Graphische Datenverarbeitung

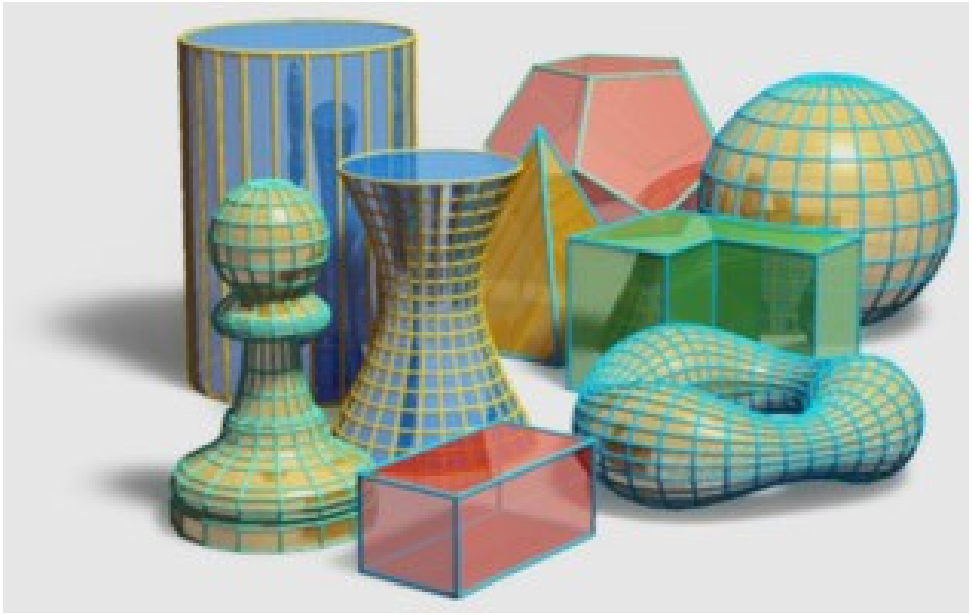
Graphische Objekte und deren Programmierung

Prof. Dr. Elke Hergenröther

h_da

Graphische Objekte und graphische Programmierung

1. Schritt: Erzeugung graphischer Objekte



Alle graphischen Objekte bestehen letztendlich aus Punkten, die mit Kanten zu Dreiecken verbunden wurden.

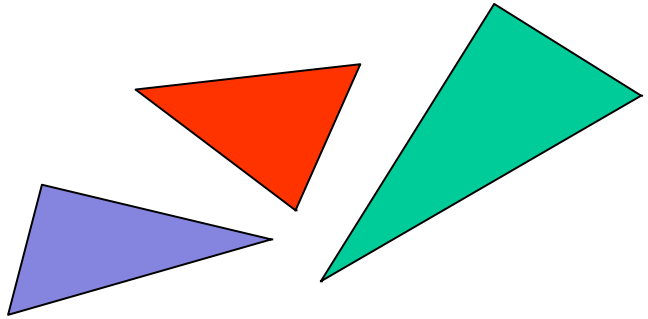
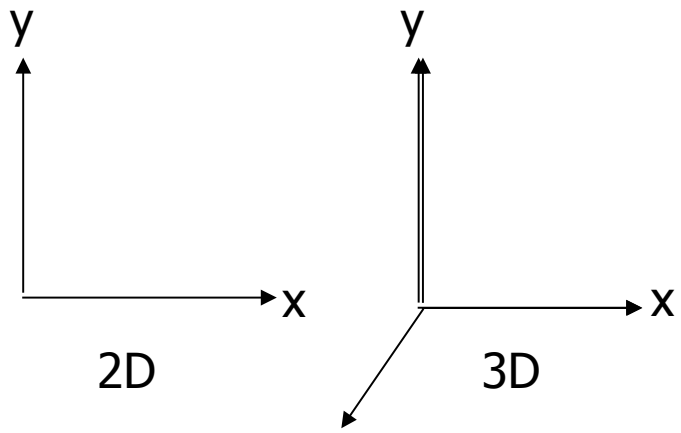
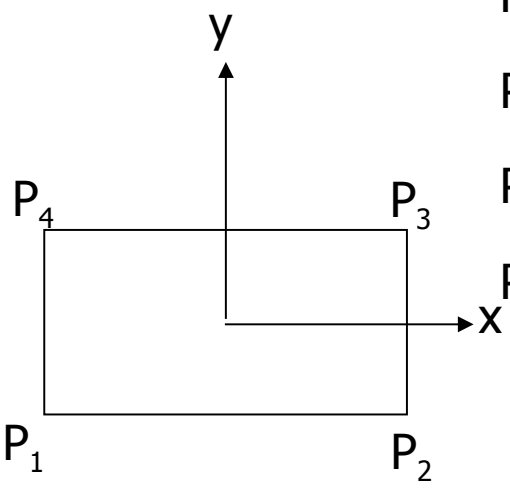


Bild entnommen aus: http://www.pytha.de/produkt/modeler_1.de.php

Jede Geometrie wird durch Punkte, Kanten, Flächen in einem bestimmten Koordinatensystem definiert



