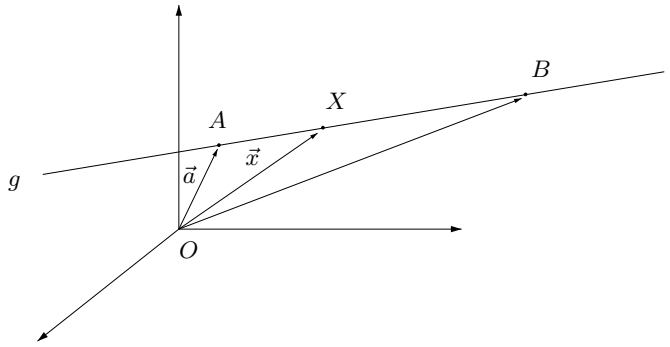
$$\boxed{d(A,B) \;=\; |\,\overrightarrow{AB}|} \tag{2.3}$$

Mit den Punkten des vorangehenden Beispiels:

$$d(A,B) = |\,\overrightarrow{AB}| = |\,\vec{b} - \vec{a}\,| = \sqrt{3^2 + 3^2 + (-6)^2} = \sqrt{54}$$

3. Geraden

Ersetzt man im obigen Beispiel 1 den Faktor 0.5 durch einen beliebigen reellen Parameter $t \in \mathbb{R}$, so erhält man einen beliebigen Punkt auf der Geraden g durch die Punkte A und B :



$$\boxed{\vec{x} = \vec{a} + t \cdot \overrightarrow{AB}} \qquad t \in \mathbb{R} \quad \text{Parameter} \tag{2.4}$$

Dies ist die *Parametergleichung* der Geraden.

Koordinaten-Form der Parametergleichung:

$A(2,5,10) \quad B(6,1,4)$

$$\vec{x} = \begin{pmatrix} 2 \\ 5 \\ 10 \end{pmatrix} + t \cdot \begin{pmatrix} 4 \\ -4 \\ -6 \end{pmatrix} \qquad \text{d.h.} \qquad \begin{array}{rcrcr} x_1 & = & 2 & + & 4t \\ x_2 & = & 5 & - & 4t \\ x_3 & = & 10 & - & 6t \end{array}$$

Spurpunkte

Die Durchstosspunkte mit den Koordinatenebenen nennt man *Spurpunkte.*

Spurpunkt S mit der x_1x_2-Ebene (Aufrissebene):

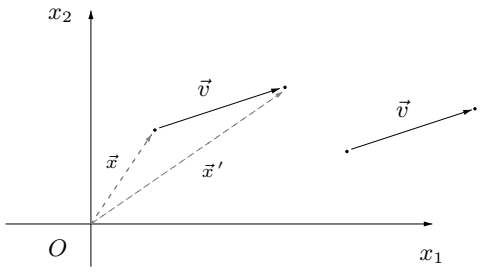
Bedingung für S: $x_3 = 0$, also folgt aus den obigen Gleichungen:

$$10 - 6t = 0, \qquad t = \tfrac{5}{3} \qquad \text{und damit} \quad S(\tfrac{26}{3},\ -\tfrac{5}{3},\ 0)$$

4. Translation

   Eine *Translation* oder *Parallelverschiebung* in $\mathbb{R}^n$ ist eine Punktabbildung in $\mathbb{R}^n$, gegeben durch einen Vektor $\vec{v} \in \mathbb{R}^n$. Jedem Punkt $\vec{x}$ wird ein Bildpunkt $\vec{x}\,' = \vec{x} + \vec{v}$ zugeordnet.

   $$T : \mathbb{R}^n \to \mathbb{R}^n : \ \vec{x} \ \mapsto \ \vec{x}\,' = \vec{x} + \vec{v}$$

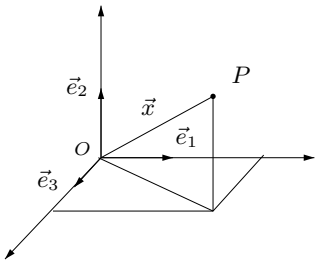


   Der Vektor $\vec{v}$ heisst Verschiebungsvektor der Translation.

Die Standardbasis von $\boldsymbol{R}^n$

Standardbasis von $\mathbb{R}^3$:

$$\vec{e}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \vec{e}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \vec{e}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$



Für einen beliebigen Vektor $\vec{x} \in \mathbb{R}^3$ folgt:

$$\vec{x} \ = \ \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \ = \ x_1\vec{e}_1 \ + \ x_2\vec{e}_2 \ + \ x_3\vec{e}_3$$

z.B.

$$\vec{x} \ = \ \begin{pmatrix} 2 \\ -5 \\ 3 \end{pmatrix} \ = \ 2\vec{e}_1 \ - \ 5\vec{e}_2 \ + \ 3\vec{e}_3$$

Eine solche Darstellung eines Vektors als Summe von Vielfachen von anderen nennt man eine *Linearkombination*. Die dabei auftretenden Faktoren nennt man die Koeffizienten der Linearkombination.

Linearkombination

Analog ist die Standardbasis von $\mathbb{R}^n$ definiert.

2.2 Das Skalarprodukt (Dotproduct)

Die Definition der Summe zweier Vektoren mit der komponentenweisen Addition, lässt sich problemlos auf eine Multiplikation von Vektoren übertragen. Das so definierte Produkt hat jedoch praktisch keine Anwendungen.

Für praktische Anwendungen eignet sich das *Skalarprodukt*. Bei diesem werden die entsprechenden Komponenten multipliziert, und die Produkte werden summiert, sodass das Resultat eine Zahl (Skalar), kein Vektor ist:

2.2.1 Definition des Skalarproduktes

Das *Skalarprodukt* von zwei Vektoren $\vec{a}, \vec{b} \in \mathbb{R}^n$ ist die reelle Zahl:

$$\boxed{\vec{a} \cdot \vec{b} \;=\; \sum_{i=1}^{n} a_i b_i \;=\; a_1 b_1 + a_2 b_2 + \cdots + a_n b_n} \qquad \text{(Skalar)}$$

Wie wir sehen werden, eignet sich das Skalarprodukt für die Berechnung von *Längen* und *Winkeln*.

Matlab-Funktion: $\mathrm{dot}(a, b)$

Beispiel:

$$\begin{pmatrix} 3 \\ -2 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 1 \\ 5 \end{pmatrix} \;=\; 3 \cdot 2 - 2 \cdot 1 + 3 \cdot 5 = 19$$

Gesetze

(1)	$\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$	Kommutativ-Gesetz
(2)	$(\vec{a} + \vec{b}) \cdot \vec{c} = \vec{a} \cdot \vec{c} + \vec{b} \cdot \vec{c}$	Additivität im ersten Argument
(3)	$(t\vec{a}) \cdot \vec{b} = t\,(\vec{a} \cdot \vec{b}), \quad t \in \mathbb{R}$	Homogenität im ersten Argument

Das zweite und dritte Gesetz bedeuten, dass das Skalarprodukt linear im ersten Argument ist. Analog ist es im zweiten Argument linear. Man sagt, es sei eine *bilineare* Funktion.

bilinear

Norm und Skalarprodukt

Das Quadrat der Norm (Länge) eines Vektors $\vec{a} \in \mathbb{R}^n$ ist das Skalarprodukt des Vektors mit sich selber:

$$\boxed{|\vec{a}\,|^{\,2} \;=\; \vec{a} \cdot \vec{a}} \qquad (2.5)$$

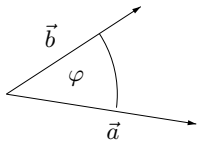
Herleitung:

$$\vec{a} \cdot \vec{a} = a_1^2 + a_2^2 + \cdots + a_n^2 = |\vec{a}|^{\,2}$$

2.2.2 Skalarprodukt und Winkel

In $\mathbb{R}^2$ und $\mathbb{R}^3$ kann das Skalarprodukt von $\vec{a}$ und $\vec{b}$ mit dem Cosinus des Zwischenwinkels berechnet werden:

$$\boxed{\vec{a} \cdot \vec{b} = |\vec{a}\,| \cdot |\vec{b}\,| \cdot \cos\varphi} \tag{2.6}$$

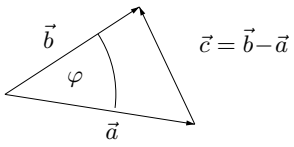


Dabei ist $\varphi = \angle(\vec{a}, \vec{b})$ der Winkel zwischen den Vektoren, $0 \leq \varphi \leq 180^o$.

Beweis:

Nach dem Cosinus-Satz für Dreiecke gilt

$$|\vec{c}\,|^2 = |\vec{a}\,|^2 + |\vec{b}\,|^2 - 2\,|\vec{a}\,|\,|\vec{b}\,|\,\cos\varphi$$



andererseits (mit $\vec{c} = \vec{b} - \vec{a}$) :

$$|\vec{c}\,|^2 = \vec{c} \cdot \vec{c} = (\vec{b} - \vec{a}) \cdot (\vec{b} - \vec{a}) = |\vec{b}\,|^2 - 2\,\vec{a} \cdot \vec{b} + |\vec{a}\,|^2$$

Durch Gleichsetzen folgt die Behauptung. □

Folgerungen

1. Orthogonalitätsbedingung

   Seien $\vec{a} \neq 0$ und $\vec{b} \neq 0$. Dann gilt

   $$\boxed{\vec{a} \cdot \vec{b} = 0 \quad \Longleftrightarrow \quad \vec{a} \perp \vec{b}} \tag{2.7}$$

   Dies folgt sofort aus 2.6 wegen $\cos 90^o = 0$ und weil 90^o die einzige Nullstelle der Cosinus-Funktion im Intervall $0 \leq \varphi \leq 180^o$ ist.

2. Winkelberechnung

   Umgekehrt kann mit der Gleichung 2.6 der Zwischenwinkel φ zweier Vektoren $\vec{a} \neq 0$ und $\vec{b} \neq 0$ berechnet werden:

   $$\cos\varphi = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}\,| \cdot |\vec{b}\,|} \tag{2.8}$$

   $$\boxed{\varphi = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{|\vec{a}\,| \cdot |\vec{b}\,|}\right)} \qquad 0 \leq \varphi \leq \pi \tag{2.9}$$

   Für Einheitsvektoren entfällt der Nenner:

   $$\varphi \;=\; \arccos(\vec{a} \cdot \vec{b}) \qquad \text{falls} \quad |\vec{a}\,| = |\vec{b}\,| = 1 \tag{2.10}$$

   Für Vektoren $\vec{a}, \vec{b} \in \mathbb{R}^n$ mit $n > 3$, wird der Zwischenwinkel φ durch diese Gleichung definiert.

Beispiel:

$\vec{a} = (\,3, -2, 1\,)\,,\ \vec{b} = (\,2, 5, 1\,)$

$\vec{a} \cdot \vec{b} = 6 - 10 + 1 = -3, \quad |\vec{a}\,| = \sqrt{14}, \quad |\vec{b}\,| = \sqrt{30}$

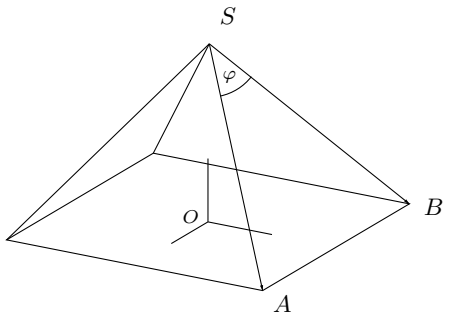
$$\varphi \ = \ \arccos \frac{-3}{\sqrt{14} \cdot \sqrt{30}} \ = \ 98.42^o$$

Merke:

Wenn das Skalarprodukt $\vec{a} \cdot \vec{b}$ negativ ist, ist der Zwischenwinkel $\varphi > 90^o$.

Beispiel

Die Cheops-Pyramide hat eine Höhe $h = 146\,m$ und eine quadratische Grundfläche mit Kantenlänge $a = 230\,m$.



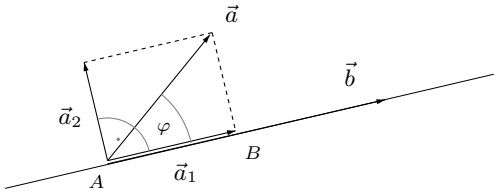
Gesucht ist der Winkel φ. (Resultat: $\varphi = 63.5^o$)
Hinweis:
Berechnen Sie die Koordinaten der Punkte A, B und S und damit die Vektoren $\overrightarrow{SA}$ und $\overrightarrow{SB}$.

2.2.3 Orthogonalzerlegung eines Vektors

Gegeben seien zwei Vektoren $\vec{a}, \vec{b} \in \mathbb{R}^n$ mit $\vec{b} \neq 0$.

Gesucht ist eine Zerlegung von $\vec{a}$ in Komponenten $\vec{a}_1$ in Richtung $\vec{b}$ und $\vec{a}_2$ senkrecht zu $\vec{b}$:

$\vec{a} = \vec{a}_1 + \vec{a}_2$



Beispiele: Zerlegung von Kräften oder Geschwindigkeiten.

Ansatz: $\vec{a}_1 = t \cdot \vec{b}, \qquad \vec{a}_2 = \vec{a} - t \cdot \vec{b}$

Bei diesem Ansatz ist gewährleistet, dass $\vec{a}_1$ parallel zu $\vec{b}$ ist, und dass $\vec{a}$ die Summe von $\vec{a}_1$ und $\vec{a}_2$ ist. Also muss nur noch der Parameter t so bestimmt werden, dass $\vec{a}_2$ senkrecht auf $\vec{b}$ steht, d.h.

$$(\vec{a} - t\,\vec{b}) \cdot b = 0$$
$$\vec{a} \cdot \vec{b} - t\,\vec{b} \cdot \vec{b} = 0$$

Daraus folgt $t = \vec{a} \cdot \vec{b}\ /\ \vec{b} \cdot \vec{b}$

Resultat:

$$\boxed{\vec{a}_1 = \frac{\vec{a} \cdot \vec{b}}{\vec{b} \cdot \vec{b}} \cdot \vec{b} \qquad \text{und} \qquad \vec{a}_2 = \vec{a} - \vec{a}_1} \tag{2.11}$$

Der Vektor $\vec{a}_1$ heisst *Orthogonalprojektion* von $\vec{a}$ auf $\vec{b}$ und der Koeffizient t *Orthogonalkomponente* von $\vec{a}$ bez. $\vec{b}$. *Orthogonalprojektion*

Beispiel

$$\vec{a} = \begin{pmatrix} 5 \\ 1 \\ -3 \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}, \quad \vec{a}_1 = \frac{10}{14} \cdot \vec{b}, \quad \vec{a}_2 = \frac{2}{14} \cdot \begin{pmatrix} 25 \\ -8 \\ -26 \end{pmatrix}$$

Falls $\vec{b}$ ein Einheitsvektor ist, entfällt der Nenner wegen $\vec{b} \cdot \vec{b} = |\vec{b}|^2 = 1$:

$$\boxed{\vec{a}_1 \;=\; (\vec{a} \cdot \vec{b}) \cdot \vec{b} \qquad \vec{a}_2 = \vec{a} - \vec{a}_1} \qquad \text{für} \quad |\vec{b}\,| = 1 \tag{2.12}$$

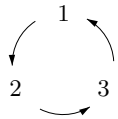
Orthogonalkomponente von $\vec{a}$ bez. einem Einheitsvektor $\vec{b}$: $\vec{a} \cdot \vec{b}$

2.3 Das Vektorprodukt (Crossproduct)

Das Vektorprodukt (Kreuzprodukt, Crossproduct) ist eine Spezialität von $\mathbb{R}^3$. Seine Hauptanwendung ist die Berechnung von *Normalenvektoren.*

Unter dem *Vektorprodukt* von zwei Vektoren $\vec{a} \in \mathbb{R}^3$ und $\vec{b} \in \mathbb{R}^3$ versteht man den Vektor $\vec{c} \in \mathbb{R}^3$, definiert durch

$$\begin{aligned} c_1 &= a_2b_3 - a_3b_2 \\ c_2 &= a_3b_1 - a_1b_3 \\ c_3 &= a_1b_2 - a_2b_1 \end{aligned}$$



Als Merkregel beachte man, dass in jeder Gleichung die ersten drei Indizes zyklisch laufen, gemäss dem dargestellten Schema:

$$1 \to 2 \to 3, \quad 2 \to 3 \to 1, \quad 3 \to 1 \to 2$$

Bezeichnung des Vektorproduktes:

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

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix} \times \begin{pmatrix} 3 \\ -2 \\ 4 \end{pmatrix} = \begin{pmatrix} -8+6 \\ 9-4 \\ -2+6 \end{pmatrix} = \begin{pmatrix} -2 \\ 5 \\ 4 \end{pmatrix}$$

Matlab-Funktion: c = cross(a,b)

Folgerung:

Das Vektorprodukt $\vec{c} = \vec{a} \times \vec{b}$ steht senkrecht auf $\vec{a}$ und $\vec{b}$.

Dies folgt sofort aus der Definition, z.B. für den Faktor $\vec{a}$:

$$\vec{a} \cdot \vec{c} = a_1(a_2b_3 - a_3b_2) + a_2(a_3b_1 - a_1b_3) + a_3(a_1b_2 - a_2b_1) = 0$$

Vektorprodukte der Basisvektoren $\vec{e}_i$:

$$\vec{e}_1 := \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \qquad \vec{e}_2 := \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \qquad \vec{e}_3 := \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$\boxed{\vec{e}_1 \times \vec{e}_2 = \vec{e}_3 \qquad \vec{e}_2 \times \vec{e}_3 = \vec{e}_1 \qquad \vec{e}_3 \times \vec{e}_1 = \vec{e}_2}$$

Gesetze des Vektorproduktes

(1) $\vec{a} \times \vec{b} = -\vec{b} \times \vec{a}$ Antikommutativ-Gesetz

(2) $\vec{a} \times \vec{a} = 0$

(3) $\vec{a} \times (\vec{b} + \vec{c}) \;=\; \vec{a} \times \vec{b} + \vec{a} \times \vec{c}$ Distributiv-Gesetze

(4) $(\vec{a} + \vec{b}) \times \vec{c} \;=\; \vec{a} \times \vec{c} + \vec{b} \times \vec{c}$

(5) $t\,(\vec{a} \times \vec{b}) \;=\; (t\vec{a}) \times \vec{b} \;=\; \vec{a} \times (t\vec{b})$ $\quad t \in \mathbb{R}$

Beispiel:

$$\vec{a} \times (\vec{b} - \vec{a}) - \vec{b} \times (\vec{b} + \vec{a}) = \vec{a} \times \vec{b} - \vec{a} \times \vec{a} - \vec{b} \times \vec{b} - \vec{b} \times \vec{a} = 2\vec{a} \times \vec{b}$$

Achtung: Das Assoziativ-Gesetz gilt *nicht*, i.a. ist

$$\vec{a} \times (\vec{b} \times \vec{c}) \;\neq\; (\vec{a} \times \vec{b}) \times \vec{c}$$

Mehrfache Vektorprodukte:

$$\vec{a} \times (\vec{b} \times \vec{c}) \;=\; (\vec{a} \cdot \vec{c})\,\vec{b} - (\vec{a} \cdot \vec{b})\,\vec{c} \tag{2.13}$$

Das doppelte Vektorprodukt ist eine Linearkombination von $\vec{b}$ und $\vec{c}$ mit den Koeffizienten $(\vec{a} \cdot \vec{c})$ und $-(\vec{a} \cdot \vec{b})$.

Beispiel:

$$\vec{a} = \begin{pmatrix} 3 \\ 1 \\ 8 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} -3 \\ 2 \\ 4 \end{pmatrix} \quad \vec{c} = \begin{pmatrix} 7 \\ -6 \\ -4 \end{pmatrix}$$

$$\vec{a} \times (\vec{b} \times \vec{c}) = -17 \cdot \vec{b} - 25 \cdot \vec{c}$$

Herleitung von 2.13:

Mit $\vec{n} = \vec{b} \times \vec{c}$ ist die erste Komponente der linken Seite von 2.13 :

$$\begin{aligned} a_2 n_3 - a_3 n_2 &= a_2 \cdot (b_1 c_2 - b_2 c_1) - a_3 \cdot (b_3 c_1 - b_1 c_3) = \\ &= (a_2 c_2 + a_3 c_3) \cdot b_1 - (a_2 b_2 + a_3 b_3) \cdot c_1 \end{aligned}$$

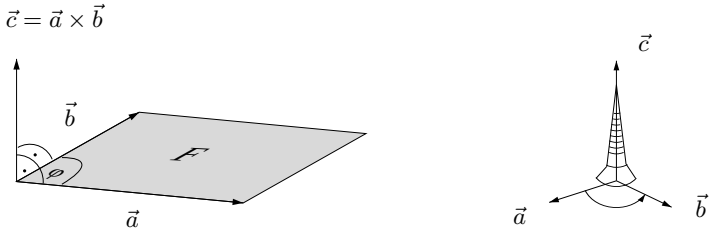
Erste Komponente der rechten Seite von 2.13:

$$(a_1 c_1 + a_2 c_2 + a_3 c_3) \cdot b_1 - (a_1 b_1 + a_2 b_2 + a_3 b_3) \cdot c_1$$

Weil sich die beiden Terme $a_1 b_1 c_1$ aufheben, ist dies identisch mit der linken Seite. Die Herleitung für die anderen Komponenten geht analog.

Geometrische Interpretation des Vektorproduktes

Seien $\vec{a} \in \mathbb{R}^3$, $\vec{b} \in \mathbb{R}^3$. Der Vektor $\vec{c} = \vec{a} \times \vec{b}$ weist die folgenden geometrischen Eigenschaften auf, durch die er eindeutig festgelegt ist:



Das Vektorprodukt $\vec{a} \times \vec{b}$ ist der Vektor $\vec{c}$, Bestimmt durch die folgenden Eigenschaften:

(1)	$\vec{c}$ steht *senkrecht* auf $\vec{a}$ und $\vec{b}$
(2)	$\lvert\vec{c}\rvert = \lvert\vec{a}\rvert \cdot \lvert\vec{b}\rvert \cdot \sin(\varphi) \;=\; F_{Parallelogramm}$
(3)	$\vec{a}$, $\vec{b}$ und $\vec{c}$ bilden in dieser Reihenfolge ein *Rechtssystem*

Die dritte Bedingung kann man sich mit der *Schraubenregel* merken (siehe Figur): *Schraubenregel*

Sei R_{ab} die Drehung, welche den Vektor $\vec{a}$ auf dem kürzesten Weg in $\vec{b}$ dreht. Dann zeigt $\vec{c}$ in die Richtung, in welche sich eine Rechtsschraube bei der Drehung R_{ab} bewegt.

Dies kann äquivalent auch mit der *Rechten-Hand-Regel* formuliert werden: *Rechte-Hand-Regel*

Man umfasse den Vektor $\vec{c}$ mit der rechten Hand, sodass der Daumen gegen die Pfeilpitze von $\vec{c}$ gerichtet ist. Dann zeigen die Finger den Drehsinn der Drehung, die den Vektor $\vec{a}$ auf dem *kürzesten* Weg in $\vec{b}$ dreht.

Beweis der geometrischen Interpretation:

(1) Siehe oben, Seite 47.

(2) Wir berechnen das Quadrat der linken und der rechten Seite der zu beweisenden Gleichung:

$$\lvert\vec{c}\rvert^2 = (a_2b_3 - a_3b_2)^2 + (a_3b_1 - a_1b_3)^2 + (a_1b_2 - a_2b_1)^2$$

$$\lvert\vec{a}\rvert^2 \cdot \lvert\vec{b}\rvert^2 \cdot \sin^2\varphi = \lvert\vec{a}\rvert^2 \cdot \lvert\vec{b}\rvert^2 \,(1 - \cos^2\varphi) = \lvert\vec{a}\rvert^2 \cdot \lvert\vec{b}\rvert^2 - (\vec{a} \cdot \vec{b})^2 =$$

$$= (a_1^2 + a_2^2 + a_3^2)\,(b_1^2 + b_2^2 + b_3^2) - (a_1b_1 + a_2b_2 + a_3b_3)^2$$

Durch Auflösen der Klammern folgt die Behauptung.

(3) Für die Basisvektoren ist dies erfüllt, z.B. ist $\vec{e}_1 \times \vec{e}_2 = \vec{e}_3$. Der allgemeine Beweis ist relativ umfangreich und für uns weniger interessant.

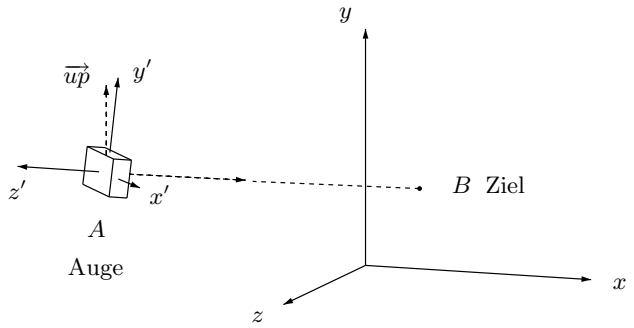
Berechnung von Normalenvektoren

Gegeben sei ein Dreieck ABC. Gesucht ist ein Normalenvektor $\vec{n}$.

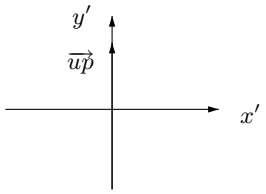
$$\vec{n} = \overrightarrow{AB} \times \overrightarrow{AC}$$

Ausrichtung eines Kamera-Systems (LookAt)

Ein geeignetes Verfahren zur Positionierung eines Kamera-Systems ist das *LookAt*–Prinzip. Dabei werden zwei Punkte A (Auge), B (Ziel) und ein Vektor $\overrightarrow{up}$ vorgegeben. LookAt



Durch die Punkte A und B ist die z'-Achse des Kamera-Systems festgelegt. Es kann folglich nur noch um diese Achse gedreht werden. Diese Drehung wird so gewählt, dass der up-Vektor im Kamera-System in y'-Richtung (Vertikalrichtung) erscheint:



Der *up*-Vektor legt also die vertikale Richtung im Kamera-System fest. Mathematisch bedeutet dies, dass der up-Vektor in der $y'z'$-Ebene liegt.

Die Richtungen $\vec{e}_1{}'$, $\vec{e}_2{}'$ und $\vec{e}_3{}'$ der Koordinaten-Achsen des Systems erhält man folgendermassen:

z'-Achse:	$\vec{e}_3{}' = -\overrightarrow{AB}$	negative Blickrichtung
x'-Achse:	$\vec{e}_1{}' = \overrightarrow{up} \times \vec{e}_3{}'$	senkrecht zu $\overrightarrow{up}$ und z'-Achse
y'-Achse:	$\vec{e}_2{}' = \vec{e}_3{}' \times \vec{e}_1{}'$	Ergänzung zu Rechtssystem

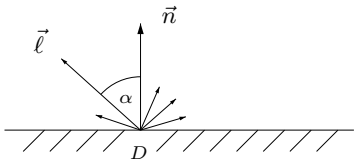
2.4 Das Beleuchtungsmodell

Beim Standard-Beleuchtungsmodell der Computergraphik werden die Helligkeiten der Punkte auf den Oberflächen von Körpern (wie in Wirklichkeit) aufgrund des reflektierten Lichtes bestimmt.

Es sind zwei Reflexionsarten zu unterscheiden, die *diffuse* (allseitige) und die *spiegelnde* (specular) Reflexion. Die diffuse Reflexion ergibt matte Oberflächen, die spiegelnde glänzende.

2.4.1 Diffuse Reflexion

Bei rauhen Oberflächen wird das einfallende Licht diffus, d.h. in alle Richtungen, reflektiert. Beispiele: Mond, Papier, Wand



D Punkt auf der Oberfläche eines Körpers
$\vec{n}$ nach aussen gerichteter Normalenvektor im Punkt D
$\vec{\ell}$ Richtung zur Lichtquelle
α Einfallswinkel des Lichtes zur Normalen

Weil das Licht nach allen Richtungen reflektiert wird, hat die Beobachtungsrichtung keinen Einfluss auf die Helligkeit (eine Wand erscheint aus allen Richtung gleich hell), hingegen ist die Einfallsrichtung des Lichtes wichtig. Bei senkrechtem Einfall ist die Helligkeit maximal, weil dann die Energie, die pro Fläche auftrifft maximal ist.

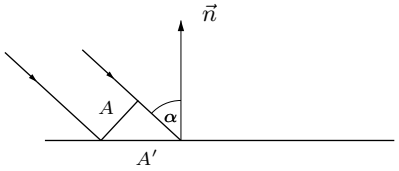
Gesetz von Lambert

$$\boxed{I = a \cdot \cos(\alpha)} \tag{2.14}$$

I Helligkeit (Intensität) des reflektierten Lichtes
a Konstante Reflexionskonstante (maximale Helligkeit)

Begründung der Gleichung 2.14

Die pro Flächeneinheit auftreffende Energie nimmt mit wachsendem α mit dem Faktor $\cos\alpha$ ab. Dies ist eine Folge des Flächen-Verhältnisses bei senkrechter und schiefer Parallelprojektion (Licht wird auf grössere Fläche verteilt):



$A : A' = \cos\alpha$

Die Berechnung von $\cos(\alpha)$ erfolgt mit dem Skalarprodukt:

$$\cos\alpha \;=\; \frac{\vec{\ell}\cdot\vec{n}}{|\vec{\ell}\,|\cdot|\vec{n}\,|}$$

Im folgenden setzen wir voraus, dass die Vektoren $\vec{\ell}$ und $\vec{n}$ *normiert* sind (Länge 1). Dadurch entfällt der Nenner, und das Gesetz von Lambert lautet:

$$\boxed{I \;=\; a\cdot\vec{\ell}\cdot\vec{n}} \qquad |\vec{\ell}\,|=1, \quad |\vec{n}|=1 \tag{2.15}$$

a Reflexionskonstante (maximale Helligkeit)

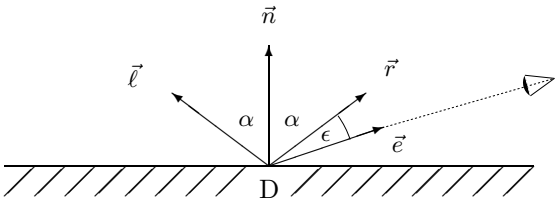
Für $\alpha > 90^o$ wird $I < 0$. Dies ist als 0 zu interpretieren, da in diesem Fall der Vektor $\vec{\ell}$ in den Körper hinein zeigt (Oberfläche nicht beleuchtet).

2.4.2 Spiegelnde Reflexion

Die spiegelnde Reflexion tritt bei glatten Oberflächen (Metall) auf und erzeugt einen Glanzfleck. Sie ist richtungsspezifisch, nach dem Gesetz

Einfallswinkel = Ausfallswinkel.

In der Praxis heisst dies, dass ein einfallender Strahl in einem *Winkelbereich* (Kegel) mit Maximum in Richtung des gespiegelten Strahls $\vec{r}$ reflektiert wird.



$\vec{n}$ Flächennormale, nach aussen gerichtet
$\vec{\ell}$ Richtung zur Lichtquelle (toLight)
$\vec{r}$ gespiegelter Strahl (Einfallswinkel = Ausfallswinkel)
$\vec{e}$ Richtung zum Beobachter (toEye)

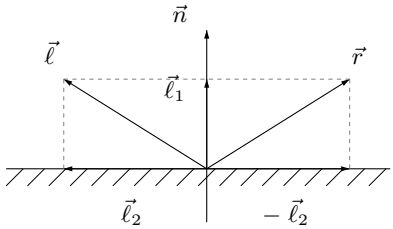
Der Winkel ϵ gibt die Abweichung der Beobachtungsrichtung von der Richtung des gespiegelten Strahls an. Er bestimmt die Intensität des gespiegelten Lichtes in Beobachtungsrichtung. Sie ist maximal für $\epsilon = 0$ und nimmt mit wachsendem Winkel ϵ ab.

Berechnung des gespiegelten Strahls

Den gespiegelten Strahl $\vec{r}$ erhält man leicht durch Orthogonalzerlegung von $\vec{\ell}$ bezüglich $\vec{n}$ mit den Gleichungen 2.12, Seite 46:

$$\vec{\ell} \;=\; \vec{\ell}_1 + \vec{\ell}_2$$

$$\vec{\ell}_1 = (\vec{\ell} \cdot \vec{n})\, \vec{n}, \quad \vec{\ell}_2 = \vec{\ell} - \vec{\ell}_1$$

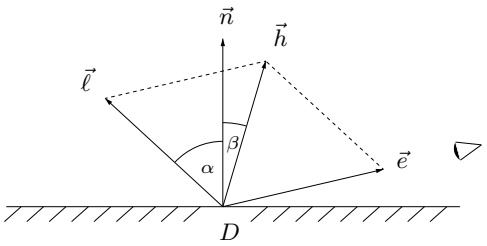


Aus der Figur ergibt sich:

$$\vec{r} \;=\; \vec{\ell}_1 + (-\vec{\ell}_2) = 2\,\vec{\ell}_1 - \vec{\ell} \tag{2.16}$$

Die Phong-Näherung

Das Modell von Phong verwendet anstelle des reflektierten Strahls $\vec{r}$ den einfacher zu berechnenden *winkelhalbierenden Vektor* $\vec{h}$ zwischen $\vec{\ell}$ und $\vec{e}$:



Konvention

Im folgenden nehmen wir an, dass die Vektoren $\vec{n}$, $\vec{\ell}$ und $\vec{e}$ normiert sind:

$$|\vec{n}| = 1, \;\; |\vec{\ell}| = 1, \;\; |\vec{e}| = 1$$

Der Vektor $\vec{h}$ heisst 'Halfway-between' Vektor . Er kann leicht berechnet werden, falls $\vec{\ell}$ und $\vec{e}$ normiert sind:

Halfway-between

$$\vec{h} \;=\; \text{normalize}(\,\vec{\ell} + \vec{e}\,)$$

Wenn $\vec{e}$ die Richtung des gespiegelten Strahls hat, ist der Winkel β zwischen $\vec{n}$ und $\vec{h}$ gleich 0. Mit zunehmender Abweichung der Beobachtungsrichtung von der Spiegelungsrichtung nimmt β zu und ist (wie der obige Winkel ϵ) ein Mass für diese Abweichung. Wenn die Strahlen $\vec{\ell}$, $\vec{e}$ und $\vec{n}$ komplanar sind, kann man zeigen, dass $\epsilon = 2 \cdot \beta$ gilt.

Helligkeit I des gespiegelten Lichtes in Richtung $\vec{e}$:

Wir nehmen wie beim Gesetz von Lambert an, dass die Helligkeit mit $\cos(\beta)$ abnimmt. Zusätzlich führt man hier noch einen Exponenten p ein, mit dem man steuern kann, wie rasch die Helligkeit mit wachsendem Winkel β abnimmt. Dies ergibt den folgenden Ansatz:

$$\boxed{I = b \cdot (\cos(\beta))^p = b \cdot (\vec{h} \cdot \vec{e})^p} \tag{2.17}$$

mit:

- b Reflexionskonstante der spiegelnden Reflexion
- p Exponent für Abnahme der spiegelnden Reflexion (specular exponent, shininess)

Der Exponent p bestimmt die Grösse des Spiegelfleckes der Lichtquelle.

Bedingung

Sei E die Ebene durch den Punkt D, senkrecht zu $\vec{n}$.

Da der gespiegelte Strahl in dieselbe Seite wie der einfallende Strahl zeigt, müssen die Lichtquelle und der Beobachter auf derselben Seiten der Ebene E sein.

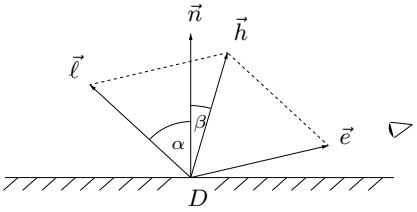
Dies ist der Fall, wenn die Orthogonalprojektionen der Vektoren $\vec{\ell}$ und $\vec{e}$ auf $\vec{n}$ gemäss Seite 46 gleichgerichtet sind, d.h. die Orthogonalkomponenten $\vec{\ell} \cdot \vec{n}$ und $\vec{e} \cdot \vec{n}$ haben dasselbe Vorzeichen. Dies ergibt die Bedingung:

$$(\vec{\ell} \cdot \vec{n}) \cdot (\vec{e} \cdot \vec{n}) \geq 0 \tag{2.18}$$

Andernfalls ist für die spiegelnde Reflexion $I = 0$ zu setzen.

2.4.3 Resultierende Gesamthelligkeit

In Wirklichkeit ist die Helligkeit eine Linearkombination der Intensitäten der diffusen und der spiegelnden Reflexion. Zusätzlich kommt noch eine Grundhelligkeit dazu, das Streulicht (ambientes Licht).



$$\boxed{I_{tot}(\vec{\ell},\vec{n},\vec{e}) \;=\; I_{ambient} + I_{diffuse} + I_{specular}} \tag{2.19}$$

$\vec{\ell}$, $\vec{n}$, $\vec{e}$ Einheitsvektoren

$I_{ambient}$	konstante Grundhelligkeit
$I_{diffuse} = a \cdot \vec{\ell} \cdot \vec{n}$	diffuse Reflexion (Lambert)
$\vec{h} = \text{normalize}(\vec{\ell} + \vec{e})$	Halfway-between Vektor
$I_{specular} = b \cdot (\vec{n} \cdot \vec{h}\,)^p$	spiegelnde Reflexion

Falls $\vec{\ell} \cdot \vec{n} < 0$ ist, setze $I_{diffuse} = 0$
Falls $(\vec{\ell} \cdot \vec{n}) \cdot (\vec{e} \cdot \vec{n}) < 0$ ist, setze $I_{specular} = 0$

Geeignete Parameter: $I_{ambient} = 0.2, \quad a = 0.4, \quad b = 0.4, \quad p = 20$

2.4.4 Implementation der Beleuchtung

Die Beleuchtungsrechnung kann in den Vertex- oder Fragment-Shader integriert werden. Im ersten Fall spricht man von *Gouraud-Shading*, im zweiten Fall von *Phong-Shading* (nicht zu verwechseln mit der Phong-Näherung).

Beim Gouraud-Shading werden die Helligkeiten in jedem Vertex berechnet und dann von OpenGL (wie alle Vertex-Attribute) für die dazwischen liegenden Fragmente (Pixel) interpoliert.

Beim Phong-Shading werden die Normalen-Vektoren der Vertices für die dazwischen liegenden Fragmente interpoliert, und die Beleuchtungsrechnung wird für jedes Fragment ausgeführt.

Das Phong-Shading ist aufwendiger als das Gouraud-Shading, es ergibt aber i.a. bessere Resultate. Wenn z.B. ein scharfer Spiegelfleck zwischen Vertices liegt, kann dieser beim Gouraud-Shading verloren gehen, wenn nur die Helligkeiten der Vertices interpoliert werden.

Unsere Shader vShader1 und fShader1 implementieren das Phong-Shading. Da die Shader-Language die Vektor- und Matrix-Algebra gut unterstützt, ist die Implementation eine direkte Umsetzung der obigen Gleichungen für die Beleuchtung.

Kapitel 3

Matrix-Algebra

Matrizen wurden zur Darstellung und Bearbeitung von linearen Gleichungssystemen eingeführt. In der Computergraphik werden sie für lineare Punktabbildungen und Koordinatentransformationen eingesetzt.

3.1 Einführung von Matrizen

Eine *Matrix* ist ein rechteckförmiges Schema von reellen Zahlen.

Matrix mit 3 Zeilen und 4 Spalten (3×4 Matrix):

$$A = \begin{pmatrix} 3.3 & -2.1 & 1.5 & 6.0 \\ 0.5 & 4.5 & 3.1 & -1.2 \\ 12.3 & 3.3 & 1.8 & 8.4 \end{pmatrix}$$

Allgemeine $m \times n$ Matrix:

Die Elemente werden mit zwei Indizes numeriert:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & & & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} = (a_{ij})_{\substack{1 \le i \le m \\ 1 \le j \le n}}$$

a_{ij}	Element in Zeile i und Spalte j. Alternative Bezeichnung: A_{ij}
m	Anzahl Zeilen
n	Anzahl Spalten

Die Menge aller $m \times n$ Matrizen wird mit $\mathbb{R}^{m \times n}$ bezeichnet.

Beispiele:

1. Matrix eines Bitmap-Bildes

   Ein Bitmap-Bild (JPG, GIF) wird beschrieben durch eine Matrix, welche für jeden Bildpunkt des Bildes (Pixel) die Farbinformationen des Punktes enthält, z.B. die Helligkeit bei einem Graustufenbild

2. Koeffizienten-Matrix eines linearen Gleichungssystems

   Lineares Gleichungssystem mit 3 Gleichungen und 4 Unbekannten:

$$\begin{array}{rcl} 3.2x_1 - 5.1x_2 + 2.4x_3 - 6.2x_4 & = & 2.5 \\ 9.2x_1 - 1.4x_2 + 6.2x_3 + 4.3x_4 & = & -4 \\ 8.1x_1 - 4.2x_2 + 9.1x_3 + 2.2x_4 & = & 3 \end{array}$$

   Koeffizienten-Matrix des Systems (3×4 Matrix):

$$A = \begin{pmatrix} 3.2 & -5.1 & 2.4 & -6.2 \\ 9.2 & -1.4 & 6.2 & 4.3 \\ 8.1 & -4.2 & 9.1 & 2.2 \end{pmatrix}$$

Spezielle Matrizen

- Zeilen- und Spalten-Vektoren

  Ein Spaltenvektor $x \in \mathbb{R}^m$ ist nichts anderes als eine $m \times 1$ Matrix und ein Zeilenvektor ist analog eine $1 \times n$ Matrix.

- Diagonalmatrix

  Eine *Diagonalmatrix* enthält nur in der Hauptdiagonalen Elemente verschieden von 0 :

$$\begin{pmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{pmatrix}$$

- Einheitsmatrix

  Die *Einheitsmatrix* $\mathbb{I}_n$ ist die $n \times n$ Diagonalmatrix mit lauter Einsen in der Diagonale, z.B. für $n = 4$:

$$\mathbb{I}_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

  Häufig lässt man den Index n weg, wenn die Dimension aus dem Zusammenhang heraus klar ist.

- Dreiecksmatrix

  Eine obere (untere) *Dreiecksmatrix* enthält nur oberhalb (unterhalb) der Hauptdiagonalen und auf der Hauptdiagonalen Elemente verschieden von 0:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{pmatrix}$$

3.2 Die Grundoperationen

Für Matrizen einer festen Dimension $m \times n$ hat man die Grundoperationen:

- Addition und Subtraktion
- Multiplikation mit einer reellen Zahl

Diese Operationen sind wie bei Vektoren komponentenweise definiert:

$$\begin{pmatrix} 2 & 4 \\ -1 & 3 \end{pmatrix} + \begin{pmatrix} 1 & 3 \\ 2 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 7 \\ 1 & 3 \end{pmatrix} \qquad 2 \cdot \begin{pmatrix} 2 & 4 \\ -1 & 3 \end{pmatrix} = \begin{pmatrix} 4 & 8 \\ -2 & 6 \end{pmatrix}$$

Mit diesen Operationen bildet $(\mathbb{R}^{m\times n}, +, \cdot)$ einen *Vektorraum*.