Kartesische Koordinatensysteme



$P_1 = \{-2.4, -1.2\}$

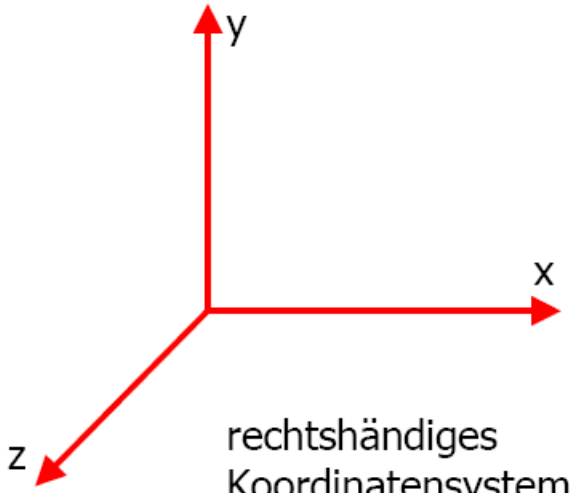
$P_2 = \{ 2.4, -1.2\}$

$P_3 = \{ 2.4, 1.2\}$

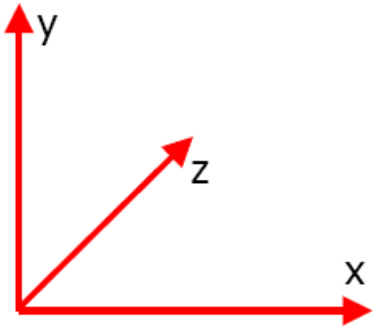
$P_4 = \{-2.4, 1.2\}$

Definition eines Rechtecks in kartesischen Koordinaten

Kartesisches Koordinatensystem

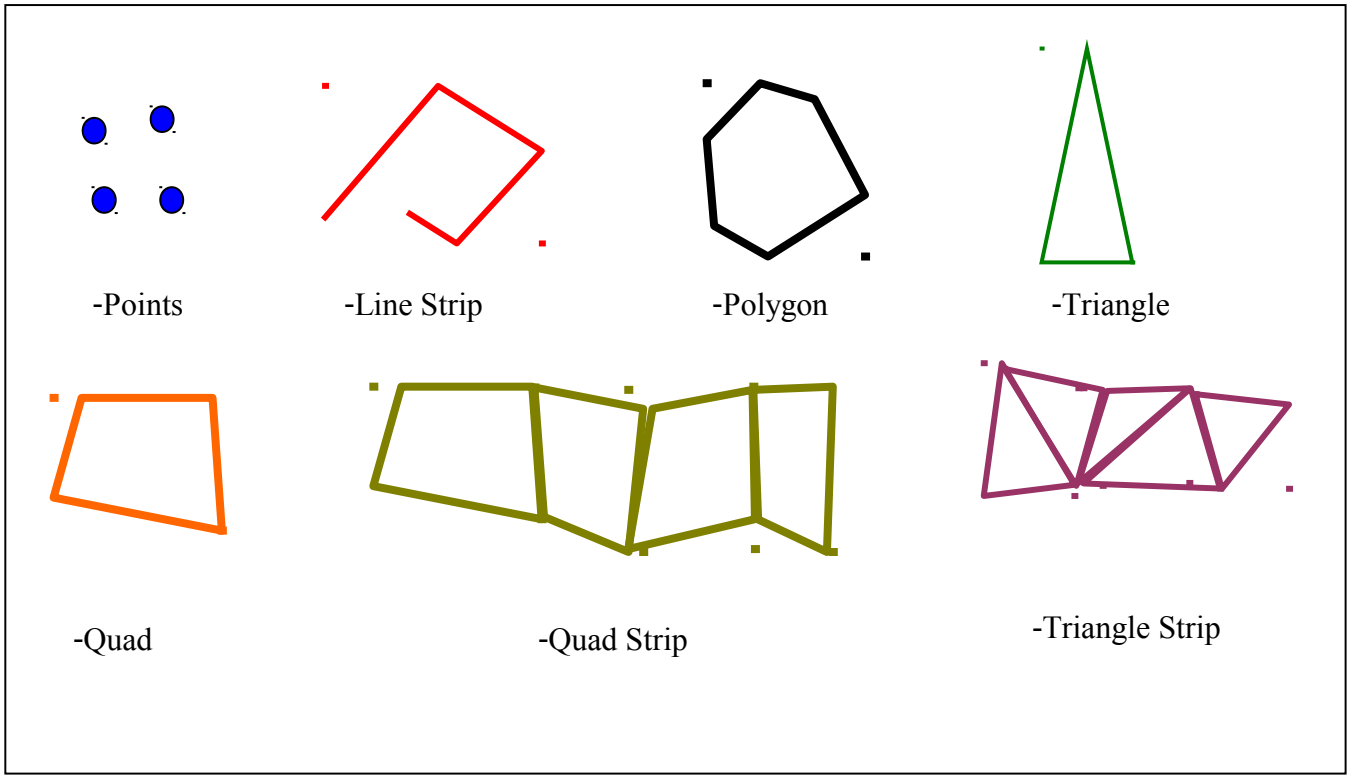


rechtshändiges
Koordinatensystem

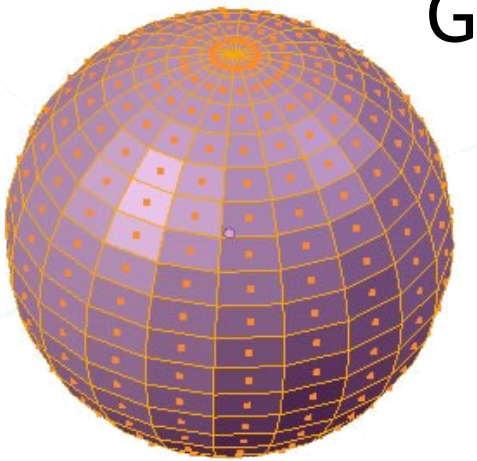


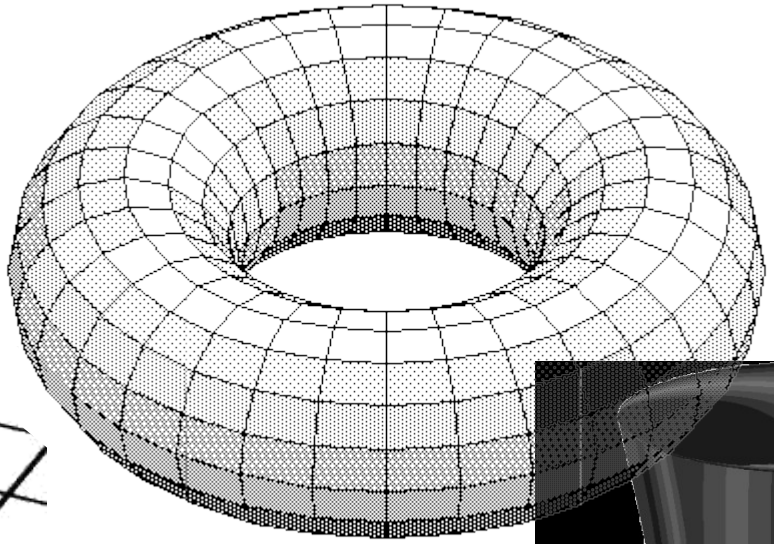
linkshändiges
Koordinatensystem

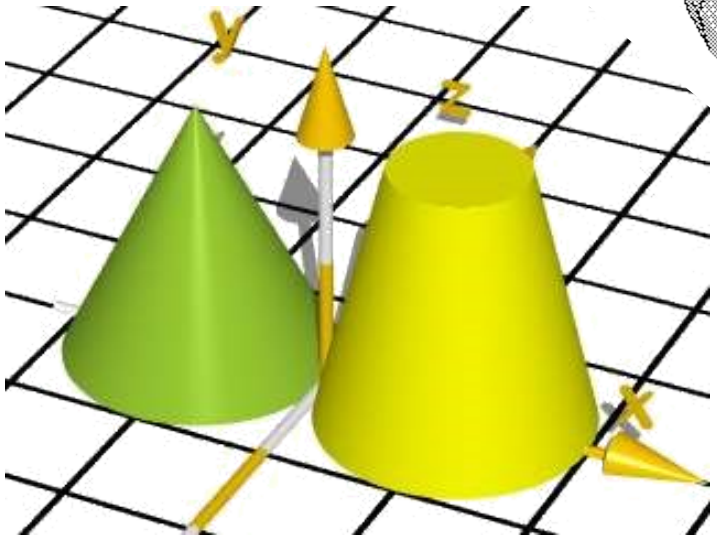
OpenGL verbindet die Punkte zu best. Primitiven

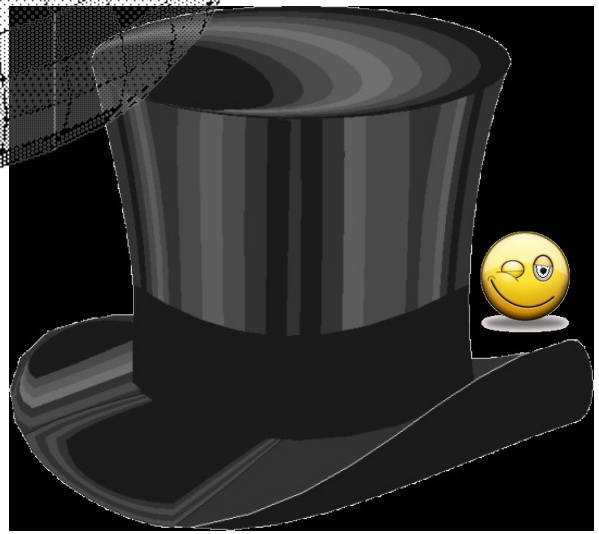


… Hilfs-Bibliotheken stellen komplexere Geometrien zur Verfügung:

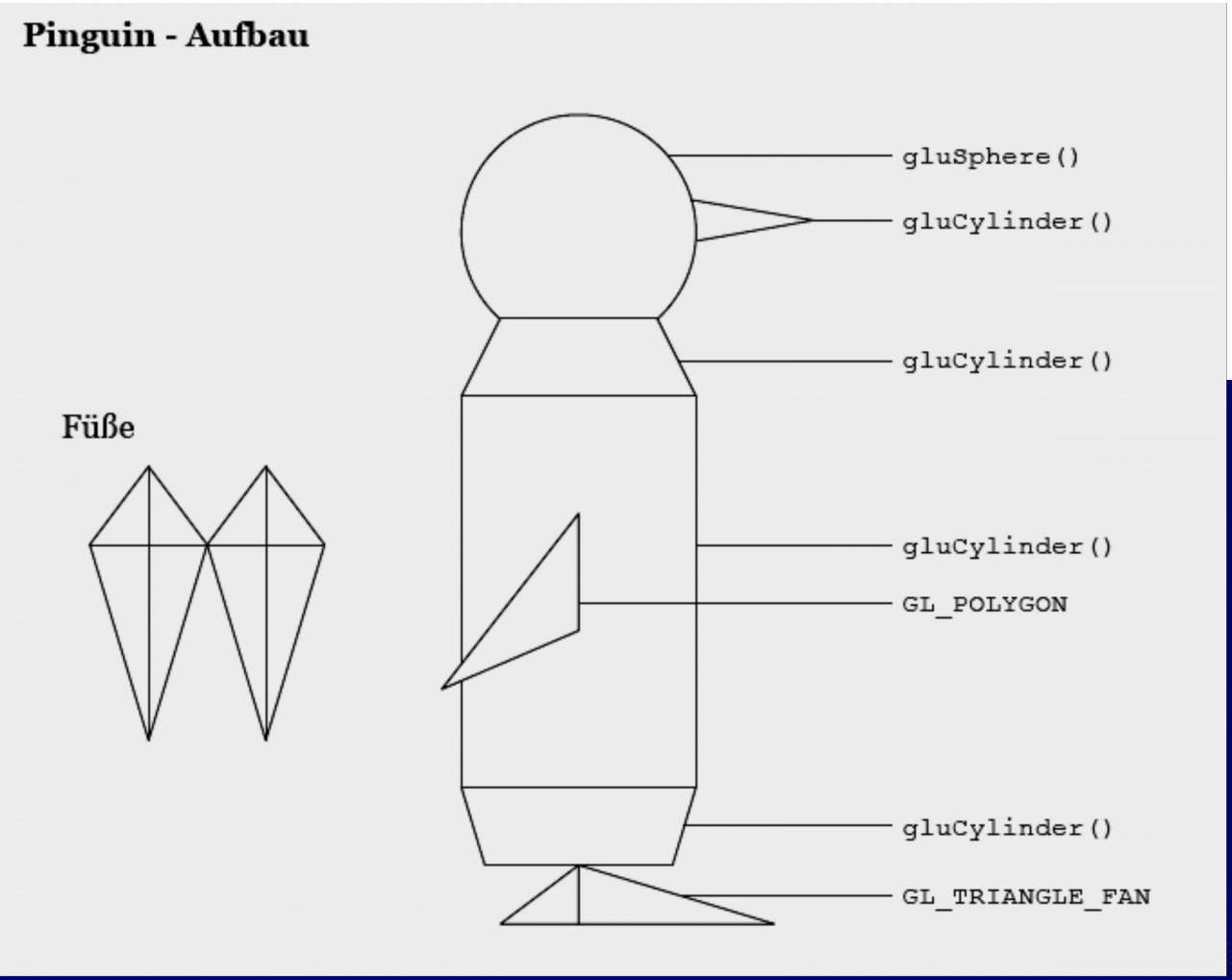


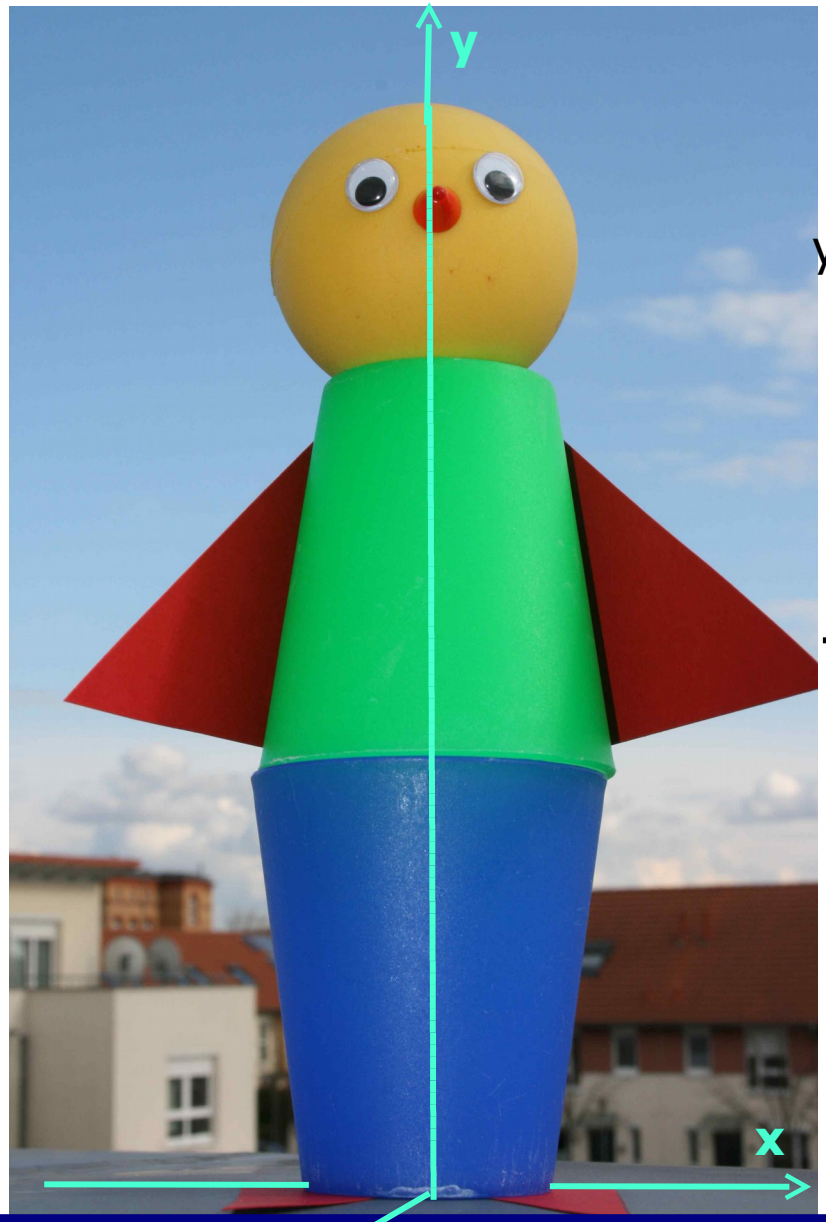




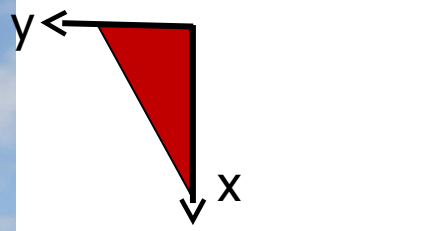


Erzeugen einer Geometrie am Beispiel eines Pinguins
(von Dominik Paulus)

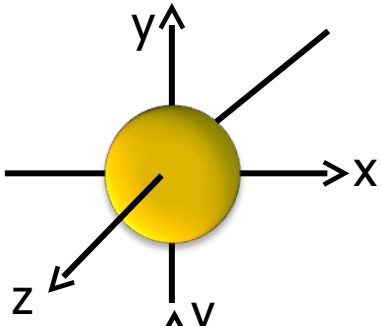




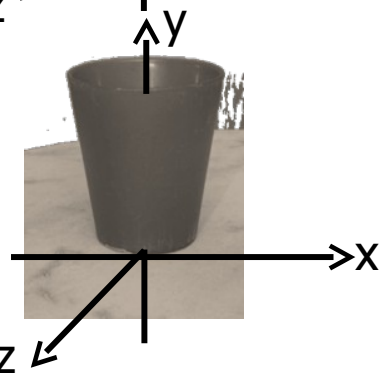
Primitive des Pinguins in ihren lokalen Koordinatensystemen



Dreieck für die Flügel und die Füße im lokalen Koord.