Transposition

Sei A eine $m \times n$ Matrix. Die *transponierte* Matrix A^τ ist die $n \times m$ Matrix, deren Zeilen die Spalten von A sind:

$$A = \begin{pmatrix} * & \cdot & \cdot & \cdot \\ * & \cdot & \cdot & \cdot \\ * & \cdot & \cdot & \cdot \end{pmatrix} \qquad \rightarrow \qquad A^\tau = \begin{pmatrix} * & * & * \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}$$

$$A = \begin{pmatrix} 3 & 2 & 1 & 6 \\ 0 & 4 & 5 & 1 \\ 2 & 3 & 1 & 8 \end{pmatrix} \qquad \rightarrow \qquad A^\tau = \begin{pmatrix} 3 & 0 & 2 \\ 2 & 4 & 3 \\ 1 & 5 & 1 \\ 6 & 1 & 8 \end{pmatrix}$$

Die i-te Zeile von A^τ ist also gleich der i-ten Spalte von A, d.h.

$$\boxed{(A^\tau)_{ij} \;=\; A_{ji}} \qquad 1 \le i \le n, \quad 1 \le j \le m$$

Folgerungen:

$$(A + B)^\tau \;=\; A^\tau + B^\tau$$

$$(t \cdot A)^\tau \;=\; t \cdot A^\tau \qquad\qquad t \in \mathbb{R}$$

Eine Matrix heisst *symmetrisch*, wenn $A^\tau = A$. Eine symmetrische Matrix muss insbesondere *quadratisch* sein, d.h gleich viele Zeilen wie Spalten haben $(m = n)$.

symmetrische Matrix

3.3 Das Matrixprodukt

Das Produkt zweier Matrizen A und B ist die wichtigste Operation der Matrix-Algebra. Es ist so definiert, dass es für lineare Gleichungssysteme und Transformationen verwendet werden kann. Dazu werden die Skalarprodukte der Zeilen von A mit den Spalten von B gebildet:

Das Element der Produktmatrix $A \cdot B$ in Zeile i und Spalte j ist das *Skalarprodukt* der i-ten Zeile von A mit der j-ten Spalte von B :

$$(A \cdot B)_{ij} \;=\; (\text{Zeile } i \text{ von } A) \;\cdot\; (\text{Spalte } j \text{ von } B) \tag{3.1}$$

Bedingung:

Die Zeilen von A müssen dieselbe Länge wie die Spalten von B haben.

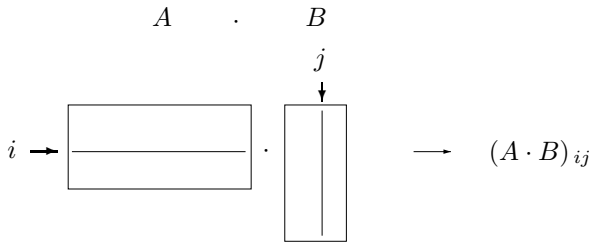
Dies ergibt die folgende Definition des Matrix-Produktes.

Definition

Sei A eine $m \times n$ und B eine $n \times r$ Matrix. Das *Produkt* $A \cdot B$ ist die Matrix $m \times r$ Matrix mit den Elementen

$$\boxed{(A \cdot B)_{ij} \;=\; a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj}} \tag{3.2}$$

für $1 \le i \le m$ und $1 \le j \le r$.



Beispiel:

$$\begin{pmatrix} . & . & . \\ 2 & 4 & 1 \\ . & . & . \end{pmatrix} \cdot \begin{pmatrix} . & 2 \\ . & 3 \\ . & 1 \end{pmatrix} \;=\; \begin{pmatrix} . & . \\ . & 17 \\ . & . \end{pmatrix}$$

Merke:

$$\begin{array}{ccccc} 3 \times 3 & \cdot & 3 \times 2 & & 3 \times 2 \\ m \times n & \cdot & n \times r & \longrightarrow & m \times r \end{array}$$

Das Falk-Schema

Für die praktische Berechnung des Matrixproduktes eignet sich das *Falk-Schema*. Bei diesem werden die Faktoren A und B in einer Tabelle angeordnet, A links unten, B rechts oben:

		4	-2	2
		5	2	-1
-2	3	7	$.$	$.$
3	-2	$.$	$.$	$.$

$$\begin{pmatrix} -2 & 3 \\ 3 & -2 \end{pmatrix} \cdot \begin{pmatrix} 4 & -2 & 2 \\ 5 & 2 & -1 \end{pmatrix} = \begin{pmatrix} 7 & 10 & -7 \\ 2 & -10 & 8 \end{pmatrix}$$

$$2 \times 2 \quad \cdot \quad 2 \times 3 \qquad \rightarrow \qquad 2 \times 3$$

Beispiele zum Matrix-Produkt:

1. Produkt einer Matrix A mit einem Spaltenvektor x

   $$\begin{pmatrix} 3 & 2 & 1 \\ 2 & 4 & 1 \\ 7 & 2 & 4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

   Für die Komponenten bedeutet dies:

   $$\begin{array}{rcl} 3\,x_1 + 2\,x_2 + x_3 & = & y_1 \\ 2\,x_1 + 4\,x_2 + x_3 & = & y_2 \\ 7x_1 + 2\,x_2 + 4\,x_3 & = & y_3 \end{array}$$

   Dies ist ein lineares Gleichungssystem. Dabei wurde x als 3×1 Matrix aufgefasst.

   Merke:

   Die Gleichung

   $$\boxed{A \cdot x \;=\; y} \tag{3.3}$$

   stellt ein *lineares Gleichungssystem* mit Koeffizientenmatrix A dar.

2. Spaltenweise Berechnung eines Matrixproduktes

   $$\begin{pmatrix} 3 & 2 & 1 \\ 2 & 4 & 1 \\ 1 & 2 & 4 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 \\ 3 & 3 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} 13 & 12 \\ 17 & 17 \\ 12 & 19 \end{pmatrix}$$

   Das Produkt kann so berechnet werden, dass man die Matrix A mit den Spaltenvektoren von B gemäss Beispiel 1 multipliziert und die so erhaltenen Spaltenvektoren nebeneinander stellt.

Gesetze des Matrixproduktes

In den folgenden Gleichungen sind A, B, C beliebige Matrizen, für welche die aufgeführten Operationen definiert sind.

$A \cdot \mathbb{I} = A \quad$ und $\quad \mathbb{I} \cdot A = A$	Einheitsmatrix
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	Assoziativgesetz
$A \cdot (B + C) = A \cdot B + A \cdot C$	Distributivgesetze
$(A + B) \cdot C = A \cdot C + B \cdot C$	

▶ *Merke:*

Das Kommutativgesetz gilt *nicht*. Im allgemeinen ist

$$A \cdot B \neq B \cdot A$$

Multiplikation und Transposition:

$$\boxed{(A \cdot B)^\tau = B^\tau \cdot A^\tau} \qquad \text{(umgekehrte Reihenfolge!)} \qquad (3.4)$$

Potenzen von Matrizen

Seien A eine quadratische Matrix und n eine ganze Zahl ≥ 0.

Wie bei reellen Zahlen definiert man:

$$A^n = A \cdot A \cdots A \qquad n \text{ Faktoren}$$

Wegen dem Assoziativ-Gesetz des Matrix-Produktes müssen keine Klammern gesetzt werden.

Aufgrund dieser Gesetze kann man mit Matrizen wie mit Zahlen rechnen, mit dem Unterschied, dass das Kommutativgesetz nicht gilt.

Beispiel:

$$\begin{aligned} A \cdot (3B + 4A) - (2A + B) \cdot A &= 3AB + 4A^2 - 2A^2 - BA = \\ &= 2A^2 + 3AB - BA \end{aligned}$$

Die letzten beiden Terme können nicht zusammengefasst werden, weil das Kommutativgesetz nicht gilt.

3.4 Die Inverse einer Matrix

Die Inverse einer reellen Zahl $a \neq 0$ ist bekanntlich ihr Kehrwert $a^{-1} = 1/a$, definiert durch

$$a \cdot a^{-1} = 1$$

Mit dem Kehrwert erhält man u.a. sofort die Lösung der linearen Gleichung

$$\begin{aligned} a \cdot x &= b \\ x &= \frac{b}{a} = a^{-1} \cdot b \end{aligned}$$

Für quadratische Matrizen kann man eine *inverse Matrix* mit analogen Eigenschaften definieren. Diese existiert aber nicht für alle Matrizen verschieden von null, sondern nur für sogenannte *invertierbare Matrizen:*

Definition:

Eine $n \times n$ Matrix A heisst *invertierbar* oder *regulär*, wenn es eine Matrix B mit den folgenden Eigenschaften gibt: *invertierbar*

$$A \cdot B = \mathbb{I} \qquad \text{und} \qquad B \cdot A = \mathbb{I} \tag{3.5}$$

Eine solche Matrix B heisst *Inverse* von A. Weil das Matrix-Produkt nicht kommutativ ist, verlangt man beide Bedingungen. Die erste Bedingung bedeutet, dass die Matrix B *rechtsinvers* zu A ist, die zweite, dass sie *linksinvers* zu A ist. *Inverse*

Folgerung

Wenn eine Matrix A eine Inverse hat, so ist diese eindeutig bestimmt.

Beweis:

Sei B' eine zweite Inverse, d.h. es gilt auch

$$A \cdot B' = \mathbb{I} \quad \text{und} \quad B' \cdot A = \mathbb{I}$$

Dann folgt:

$$B' = B' \cdot \mathbb{I} = B' \cdot (A \cdot B) == (B' \cdot A) \cdot B = B \qquad \square$$

Die eindeutige Inverse einer invertierbaren Matrix A wird mit A^{-1} bezeichnet. Also: A^{-1}

$$\boxed{A \cdot A^{-1} = \mathbb{I} \qquad \text{und} \qquad A^{-1} \cdot A = \mathbb{I}} \tag{3.6}$$

Bemerkung

Für den Nachweis, dass eine Matrix B die Inverse von A ist, genügt es zu zeigen, dass sie *eine* der beiden Bedingungen $A \cdot B = \mathbb{I}$ und $B \cdot A = \mathbb{I}$ erfüllt. Nach einem Satz der Linearen Algebra erfüllt sie dann die andere auch.

Gesetze der Inversen

Seien A und B invertierbare $n \times n$ Matrizen. Dann gelten

$$\boxed{\begin{aligned} (\, A \cdot B \,)^{-1} &= B^{-1} \cdot A^{-1} \\ (\, A^{\tau} \,)^{-1} &= (\, A^{-1})^{\tau} \\ (\, A^{-1} \,)^{-1} &= A \end{aligned}} \tag{3.7}$$

Beweis:

Verifikation der Inversen-Bedingung, z.B. für das erste Gesetz:

$$(A \cdot B) \cdot (B^{-1} \cdot A^{-1}) = A \cdot (B \cdot B^{-1}) \cdot A^{-1} = A \cdot \mathbb{I} \cdot A^{-1} = \mathbb{I}$$

Beispiele und Folgerungen:

1. Diagonalmatrix

$$D = \begin{pmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{pmatrix} \qquad D^{-1} = \begin{pmatrix} \frac{1}{a_1} & 0 & 0 \\ 0 & \frac{1}{a_2} & 0 \\ 0 & 0 & \frac{1}{a_3} \end{pmatrix}$$

   Probe ! Bedingung: alle $a_i \neq 0$.

2. $A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ $\qquad A^{-1} = A$ Probe !

3. Auflösung von linearen Gleichungssystemen

   Sei A eine invertierbare $n \times n$ Matrix mit der Inversen A^{-1}. Wir betrachten das Gleichungssystem

   $$A \cdot x = b$$

   Auflösung nach x :

   Wir multiplizieren beide Seiten von links mit A^{-1} :

   $$A^{-1} \cdot A \cdot x = A^{-1} \cdot b$$

   Mit $A^{-1}A = \mathbb{I}$ folgt:

   $$\boxed{x = A^{-1}\, b} \qquad \text{Lösung des Systems} \tag{3.8}$$

   Probe! Nach dem gleichen Verfahren kann auch eine *Matrix-Gleichung*

   $$A \cdot X = B$$

   mit der Inversen von A gelöst werden:

   $$X = A^{-1}B$$

Berechnung der Inversen

Die Inverse einer Matrix wird mit dem *Gauss'schen Algorithmus* beechnet. Alternativ gibt es eine Formel für die Inverse mit *Determinanten*. Diese eignet sich für Matrizen der Grösse ≤ 4. Für grössere Matrizen ist der Gauss'sche Algorithmus wesentlich effizienter. Die Methode inverse der Klasse Mat4 verwendet die Formel mit Determinanten.

Wir benötigen diese Verfahren nicht, da wir nur Inverse von Matrizen von Drehungen und Translationen benötigen, welche direkt angegeben werden können (entgegengesetzte Drehwinkel bzw. Verschiebungen).

Kapitel 4

Lineare Abbildungen

4.1 Definition einer linearen Abbildung

Eine *lineare Abbildung* oder *lineare Transformation* von $\mathbb{R}^n$ nach $\mathbb{R}^m$ ist eine Punktabbildung

$$T : \mathbb{R}^n \to \mathbb{R}^m : x \mapsto y = T(x) \,,$$

welche die folgenden *Linearitätsbedingungen* erfüllt:

$$\boxed{\begin{array}{lcll} T(a+b) & = & T(a)+T(b) & \text{Additivität} \\ T(t\cdot a) & = & t\cdot T(a) & \text{Homogenität} \end{array}} \tag{4.1}$$

für alle $a, b \in \mathbb{R}^n$ und $t \in \mathbb{R}$.

Beispiel: $T : \mathbb{R}^n \to \mathbb{R}^n : x \mapsto T(x) = c \cdot x$ Streckung, $c \in \mathbb{R}$

Wie wir sogleich sehen werden, haben die Linearitätsbedingungen zur Folge, dass die Abbildung mit einer Matrix und linearen Abbildungsgleichungen für die Koordinaten dargestellt werden kann.

Eine lineare Abbildung heisst auch *Vektorraum-Homomorphismus*, da die Vektorraum-Operationen unter der Abbildung erhalten bleiben. Analog ist eine lineare Abbildung zwischen beliebigen Vektorrräumen definiert.

Homomorphismus

Folgerungen

1. Der Nullpunkt wird auf den Nullpunkt abgebildet: $T(0) = 0$

   Beweis:

   Wegen $0 = 0 + 0$ folgt aus der Linearitätsbedingung

   $$T(0) = T(0) + T(0).$$

   Daraus folgt die Behauptung durch Subtraktion von $T(0)$ auf beiden Seiten dieser Gleichung.

2. Eine *Translation* mit Verschiebungsvektor v ungleich 0 ist *keine* lineare Abbildung.

   Dies folgt aus der vorangehenden Folgerung, weil der Nullpunkt auf v (also nicht auf 0) abgebildet wird.

3. Bild einer Linearkombination

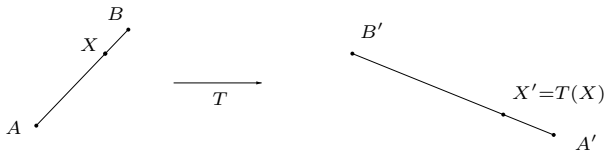
   $$T(s \cdot a + t \cdot b) \;=\; s \cdot T(a) + t \cdot T(b) \qquad \text{für alle} \quad s,t \in \mathbb{R}, \quad a,b \in \mathbb{R}^n$$

   Beweis:

   Mit den Linearitätsbedingungen für T erhält man

   $$T(s \cdot a + t \cdot b) = T(s \cdot a) + T(t \cdot b) = s \cdot T(a) + t \cdot T(b)$$

4. Ein Teilungspunkt X einer Strecke AB wird auf den entsprechenden Teilungspunkt X' der Strecke $A'B'$ abgebildet, wobei $A' = T(A)$ und $B' = T(B)$. Damit werden die Punkte der Strecke AB auf die Punkte der Strecke $A'B'$ abgebildet.



   Beweis:

   Sind $\vec{a}$, $\vec{b}$ und $\vec{x}$ die Ortsvektoren der Punkte, so gilt:

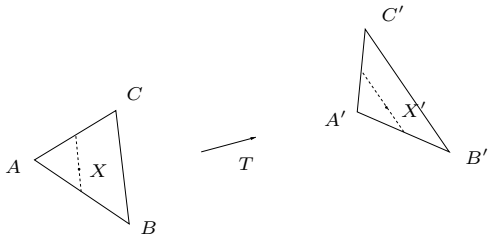
   $$\vec{x} = \vec{a} + t \cdot (\vec{b} - \vec{a}) = (1-t)\,\vec{a} + t\,\vec{b}$$

   Mit der Folgerung 3 folgt:

   $$T(\vec{x}) = (1-t)\,T(\vec{a}) + t\,T(\vec{b})$$

   Damit ist $T(\vec{x})$ der entsprechende Teilungspunkt der Strecke $A'B'$.

5. Die Punkte eines Dreiecks ABC werden auf die Punkte des Dreiecks $A'B'C'$ abgebildet, wobei $A' = T(A)$, $B' = T(B)$ und $C' = T(C)$.



   Dies folgt sofort aus der vorangehenden Folgerung mit den gestrichelt dargestellten Hilfsstrecken.

4.2 Die Matrix einer linearen Abbildung

Es besteht eine umkehrbar eineindeutige Beziehung zwischen linearen Abbildungen und Matrizen, d.h. zu jeder Matrix gehört eine lineare Transformation und umgekehrt.

Gegeben sei eine Matrix

$$A = \begin{pmatrix} 2 & 4 & 3 \\ -1 & 3 & 2 \\ 4 & -2 & 1 \end{pmatrix}$$

Wir betrachten die Punktabbildung

$$T_A : \mathbb{R}^3 \to \mathbb{R}^3 : x \mapsto y$$

gegeben durch die Abbildungsgleichung

$$\boxed{y \;=\; A \cdot x} \tag{4.2}$$

Aus den Gesetzen der Matrixmultiplikation folgt sofort, dass diese Abbildung die Linearitätsbedingungen erfüllt, z.B.

$$A \cdot (x + y) \;=\; A \cdot x + A \cdot y$$

Die Abbildung T_A heisst die zu A *assoziierte lineare Abbildung.*

Die Abbildungsgleichung $y = A \cdot x$ bedeutet in Komponenten:

$$\begin{array}{rcrcrcr} y_1 & = & 2x_1 & + & 4x_2 & + & 3x_3 \\ y_2 & = & -x_1 & + & 3x_2 & + & 2x_3 \\ y_3 & = & 4x_1 & - & 2x_2 & + & x_3 \end{array}$$

Dies sind *lineare Abbildungsgleichungen.*

Die Bilder der Basisvektoren e_j sind die Spaltenvektoren der Matrix A:

$$\boxed{T_A(e_j) \;=\; j\text{-te Spalte der Matrix } A} \tag{4.3}$$

Dies folgt sofort aus 4.2, z.B. für e_2:

$$T_A(e_2) \;=\; A \cdot e_2 \;=\; \begin{pmatrix} 2 & 4 & 3 \\ -1 & 3 & 2 \\ 4 & -2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \;=\; \begin{pmatrix} 4 \\ 3 \\ -2 \end{pmatrix}$$

Allgemein sei A eine beliebige $m \times n$ Matrix.

Die zu A assoziierte lineare Abbildung ist definiert durch

$$\boxed{T_A : \mathbb{R}^n \to \mathbb{R}^m : x \mapsto y = A \cdot x}$$

Sie erfüllt 4.3. Nach dem folgenden Satz hat jede lineare Abbildung diese Form.

Satz 4.2.1

Sei $T : \mathbb{R}^n \to \mathbb{R}^m$ eine beliebige lineare Abbildung.

Dann gibt es genau eine $m \times n$ Matrix A, sodass T die zu A assoziierte lineare Abbildung T_A ist, d.h.

$$\boxed{T(x) \;=\; A \cdot x} \qquad \text{für alle} \quad x \in \mathbb{R}^n \tag{4.4}$$

Die Spalten der Matrix A sind gegeben durch die Bilder $T(e_j)$ der Basisvektoren $e_j,\ j = 1 \ldots n$.

Beweis:

Wir setzen

$$a_j \;:=\; T(e_j) \qquad \text{für} \quad j = 1..n$$

Weiter sei A die Matrix mit den Spalten a_j, und x sei ein beliebiger Vektor in $\mathbb{R}^n$ mit Basisdarstellung

$$x = x_1 e_1 + \cdots + x_n e_n$$

Wegen der Linearität von T und T_A folgt:

$$T(x) \;=\; x_1 T(e_1) + \cdots + x_n T(e_n) \;=\; x_1 a_1 + \cdots + x_n a_n$$

$$T_A(x) = A \cdot (x_1 e_1 + \cdots + x_n e_n) \;=\; x_1 A \cdot e_1 + \cdots + x_n A \cdot e_n \;=\; x_1 a_1 + \cdots + x_n a_n$$

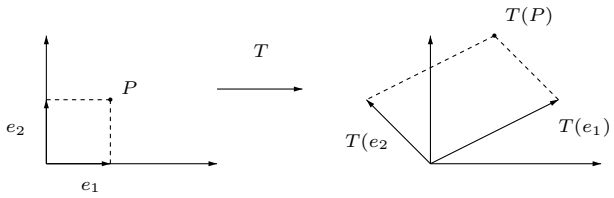
Eindeutigkeit von A:

Nach 4.3 sind die Spalten von A eindeutig durch die Abbildung T festgelegt, also ist A eindeutig bestimmt.

Folgerung:

Zur Festlegung einer linearen Abbildung genügt es, die Bilder der Basisvektoren beliebig vorzugeben. Dadurch ist die Matrix gemäss Gleichung 4.3 bestimmt. Beispiel in der Ebene:

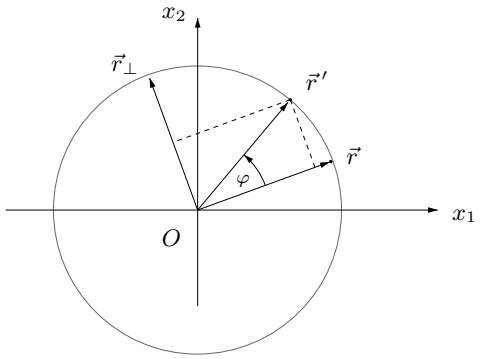
$$T(e_1) = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \quad T(e_2) = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \qquad A = \begin{pmatrix} 2 & -1 \\ 1 & 1 \end{pmatrix}$$



Beispiele:

1. Drehung in der Ebene

   Wir betrachten eine Drehung um den Nullpunkt mit Drehwinkel φ. Gemäss Konvention ist der Drehsinn für positive Drehwinkel im Gegenuhrzeigersinn.



$\vec{r} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ Ortsvektor eines beliebigen Punktes der Ebene

$\vec{r}\,' = \begin{pmatrix} x_1' \\ x_2' \end{pmatrix}$ Ortsvektor des gedrehten Punktes

Wir führen einen Vektor $\vec{r}_\perp$ senkrecht zu $\vec{r}$ ein: $\vec{r}_\perp = \begin{pmatrix} -x_2' \\ x_1' \end{pmatrix}$

(Verifikation der Orthogonalität mit Skalarprodukt). Gemäss der Figur gilt im dargestellten Kreis (wie im Einheitskreis):

$$\vec{r}\,' = \cos(\varphi) \cdot \vec{r} + \sin(\varphi) \cdot \vec{r}_\perp$$

Durch Einsetzen der Komponenten der Vektoren folgen die Abbildungsgleichungen der Drehung:

$$\begin{aligned} x_1' &= \cos(\varphi) \cdot x_1 - \sin(\varphi) \cdot x_2 \\ x_2' &= \sin(\varphi) \cdot x_1 + \cos(\varphi) \cdot x_2 \end{aligned}$$

Matrixschreibweise:

$$\vec{x}\,' = R(\varphi) \cdot \vec{x}$$

mit der Matrix

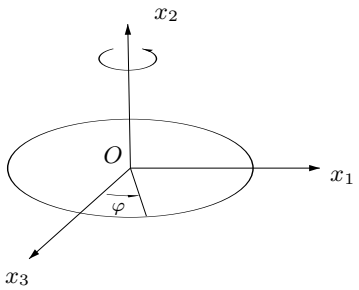
$$\boxed{R(\varphi) = \begin{pmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{pmatrix}} \qquad \text{Drehmatrix} \qquad (4.5)$$

Probe: Die Spalten sind die gedrehten Basisvektoren

2. Drehungen um die Koordinatenachsen des Raumes

   Drehungen im Raum um die Koordinatenachsen sind einfache Erweiterungen der Drehungen in der Ebene.

   Drehung um die x_2-Achse:



   Die x_2 Koordinate bleibt bei der Drehung unverändert, und die (x_1, x_3) Koordinaten werden mit einer Drehung in der Ebene transformiert. Zur Berechnung der Matrix bestimmen wir die Bilder der Basisvektoren mit Hilfe der Figur:

$$e_1' = \begin{pmatrix} \cos\varphi \\ 0 \\ -\sin\varphi \end{pmatrix}, \qquad e_2' = e_2, \qquad e_3' = \begin{pmatrix} \sin\varphi \\ 0 \\ \cos\varphi \end{pmatrix}$$

   Dies ergibt die Matrix der Drehung:

$$R_2(\varphi) = \begin{pmatrix} \cos\varphi & 0 & \sin\varphi \\ 0 & 1 & 0 \\ -\sin\varphi & 0 & \cos\varphi \end{pmatrix} \qquad \text{Drehung um } x_2\text{-Achse}$$

   analog:

$$R_1(\varphi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{pmatrix} \qquad \text{Drehung um } x_1\text{-Achse}$$

$$R_3(\varphi) = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \text{Drehung um } x_3\text{-Achse}$$

   Drehrichtung (Rechte Hand Regel):

   Hält man die rechte Hand so, dass der Daumen in die positive Richtung der Drehachse (Koordinatenachse) zeigt, so zeigen die Finger für positive Drehwinkel in die Drehrichtung.

3. Vektorprodukt als lineare Abbildung

   Sei $\vec{a} \in \mathbb{R}^3$. Wir betrachten die Abbildung

   $$T : \mathbb{R}^3 \to \mathbb{R}^3 : \vec{x} \mapsto \vec{y} = \vec{a} \times \vec{x}$$

   Diese ist gemäss den Gesetzen des Vektorproduktes linear, z.B. ist

   $$T(\vec{x} + \vec{x}\,') = \vec{a} \times (\vec{x} + \vec{x}\,') = \vec{a} \times \vec{x} + \vec{a} \times \vec{x}\,' = T(\vec{x}) + T(\vec{x}\,')$$

   Zur Berechnung der Matrix der Abbildung bestimmen wir die Bilder der Basisvektoren:

   $$T(\vec{e}_1) = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ a_3 \\ -a_2 \end{pmatrix}$$

   $$T(\vec{e}_2) = \begin{pmatrix} -a_3 \\ 0 \\ a_1 \end{pmatrix}, \qquad T(\vec{e}_3) = \begin{pmatrix} a_2 \\ -a_1 \\ 0 \end{pmatrix}$$

   Matrix der Abbildung (antisymmetrische Matrix):

   $$A = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix}$$

   Resultat:

   $$\vec{a} \times \vec{x} \;=\; A \cdot \vec{x} \qquad \text{für alle } \; \vec{x} \in \mathbb{R}^3 \tag{4.6}$$

4. Orthogonalprojektion auf eine Gerade

   Gegen sei ein Einheitsvektor $\vec{b} \in \mathbb{R}^3$. Wir betrachten die Abbildung, die einen beliebigen Vektor $\vec{x} \in \mathbb{R}^3$ orthogonal auf die Richtung $\vec{b}$ projiziert (Orthogonalzerlegung, siehe Seite 46) :

   $$T : \mathbb{R}^3 \to \mathbb{R}^3 : \vec{x} \mapsto \vec{y} = (\vec{x} \cdot \vec{b})\,\vec{b} \qquad \text{(ohne Nenner wegen } |\vec{b}\,| = 1)$$

   Die Linearitätsbedingungen sind aufgrund der Gesetze des Skalarproduktes erfüllt.

   Zur Berechnung der Matrix der Abbildung betrachten wir die Bilder der Basisvektoren:

   $$T(e_j) = (\vec{e}_j \cdot \vec{b})\,b = b_j\,\vec{b}$$

   Matrix der Abbildung:

   $$B \;=\; \begin{pmatrix} b_1 b_1 & b_1 b_2 & b_1 b_3 \\ b_2 b_1 & b_2 b_2 & b_2 b_3 \\ b_3 b_1 & b_3 b_2 & b_3 b_3 \end{pmatrix} \;=\; \vec{b} \cdot \vec{b}^{\,\tau}$$

   Probe!

4.3 Zusammensetzungen

Abbildungen können zusammengesetzt, d.h. hintereinander ausgeführt werden, wenn die Bilder der ersten Abbildung im Definitionsbereich der zweiten liegen. Dies ist (z.B.) der Fall für lineare Abbildungen von $\mathbb{R}^n$ in sich, d.h. für quadratische Matrizen.

Gegeben seien zwei lineare Abbildungen von $\mathbb{R}^n$ in sich mit $n \times n$ Matrizen A und B. Wir betrachten die assoziierten linearen Abbildungen:

$$A : \mathbb{R}^n \to \mathbb{R}^n \qquad B : \mathbb{R}^n \to \mathbb{R}^n$$

Zusammensetzung der Abbildungen, zuerst A dann B:

1. Abbildung $\quad y = A \cdot x$

2. Abbildung $\quad z = B \cdot y$

Resultat: $\quad z = B \cdot (A \cdot x) = (B \cdot A) \cdot x = C \cdot x$

Dabei wurde das Assoziativgesetz der Matrixmultiplikation verwendet.

Die Zusammensetzung der Abbildungen, zuerst A, dann B, ist die lineare Abbildung mit der Matrix

$$C \;=\; B \cdot A \tag{4.7}$$

Man beachte die Reihenfolge der Faktoren, die erste Abbildung steht *rechts*. Die Reihenfolge ist bedeutend, da das Matrizenprodukt nicht kommutativ ist. Die Gleichung gilt analog auch für nicht quadratische Matrizen, wenn diese miteinander multipliziert werden können.

Merke:

Bei der Zusammensetzung von linearen Abbildungen kommt es wie bei dem Produkt von Matrizen auf die Reihenfolge an.

Beispiel:

Erste Abbildung: $\quad y = \begin{pmatrix} 1 & 3 \\ 2 & 7 \end{pmatrix} \cdot x$

Zweite Abbildung: $\quad z = \begin{pmatrix} 2 & 0 \\ 1 & 1 \end{pmatrix} \cdot y_1$

Resultierende Abbildung:

$$z = \begin{pmatrix} 2 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 3 \\ 2 & 7 \end{pmatrix} \cdot x = \begin{pmatrix} 2 & 6 \\ 3 & 10 \end{pmatrix} \cdot x$$

4.4 Inverse und Umkehrabbildung

Sei A eine invertierbare $n \times n$ Matrix. Dann hat die Gleichung

$$y = A \cdot x \tag{4.8}$$

für jedes $y \in \mathbb{R}^n$ nach 3.8 (Seite 63) die eindeutige Lösung

$$x = A^{-1} \cdot y$$

Dies bedeutet, dass die assoziierte lineare Abbildung T_A umkehrbar ist, und die Umkehrabbildung die Matrix A^{-1} hat.

Umkehrung einer Drehung

Für eine Drehmatrix $R(\varphi)$ ist die Inverse einfach die Drehmatrix zum entgegengesetzten Drehwinkel:

$$R(\varphi)^{-1} \;=\; R(-\varphi)$$

Probe:

$$R(\varphi) \;=\; \begin{pmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{pmatrix}, \quad R(-\varphi) \;=\; \begin{pmatrix} \cos(\varphi) & \sin(\varphi) \\ -\sin(\varphi) & \cos(\varphi) \end{pmatrix}$$

Dabei wurden die Eigenschaften $\cos(-\varphi) = \cos(\varphi)$ und $\sin(-\varphi) = -\sin(\varphi)$ der Cosinus- und Sinus-Funktionen verwendet.
Mit der Formel $\cos^2(\varphi) + \sin^2(\varphi) = 1$ folgt:

$$R(\varphi) \cdot R(-\varphi) \;=\; \mathbb{I}.$$

Dies bedeutet, dass $R(-\varphi)$ die Inverse von $R(\varphi)$ ist. Man beachte, dass die Inverse gerade die *Transponierte* von $R(\varphi)$ ist, was für alle Drehungen gilt (siehe unten).

4.5 Allgemeine Drehung im Raum

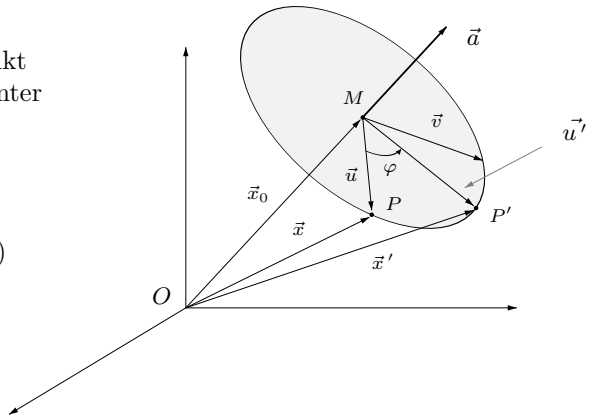
Die Formel von Rodrigues

Wir betrachten eine Drehung im Raum mit beliebigerDrehachse durch den Nullpunkt.

$\vec{a}$ Vektor in Richtung der Drehachse. Dies sei im folgenden immer ein *Einheitsvektor* : $|\vec{a}| = 1$
φ Drehwinkel

Der *Drehsinn* der Drehung sei gemäss der rechten Hand Regel festgelegt: Hält man den Daumen der rechten Hand in Richtung des Vektors $\vec{a}$, so zeigen die Finger für positive φ in die Drehrichtung.

P beliebger Raumpunkt
P' Bildpunkt von P unter der Drehung
$\vec{x}$ Ortsvektor von P
$\vec{x}\,'$ Ortsvektor von P'
E Ebene MPP' (Normalebene zu $\vec{a}$)



Zu Berechnung von $\vec{x}\,'$ zerlegen wir den Ortsvektor $\vec{x}$ in Komponenten parallel und senkrecht zu $\vec{a}$ (Orthogonalzerlegung, S. 46):

$$\vec{x} = \vec{x}_o + \vec{u} \qquad \text{mit} \qquad \vec{x}_o = (\vec{x} \cdot \vec{a})\,\vec{a}\,, \qquad \vec{u} = \vec{x} - \vec{x}_o$$

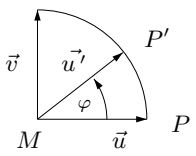
Es folgt:

$$\vec{x}\,' = \vec{x}_o + \vec{u}\,', \tag{1}$$

wobei $\vec{u}\,'$ der in der Ebene E gedrehte Vektor $\vec{u}$ ist. Damit ist das Problem auf eine Drehung in der Ebene E zurückgeführt. Zur Darstellung dieser Drehung führen wir einen Vektor $\vec{v}$ in E ein, der senkrecht auf $\vec{u}$ steht:

$$\vec{v} = \vec{a} \times \vec{u} = \vec{a} \times (\vec{x} - \vec{x}_o) = \vec{a} \times \vec{x} \quad (\text{da } \vec{a} \text{ parallel } \vec{x}_o)$$

Der Vektor $\vec{v}$ hat dieselbe Länge wie $\vec{u}$, da $\vec{a}$ senkrecht auf $\vec{u}$ steht und die Länge 1 hat. Damit folgt aus der nebenstehenden Figur (wie im Einheitskreis):



$$\vec{u}\,' = \cos\varphi \cdot \vec{u} + \sin\varphi \cdot \vec{v}$$

Eingesetzt in Gleichung (1):

$$\begin{aligned}\vec{x}\,' &= \vec{x}_o + \cos\varphi \cdot \vec{u} + \sin\varphi \cdot \vec{v} \\ &= \vec{x}_o + \cos\varphi \cdot (\vec{x} - \vec{x}_o) + \sin\varphi \cdot \vec{v} \\ &= (1 - \cos\varphi) \cdot \vec{x}_o + \cos\varphi \cdot \vec{x} + \sin\varphi \cdot \vec{v}\end{aligned}$$

Resultat :

$$\boxed{\vec{x}\,' \;=\; (1 - \cos\varphi) \cdot (\vec{x} \cdot \vec{a})\;\vec{a} \;+\; \cos\varphi \cdot \vec{x} \;+\; \sin\varphi \cdot \vec{a} \times \vec{x}} \tag{4.9}$$

Dies ist die *Rodrigues Formel* für eine Drehung. Bedingung: $|\vec{a}\,| = 1$ *Rodrigues Formel*

Matrix-Darstellung

Mit den auf Seite 71 eingeführten Matrizen für die Orthogonalprojektion und das Vektorprodukt kann die Rodrigues-Gleichung in Matrix-Form geschrieben werden:

$$x' \;=\; R(\varphi, \vec{a}) \cdot x$$

mit:

$$\boxed{R(\varphi, \vec{a}) \;=\; (1 - \cos\varphi) \cdot P + \cos\varphi \cdot \mathbb{I} + \sin\varphi \cdot A} \tag{4.10}$$

$$P = \begin{pmatrix} a_1a_1 & a_1a_2 & a_1a_3 \\ a_2a_1 & a_2a_2 & a_2a_3 \\ a_3a_1 & a_3a_2 & a_3a_3 \end{pmatrix} \;=\; \vec{a} \cdot \vec{a}^{\,\tau} \qquad A = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix}$$

Resultat:

Mit den Abkürzungen

$$c = \cos(\varphi),\; s = \sin(\varphi)$$

lautet die Drehmatrix $R(\varphi, \vec{a})$:

$$\begin{pmatrix} (1-c)\,a_1a_1 + c & (1-c)\,a_1a_2 - s\,a_3 & (1-c)\,a_1a_3 + s\,a_2 \\ (1-c)\,a_2a_1 + s\,a_3 & (1-c)\,a_2a_2 + c & (1-c)\,a_2a_3 - s\,a_1 \\ (1-c)\,a_3a_1 - s\,a_2 & (1-c)\,a_3a_2 + s\,a_1 & (1-c)\,a_3a_3 + c \end{pmatrix} \tag{4.11}$$

Beispiel:

Seien $\vec{a} = \frac{1}{\sqrt{3}} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ und $\varphi = 30^o$.

$$R(\varphi, \vec{a}) = \begin{pmatrix} 0.9107 & -0.2440 & 0.3333 \\ 0.3333 & 0.9107 & -0.2440 \\ -0.2440 & 0.3333 & 0.9107 \end{pmatrix}$$

Bemerkungen

1. Die Spur einer Drehmatrix

   Unter der *Spur* einer beliebigen quadratischen Matrix A versteht man die Summe der Elemente der Hauptdiagonalen: *Spur*

   $$\mathrm{Spur}(A) = a_{11} + \cdots + a_{nn}$$

   Für eine Drehmatrix erhält man aus der Darstellung 4.11 sofort:

   $$\boxed{\mathrm{Spur}(R) = 1 + 2\cos(\varphi)} \qquad \varphi \quad \text{Drehwinkel} \tag{4.12}$$

   Aus dieser Gleichung kann der Drehwinkel einer numerisch gegebenen Drehmatrix sofort berechnet werden.

2. Die Inverse einer Drehmatrix ist die Transponierte

   Sei R eine Drehmatrix. Dann gilt

   $$\boxed{R^\tau \cdot R = \mathbb{I}}, \qquad \text{d.h.} \qquad \boxed{R^{-1} \;=\; R^\tau} \tag{4.13}$$

   Beweis:

   In den Spalten der Matrix R stehen (wie bei jeder linearen Abildung) die Bilder e'_j der Basisvektoren e_j unter der Drehung. Folglich enthält die Transponierte R^τ in den Zeilen die e'_j. Es folgt, dass das Produkt $R^\tau \cdot R$ aus den Skalarprodukten $e'_i \cdot e'_j$ besteht:

   $$R^\tau \cdot R = (e'_i \cdot e'_j)_{1 \le i,j \le 3}$$

   Dies ist die Einheitsmatrix, weil die gedrehten Basisvektoren e'_i wie die e_i orthonormal sind, d.h. die Skalarprodukte sind gleich 0 für $i \neq j$ und gleich 1 für $i = j$.

4.6 Links- statt Rechtsmultiplikationen

Gewisse Autoren und Systeme (z.B. Microsoft XNA) stellen lineare Abbildungen mit Links- statt Rechtsmultiplikationen dar (pre- statt postmultiply). Die Funktionsgleichung lautet in diesem Fall

$$y = x \cdot A$$

Dabei müssen x und y Zeilen- statt Spaltenvektoren sein. Die Gleichung kann mittels Transposition in unsere Form mit Spaltenvektoren umgeformt werden:

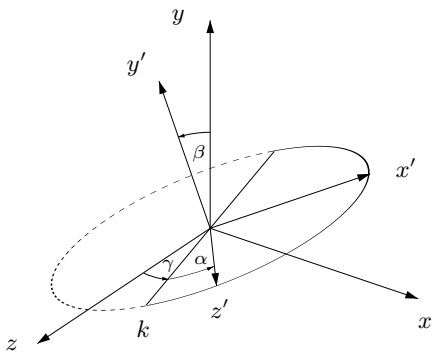
$$y^\tau = A^\tau \cdot x^\tau$$

Dabei wurde das Gesetz $(AB)^\tau = B^\tau A^\tau$ verwendet (siehe Seite 61).

4.7 Die Euler'schen Winkel

Mit den Euler'schen Winkel kann eine beliebige Drehung als Produkt von drei Drehungen um die Koordinatenachsen dargestellt werden.

1. Drehung um die y-Achse, Winkel α
2. Drehung um die z-Achse, Winkel β
3. Drehung um die y-Achse, Winkel γ



Die Schnittgerade k der xz-Ebene mit der $x'z'$-Ebene bestimmt die Winkel α und γ.

Man beachte, dass alle Drehungen um die raumfesten absoluten Achsen erfolgen.

Für die Vorstellung ist es einfacher die Drehungen rückgängig zu machen:

Rückdrehung in die Ausgangslage:

Wegen der allgemeinen Formel $(AB)^{-1} = B^{-1}A^{-1}$ (Seite 63) müssen die Rückdrehungen in umgekehrter Reihenfolge ausgeführt werden:

1. Drehung um die y-Achse, Winkel $-\gamma$
2. Drehung um die z-Achse, Winkel $-\beta$
3. Drehung um die y-Achse, Winkel $-\alpha$

4.8 Affine Abbildungen

Eine *affine Abbildung* in $\mathbb{R}^n$ ist die Zusammensetzung einer linearen Abbildung mit einer anschliessenden Translation.

$$\boxed{y = A_o \cdot x + v} \qquad x, y \in \mathbb{R}^n \tag{4.14}$$

$A_o = (a_{ij})$ $n \times n$ Matrix der linearen Abbildung

$v \in \mathbb{R}^n$ Verschiebungsvektor

Dies sind für $v \neq 0$ *keine* linearen Abbildungen, da der Nullpunkt verschoben wird. Im folgenden beschränken wir uns auf den Fall $\mathbb{R}^3$.

Translationen

Ist A die Einheitsmatrix, so erhält man eine *Translation*:

$$T_v(x) = x + v \qquad x \in \mathbb{R}^3$$

Homogene Koordinaten

Für die Darstellung von affinen Transformationen mit Matrizen führt man *homogene Koordinaten* für Punkte in $\mathbb{R}^3$ ein:

Jedem Punkt $P(x_1, x_2, x_3)$ des dreidimensionalen Raumes wird eine vierte Koordinate zugeordnet, welche fix gleich 1 gesetzt wird:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix} \in \mathbb{R}^4 \qquad \text{homogene Koordinaten}$$

Diese erweiterten Koordinaten eines Punktes nennt man *homogene Koordinaten* des Punktes.