Pinguinkopf im lokalen Koordinatensystem



Becher im lokalen Koordinatensystem

Pinguin im Weltkoordinatensystem

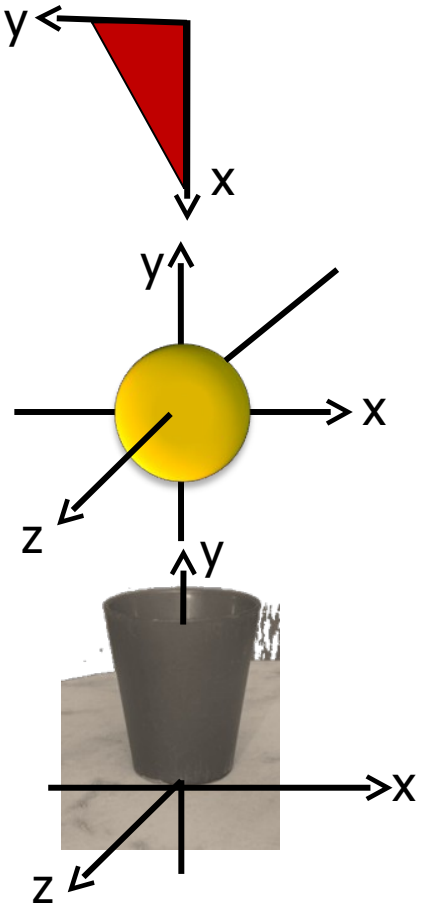
Was fehlt noch zur vollständigen Modellierung des Pinguins?

- Neben den geometrischen Größen müssen noch die graphischen Attribute zugeordnet werden:

  Graphische Attribute beziehen sich auf das „Aussehen“ der Objekte:

- Beispiel für graphische Attribute:
  - Farbe
  - Synthetische Textur auf die Fläche gemappt (=„geklebt“)
  - JPEG-Bild auf Fläche gemappt
  - …

Wie kann man die Primitive im lokalen Koordinatensystem „manipulieren“, um einen Pinguin zu konstruieren?

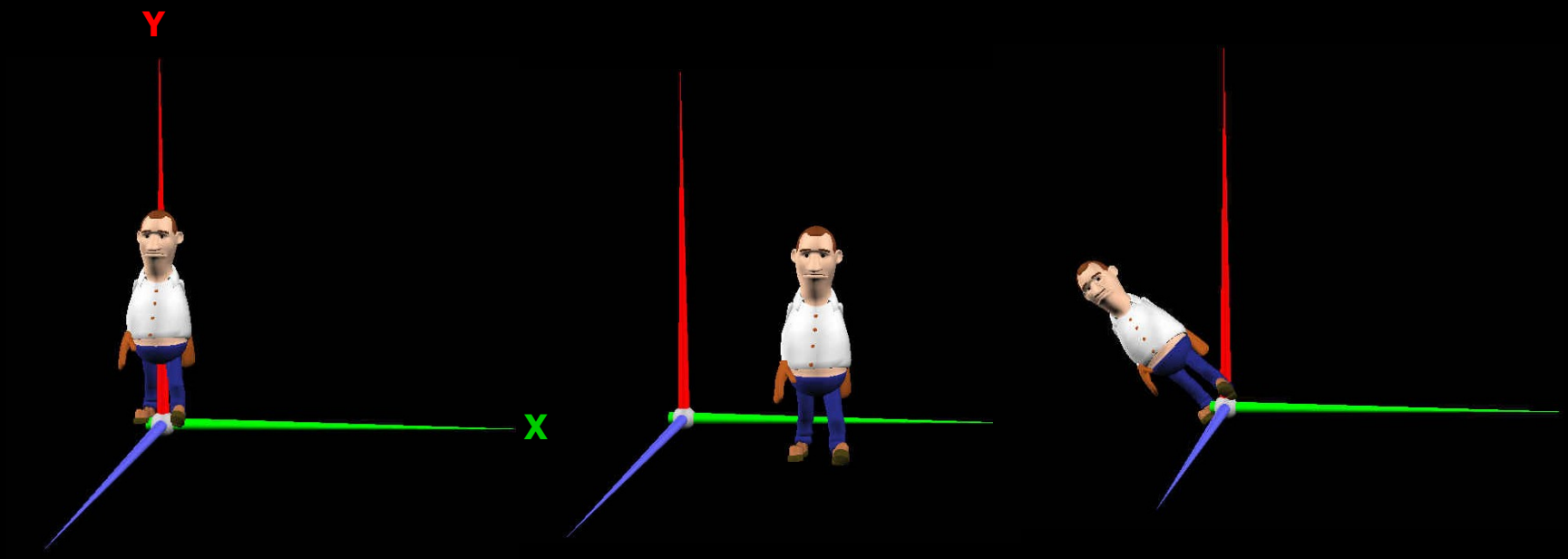


Man kann sie:

- verschieben (translieren)
- drehen (rotieren)
- vergrößern und verkleinern (skalieren)

Diese „Manipulationen“ nennt man Transformationen!

Beispiele für Transformationen im 3D-Raum



Rechtshändiges Koordinatensystem

Translation

glTranslatef(x, y, z);

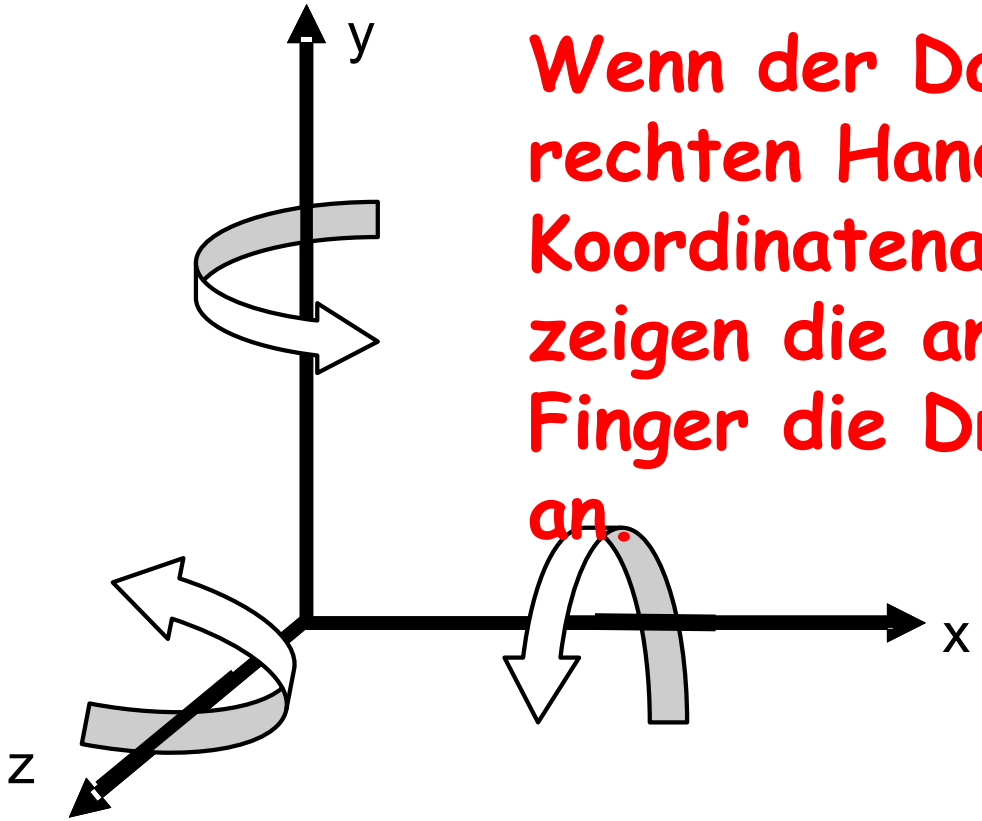
Rotation

glRotatef(Winkel, x, y, z);

Mit x, y, z: Rotationsachse

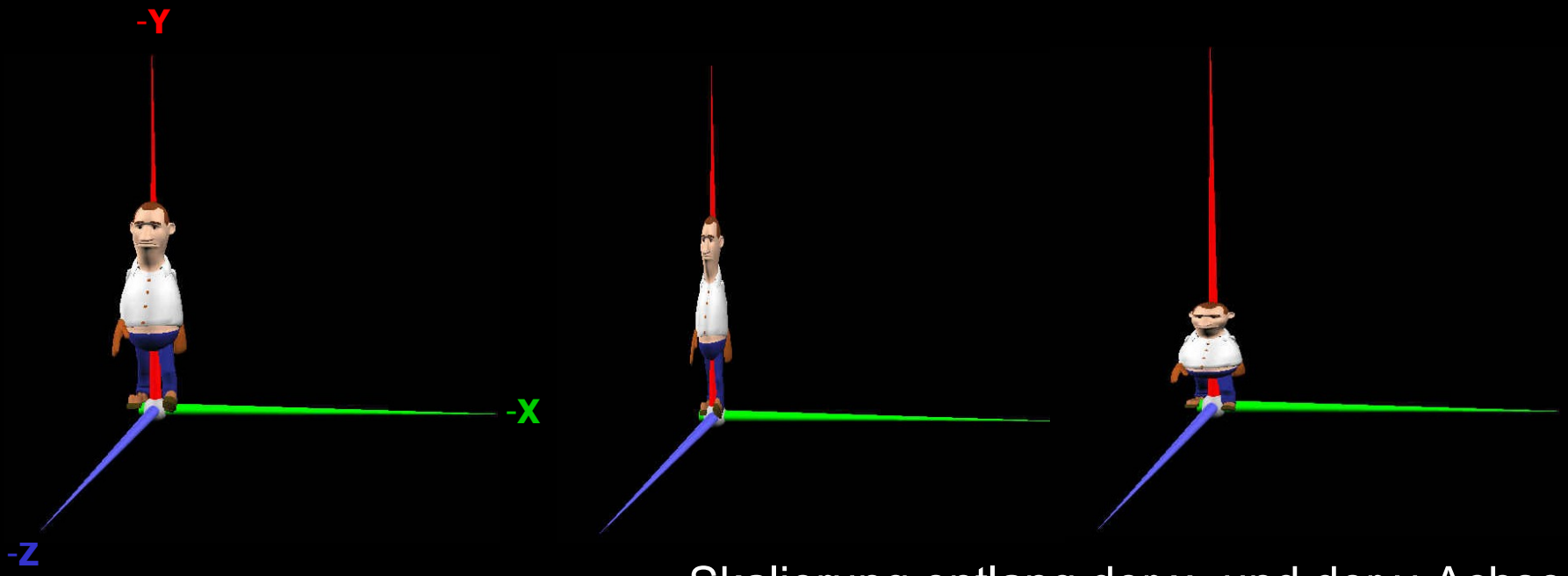
Um welche Achse wird die Geometrie bei der Rotation gedreht?