Die Matrix einer Translation

Die Matrix einer Translation in $\mathbb{R}^3$ mit Verschiebungsvektor $v \in \mathbb{R}^3$ ist die 4×4 Matrix:

$$A = \begin{pmatrix} 1 & 0 & 0 & v_1 \\ 0 & 1 & 0 & v_2 \\ 0 & 0 & 1 & v_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Diese wird auf die homogenen Koordinaten x eines Raumpunktes angewandt:

$$y = A \cdot x$$

Die Gleichung bedeutet für die Komponenten:

$$\begin{aligned} y_1 &= x_1 + v_1 \\ y_2 &= x_2 + v_2 \\ y_3 &= x_3 + v_3 \\ y_4 &= 1 \end{aligned}$$

Nimmt man am Schluss wieder die euklidischen Koordinaten von y, so ergibt sich eine Translation in $\mathbb{R}^3$. So kann eine beliebige 4×4 Matrix A als Punktabbildung in $\mathbb{R}^3$ aufgefasst werden: Man geht von den euklidischen Koordinaten zu homogenen Koordinaten über, wendet die Matrix A an und geht zurück zu euklidischen Koordinaten:

$$\begin{array}{ccccccc} \mathbb{R}^3 & \rightarrow & \mathbb{R}^4 & \rightarrow & \mathbb{R}^4 & \rightarrow & \mathbb{R}^3 \\ \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} & \mapsto & x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix} & \mapsto & y = A \cdot x & \mapsto & \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \end{array}$$

Die Matrix einer affinen Abbildung

Die Matrix einer beliebigen affinen Abbildung

$$y = A_o \cdot x + v \tag{4.15}$$

ist die Matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & v_1 \\ a_{21} & a_{22} & a_{23} & v_2 \\ a_{31} & a_{32} & a_{33} & v_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$A_o = (a_{ij})$ 3×3 Matrix der linearen Abbildung
v Verschiebungsvektor in $\mathbb{R}^3$

Mit dieser Matrix ist die Abbildungsgleichung 4.15 äquivalent zur Gleichung

$$\boxed{y \;=\; A \cdot x} \tag{4.16}$$

angewandt auf die homogenen Koordinaten. Probe!

Bemerkung:

Wendet man die Abbildungsgleichung 4.16 auf einen Vektor $n \in \mathbb{R}^4$ mit homogener Komponente 0 (statt 1) an, so entfällt die Translation:

$$A \cdot n \;=\; \begin{pmatrix} a_{11} & a_{12} & a_{13} & v_1 \\ a_{21} & a_{22} & a_{23} & v_2 \\ a_{31} & a_{32} & a_{33} & v_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} n_1 \\ n_2 \\ n_3 \\ 0 \end{pmatrix} \;=\; \begin{pmatrix} A_o \cdot n \\ 0 \end{pmatrix}$$

Aus diesem Grund werden Normalenvektoren mit homogener Koordinate 0 gespeichert, da sie freie Vektoren sind, die *Richtungen* beschreiben.

Zusammensetzungen

Mit der Abbildungsgleichung 4.16 folgt wie bei linearen Abbildungen:

Die Zusammensetzung zweier affinen Abildungen mit Matrizen A und B ist die affine Abbildung mit der Matrix $B \cdot A$, wobei A die Matrix der ersten Abbildung ist.

Beispiel:

Drehung um eine Achse im Raum, die nicht durch den Nullpunkt geht

1. Translation T, die die Achse in den Nullpunkt verschiebt
2. Drehung um die Achse durch den Nullpunkt
3. inverse Translation (entgegengesetzter Verschiebungsvektor)

Matrizen der Standard-Abbildungen

1. Translation

$$T(v_1, v_2, v_3) = \begin{pmatrix} 1 & 0 & 0 & v_1 \\ 0 & 1 & 0 & v_2 \\ 0 & 0 & 1 & v_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2. Skalierung (Streckung)

$$S(q_1, q_2, q_3) = \begin{pmatrix} q_1 & 0 & 0 & 0 \\ 0 & q_2 & 0 & 0 \\ 0 & 0 & q_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3. Drehungen um die Koordinatenachsen

$$R_1(\varphi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$c = \cos(\varphi), \quad s = \sin(\varphi)$

positiver Drehsinn gemäss rechter Handregel

$$R_2(\varphi) = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_3(\varphi) = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4. Drehung um eine beliebige Achse durch den Nullpunkt

   Der Drehwinkel sei φ und die Richtung der Achse sei gegeben durch einen *Einheitsvektor* $\vec{a}$

   $$\vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \qquad |\vec{a}| = 1$$

   Ergänzung der 3×3 Matrix von Seite 75 zu einer 4×4 Matrix:

   $$R(\varphi,\, a_1, a_2, a_3) =$$

   $$= \begin{pmatrix} a_1a_1(1-c)+c & a_1a_2(1-c)-a_3 \cdot s & a_1a_3(1-c)+a_2 \cdot s & 0 \\ a_2a_1(1-c)+a_3 \cdot s & a_2a_2(1-c)+c & a_2a_3(1-c)-a_1 \cdot s & 0 \\ a_3a_1(1-c)-a_2 \cdot s & a_3a_2(1-c)+a_1 \cdot s & a_3a_3(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

   Mit:

   $$c = \cos(\varphi), \quad s = \sin(\varphi) \qquad \varphi: \text{ Drehwinkel}$$

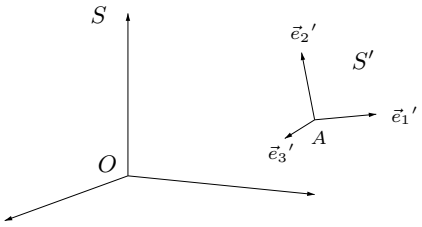
   Positiver Drehsinn: gemäss rechter Handregel

Kapitel 5

Koordinaten-transformationen

5.1 Ortsbasen

Unter einer *Ortsbasis* verstehen wir ein Koordinatensystem (Beispiel Kamera-System) in einer allgemeinen Lage, gegeben durch einen Ursprung A und drei paarweise zueinander senkrechte Einheitsvektoren $\vec{e}_1\,', \vec{e}_2\,', \vec{e}_3\,'$.



$S = (O,\, \vec{e}_1, \vec{e}_2, \vec{e}_3)$ absolutes System

$S' = (A,\, \vec{e}_1\,', \vec{e}_2\,', \vec{e}_3\,')$ Ortsbasis mit Ursprung A

Die Lage-Matrix von S'

Das System S' entsteht aus S durch eine Drehung R und anschliessende Translation in den Punkt $A = (a_1, a_2, a_3)$. Die Matrix M dieser affinen Transformation heisst *Lage-Matrix* des Systems S'.

$$M = \begin{pmatrix} \boxed{R} & \begin{matrix} a_1 \\ a_2 \\ a_3 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{pmatrix} \qquad R\,\vec{e}_j = \vec{e}_j\,' \qquad j = 1, 2, 3$$

Die Drehung R dreht die Basisvektoren $\vec{e}_j$ in die neuen Richtungen $\vec{e}_j\,'$.

Merke:

Die zur Lage-Matrix M gehörige affine Transformation führt das absolute System in die neue Lage S' über.

5.2 Die Transformationsgleichungen

Jeder Raumpunkt P hat Koordinaten x in Bezug auf das absolute System S und x' in Bezug auf S'. Mit der Lage-Matrix M können die Koordinaten umgerechnet werden:

Satz 5.2.1

Für einen beliebigen Raumpunkt P gilt:

$$\boxed{x = M \cdot x'} \quad \text{und} \quad \boxed{x' = M^{-1} \cdot x} \tag{5.1}$$

x Koordinaten von P im System S

x' Koordinaten von P im System S'

Dabei sind x und x' die homogenen Koordinaten als Spaltenvektoren.

Merke:

Die Matrix M berechnet (entgegen den Erwartungen) die *absoluten* Koordinaten x aus den *neuen* x' und M^{-1} die neuen aus den absoluten.

Herleitung:

Sei P ein fester Raumpunkt mit Koordinaten x im absoluten System. Die affine Transformation M bewegt das absolute System in die Lage S'. Für einen *mitbewegten* Beobachter bewegen sich dabei die festen Raumpunkte *entgegengesetzt*, d.h. mit M^{-1}. Der Punkt P erscheint daher für den Beobachter nach der Bewegung im System S' an der Position

$$x' = M^{-1} \cdot x$$

Die erste Gleichung der Behauptung folgt daraus durch Multiplikation beider Seiten mit M.

Beispiele:

1. Reine Translation eines Systems

   Das System S' werde mit einem Verschiebungsvektor $\vec{a}$ aus der Ausgangslage verschoben.

   Der absolute Nullpunkt O erscheint dabei im System S' offensichtlich an der Position $\vec{x}' = -\vec{a}$. Wir überprüfen dies mit den Transformationsgleichungen:

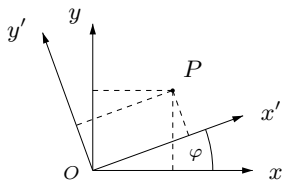
   Die Lage-Matrix M des Systems S' ist die Matrix der Translation mit Verschiebungsvektor $\vec{a}$, also ist M^{-1} die Translationsmatrix zum Vektor $-\vec{a}$. Mit den Gleichungen 5.1 folgt wie erwartet:

$$x' = M^{-1} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & -a_1 \\ 0 & 1 & 0 & -a_2 \\ 0 & 0 & 1 & -a_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -a_1 \\ -a_2 \\ -a_3 \\ 1 \end{pmatrix}$$

2. Drehung in der Ebene

   Mit $M = R(\varphi)$ gilt
   nach Gleichung 5.1 :

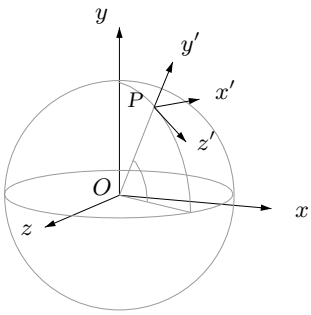
   $$\begin{pmatrix} x' \\ y' \end{pmatrix} = R(-\varphi) \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$



3. Koordinatensystem auf der Erdoberfläche

   Sei P ein Punkt auf der Erdoberfläche, gegeben durch seine Längen- und Breitengrade, z.B. Zürich: $\alpha = 8.55^o$ östliche Länge, $\beta = 47.38^o$ nördliche Breite. Erdradius $r_E = 6370\,km$.

   Ortsbasis in P:



   Das System wird mit einer Translation und zwei Drehungen um die Koordinatenachsen des absoluten Systems aus der Ausgangslage in die gewünschte Lage gebracht:

   (a) Translation T in y-Richtung zum Nordpol

   (b) Drehung R_1 um die x-Achse, Drehwinkel $90^o - \beta$ (auf den Breitengrad β hinunter)

   (c) Drehung R_2 um die y-Achse, Drehwinkel α

   Resultierende Transformation:

   $$M = R_2 \cdot R_1 \cdot T \qquad \text{Lage-Matrix}$$

   Die Inverse kann nach der allgemeinen Formel $(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$ berechnet werden:

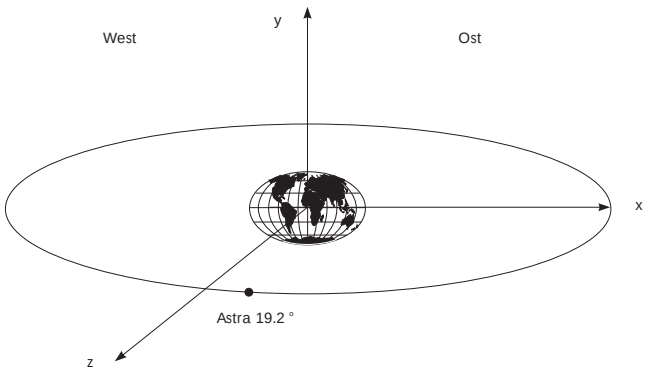
   $$M^{-1} = T^{-1} \cdot R_1^{-1} \cdot R_2^{-1}$$

   Die Inversen der Bewegungen sind einfach die Drehungen mit entgegengesetzten Drehwinkeln, bzw. die Translation mit $-\vec{a}$.

   Koordinatentransformation: $x' = M^{-1} \cdot x$

Anwendung:

Blickrichtung vom Punkt P aus zum Astra-Satelliten



Gesucht ist die Richtung (Azimut, Elevation) der Blickrichtung von P zum Astra-Satelliten im lokalen System im Punkt P.

Lösungsweg:

Die Koordinaten des Satelliten werden im absoluten System berechnet und dann in das lokale System transformiert.

Positionsangaben für Astra im absoluten System (Aequatorsystem):

Höhe über der Erdoberfläche $h = 35680\,km$, d.h. $r = 42050\,km$ vom Erdmittelpunkt entfernt, $\alpha_A = 19.2^o$ östliche Länge.

Resultate der Aufgabe:

Absolute Koordinaten: $x = r\sin(\alpha_A),\ y = 0,\ z = r\cos(\alpha_A)$

(Wegen den Koordinatenkonventionen der Computergraphik sind Sinus und Cosinus vertauscht.)

Lokale Koordinaten (km): $x' = 7771,\ y' = 21613,\ z' = 30410$

Azimut: $\tan(\varphi) = \frac{x'}{z'} \qquad \varphi = 14.24^o$

Elevation: $\tan(\theta) = \frac{y'}{\sqrt{x'^2+z'^2}} \qquad \theta = 34.55^o$

5.3 Bewegungen eines Koordinatensystems

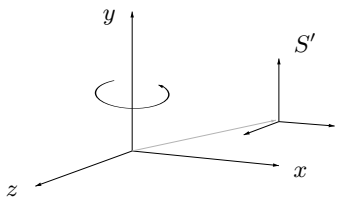
Gegeben sei ein Koordinatensystem S' in einer allgemeinen Lage, gegeben durch die Lage-Matrix M.

Frage:

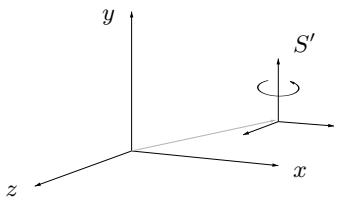
Wie ändert sich die Matrix M, wenn das System mit einer Drehung oder Translation transformiert wird ?

Drehungen

Es sind zwei Fälle zu unterscheiden: Die Drehung kann im absoluten System erfolgen (absolute Drehung), oder relativ zum momentanen System S' (relative Drehung). Im ersten Fall geht die Drehachse durch O, im zweiten Fall durch den Ursprung des Systems S' :



absolute Drehung des Systems S'



relative Drehung des Systems S'

Es gilt:

Sei R eine Drehmatrix. Bei einer *absoluten* Drehung von S' mit R wird M von links mit R multipliziert, bei einer relativen von rechts:

(a) $\boxed{M' = R \cdot M}$ absolute Drehung mit R (5.2)

(b) $\boxed{M' = M \cdot R}$ relative Drehung mit R (5.3)

Beweis:

(a) Die resultierende Lage des Systems entsteht durch die affine Transformation M, gefolgt von der Drehung R. Die Lage-Matrix M' für die resultierende Lage von S' ist also gegeben durch

$$M' = R \cdot M$$

(b) Die relative Drehung R' mit Drehachse durch den Ursprung von S' kann so erreicht werden, dass S' mit M^{-1} in die Ausgangslage S zurücktransformiert wird, dann mit R gedreht und schliesslich wieder mit M transformiert wird, d.h.

$$R' = M \cdot R \cdot M^{-1}$$

Dies ist die Drehung, welche das System S' von der momentanen Lage in die neue Lage S'' weiter dreht. Für die gesuchte Lage-Matrix M' von S'' kommt noch die Transformation M davor, welche S in die Lage S' vor der Drehung bringt:

$$M' \;=\; R' \cdot M \;=\; (M \cdot R \cdot M^{-1}) \cdot M \;=\; M \cdot R$$

Jetzt wird M also von rechts mit R multipliziert □

Die Behauptung (b) kannn auch direkt mit den Tranformationsgleichungen eingesehen werden: Sei S'' das System nach der Drehung, und sei P ein Punkt mit Koordinaten x, x' und x'' in Bezug auf S, S' und S''. Dann gilt:

$x = Mx'$.

Weiter gilt im System S'

$x' = R\,x''$,

weil R die relative Lage von S'' bezüglich S' beschreibt. Zusammen:

$x = Mx' = M \cdot (R\,x'') = (M \cdot R)\,x''$

Die Matrix des resultierenden Systems S'' ist also $M \cdot R$.

Derselbe Sachverhalt gilt auch für Translationen. Bei einer absoluten Translation T werden die Verschiebungsstrecken im absoluten System interpretiert, bei einer relativen im momentanen System S'. Je nachdem wird die Matrix M bei der Translation des Systems von links bzw. von rechts mit T multipliziert.

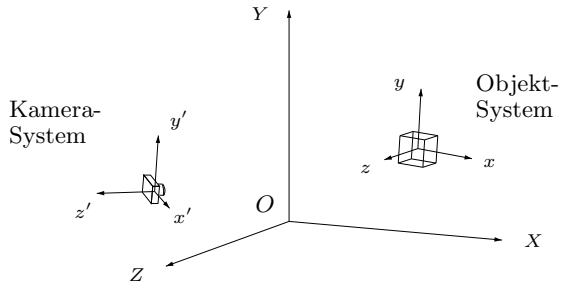
Resultat:

Satz 5.3.1

Eine *Linksmultiplikation* von M mit der Matrix einer Drehung oder Translation entspricht geometrisch einer *absoluten* Drehung bzw. Translation des Systems, eine *Rechtsmultiplikation* einer *relativen.*

5.4 Die ModelView-Transformation

Wir betrachten die oben eingeführte Konfiguration für 3D Darstellungen mit dem Objekt- und dem Kamera-System.



Gesucht ist die ModelView-Transformation, d.h. die Transformation der Koordinaten eines Punktes vom Objekt- in das Kamera-System.

Lage-Matrizen:

M	Lage-Matrix des Objekt-Systems
M'	Lage-Matrix des Kamera-Systems

Koordinaten eines Raumpunktes (homogene Koordinaten):

x	Koordinaten im Objekt-System	Object coordinates
X	absolute Koordinaten	World coordinates
x'	Koordinaten im Kamera-System	Eye coordinates

Koordinaten-Transformationen:

Die ModelView-Transformation besteht aus zwei Schritten:

1. Transformation vom Objekt-System in das absolute System

   $$X = M \cdot x\,, \qquad M \;\; \textit{Model-Matrix}$$ *Model-Matrix*

2. Transformation vom absoluten in das Kamera-System

   Hier kommt die *Inverse* der Lage-Matrix M' zum Einsatz, da die Transformation vom absoluten in das neue System verlangt ist.

   $$x' = (M')^{-1} \cdot X = V \cdot X\,, \qquad V = (M')^{-1} \quad \textit{View-Matrix}$$ *View-Matrix*

   Merke:

   Die View-Matrix ist die Inverse der Lage-Matrix des Kamera-Systems.

Die beiden Transformationen können auch zusammengefasst werden:

$$\boxed{x' = V \cdot M \cdot x} \qquad \text{ModelView-Transformation} \tag{5.4}$$

5.4.1 Berechnung der View-Matrix

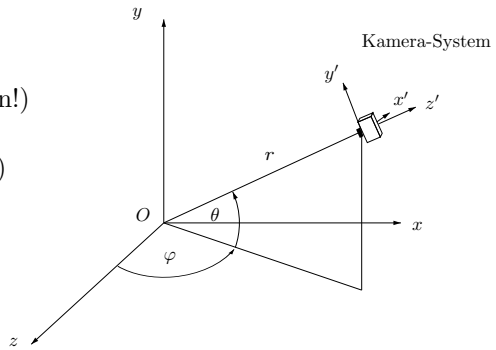
Die View-Matrix V des Kamera-Systems ist die Inverse der Lage-Matrix M, welche das Kamera-System von der Ausgangslage in die gewünschte Lage transformiert.

Azimut-Elevation Positionierung

Eine Kamera befinde sich auf der z-Achse im Abstand r vom Nullpunkt mit Blickrichtung gegen den Nullpunkt O.

Wir stellen uns vor, die Kamera sei auf einem Schwenkarm montiert, der mit zwei Drehungen um die absoluten Koordinatenachsen in die dargestellte Lage gebracht wird:

1. Drehung um die x Achse, Drehwinkel $-\theta$ (nach oben!)
2. Drehung um die (absolute) y-Achse, Drehwinkel φ



r Abstand von O
φ Azimutwinkel
θ Elevationswinkel

Man beachte, dass gemäss der Konvention für den Drehsinn von Drehungen (rechte Hand Regel) für die Drehung nach oben der Winkel $-\theta$ eingesetzt werden muss.

Für die Transformation des Kamera-Systems von der Ausgangslage in die gewünschte Lage kommt noch eine vorausgehende Translation in z-Richtung hinzu. Somit lautet die Lage-Matrix M des Systems:

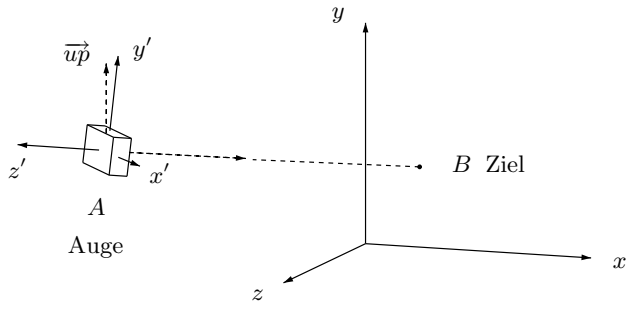
$$M = \mathrm{R}_2(\varphi) \cdot \mathrm{R}_1(-\theta) \cdot \mathrm{T}(0, 0, r)$$

View-Matrix:

$$V = M^{-1} \;=\; \mathrm{T}(0, 0, -r) \cdot \mathrm{R}_1(\theta) \cdot \mathrm{R}_2(-\varphi)$$

LookAt-Positionierung

Ein Kamera-System soll gemäss dem LookAt-Prinzip ausgerichtet werden (vgl. Seite 50).



Basisvektoren des Kamera-Systems (vgl. Seite 50):

$$\begin{aligned} \vec{e}_3{}' &= -\overrightarrow{AB} && \text{negative Blickrichtung} \\ \vec{e}_1{}' &= \overrightarrow{AB} \times \overrightarrow{up} \\ \vec{e}_2{}' &= \vec{e}_3{}' \times \vec{e}_1{}' \end{aligned}$$

Die Lage des Kamera-Systems entsteht aus der Ausgangslage mit den folgenden Transformationen:

1. Drehung R, welche die absoluten Achsen in die neue Lage dreht. R ist die Matrix, welche in den Spalten die normierten Vektoren e'_j enthält.
2. Translation T nach A.