Demonstration des mathematisch positiven Drehsinns



Wenn der Daume der rechten Hand die Koordinatenachse bildet, zeigen die angewinkel-ten Finger die Drehrichtung an.

Transformation im 3D-Raum: Skalierung

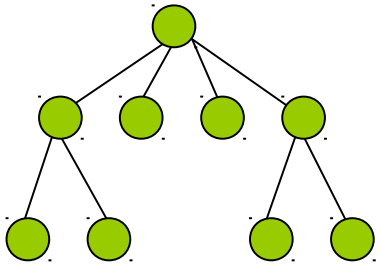


Skalierung entlang der x- und der y-Achse

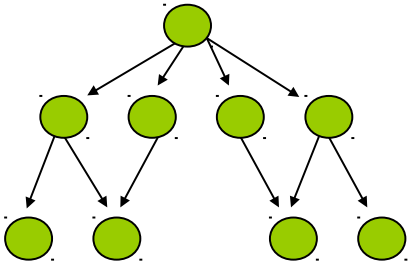
glScalef(x, y, z);

Der Szenengraph

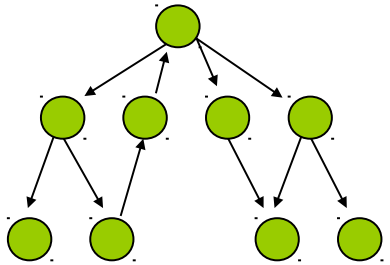
Unterschied Baum und Graph



Baum



gerichteter
zyklenfreier Graph



gerichteter
nicht-zyklenfreier Graph

Szenengraph

Szenengraph besteht aus mindestens 3 Knotentypen:

Gruppen

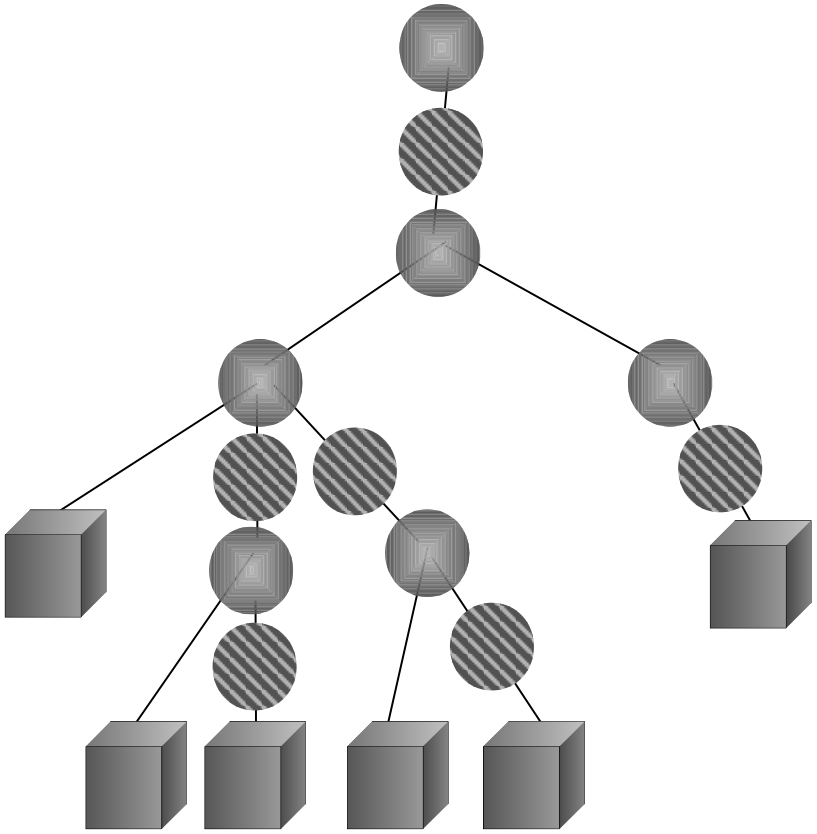
Geometrien (inkl. Materialeigenschaften)

Transformationen

Szenengraph dient zur Verwaltung einer komplexen Szene:

- Gruppierung von Geometrien zu Gruppen
- Gruppierung von Gruppen zu Gruppen
- Gruppierung von Gruppen zu einer Szene

Szenengraphstruktur:



- gerichtet
- azyklisch
- Geometrie / darstellbare Primitive in den Blättern

Welchen Vorteil hat ein gerichteter azyklischer Szenengraphen gegenüber Szenengraphen, die als Bäume realisiert wurden?

(Sehen wir dann beim Pinguin)

Erzeugung des Pinguins mit Hilfe eines Szenengraphen…

Folgende Primitive stehen zur Verfügung:

- Becher mit der Öffnung nach oben
- Ball (bereits mit Augen und Nase)
- [Ein Dreieck]

Szenengraph des Pinguins

Dreieck für die Flügel und die Füße

Szenengraph des Pinguins

Dreieck für die Flügel und die Füße

Allgemeines zu OpenGL

- OpenGL (Open Graphics Library) ist eine Spezifikation einer API für 3D-Graphik
- OpenGL spezifiziert (standardisiert) rund 250 Befehle
- Die Implementierung der Befehle findet man in den Grafikkartentreibern
  - Befehle werden dann entweder von der Grafikkarte ausgeführt
  - oder auf der CPU
- OpenGL ist ein Renderingsystem keine Modellierungssoftware: komplexe Modelle müssen aus einfachen graphischen Primitiven aufgebaut werden.

Allgemeines zu OpenGL

OpenGL ist eine State-Machine:

- Funktionen verändern den internen Zustand, bzw. verwenden ihn zur Darstellung.
- Das heißt einmal angeschaltet, bleibt der betreffende Zustand aktiv bis er wieder ausgeschaltet oder umgeschaltet wird.

OpenGL ist sehr „explizit“:

- Was nicht explizit aktiviert wurde, bleibt aus.
- Beispiel: Es nutzt nichts die Transparenz zu setzen, wenn man nicht explizit gesagt hat, dass Transparenzen berechnet werden sollen.

GLUT (OpenGL Utility Toolkit)

Übernimmt Plattform unabhängig:

- Darstellung von Fenstern
- Tastatureingaben und Ausgaben
- Funktionen zum Zeichnen einfacher geometrischer Objekte (Torus, Zylinder, Kugel,...)

GLUT (OpenGL Utility Toolkit)

Initialisierung über:

```
glutInit(&argc, argv);
glutInitDisplayMode( GLUT_DEPTH |GLUT_RGB );
glutCreateWindow(„Name");
```

Zeichenfunktion wird automatisch von der glutMainLoop aufgerufen.

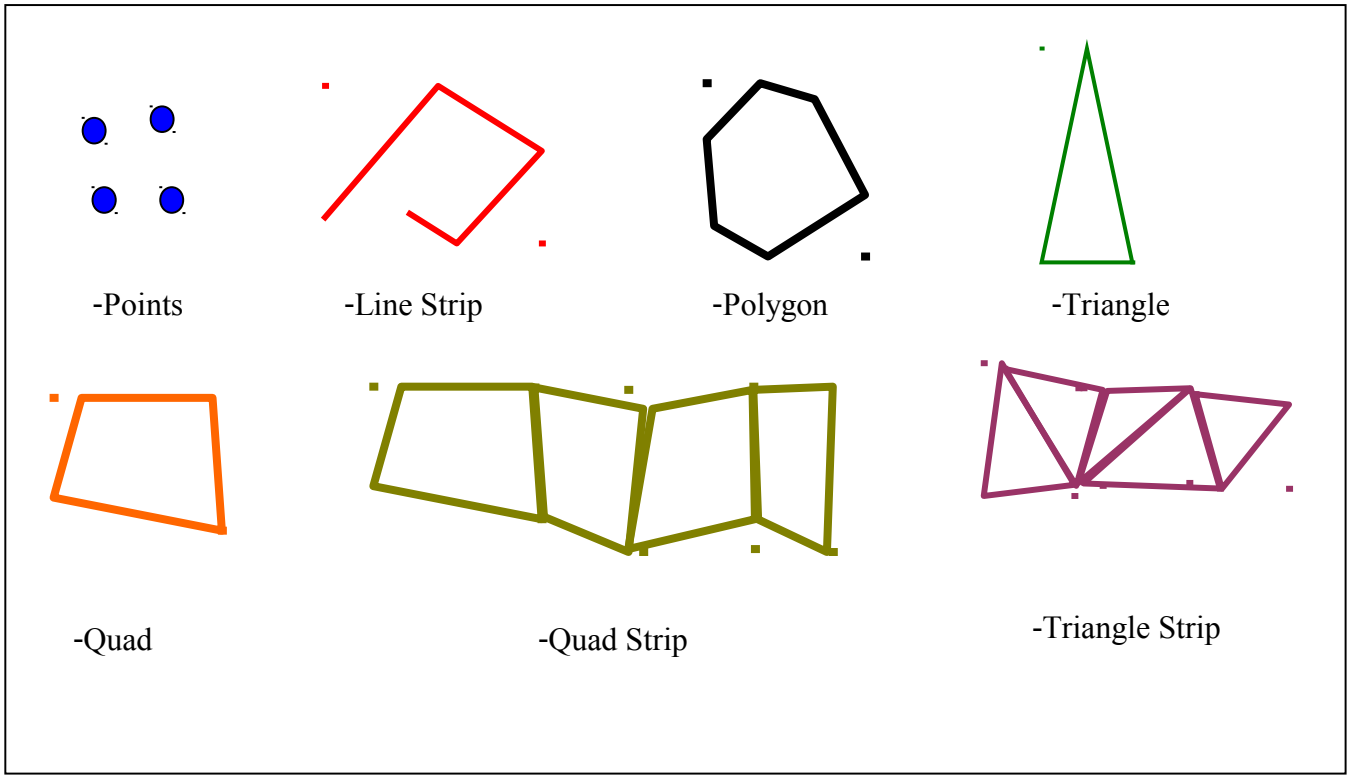
Welche Funktion als Zeichenfunktion genutzt werden soll, wird mit dieser Funktion festgelegt:

```
glutDisplayFunc(display);
```

Aufruf der Hauptschleife:

```
glutMainLoop();
```

OpenGL Primitive

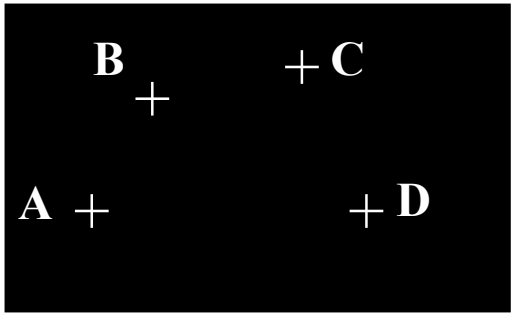


Definition der OpenGL Primitive

```
glBegin ( Typ des Primitivs )
    Liste von Eckpunkten (= Vertices)
glEnd ();
```

Beispiel:

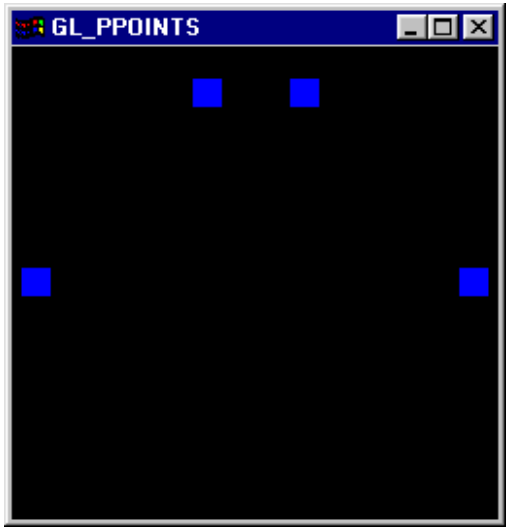
```
glBegin ( GL_POINTS );
    glVertex3f ( xA, yA, zA );
    glVertex3f ( xB, yB, zB );
    glVertex3f ( xC, yC, zC );
    glVertex3f ( xD, yD, zD );
glEnd ();
```



Achtung:
Dies ist nicht die Bildschirm-ausgabe. Wegen der Größe der Punkte (1 Pixel) wird auf dem Schirm nur sehr wenig zu sehen sein

OpenGL Primitiv: Punkt

```
glPointSize (  15.0 );
glBegin ( GL_POINTS );
   glColor4f  ( 0.0f, 0.0f, 1.0f, 1.0f );
   glVertex3f ( -0.9f, 0.0f, 0.0f );
   glVertex3f ( -0.2f, 0.8f, 0.0f );
   glVertex3f ( +0.2f, 0.8f, 0.0f );
   glVertex3f ( +0.9f, 0.0f, 0.0f );
glEnd ( );
```

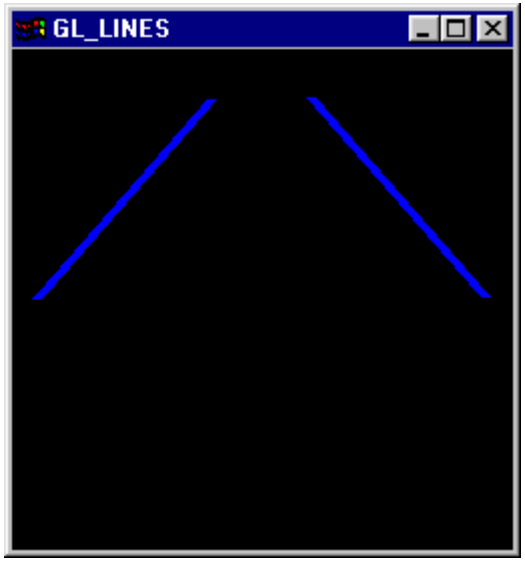


OpenGL ist eine State-Machine:

- Funktionen verändern den internen Zustand.
- Das heißt, einmal angeschaltet, bleibt der betreffende Zustand aktiv, bis er wieder ausgeschaltet oder umgeschaltet wird.
- **Beispiel:** `glColor4f`

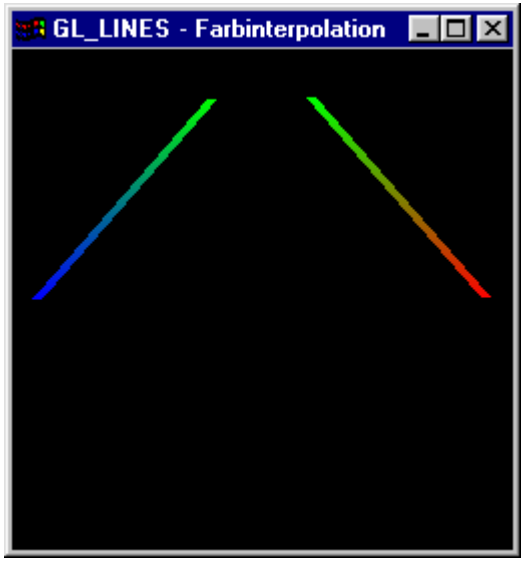
OpenGL Primitiv: Linie

```
glLineWidth (  5.0 );
glBegin ( GL_LINES );
  glColor4f  ( 0.0f, 0.0f, 1.0f, 1.0f );
  glVertex3f ( -0.9f, 0.0f, 0.0f );
  glVertex3f ( -0.2f, 0.8f, 0.0f );
  glVertex3f ( +0.2f, 0.8f, 0.0f );
  glVertex3f ( +0.9f, 0.0f, 0.0f );
glEnd ( );
```



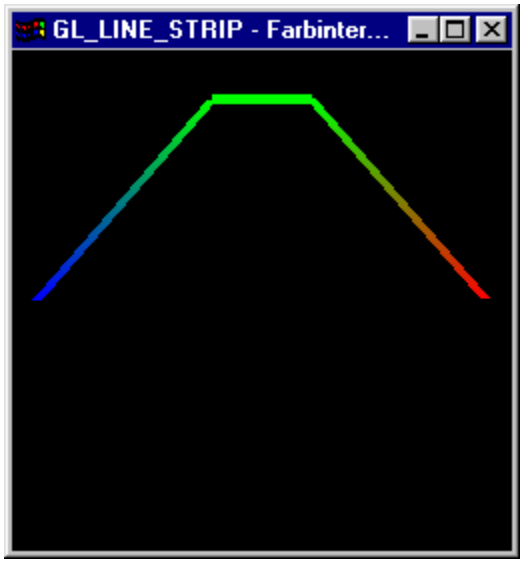
OpenGL Primitiv: Linie

```
glLineWidth (  5.0 );
glBegin ( GL_LINES );
    glColor4f  ( 0.0f, 0.0f, 1.0f, 1.0f );
    glVertex3f ( -0.9f, 0.0f, 0.0f );
    glColor4f  ( 0.0f, 1.0f, 0.0f, 1.0f );
    glVertex3f ( -0.2f, 0.8f, 0.0f );
    glVertex3f ( +0.2f, 0.8f, 0.0f );
    glColor4f  ( 1.0f, 0.0f, 0.0f, 1.0f );
    glVertex3f ( +0.9f, 0.0f, 0.0f );
glEnd ( );
```



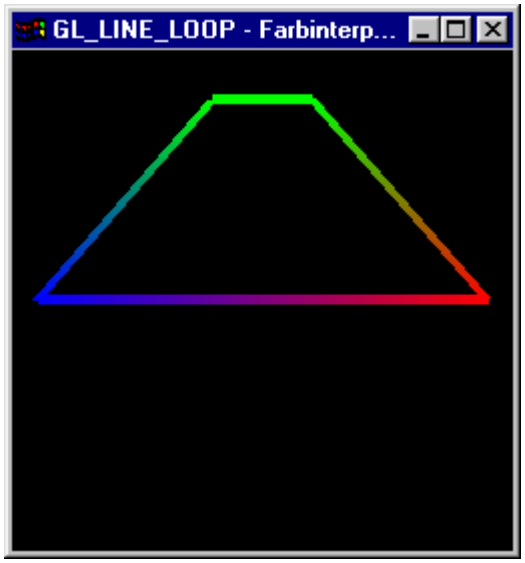
OpenGL Primitiv: Linienzug

```
glLineWidth (  5.0 );
glBegin ( GL_LINE_STRIP );
    glColor4f  ( 0.0f, 0.0f, 1.0f, 1.0f );
    glVertex3f ( -0.9f, 0.0f, 0.0f );
    glColor4f  ( 0.0f, 1.0f, 0.0f, 1.0f );
    glVertex3f ( -0.2f, 0.8f, 0.0f );
    glVertex3f ( +0.2f, 0.8f, 0.0f );
    glColor4f  ( 1.0f, 0.0f, 0.0f, 1.0f );
    glVertex3f ( +0.9f, 0.0f, 0.0f );
glEnd ( );
```



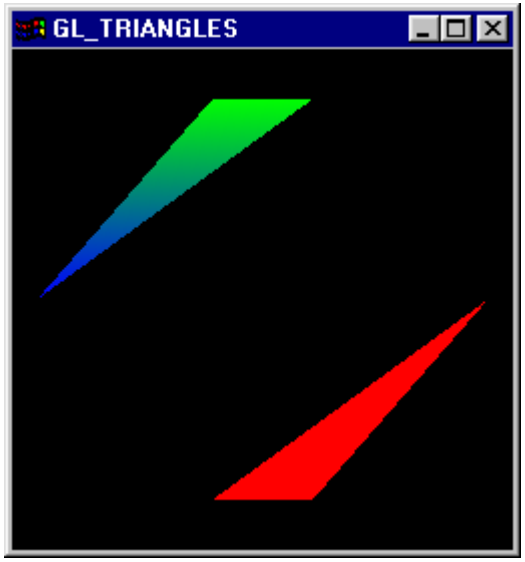
OpenGL Primitiv: Geschlossener Linienzug

```
glLineWidth (  5.0 );
glBegin ( GL_LINE_LOOP );
    glColor4f  ( 0.0f, 0.0f, 1.0f, 1.0f );
    glVertex3f ( -0.9f, 0.0f, 0.0f );
    glColor4f  ( 0.0f, 1.0f, 0.0f, 1.0f );
    glVertex3f ( -0.2f, 0.8f, 0.0f );
    glVertex3f ( +0.2f, 0.8f, 0.0f );
    glColor4f  ( 1.0f, 0.0f, 0.0f, 1.0f );
    glVertex3f ( +0.9f, 0.0f, 0.0f );
glEnd ( );
```



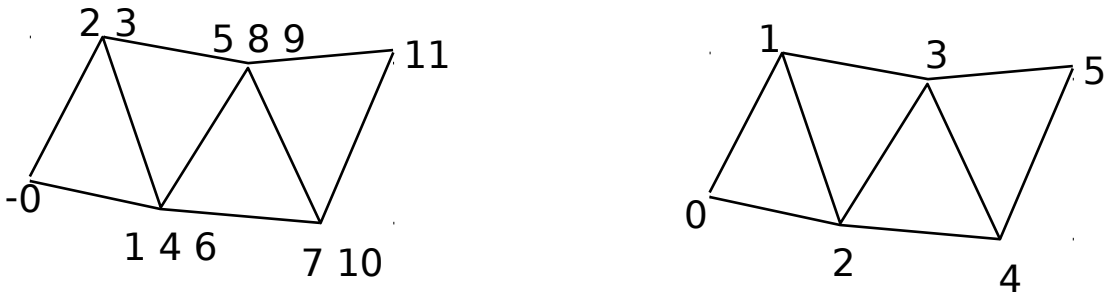
OpenGL Primitiv: Dreieck

```
glBegin(GL_TRIANGLES);
    glColor4f ( 0.0f, 0.0f, 1.0f, 1.0f );
   glVertex3f(-0.9f, 0.0f, 0.0f );
   glColor4f ( 0.0f, 1.0f, 0.0f, 1.0f );
   glVertex3f(-0.2f, 0.8f, 0.0f );
   glVertex3f(+0.2f, 0.8f, 0.0f );
   glColor4f ( 1.0f, 0.0f, 0.0f, 1.0f );
   glVertex3f(+0.9f, 0.0f, 0.0f );
   glVertex3f(+0.2f,-0.8f, 0.0f );
   glVertex3f(-0.2f,-0.8f, 0.0f );
glEnd();
```



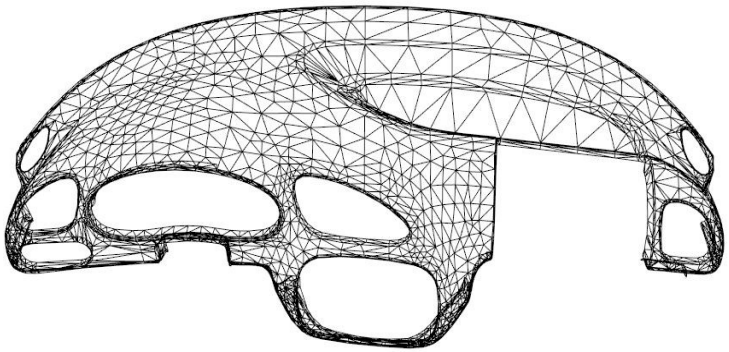
OpenGL Primitiv: Dreiecksstreifen
(Triangle Strip)

- Ziel ist es, möglichst wenig Elemente anzulegen.
- Eckpunkte können durch zusammenhängende Strips recycelt werden.