Matrix des resultierenden Systems:

$$\boxed{M = T \cdot R}$$

View-Matrix (LookAt-Matrix)

$$\boxed{V \;=\; M^{-1} \;=\; R^{-1} \cdot T^{-1}}$$

Dabei wurde die allgemeine Formel $(AB)^{-1} = B^{-1}A^{-1}$ verwendet. Die inverse Translation ist die Translation mit entgegengesetztem Verschiebungsvektor und R^{-1} ist gleich R^{τ} (siehe Seite 76).

Numerisches Beispiel:

$A(1,1,1), \quad B(5,4,1), \quad \overrightarrow{up} = \vec{e}_3$

$$M = T \cdot R = \begin{pmatrix} \frac{3}{5} & 0 & -\frac{4}{5} & 1 \\ -\frac{4}{5} & 0 & -\frac{3}{5} & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad V = R^{\tau} \cdot T^{-1} = \begin{pmatrix} \frac{3}{5} & -\frac{4}{5} & 0 & 0.2 \\ 0 & 0 & 1 & -1 \\ -\frac{4}{5} & -\frac{3}{5} & 0 & 1.4 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In den Spalten von R stehen die normierten Basisvektoren $\vec{e_j}{}'$.

5.4.2 Bewegungen des Objekt-Systems

Ein Objekt-System sei gegeben durch seine Lage-Matrix M (Model-Matrix). Nach dem Satz 5.3.1 (Seite 87) wirken sich Drehungen und Translationen des Systems wie folgt auf die Matrix M aus:

Bewegung	Matrix	Wirkung auf M
absolute Drehung	R	$M = R \cdot M$
relative Drehung	R	$M = M \cdot R$
absolute Translation	T	$M = T \cdot M$
relative Translation	T	$M = M \cdot T$

Merke:

Eine Rechtsmultiplikation (postMultiply) bedeutet eine relative Bewegung des Systems, eine Linksmultiplikation eine absolute.

Beispiel:

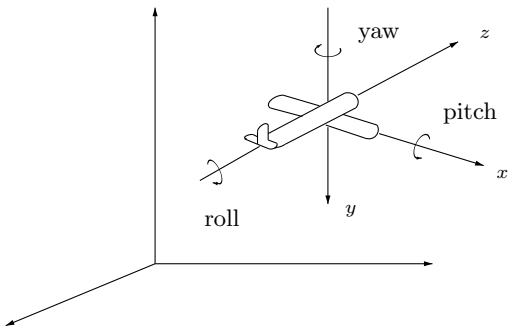
Pitch- Yaw- und Roll-Bewegungen (Aviatik)

Ein System sei fest mit einem Flugzeug verbunden. In der Aviatik wählt man die vertikale Achse (y-Achse) nach unten, sodass die horizontale Achse (x-Achse) in Flugrichtung nach rechts zeigt.

Roll-Drehung:

$M = M \cdot R$ (relative Drehung)

R Drehung um y-Achse



5.4.3 Bewegungen des Kamera-Systems

Sei M die Lage-Matrix des Kamera-Systems. Von Interesse ist jedoch die View-Matrix $V = M^{-1}$. Also müssen die Veränderungen der Lage-Matrix M bei Bewegungen auf die View-Matrix V übertragen werden. Dabei kommt die Formel $(AB)^{-1} = B^{-1} \cdot A^{-1}$ zum Einsatz. Daher kehrt sich einiges um.

Drehungen und Translationen wirken sich wie folgt auf die Matrizen M und V des Kamera-Systems aus. Die Auswirkungen auf M sind gleich wie beim Objekt-System.

Bewegung	Matrix	Wirkung auf M	Wirkung auf V
absolute Drehung	R	$M = R \cdot M$	$V = V \cdot R^{-1}$
relative Drehung	R	$M = M \cdot R$	$V = R^{-1} \cdot V$
absolute Translation	T	$M = T \cdot M$	$V = V \cdot T^{-1}$
relative Translation	T	$M = M \cdot T$	$V = T^{-1} \cdot V$

Rechtsmultiplikationen der View-Matrix bedeuten also *absolute* Bewegungen des Kamera-Systems, Linksmultiplikationen *relative* Bewegungen.

Man beachte, dass R^{-1} einfach die Drehung mit dem entgegengesetzten Drehwinkel und T^{-1} die Translation mit entgegengesetzten Verschiebungsstrecken sind.

Herleitung der Tabelle

Die Spalte für M ist gleich wie beim Objekt-System (gemäss Satz 5.3.1, Seite 87). Die Spalte für V entsteht durch Invertieren, z.B. wird aus der Gleichung

$$M = R \cdot M$$

durch Invertieren beider Seiten:

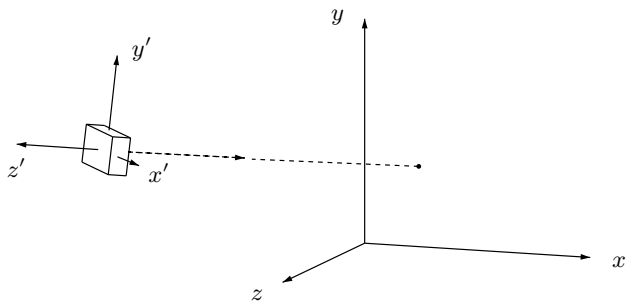
$$M^{-1} = (R \cdot M)^{-1} = M^{-1} \cdot R^{-1}$$

also

$$V = V \cdot R^{-1}$$

Beispiele von Kamerabewegungen

1. Relative Drehungen und Translationen des Kamera-Systems



Drehung um die y'-Achse mit Drehwinkel φ (relative Drehung):

$$V = R_2(-\varphi) \cdot V$$

Translation in z'-Richtung um eine Strecke a:

$$V = T(0, 0, -a) \cdot V$$

$a > 0$ Translation in positive z'-Richtung
$a < 0$ Translation in negative z'-Richtung, d.h. in Blickrichtung

2. Fernrohr

   Eine Kamera sei wie das dargestellte Fernrohr drehbar um zwei Achsen montiert.

   Die Drehung für die Elevation ist eine relative Drehung um die x'-Achse des Kamera-Systems (Linksmultiplikation von V mit R^{-1}).

   Die Drehung um die vertikale Achse ist eine absolute Drehung um die Parallele zur absoluten y-Achse durch das Fernrohr.

   Diese Drehung kann mit den folgenden drei Transformationen realisiert werden:

   1. Translation T, welche die Drehachse so verschiebt, dass sie durch den Nullpunkt O geht
   2. Drehung um die Achse durch O
   3. Rückverschiebung mit T^{-1}

Kapitel 6

Die Projektionsmatrix

6.1 Zentralprojektion

Die *Zentralprojektion* oder *Perspektive* ist eine Abbildung, welche die Punkte des Raumes auf eine Ebene (Bildebene) projiziert. Sie entspricht exakt unserer visuellen Wahrnehmung und ergibt die bekannten perspektiven Effekte: parallele Geraden erscheinen i.a. nicht parallel, sondern schneiden sich in Fluchtpunkten.

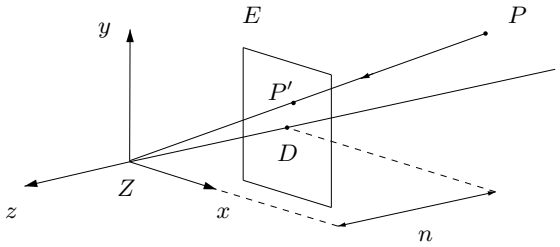
Aufgrund dieser Effekte kann eine Zentralprojektion keine lineare oder affine Abbildung sein, da bei diesen die Bilder von parallelen Geraden parallel sind.

Definition der Zentralprojektion

Eine Zentralprojektion ist mathematisch gegeben durch eine beliebige Ebene E im Raum und einen Punkt Z, der nicht in E liegt.

Die Ebene E ist die *Bildebene* (Bildschirm) und Z ist das *Zentrum* der Projektion (Auge).

Das Bild eines Raumpunktes P ist der Durchstosspunkt P' des Strahls ZP mit der Ebene E. Da in OpenGL die Projektion auf die Bildebene im Kamera-System erfolgt, benötigen wir nur die folgende spezielle Konfiguration mit Zentrum im Nullpunkt (Kamera) und einer Bildebene parallel zur xy-Ebene, welche die z-Achse in einem Punkt mit negativer z-Koordinate schneidet.



Z	Zentrum der Projektion (Auge)
P	beliebiger Punkt
P'	Bildpunkt
$n > 0$	Abstand der Bildebene von O (n = near)
$D(0, 0, -n)$	Durchstosspunkt der z-Achse mit E

Der Abstand n ist bei der Zentralprojektion immer *positiv*. Aus dem folgenden Fenstermodell ist ersichtlich, dass die Zentralprojektion wahrheitsgetreue Bilder in der Bildebene ergibt.

Das Fenstermodell

Wir betrachten einen Gegenstand, z.B. einen Würfel, durch ein Fenster (virtual Screen). Das Auge sieht die Lichtstrahlen, die von den Punkten des Körpers durch das Fenster zum Auge gelangen. Wir malen in Gedanken für jeden Punkt P des Körpers den Durchstosspunkt des Lichtstrahls mit dem Fenster in der Farbe des Punktes P. Der gemalte Durchstosspunkt ist nichts anderes als der Bildpunkt von P unter der Zentralprojektion mit dem Auge als Zentrum und dem Fenster als Bildebene.

So entsteht ein wahrheitsgetreues Bild des Gegenstandes auf dem Fenster (Bildschirm). Der Gegenstand kann entfernt werden, ohne Änderung für den Beobachter.

Dabei ist zu beachten, dass bei der Betrachtung des Bildes auf dem Bildschirm der Abstand der Augen zum Bild so gewählt werden muss, dass die gleichen Betrachtungswinkel zu den Bildpunkten wie bei der Zentralprojektion von O aus entstehen. Sonst treten unnatürliche Verzerrungen auf (Kugeln erscheinen als zu stark verzerrte Ellipsoide usw.).

Die Abbildungsgleichungen

Für unsere spezielle Konfiguration sind die Abbildungsgleichungen der Zentralprojektion sehr einfach.

Sei $P(x, y, z)$ ein beliebiger Raumpunkt.

Ansatz für den Bildpunkt $P'(x', y', z')$: $\overrightarrow{OP'} = t \cdot \overrightarrow{OP}$,

d.h.

$$\begin{aligned} x' &= t \cdot x \\ y' &= t \cdot y \\ z' &= t \cdot z \end{aligned}$$

Da P' in E liegen muss, folgt $z' = -n$, d.h. $t = -\frac{n}{z}$.

Damit:

$$\begin{aligned} x' &= -n \cdot \frac{x}{z} \\ y' &= -n \cdot \frac{y}{z} \\ z' &= -n \end{aligned} \tag{6.1}$$

Wegen dem Nenner z sind dies *nichtlineare* Gleichungen, die infolgedessen *nicht* mit einem Matrixprodukt dargestellt werden können

Man umgeht dieses Problem so, dass man die Division durch z in einem nachträglichen Schritt ausführt, der sogenannten *perspektiven Division*. Ohne den Nenner z sind die Gleichungen linear.

perspektive Division

Matrix-Darstellung

Sei P die Matrix

$$P = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Die Gleichung

$$x' = P \cdot x$$

angewandt auf die homogenen Koordinaten x eines Punktes P ergibt die Gleichungen:

$$\begin{aligned} x' &= n \cdot x \\ y' &= n \cdot y \\ z' &= n \cdot z \\ w' &= -z \end{aligned}$$

Perspektive Division:

Dividiert man die Komponenten durch die homogene (vierte) Komponente, so erhält man die homogenen Koordinaten des gewünschten Bildpunkt P' gemäss den Gleichungen 6.1.

Die perspektive Division kann als Erweiterung des Konzeptes der homogenen Koordinaten aufgefasst werden: die Raumkoordinaten eines Punktes sind die ersten drei Komponenten der homogenen Koordinaten, dividiert durch die vierte Komponente. Wenn diese 1 ist, erübrigt sich die Division.

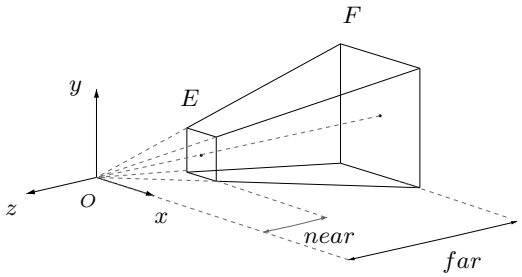
Das ViewingVolume

Die vollständige Projektionsmatrix enthält noch zusätzliche Terme für eine Transformation eines ViewingVolumes in den Standardwürfel mit Eckpunkt-Koordinaten ± 1. Dies ermöglicht ein einfaches Zuschneiden der Figuren auf den Sichtbereich (Clipping).

Das *ViewingVolume* der Zentralprojektion entspricht dem tatsächlichen Sichtbereich gemäss dem obigen Fenstermodell. Es besteht aus allen Punkten des Raumes, deren Projektionsstrahlen das Fenster durchstossen. Dies ist eine abgeschnitte Pyramide.

Aus technischen Gründen wird noch eine *far clipping plane* parallel zur Bildebene eingeführt, damit der z-Bereich endlich ist.

Damit ist das Viewing-Volume der Zentralprojektion ein Pyramidenstumpf (Frustum):



E near clipping Plane (Bildebene)
F far clipping Plane

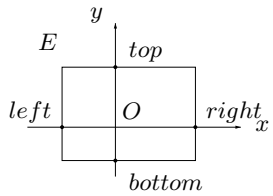
Festlegung des ViewingVolumes

- $near$, far Abstände der Ebenen E und F von O (Kamera). Sie müssen beide *positiv* sein.

  Die Entfernung der Ebene F beeinflusst nur den Clipping-Bereich, sie hat keinen Einfluss auf die Form und Grösse des Bildes, da die Zentralprojektion durch O und E bestimmt ist.

  Die Ebenen E und F sind parallel zur xy-Ebene und durchstossen die z-Achse in den Punkten $(0, 0, -near)$ bzw. $(0, 0, -far)$.

- Das Rechteck in der Ebene E wird durch die Grenzen auf der x- und y-Achse festgelegt: $left$, $right$, $bottom$, top



Das Rechteck in E entspricht dem Filmrechteck der Kamera. Es wird mit der Viewport-Transformation auf das Viewport-Rechteck transformiert. Eine Veränderung der Grösse des Rechteckes wirkt folglich wie der Zoom einer Kamera.

Vollständige Projektionsmatrix:

Die Matrix ist durch die Parameter des ViewingVolumes festgelegt.
Abkürzungen: $\ell = left$, $r = right$, $b = bottom$, $t = top$

$$P = \begin{pmatrix} \frac{2n}{r-\ell} & 0 & \frac{r+\ell}{r-\ell} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{6.2}$$

Zugehörige Abbildungsgleichungen:

$$\begin{array}{lcrcl} x' & = & \frac{2n}{r-\ell} \cdot x & + & \frac{r+\ell}{r-\ell} \cdot z \\ y' & = & \frac{2n}{t-b} \cdot y & + & \frac{t+b}{t-b} \cdot z \\ z' & = & -\frac{f+n}{f-n} \cdot z & - & \frac{2fn}{f-n} \\ w' & = & -z & & \end{array}$$

Nach der perspektiven Division:

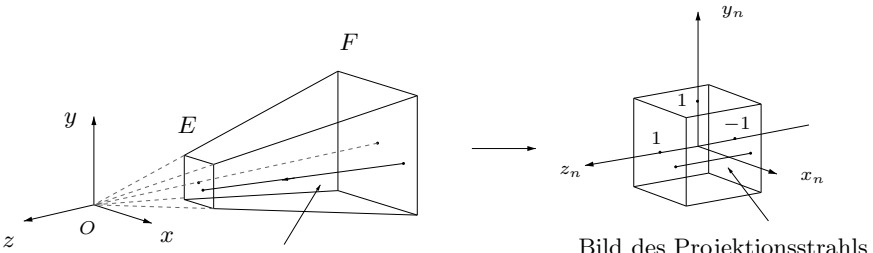
$$\begin{array}{lcrcl} x_n & = & -\frac{2n}{(r-\ell)} \cdot \frac{x}{z} & - & \frac{r+\ell}{r-\ell} \\ y_n & = & -\frac{2n}{(t-b)} \cdot \frac{y}{z} & - & \frac{t+b}{t-b} \\ z_n & = & \frac{f+n}{f-n} & + & \frac{2fn}{(f-n)} \cdot \frac{1}{z} \end{array}$$

Dies sind die *normierten Device-Koordinaten*. Sie liegen für die Punkte des ViewingVolumes im Standardwürfel mit Eckpunkt-Koordinaten ± 1.

normierte Device-Koordinaten

Die Koordinaten (x_n, y_n) eines Punktes entsprechen den oben berechneten Koordinaten der Zentralprojektion des Punktes mit zusätzlicher Transformation in den Standardwürfel. Geometrisch bedeutet dies, dass Sehstrahlen

auf Geraden abgebildet werden, die parallel zur z_n-Achse verlaufen:



Die z_n-Werte sind infolge der perspektiven Division nichtlineare Funktionen der z-Werte. Sie besitzen aber nach wie vor die nötigen Eigenschaften von Tiefeninformationen für den Depth-Test, da die Funktion $f(x) = \frac{1}{x}$ monoton fallend ist:

Wenn z von $-n$ nach $-f$ läuft, so läuft z_n von -1 nach $+1$, d.h. weit entfernte Punkte haben grosse z-Werte.

Bemerkung:

Die Koordinaten (x, y', z', w') *vor* der perspektiven Division werden für das Clipping verwendet. Sie heissen *clip coordinates*. Das Clipping muss *vor* der perspektiven Division erfolgen, damit Punkte mit $z = 0$ zur Vermeidung einer Nulldivision ausgeschieden werden. Diese Punkte liegen wegen der Bedingung $near > 0$ nicht im ViewingVolume.

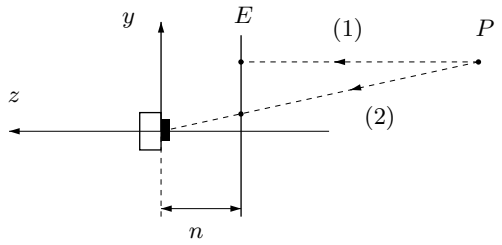
clip coordinates

Java Funktion für die Berechnung der Projektionsmatrix:

```
static Mat4 perspective(left, right, bottom, top, near, far)
```

6.2 Die Normalprojektion

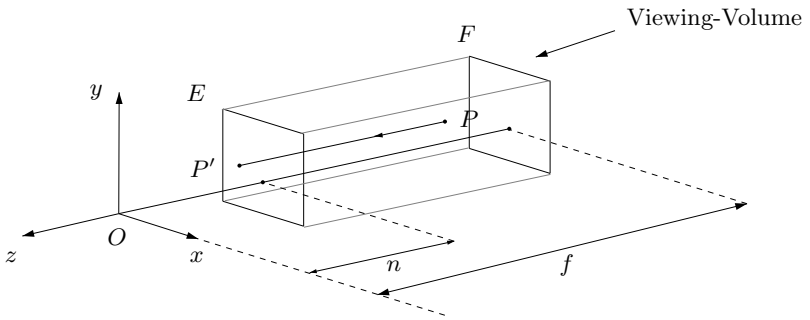
Für weit entfernte Punkte im Bereich in der Blickrichtung fallen die Projektionsstrahlen der Zentralprojektion näherungsweise senkrecht auf die Bildebene. Daraus folgt, dass bei grossen Abständen die Normalprojektion als Näherung für die Zentralprojektion verwendet werden kann.



E Bildebene
P Vertex
(1) Normalprojektion
(2) Zentralprojektion
n Abstand der Ebene E von O

Die Normalprojektion ergibt i.a. ebenfalls sehr gute Bilder. Das Viewing-Volume der Orthogonalprojektion ist ein Quader mit achsenparallelen Kanten, begrenzt durch die Grenzen

$$\ell = left,\ r = right,\ b = bottom,\ t = top,\ n = near,\ f = far$$



Bei der Normalprojektion darf n auch negativ sein, sodass der Nullpunkt im ViewingVolume liegt.

Java Funktion für die Berechnung der Projektionsmatrix:

```
static Mat4 ortho(left, right, bottom, top, near, far)
```

(Gleiche Parameterliste wie die Funktion Mat4.perspective.)

6.3 Übersicht Transformationskonzept

Quelle:

OpenGL ES

Common/Common-Lite Profile Specification
Version 1.1.12 (Full Specification)

OpenGL ES ist die OpenGL-Version für Embedded Systems, z.B. für Android Smartphones. WebGL ist weitgehend identisch mit OpenGL ES.

2.10 Coordinate Transformations

Vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how this transformation is controlled.

Figure 2.5 diagrams the sequence of transformations that are applied to vertices.

1. The vertex coordinates that are presented to the GL are termed *object coordinates*.
2. The *model-view* matrix is applied to these coordinates to yield *eye* coordinates.
3. Then another matrix, called the *projection* matrix, is applied to eye coordinates to yield *clip* coordinates. A perspective division is carried out on clip coordinates to yield *normalized device* coordinates.
4. A final *viewport* transformation is applied to convert these coordinates into *window coordinates*.

Object coordinates, eye coordinates, and clip coordinates are four-dimensional, consisting of x, y, z, and w coordinates (in that order). The model-view and projection matrices are thus 4×4.