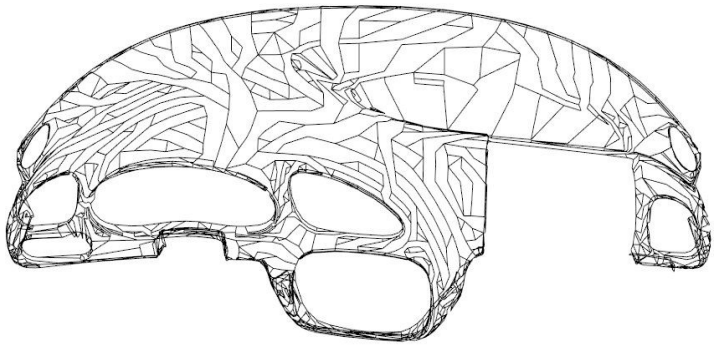


Anzahl der Punkte ohne Streifen- und mit Streifenbildung

OpenGL Primitiv: Dreiecksstreifen



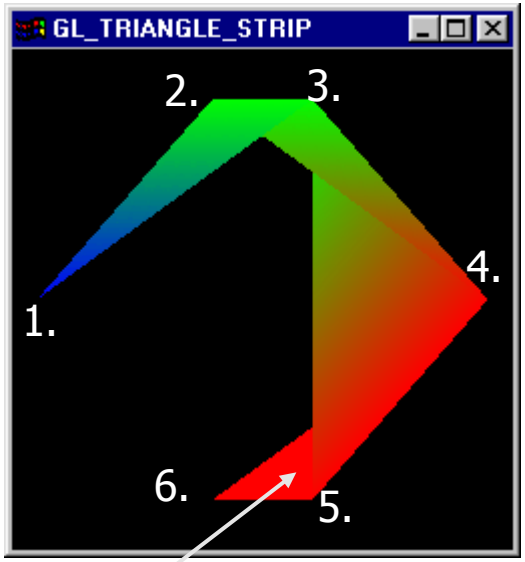
4320 Dreiecke
12960 Eckpunkte



905 Strips
6127 Eckpunkte

OpenGL Primitiv: Dreiecksstreifen

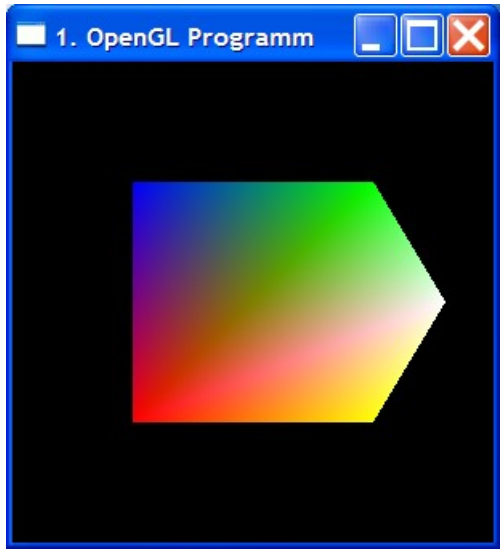
```
glBegin(GL_TRIANGLE_STRIP);
1.  glColor4f ( 0.0, 0.0, 1.0, 1.0f );
    glVertex3f(-0.9, 0.0, 0.0 );
2.  glColor4f ( 0.0, 1.0, 0.0, 1.0f );
3.  glVertex3f(-0.2, 0.8, 0.0 );
    glVertex3f( 0.2, 0.8, 0.0 );
4.  glColor4f ( 1.0, 0.0, 0.0, 1.0f );
5.  glVertex3f( 0.9, 0.0, 0.0 );
    glVertex3f( 0.2,-0.8, 0.0 );
6.  glVertex3f(-0.2,-0.8, 0.0 );
glEnd();
```



Das zuletzt definierte Dreieck wird zuerst gezeichnet!

OpenGL Primitiv: Polygon

```
glBegin( GL_POLYGON );
  glColor4f( 1., 0., 0., 1.);
  glVertex2f( -0.5, -0.5);
  glColor4f( 1., 1., 0., 1.);
  glVertex2f(  0.5, -0.5);
  glColor4f( 1., 1., 1., 1.);
  glVertex2f(  0.8, 0.);
  glColor4f( 0., 1., 0., 1.);
  glVertex2f(  0.5,  0.5);
  glColor4f( 0., 0., 1., 1.);
  glVertex2f( -0.5,  0.5);
glEnd();
```



Graphische Datenverarbeitung

Graphische Objekte & graphische Programmierung

Teil 2

Prof. Dr. Elke Hergenröther

h_da

Transformationen in OpenGL

Translation: **Geometrie wird um den Vektor (x, y, z) verschoben!**

```
glTranslatef( x, y, z);
```

Rotationen: **Geometrie wird um den angegebenen Winkel (gegen den Uhrzeigersinn) um die Achse (x, y, z) rotiert.**

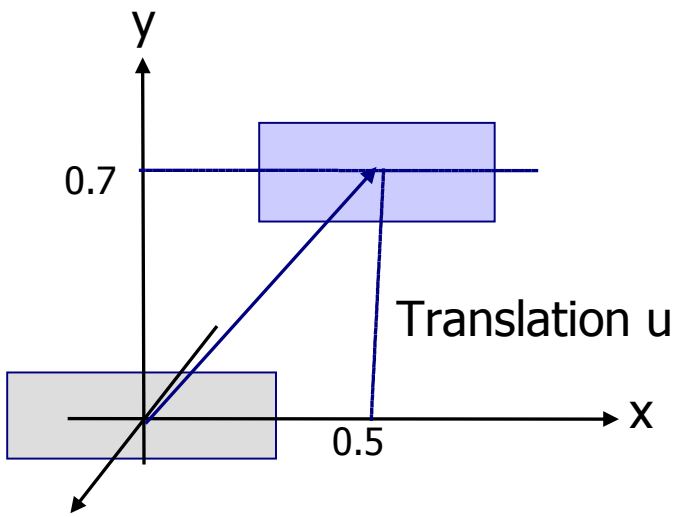
```
glRotatef( Winkel, x, y, z);
```

x, y, z: Rotationsachse

Skalierung:

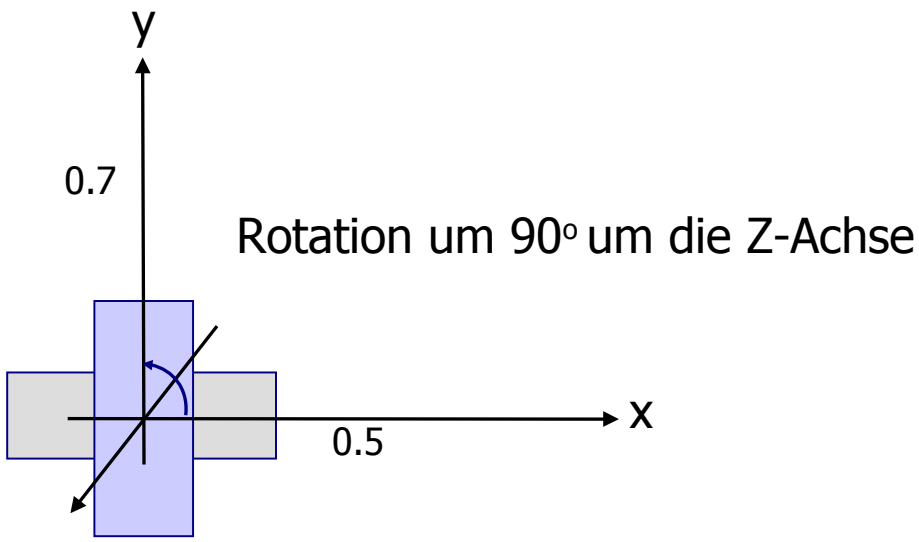
```
glScalef( x, y, z);
```

Beispiel für eine Translation

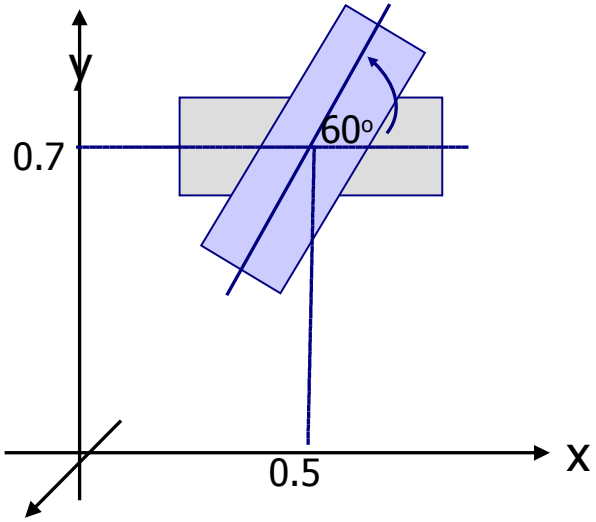


Translation um den Vektor (0.5, 0.7, 0.)

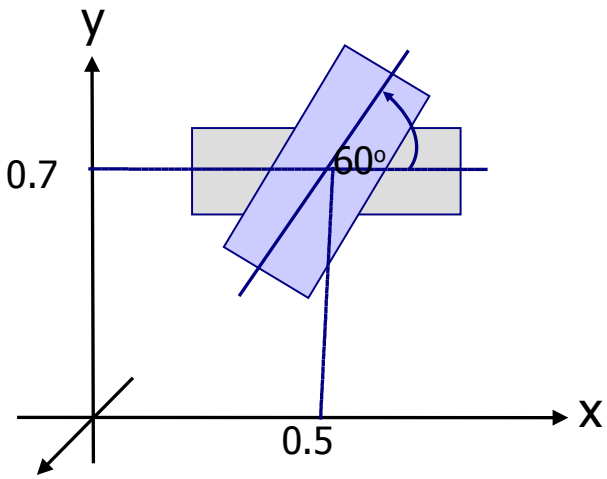
Beispiel für eine Rotation



Welche Transformationen sind notwendig um das Rechteck, wie vorgegeben, zu drehen?