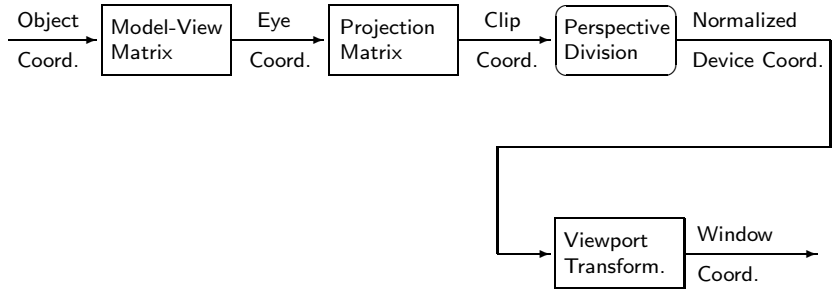
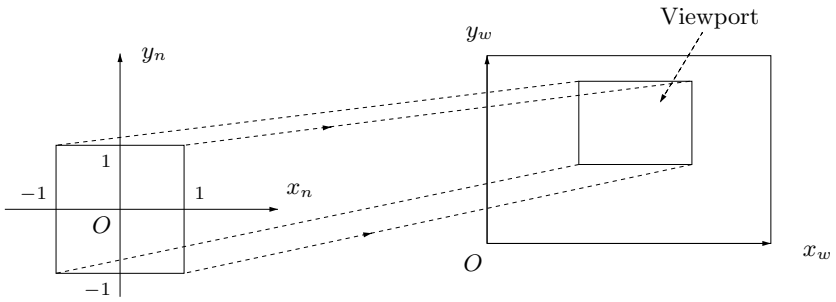


Figure 2.5. Vertex Transformation Sequence

Die Viewport-Transformation

Der Viewport ist das Rechteck auf dem Bildschirm, in welches das Bild rechteckfüllend ausgegeben wird. Dies ist ein Teilrechteck des Windows des Programmes. Der Viewport ist also massgebend für die Grösse des Bildes.

Die Viewport-Transformation transformiert die normierten Device-Koordinaten x_n und y_n entsprechend dem Viewport in Bildschirm-Koordinaten relativ zum Window des Programmes.



Normierte Device-Koordinaten Window-Koordinaten

Das Viewport-Rechteck sei gegeben durch die Window-Koordinaten (x_o, y_o) der linken unteren Ecke und die Breite w und Höhe h des Rechteckes im Window (alle Grössen in Pixel-Einheiten).

Transformationsgleichungen:

$$\begin{aligned} x_w &= x_o + \tfrac{w}{2} \cdot (x_n + 1) \\ y_w &= y_o + \tfrac{h}{2} \cdot (y_n + 1) \end{aligned}$$

(x_n, y_n) Normierte Device-Koordinaten eines Punktes
(x_w, y_w) Window-Koordinaten des Punktes

OpenGL-Methode für die Festlegung des Viewports-Rechteckes:

```
gl.glViewport(x0, y0, w, h);
```

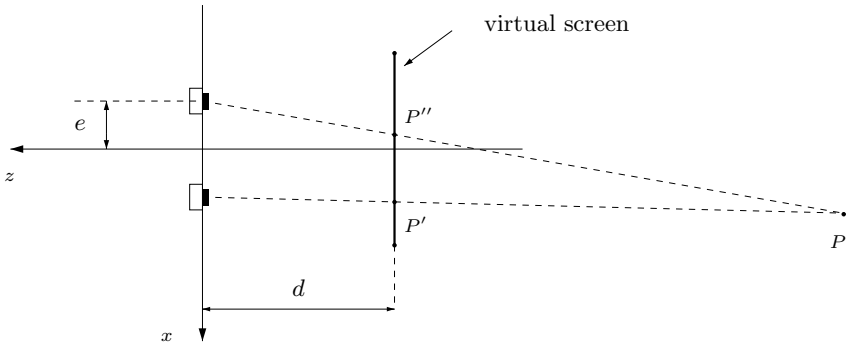
6.4 3D Stereo-Darstellungen

Quelle:

Implementing Stereoscopic 3D in Your Applications, *S.Gateau*, NVIDIA

Unseres räumliches Sehvermögen beruht auf der Fähigkeit unseres Gehirnes, aus den beiden leicht unterschiedlichen Bildern des linken und rechten Auges ein Bild mit Tiefeninformationen zu erzeugen.

3D Stereo Konfiguration (Grundriss):



e halber Abstand der Kameras
d Abstand der virtuellen Leinwand von den Kameras
P beliebiger Raumpunkt

Die virtuelle Leinwand entspricht dem Fenster des Fenstermodelles (S. 95). Ihre Grösse und ihr Abstand von den Kameras werden beliebig gewählt.

Die Differenz der x-Koordinaten der Projektionspunkte P' und P'' eines Punktes P heisst *Parallaxe* des Punktes P :

$$\Delta x = x_{P'} - x_{P''} \qquad \text{Parallaxe}$$

Aus der Parallaxe eines Punktes ist ersichtlich, ob der Punkt vor oder hinter der virtuellen Leinwand liegt:

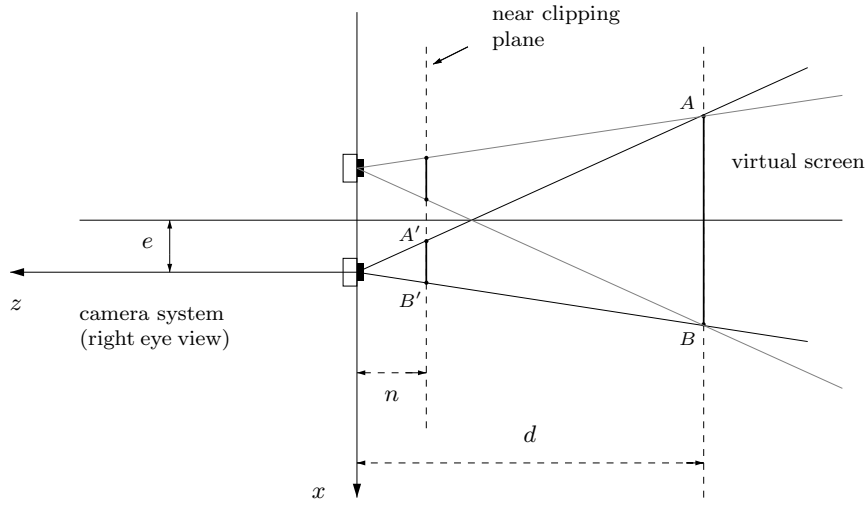
Parallaxe	Lage von P
$\Delta x > 0$	P hinter der Leinwand
$\Delta x = 0$	P auf der Leinwand
$\Delta x < 0$	P vor der Leinwand

Dies ist ersichtlich, wenn man P gegen die Kameras verschiebt.

Viewing-Volumes (Sichtpyramiden)

Die Wahl der Sichtpyramiden für die Zentralprojektion ist wichtig für einen realistischen 3D Effekt. Sie werden folgendermassen gewählt:

- Die near clipping plane wird *vor* der virtuellen Leinwand gewählt, sodass auch Objekte *vor* der Leinwand im Viewing-Volume liegen können.
- Die Grenzen $left$ und $right$ werden (aufgrund des Fenstermodelles) so bestimmt, dass beide Augen denselben Ausschnitt der virtuellen Leinwand sehen.



e	Abstand der Augen von der Symmetrieachse
n (near)	Abstand der near clipping plane
$w = \overline{AB}$	Breite der virtuellen Leinwand

Viewing-Volume für rechte Kamera:

Die Grenzen $left$ und $right$ des Viewing-Volumes sind die x-Koordinaten der Punkte A' und B' im System der Kamera. Sie werden folgendermassen berechnet:

Die Randpunkte A und B der virtuellen Leinwand haben im *zentralen* Kamerasystem die x-Koordinaten $-\frac{w}{2}$ bzw. $\frac{w}{2}$, also im System der rechten Kamera:

$$x_A = -\frac{w}{2} - e, \qquad x_B = \frac{w}{2} - e$$

Die Grenzen $left$ und $right$, d.h. die x-Koordinaten der Punkte A' und B' erhält man mit dem Strahlensatz:

$$x_{A'} : x_A \;=\; n : d, \qquad \boxed{left \;=\; x_{A'} \;=\; \frac{n}{d} \cdot x_A}$$

$$x_{B'} : x_B \;=\; n : d, \qquad \boxed{right \;=\; x_{B'} \;=\; \frac{n}{d} \cdot x_B}$$

Der Parameter far des Viewing-Volumes kann beliebig gewählt werden.

Linke Kamera

Für die linke Kamera ist in den obigen Gleichungen für x_A und x_B die Verschiebung '$-e$' durch '$+e$' zu ersetzen.

Positionierung der Kamera-Systeme

Bei der Erzeugung der Bilder wird das Kamera-System zunächst wie für eine Mono-Kamera in die gewünschte Position gebracht. Dann wird das System für das Bild des rechten Auges relativ in x-Richtung um e verschoben, für das Bild des linken Auges um $-e$.

Kapitel 7

Quaternionen

Quaternionen ermöglichen eine Darstellung von Drehungen in $\mathbb{R}^3$ mit Vektoren anstelle von Matrizen. Dabei werden für eine Drehung anstelle der 9 Elemente der Drehmatrix nur vier Werte in einem Vektor $q \in \mathbb{R}^4$ gespeichert: der Drehwinkel und drei Komponenten der Richtung der Drehachse. Dies ist für diverse Situationen vorteilhaft.

7.1 Defintion der Quaternionen

Ein Quaternion ist ein Element von $\mathbb{R}^4$, d.h. ein Vektor mit vier reellen Komponenten:

$$q = (q_o, q_1, q_2, q_3) \ \in \ \mathbb{R}^4$$

Die Komponente $q_o \in \mathbb{R}$ heisst *Skalaranteil* von q. Die Komponenten q_1 bis q_3 werden zu einem Vektor $\vec{q} \in \mathbb{R}^3$ zusammengefasst, dem *Vektoranteil* des Quaternions. Mit diesem erhält man die äquivalente Darstellung eines Quaternions:

$$q = (q_o, \vec{q}) \qquad \vec{q} = (q_1, q_2, q_3) \in \mathbb{R}^3$$

q_o Skalaranteil
$\vec{q}$ Vektoranteil

Der Skalaranteil wird manchmal auch als letzte Komponente q_4 gespeichert.

Grundoperationen

Die Vektoroperationen und die Norm von $\mathbb{R}^4$ werden unverändert für Quaternionen übernommen.

Quaternionenprodukt

Die entscheidende Neuerung der Quaternionen ist ein *Produkt* von zwei Quaternionen:

$$c = a \cdot b \qquad a, b, c \in \mathbb{R}^4$$

Dieses Produkt erüllt das Assoziativ- und Distributivgesetz und ermglicht eine eindeutige Division durch Quaternionen ungleich 0. Das Kommutativgesetz gilt jedoch nicht.

Mit diesen Operationen bilden die Quaternionen eine sogenannte *Divisionsalgebra*, wie die komplexen und die reellen Zahlen

Divisionsalgebra

Reelle und reine Quaternionen

Ein Quaternion mit Vektoranteil $\vec{0}$ heisst *reelles Quaternion* und eines mit Skalaranteil 0 heisst *reines Quaternion*.

reelles Quaternion: $q = (q_o, \vec{0}), \quad q_o \in \mathbb{R}$
reines Quaternion: $q = (0, \vec{q}), \quad \vec{q} \in \mathbb{R}^3$

So können reelle Zahlen $x \in \mathbb{R}$ und Vektoren $\vec{a}$ von $\mathbb{R}^3$ als Quaternionen aufgefasst werden:

$$\begin{array}{rcl} x \in \mathbb{R} & \leftrightarrow & (x, \vec{0}) \in \mathbb{R}^4 \\ \vec{a} \in \mathbb{R}^3 & \leftrightarrow & (0, \vec{a}) \in \mathbb{R}^4 \end{array}$$

Im folgenden identifizieren wir x und $\vec{a}$ mit den zugehörigen Quaternionen.

Basisdarstellung

Die Basisvektoren von $\mathbb{R}^3$ werden mit $\vec{i}$, $\vec{j}$ und $\vec{k}$ bezeichnet:

$$\vec{i} = (1,0,0) \qquad \vec{j} = (0,1,0) \qquad \vec{k} = (0,0,1)$$

Mit diesen Basisvektoren kann der Vektoranteil $\vec{q}$ eines Quaternions q folgendermassen dargestellt werden:

$$\vec{q} = q_1\,\vec{i} + q_2\,\vec{j} + q_3\,\vec{k}$$

Im folgenden fassen wir q_o als reelles und $\vec{i}$, $\vec{j}$ und $\vec{k}$ als reine Quaternionen auf. Damit folgt die Basisdarstellung von q:

$$q = q_o + q_1\,\vec{i} + q_2\,\vec{j} + q_3\,\vec{k}$$

7.2 Die Multiplikation

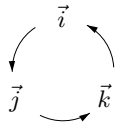
Das Quaternionen-Produkt basiert auf dem Skalar- und Vektor-Produkt von $\mathbb{R}^3$. Zur Unterscheidung der verschiedenen Produkte führen wir die folgenden Bezeichnungen ein:

$\vec{a} \cdot \vec{b}$	Skalarprodukt (Dotproduct) in $\mathbb{R}^n$
$\vec{a} \times \vec{b}$	Vektorprodukt (Crossproduct) in $\mathbb{R}^3$
$p \bullet q$	Quaternionenprodukt gemäss nachfolgenden Definitionen

Das Quaternionenprodukt wird zunächst für die Basisvektoren festgelegt:

$$\begin{array}{lll} \vec{i} \bullet \vec{i} = -1 & \vec{j} \bullet \vec{j} = -1 & \vec{k} \bullet \vec{k} = -1 \\ \vec{i} \bullet \vec{j} = \vec{k} & \vec{j} \bullet \vec{k} = \vec{i} & \vec{k} \bullet \vec{i} = \vec{j} \\ \vec{j} \bullet \vec{i} = -\vec{k} & \vec{k} \bullet \vec{j} = -\vec{i} & \vec{i} \bullet \vec{k} = -\vec{j} \end{array} \tag{7.1}$$

Die gemischten Produkte sind identisch mit den Vektorprodukten der entsprechenden Basisvektoren in $\mathbb{R}^3$ gemäss der nebenstehenden zyklischen Anordnung.



Man kann verifizieren, dass diese Definitionen das Assoziativgesetz $x \bullet (y \bullet z) = (x \bullet y) \bullet z$ erfüllen.

Produkt zweier beliebiger Quaternionen

Durch die Definitionen der obigen Produkte ist das Produkt zweier beliebigen Quaternionen $a = (a_o, a_1, a_2, a_3)$ und $b = (b_o, b_1, b_2, b_3)$ bestimmt, da man im folgenden Ausdruck einfach ausmultiplizieren kann:

$$a \bullet b = (a_o + a_1\,\vec{i} + a_2\,\vec{j} + a_3\,\vec{k}) \bullet (b_o + b_1\,\vec{i} + b_2\,\vec{j} + b_3\,\vec{k})$$

So erhält man die folgenden Definitions-Gleichungen für die Komponenten des Produktes $c = a \bullet b$:

$$\boxed{\begin{array}{lcl} c_o &=& a_o b_o - (a_1 b_1 + a_2 b_2 + a_3 b_3) \\ c_1 &=& a_o b_1 + a_1 b_o + a_2 b_3 - a_3 b_2 \\ c_2 &=& a_o b_2 + a_2 b_o + a_3 b_1 - a_1 b_3 \\ c_3 &=& a_o b_3 + a_3 b_o + a_1 b_2 - a_2 b_1 \end{array}} \tag{7.2}$$

Die Definition 7.2 kann äquivalent mit dem Skalar- und Vektorprodukt geschrieben werden, wie man leicht verifiziert:

$$\boxed{a \bullet b \;=\; (a_o\, b_o - \vec{a} \cdot \vec{b},\; a_o\,\vec{b} + b_o\,\vec{a} + \vec{a} \times \vec{b}\,)} \tag{7.3}$$

Speziell für reine Quaternionen:

$$\vec{a} \bullet \vec{b} = (-\vec{a} \cdot \vec{b},\; \vec{a} \times \vec{b}\,)$$

Das Quaternionenprodukt ist also eine Kombination des Skalar- und Vektorproduktes von $\mathbb{R}^3$.

Beispiel:

$a = (3, 1, 4, 5), \quad b = (2, 3, 1, 2)$

$c_o = a_o b_o - \vec{a} \cdot \vec{b} = -11, \quad \vec{c} = a_o \vec{b} + b_o \vec{a} + \vec{a} \times \vec{b} = (14, 24, 5)$

$c = (c_o, \vec{c}) = (-11, 14, 24, 5)$

Gesetze des Quaternionenproduktes

$a \cdot (b \cdot c)$	$=$	$(a \cdot b) \cdot c$	Assoziativgesetz
$a \cdot (b + c)$	$=$	$a \cdot b + a \cdot c$	Distributivgesetz
$(a + b) \cdot c$	$=$	$a \cdot c + b \cdot c$	

Achtung:

Das Quaternionenprodukt ist *nicht* kommutativ:

$$a \cdot b \neq b \cdot a \qquad \text{i.a.}$$

Wenn einer oder beide Faktoren reelle Quaternionen sind, so ist das Quaternionenprodukt gleich dem normalen Produkt:

$$3 \cdot a = a \cdot 3 = 3\,a$$

7.3 Konjugation und Inversion

Sei $q = (q_o, \vec{q})$ ein Quaternion. Das *konjugierte Quaternion* $\overline{q}$ ist definiert durch

$$\overline{q} = (q_o, -\vec{q}) \qquad \text{konjugiertes Quaternion}$$

Folgerung:

q	reelles Quaternion	$\Longleftrightarrow$	$q = \overline{q}$
q	reines Quaternion	$\Longleftrightarrow$	$q = -\overline{q}$

Gesetze der Konjugation:

1. $$q \cdot \overline{q} \;=\; \overline{q} \cdot q \;=\; |q|^2 \tag{7.4}$$
2. $$\overline{q_1 \cdot q_2} = \overline{q}_2 \cdot \overline{q}_1 \tag{7.5}$$
3. $$|q_1 \cdot q_2| = |q_1|\,|q_2| \tag{7.6}$$
4. $$\vec{a} \cdot \vec{b} = \overline{\vec{b} \cdot \vec{a}} \qquad \text{für reine Quaternionen } \vec{a} \text{ und } \vec{b} \tag{7.7}$$

Inversion

Sei q ein Quaternion verschieden von 0, und sei

$$q_1 = \frac{\overline{q}}{|q|^2}$$

Wegen $q \cdot \overline{q} = |q|^2$ folgt:

$$q \cdot q_1 = q_1 \cdot q = 1$$

D.h. q_1 ist das Inverse von q. Damit gilt:

Jedes Quaternion $q \neq 0$ ist invertierbar mit Inversem

$$\boxed{q^{-1} = \frac{\overline{q}}{|q|^2}} \tag{7.8}$$

Beispiel:

$q = (3, 5, 2, 1)$ $\qquad q^{-1} = (3, -5, -2, -1) \,/\, 39$

Für ein Einheitsquaternion entfällt der Nenner in 7.8 :

$$\boxed{q^{-1} = \overline{q}} \qquad \text{falls} \quad |q| = 1 \tag{7.9}$$

Mit den Inversen können *Divisionen* ausgeführt werden. Man sagt die Quaternionen bilden eine *Divisionsalgebra*. Divisionsalgebra

7.4 Quaternionen und Drehungen

Das Quaternion einer Drehung

Sei $R(\varphi, \vec{n})$ die Drehung in $\mathbb{R}^3$ mit Drehwinkel φ und Drehachse durch O in Richtung $\vec{n}$. Im folgenden sei dabei immer vorausgesetzt, dass $\vec{n}$ normiert ist: $|\vec{n}\,| = 1$.

Das Quaternion zu einer Drehung $R = R(\varphi, \vec{n})$ ist definiert durch:

$$\boxed{q = (\cos(\tfrac{\varphi}{2}),\ \sin(\tfrac{\varphi}{2}) \cdot \vec{n}\,)} \tag{7.10}$$

d.h. $\quad q_o = \cos(\frac{\varphi}{2}), \quad q_1 = \sin(\frac{\varphi}{2}) \cdot n_1, \quad q_2 = \sin(\frac{\varphi}{2}) \cdot n_2, \quad q_3 = \sin(\frac{\varphi}{2}) \cdot n_3$

Diese Definition ist so gemacht, dass die Drehung algebraisch mit dem Quaternionenprodukt dargestellt werden kann, als Alternative zur Matrix-Multiplikation (Gleichung 7.12, Seite 112).

Folgerungen

1. Das Quaternion q einer Drehung ist ein *Einheitsquaternion*, d.h. es gilt *Einheitsquaternion*

   $$|q| = 1$$

   Dabei ist $||$ die Norm von $\mathbb{R}^4$. Dies folgt sofort mit der Voraussetzung, dass $|\vec{n}| = 1$ ist:

   $$|q|^2 = q_o^2 + q_1^2 + q_2^2 + q_3^2 \;=\; \cos^2(\frac{\varphi}{2}) + \sin^2(\frac{\varphi}{2}) \cdot |\vec{n}\,|^2 \;=\; 1$$

   Dabei wurde die Pythagoras-Formel für Cosinus und Sinus verwendet.

2. Eine Drehung $R(\varphi, \vec{n})$ ist geometrisch äquivalent zu $R(-\varphi, -\vec{n})$. Die zugehörigen Quaternion sind identisch (wegen $\cos(-x) = \cos(x)$ und $\sin(-x) = -\sin(x)$).

3. Jedes Einheitsquaternion stellt eine Drehung dar.

   Sei $q = (q_o, \vec{q})$ ein Einheitsquaternion und sei zunächst $\vec{q} \neq 0$.

   Wir setzen:

   $$\boxed{\begin{array}{lcll} \varphi & = & 2\,\arccos(q_o) & 0 \leq \varphi \leq 360^o \\ \vec{n} & = & \dfrac{\vec{q}}{|\vec{q}|} & |\vec{\mathrm{q}}| = \sqrt{\mathrm{q}_1^2 + \mathrm{q}_2^2 + \mathrm{q}_3^2} \end{array}} \qquad (7.11)$$

   Behauptung: q ist das Quaternion zur Drehung $R(\varphi, \vec{n})$

   Nachweis der Gleichung 7.10:

   Nach Definition von φ gilt $q_o = \cos(\frac{\varphi}{2})$. Weiter, weil q nach Voraussetzung ein Einheitsquaternion ist:

   $$q_o^2 + |\vec{q}|^2 = 1$$

   $$|\vec{q}| = \sqrt{1 - q_o^2} = \sqrt{1 - \cos^2(\tfrac{\varphi}{2})} = \sin(\tfrac{\varphi}{2})$$

   Dabei wurde verwendet, dass $0^o \leq \frac{\varphi}{2} \leq 180^o$ und damit $\sin(\frac{\varphi}{2}) \geq 0$ ist.
   Es folgt:

   $$\vec{q} = |\vec{q}| \cdot \vec{n} = \sin(\tfrac{\varphi}{2}) \cdot \vec{n}$$

   Es folgt, dass q die gewünschte Darstellung 7.10 hat. □

   Damit ist jedem Einheitsquaternion mit $\vec{q} \neq 0$ eine Drehung $R(\varphi, \vec{n})$ zugeordnet.

   Spezialfall $\vec{q} = \vec{0}$:

   Ist q ein Einheitsquaternion mit $\vec{q} = \vec{0}$, so ist $q = (1, \vec{0})$ oder $q = (-1, \vec{0})$. In diesem Fall ist die gesuchte Drehung die Identität ($\varphi = 0^o$ bzw. $\varphi = 360^o$ und $\vec{n}$ beliebig, da $\sin(\frac{\varphi}{2}) = 0$ ist).

Beispiele:

1. Drehung um die x_1-Achse mit Drehwinkel 60^o:

$$\begin{aligned} q_o &= \cos(30^o) = \tfrac{\sqrt{3}}{2} \\ \vec{q} &= \sin(30^o) \cdot (1,0,0)) = \tfrac{1}{2} \cdot (1,0,0) \\ q &= (\tfrac{\sqrt{3}}{2}, \tfrac{1}{2}, 0, 0) \end{aligned}$$

2. Drehung mit Drehwinkel 0, d.h. $R = \mathbb{I}$ (Identität) :

$$q = (1,0,0,0)$$

3. Gegeben sei das Einheitsquaternion $q = (-0.911, 0.2, 0.2, 0.3)$
   Zugehörige Drehung:

$$\begin{aligned} \varphi &= 2 \cdot \arccos(q_o) = 311.29^o \\ \vec{n} &= \tfrac{\vec{q}}{|\vec{q}|} = (0.4851, 0.4851, 0.7276) \end{aligned}$$

Darstellung einer Drehung mit dem Quaternionenprodukt

Satz 7.4.1

Sei $R = R(\varphi, \vec{n})$ eine Drehung mit zugehörigem Quaternion q. Dann ist für jedes reine Quaternion $\vec{x} \in \mathbb{R}^3$ auch $q \bullet \vec{x} \bullet \overline{q}$ ein reines Quaternion, und es gilt:

$$\boxed{R\,\vec{x} \;=\; q \bullet \vec{x} \bullet \overline{q}} \tag{7.12}$$

Das Bild eines Vektors $\vec{x} \in \mathbb{R}^3$ unter der Drehung kann also mit dem Quaternionenprodukt berechnet werden.

Beweis:

Sei $x = (0, \vec{x})$ ein reines Quaternion, und sei

$$x' = q \bullet x \bullet \overline{q}$$

a) x' ist ein reines Quaternion:

$$\overline{x'} = \overline{q \bullet x \bullet \overline{q}} = \overline{\overline{q}} \bullet \overline{x} \bullet \overline{q} = q \bullet (-x) \bullet \overline{q} = -x'$$

b) Berechnung von $x' = q \bullet x \bullet \overline{q}$

Wir berechnen zuerst das rechtsstehende Produkt $w = x \bullet \overline{q}$:

$$w = x \bullet \overline{q} = (0, \vec{x}) \bullet (q_o, -\vec{q}) = (\vec{x} \cdot \vec{q},\; q_o\,\vec{x} - \vec{x} \times \vec{q})$$

d.h.

$$w_o = \vec{x} \cdot \vec{q}, \qquad \vec{w} = q_o\,\vec{x} + \vec{q} \times \vec{x}$$

Damit ist der Vektor-Anteil von $x' = q \bullet w$:

$$\begin{aligned}\vec{x}\,' &= q_o\,\vec{w} + w_o\,\vec{q} + \vec{q}\times\vec{w} = \\ &= q_o\,(q_o\,\vec{x} + \vec{q}\times\vec{x}) + (\vec{x}\cdot\vec{q})\,\vec{q} + \vec{q}\times(q_o\,\vec{x} + \vec{q}\times\vec{x}) = \\ &= q_o^2\,\vec{x} + 2q_o\,\vec{q}\times\vec{x} + (\vec{x}\cdot\vec{q})\,\vec{q} + \vec{q}\times(\vec{q}\times\vec{x})\end{aligned}$$

Der letzte Term kann mit der allgemeinen Vektoridentität

$$\vec{a}\times(\vec{b}\times\vec{c}) \;=\; (\vec{a}\cdot\vec{c})\,\vec{b} - (\vec{a}\cdot\vec{b})\,\vec{c}$$

(siehe S. 48) umgeformt werden. Damit erhält man

$$\vec{x}\,' = q_o^2\,\vec{x} + 2q_o\,\vec{q}\times\vec{x} + (\vec{x}\cdot\vec{q})\,\vec{q} + (\vec{q}\cdot\vec{x})\,\vec{q} - (\vec{q}\cdot\vec{q})\,\vec{x}$$

und nach Zusammenfassung von Termen:

$$\vec{x}\,' = (q_o^2 - \vec{q}\cdot\vec{q})\,\vec{x} + 2q_o\,\vec{q}\times\vec{x} + 2(\vec{q}\cdot\vec{x})\,\vec{q} \tag{7.13}$$

Nach Voraussetzung ist

$$q = (\cos(\tfrac{\varphi}{2}),\, \sin(\tfrac{\varphi}{2})\,\vec{n})$$

d.h.

$$q_o = c, \quad \vec{q} = s\,\vec{n} \qquad \text{mit} \quad c = \cos(\tfrac{\varphi}{2}),\; c = \sin(\tfrac{\varphi}{2})$$

Eingesetzt in 7.13:

$$\vec{x}\,' = (c^2 - s^2)\,\vec{x} + 2cs\,\vec{n}\times\vec{x} + 2s^2(\vec{n}\cdot\vec{x})\,\vec{n}$$

Nach den Formeln der Trigonometrie für doppelte Winkel gilt:

$$c^2 - s^2 = \cos(\varphi), \quad 2cs = \sin(\varphi), \quad 2s^2 = 1 - \cos(\varphi)$$

Resultat:

$$\vec{x}\,' = \cos(\varphi)\cdot\vec{x} + \sin(\varphi)\cdot\vec{n}\times\vec{x} + (1 - \cos(\varphi))\cdot(\vec{n}\cdot\vec{x})\,\vec{n} \tag{7.14}$$

Dies ist die Formel von Rodrigues für die Drehung $R(\varphi, \vec{n})$ (siehe S. 75). □

Folgerungen

1. Die Matrix $R(\varphi, \vec{n})$ der Drehung eines Einheitsquaternions q kann direkt mit den Komponenten von q dargestellt werden:

$$R = \begin{pmatrix} q_o^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 - q_oq_3) & 2(q_1q_3 + q_oq_2) \\ 2(q_1q_2 + q_oq_3) & q_o^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 - q_oq_1) \\ 2(q_1q_3 - q_oq_2) & 2(q_2q_3 + q_oq_1) & q_o^2 - q_1^2 - q_2^2 + q_3^2 \end{pmatrix} \tag{7.15}$$

   Man beachte, dass dabei keine Cosinus- und Sinus-Terme auftreten.

Herleitung:

Die Spalten der Matrix einer linearen Abbildung sind die Bilder der Basisvektoren. Setzt man die Basisvektoren von $\mathbb{R}^3$ in die Gleichung 7.13 ein, so erhält man sofort die Matrix 7.15. □

Die Diagonalelemente von R können noch etwas vereinfacht werden. Da q ein Einheitsquaternion ist, gilt

$$q_o^2 + q_1^2 + q_2^2 + q_3^2 = 1$$

Setzt man dies in die Diagonalelemente von R ein, so folgt:

$$R = \begin{pmatrix} 2(q_o^2 + q_1^2) - 1 & 2(q_1 q_2 - q_o q_3) & 2(q_1 q_3 + q_o q_2) \\ 2(q_1 q_2 + q_o q_3) & 2(q_o^2 + q_2^2) - 1 & 2(q_2 q_3 - q_o q_1) \\ 2(q_1 q_3 - q_o q_2) & 2(q_2 q_3 + q_o q_1) & 2(q_o^2 + q_3^2) - 1 \end{pmatrix} \tag{7.16}$$

2. Zusammensetzungen von Drehungen

Seien R_1 und R_2 zwei Drehungen in $\mathbb{R}^3$ mit zugehörigen Quaternionen q_1 und q_2. Dann gilt:

Das Quaternion der Zusammensetzung $R_2 \cdot R_1$ ist gleich dem Produkt der Quaternionen $q_2 \bullet q_1$.

Beweis:

$R_2(R_1\,\vec{x}\,) = q_2 \bullet (q_1 \bullet \vec{x} \bullet \overline{q}_1) \bullet \overline{q}_2 = (q_2 \bullet q_1) \bullet \vec{x} \bullet (\overline{q_2 \bullet q_1})$

Beispiel:

R_1: Drehung um x-Achse mit Drehwinkel $\varphi_1 = 60^o$
R_2: Drehung um z-Achse mit Drehwinkel $\varphi_2 = 45^o$

$q_1 = \ (\cos(30^o),\, \sin(30^o)\,\vec{i}\,) \ = \ \cos(30^o) + \sin(30^o)\,\vec{i}$
$q_2 = \ (\cos(22.5^o),\, \sin(22.5^o)\,\vec{k}\,) \ = \ \cos(22.5^o) + \sin(22.5^o)\,\vec{k}$

Zusammensetzung $R_2 R_1$: $\quad q = q_2 \bullet q_1$

Mit den Abkürzungen
$c = \cos(30^o),\ \ s = \sin(30^o),\ \ c' = \cos(22.5^o),\ \ s' = \sin(22.5^o)$
folgt:

$$\begin{aligned} q \ &= \ q_2 \bullet q_1 = (c' + s'\vec{k}\,) \bullet (c + s\,\vec{i}\,) \ = \\ &= \ c'c + c's\,i + s'c\,\vec{k} + s'\vec{j} \ = \ (0.8001,\, 0.4619,\, 0.1913,\, 0.3314) \end{aligned}$$

Parameter der resultierenden Drehung:

$$\varphi = 2 \cdot \arccos(0.8001) = 73.72^o \qquad \vec{n} = \frac{\vec{q}}{|\vec{q}\,|} = \begin{pmatrix} 0.7701 \\ 0.3190 \\ 0.5525 \end{pmatrix}$$

Insbesondere kann so die Drehachse der resultierenden Drehung elegant berechnet werden.

7.5 Interpolation von Drehungen (SLERP)

Ein wichtiges Beispiel für den Einsatz von Quaternionen ist die Interpolation von Drehungen.

Problemstellung

Gegeben seien zwei Drehungen R_o und R_1 in $\mathbb{R}^3$.

Gesucht sind Drehungen $R(t)$ für $0 \leq t \leq 1$, welche R_o stetig in R_1 überführen:

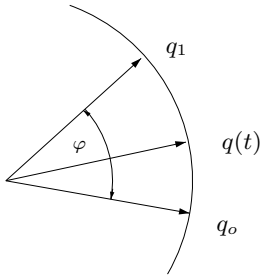
$$R(0) = R_o \qquad \text{und} \qquad R(1) = R_1$$

Dies kommt z.B. zum Einsatz, wenn ein Körper oder ein Kamera-System kontinuierlich von einer Ausgangslage, gegeben durch eine Drehung R_o, in eine Endlage R_1 bewegt werden soll.

Indem man die Drehungen R_o und R_1 durch ihre Einheitsquaternionen q_o und q_1 darstellt, ist das Problem auf die Interpolation von Einheitsquaternionen, d.h. auf Vektoren in $\mathbb{R}^4$ zurückgeführt.

Dabei ist zu beachten, dass die interpolierten Quaternionen $q(t)$ Einheitsquaternionen sein müssen, daher kann die lineare Interpolation (entlang einer Geraden) nicht verwendet werden.

Man verwendet die *spherical linear Interpolation* (SLERP) für Vektoren in $\mathbb{R}^n$. Bei dieser wird der Vektor q_o in der von q_o und q_1 aufgespannten Ebene gedreht. Die Spitze des Vektors bewegt sich dabei auf der Einheitskugel, daher der Name 'spherical'.



Spherical Linear Interpolation in $\mathbb{R}^n$

Seien q_o und q_1 Einheitsvektoren in $\mathbb{R}^n$. Der interpolierte Einheitsvektor $q(t)$ für $0 \leq t \leq 1$ ist definiert durch

$$\boxed{q(t) \;=\; \frac{\sin((1-t)\,\varphi)}{\sin\varphi} \cdot q_o \;+\; \frac{\sin(t\,\varphi)}{\sin\varphi} \cdot q_1} \tag{7.17}$$

Dabei ist φ der Winkel zwischen den Vektoren q_o und q_1 in $\mathbb{R}^n$, d.h.:

$$\varphi = \arccos(q_o \cdot q_1) \qquad (\text{'}\cdot\text{'} = \text{Skalarprodukt in } \mathbb{R}^4)$$

Bedingung:

Es gilt $\sin(\varphi) \neq 0$, d.h. die Vektoren q_o und q_1 sind nicht kollinear.

Die Bedingung schliesst die Fälle $q_o = q_1$ bzw. $q_o = -q_1$ aus.

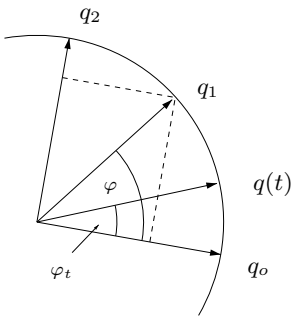
Für den Einsatz zur Interpolation von Drehungen ergibt dies keine Einschränkung, da in beiden Fällen die Drehungen R_o und R_1 identisch sind, sodass $R(t) = R_o$ für jedes t gesetzt werden kann.

Herleitung der Formel 7.17

Seien $q_o, q_1 \in \mathbb{R}^n$ Einheitsvektoren.

$$\varphi = \arccos(q_o \cdot q_1)$$

$$\varphi_t = t \cdot \varphi$$



Wir führen einen Einheitsvektor q_2 senkrecht zu q_o in der von q_o und q_1 aufgespannten Ebene ein (siehe Figur). Mit q_2 lautet der gedrehte Vektor $q(t)$:

$$q(t) = \cos(\varphi_t)\, q_o + \sin(\varphi_t)\, q_2 \tag{7.18}$$

Berechnung von q_2 :

Gemäss Figur gilt:

$$q_1 = \cos(\varphi) \cdot q_o + \sin(\varphi) \cdot q_2,$$

d.h.

$$q_2 = \frac{q_1 - \cos(\varphi) \cdot q_0}{\sin(\varphi)}$$

(Man verifiziert sofort mit dem Skalarprodukt, dass q_2 tatsächlich senkrecht zu q_o ist).

Eingesetzt in 7.18:

$$\begin{aligned} q(t) &= \cos(\varphi_t)\, q_o + \frac{\sin(\varphi_t)}{\sin(\varphi)} (q_1 - \cos(\varphi)\, q_o) \\ &= \frac{\cos(\varphi_t)\sin(\varphi)\, q_o + \sin(\varphi_t)\, q_1 - \sin(\varphi_t)\cos(\varphi)\, q_o}{\sin(\varphi)} \\ &= \frac{\sin(\varphi - \varphi_t)\, q_o + \sin(\varphi_t)\, q_1}{\sin(\varphi)} \end{aligned}$$

Im letzten Schritt wurde das Additionstheorem für die Sinus-Funktion verwendet. □

Direkte Interpolation der Drehungen

Alternativ können auch direkt die Parameter der Drehungen R_o und R_1 interpoliert werden: Der Drehwinkel wird linear interpoliert, und die Drehachse mit SLERP für Vektoren. So werden keine Quaternionen benötigt.

Kapitel 8

Ergänzungen

8.1 Nebel und Dunst

Bei Nebel und Dunst verblassen die Farben der Objekte mit wachsendem Abstand vom Beobachter. Dies kann leicht im Vertex-Shader implementiert werden. Dazu wird die Farbe der Vertices in Abhängigkeit von seiner z-Koordinate einer Hintergrundfarbe (Nebel) angeglichen. Mathematisch ergibt dies eine Interpolation:

$$\boxed{\text{interpolatedColor} = s \cdot \text{computedColor} + (1-s) \cdot \text{fogColor}}$$

computedColor	Vertex-Farbe (vec3) nach Beleuchtungs-Berechnung
fogColor	Hintergrundfarbe (Nebel)
interpolatedColor	interpolierte Vertex-Farbe
s	Interpolationsfaktor

Der Interpolationsfaktor s nimmt mit wachsendem Abstand z des Vertex vom Beobachter von 1 auf 0 ab, z.B. linear oder (besser) mit einer Exponentialfunktion:

$$s = e^{-(d \cdot z)^2} \qquad d \text{ Konstante (Nebeldichte)}$$

Zusatzcode für Fragment-Shader fShader1:

```
float d = 0.015;
vec3 fogColor = vec3(0,0,0.8);

float z = d * fPosition.z;
float s = exp(-z*z);
vec3 computedColor = s*computedColor + (1-s)*fogColor;
```

Dieser Code wird sinnvollerweise nur bei Bedarf ausgeführt, z.B. bei einem speziellen Wert der Variabeln shadingLevel.

8.2 Texturen

Siehe auch

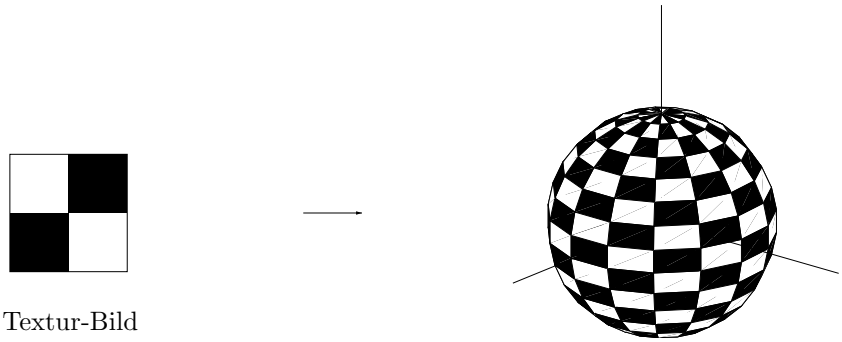
https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial

Texturen sind Oberflächenmuster, die realistischere Darstellungen von Körpern ermöglichen. Technisch gesehen ist eine *Textur* (2D Textur) ein beliebiges digitales Bitmap-Bild (JPG, GIF), welches auf die Oberfläche eines Körpers abgebildet wird.

Ein Bildpunkt $T(i, j)$ einer Textur heisst *Texel* (Texture Pixel) und repräsentiert eine Farbe (RGB). *Texel*

Die Textur kann einfach oder wiederholt als Muster abgebildet werden.

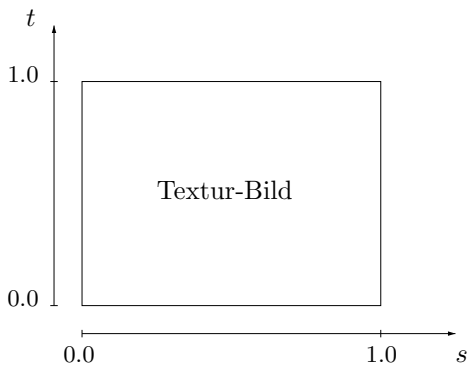
Beispiel: Schachbrett-Muster



Textur-Bild

Textur-Koordinaten

Für die Abbildung der Textur auf eine Fläche werden jedem Vertex der Fläche neben den bisherigen Attributen *Textur-Koordinaten* (s, t) zugeordnet. Dies sind normierte Koordinaten für die Textur gemäss der folgenden Figur.



Diese normierten Koordinaten sind unabhängig von der Auflösung $m \times n$ des Textur-Bildes. Die Texels des Bildes werden gleichmässig als Gitterpunkte im Einheits-Quadrat $[0,1] \times [0,1]$ angeordnet. Die linke untere Ecke des Textur-Bildes hat die Koordinaten $(0.0, 0.0)$, die rechte obere Ecke $(1.0, 1.0)$.

Koordinaten ausserhalb $[0.0, 1.0]$ sind ebenfalls zulässig, ihre Interpretation wird durch den *Texture Wrapping Mode* festgelegt. Defaultmässig wird der ganzzahlige Teil der Koordinaten auf 0 gesetzt, z.B.

$$4.xxxx \rightarrow 0.xxxx$$

was eine periodische Wiederholung der Textur auf jede Seite ergibt.

Schliesslich wird für den Punkt (s,t) das Texel mit dem kleinsten Abstand zum Punkt gewählt. Bei Bedarf kann lineare Interpolation der vier benachbarten Texels spezifiziert werden.

Verwendung von Texturen

Die Texels einer Textur beeinflussen die Farben aller Fragmente (Pixels) der Figur, sie werden daher im *Fragment-Shader* ausgewertet.

Kapitel 9

Referenzen

OpenGL/WebGL

[1] D. Shreiner, et a. “OpenGL Programming Guide”,
Addison Wesley, USA, 2013, Eighth Edition

[2] E. Angel, D. Shreiner, “Interactive ComputerGraphics with WebGL”,
Pearson Education, England, 2015, Seventh Edition

[3] V. Scott Gordon, J. Clevenger, “Computer Graphics Programming in OpenGL with Java”,
Mercury Learning and Information, Dulles, Virginia, 2019, Second Edition

Mathematik

[4] E. Lengyel, “Mathematics for 3D Game Programming and Computer Graphics”,
Course Technology, Boston, 2012

[5] H. JungHyun, “3D Graphics for Game Programming”,
CRC Press, USA, 2011