Quadrat soll um 60° gedreht werden:



3. Rücktransformation in die Originalposition
2. Rotation um die Z-Achse
1. Transformation in den Ursprung

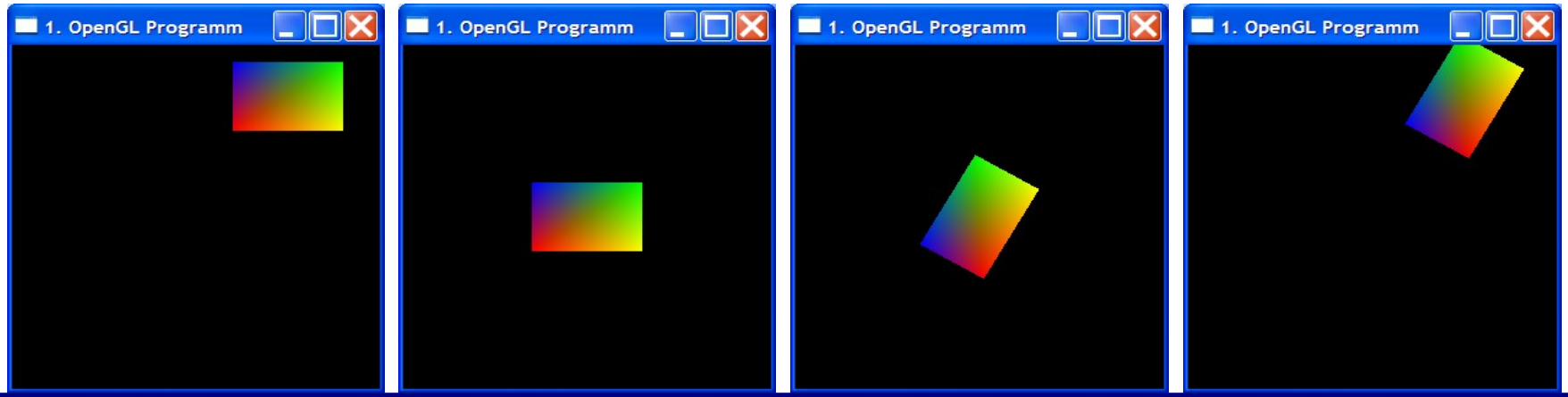
Reihenfolge in der die Transformationen angewendet werden ist wichtig!

Quadrat soll um 60° gedreht werden:

Transformationen werden in **umgekehrter** Reihenfolge angegeben:

```
glTranslatef( 0.5, 0.7, 0.);
GlRotatef( 60., 0., 0., 1.);
GlTranslatef( -0.5, -0.7, 0.);
Quadrat();
```

Reihenfolge in der die Transformationen auf das Quadrat angewendet werden:



Quadrat soll um 60° gedreht werden:

```
glTranslatef( 0.5, 0.7, 0.);
glRotatef( 60., 0., 0., 1.);
glTranslatef( -0.5, -0.7, 0.);

glBegin( GL_POLYGON );
     glColor4f( 1., 0., 0., 1.);
     glVertex3f( 0.2, 0.5, 0);
     glVertex3f(  0.8, 0.5, 0);
     glVertex3f(  0.8,  0.9, 0);
     glVertex3f( 0.2,  0.9, 0);
glEnd();
```

Entspricht der Funktion: Quadrat()

Warum werden die Transformationen in umgekehrter Reihenfolge angegeben?

- Alle Transformationen werden durch Matrixmultiplikationen realisiert.
- Matrixmultiplikationen sind nicht kommutativ (d.h. sie sind nicht in ihrer Reihenfolge vertauschbar)

- Beispiel: a.) liefert nicht dasselbe Ergebnis, wie b.)

a.)

$$\begin{bmatrix} 3 \\ 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ -1 & (\\ 0 & (\\ 0 & (\end{bmatrix}$$

b.)

$$\begin{bmatrix} 5 \\ -3 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & (\\ 0 & \\ 0 & (\\ 0 & (\end{bmatrix}$$

Warum werden die Transformationen in umgekehrter Reihenfolge ausgewertet?

- Alle Transformationen werden durch Matrixmultiplikationen realisiert.
- Matrixmultiplikationen sind nicht kommutativ (d.h. sie sind nicht in ihrer Reihenfolge vertauschbar)
- Äquivalente Berechnungen von P‘:

$$\vec{P}' = M_2 \cdot (M_1 \cdot \vec{P}) \Leftarrow$$

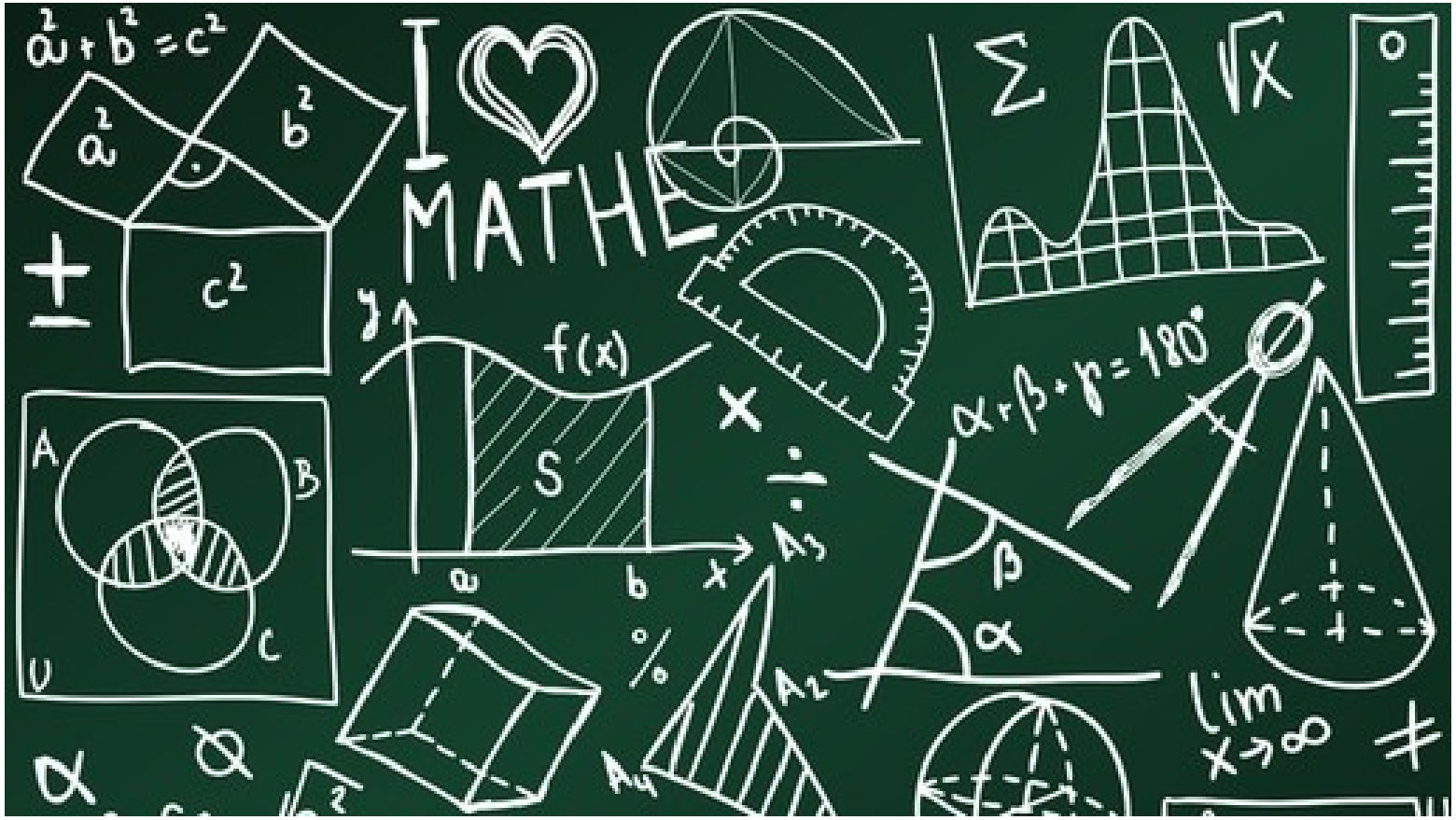
$$\vec{P}' = M_2 \cdot \vec{P}^{M_1} \Leftrightarrow \vec{P}$$

Intuitive Vorgehensweise wenn man „von Hand“ transformiert:

P wird zunächst mit M_1 und dann wird der Ergebnisvektor mit M_2 multipliziert

Vorgehensweise von OpenGL:

Erstellen einer akkumulierten Matrix (M_2 wird mit M_1 multipliziert).



Bildquelle: http://www.ors.bb.schule-bw.de/index.php/ag-s-und-projekte/ag-mathe

X-Rotation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z-Rotation in 3D

$$\begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale in 3D

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(4x4)*(4x1) = (4x1)

Y-Rotation in 3D

$$\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix Multiplication

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ q \end{bmatrix}$$

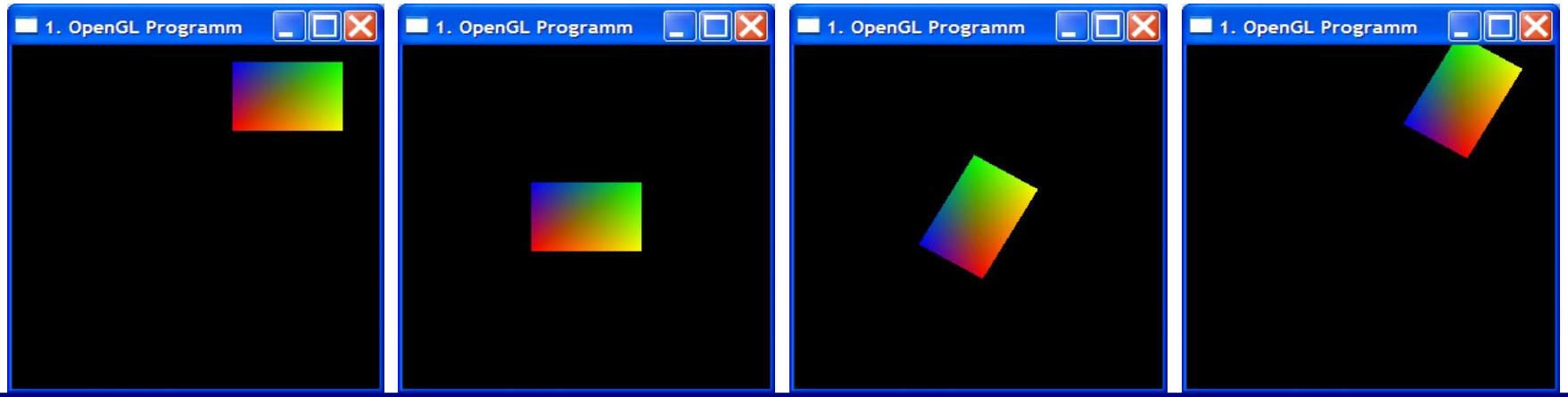
Quadrat soll um 60° gedreht werden:

Transformationen werden in **umgekehrter** Reihenfolge angegeben:

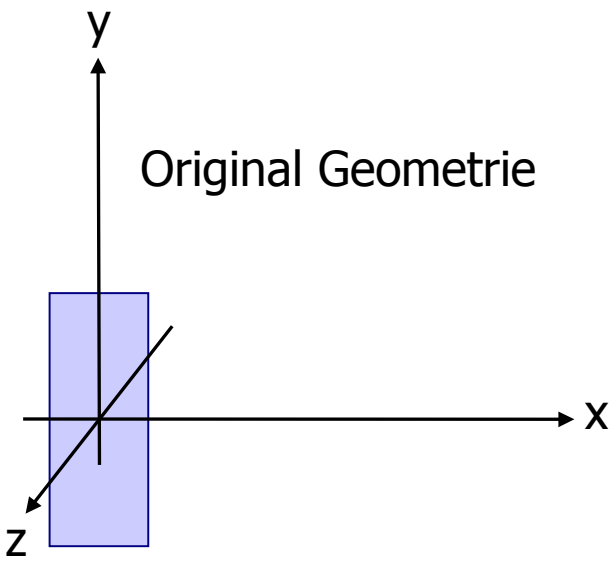
```
glTranslatef( 0.5, 0.7, 0.);    // MT2
glRotatef( 60., 0., 0., 1.);    // MR
glTranslatef( -0.5, -0.7, 0.); // MT1
Quadrat();
```

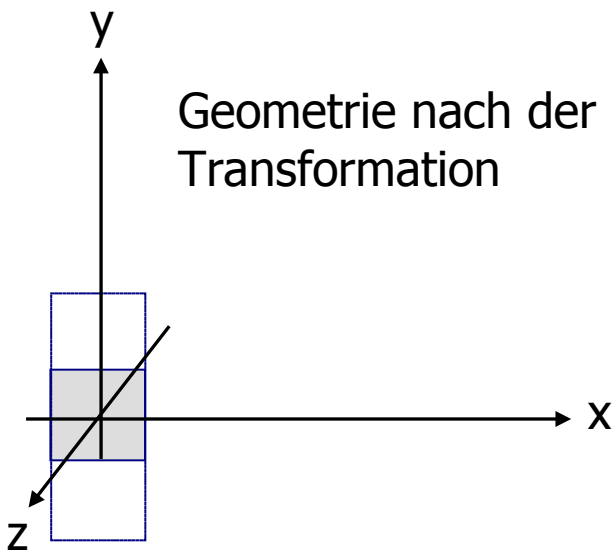
Reihenfolge in der die Transformationen auf die akkumulierte Matrix M multipliziert werden:

$M = M_{T2} * M_R * M_{T1} *$ Punkt



Beispiel für eine Skalierung

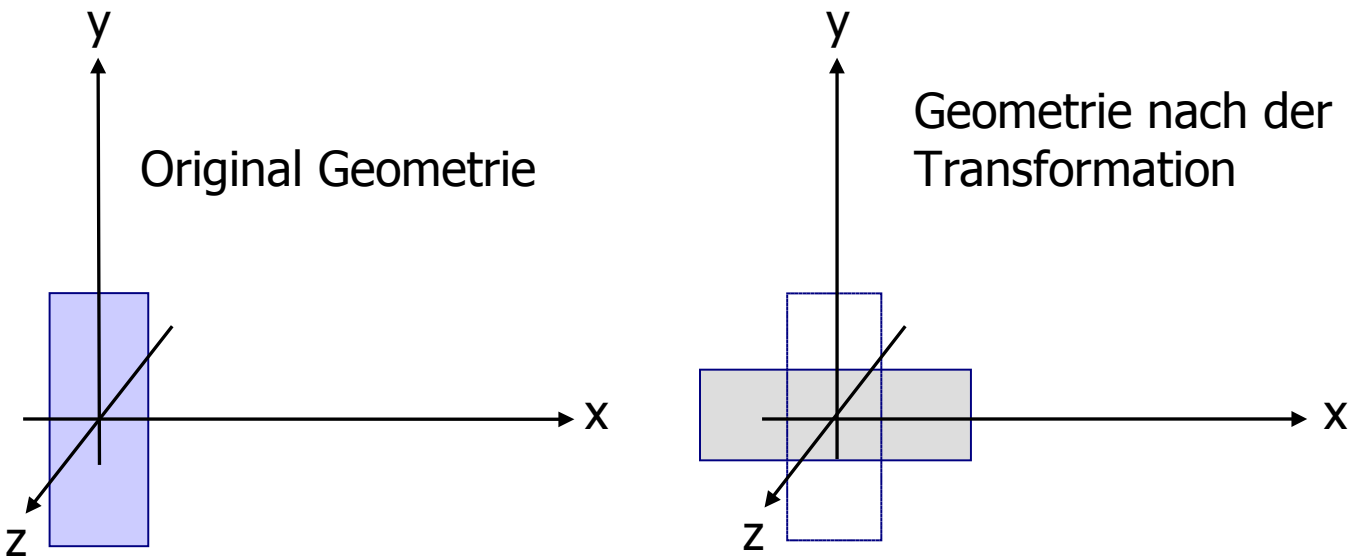




Skalierung um die Faktoren:

x = 1, y = ? und z = ?

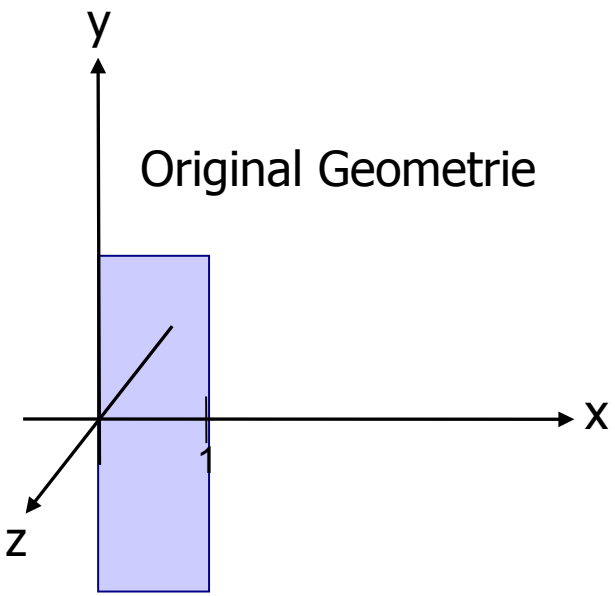
Beispiel für eine Skalierung

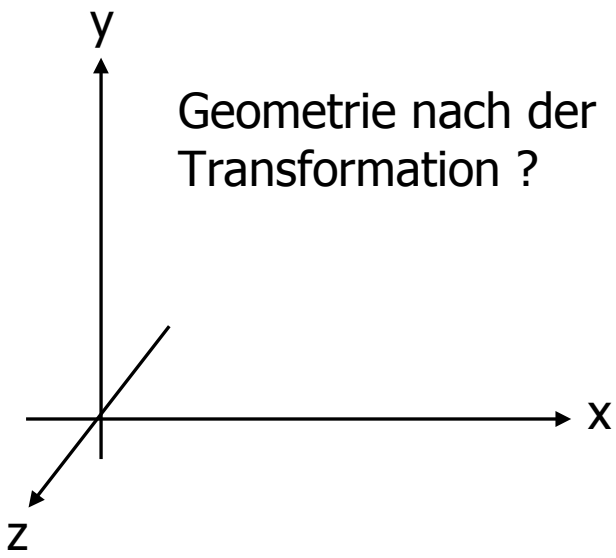


Skalierung um die Faktoren:

x = ?, y = ? und z = 1

Beispiel für eine Skalierung

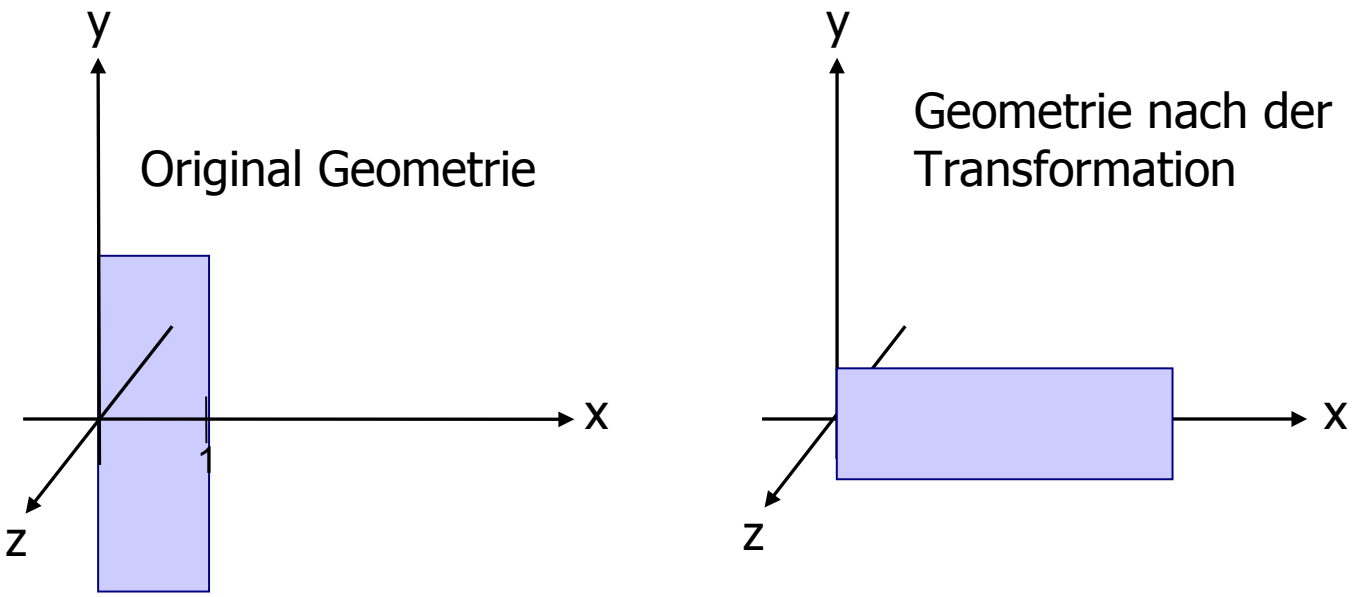




Wie sieht die Geometrie nach der Skalierung mit diesen Faktoren aus?

x = 3 , y = 0.3 und z = 1

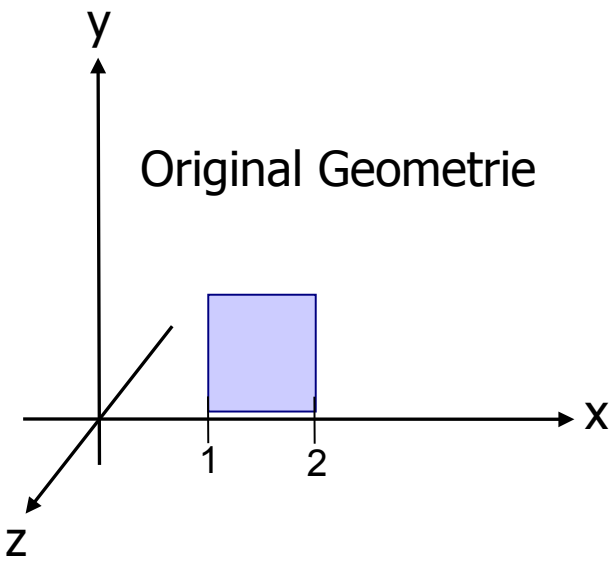
Beispiel für eine Skalierung

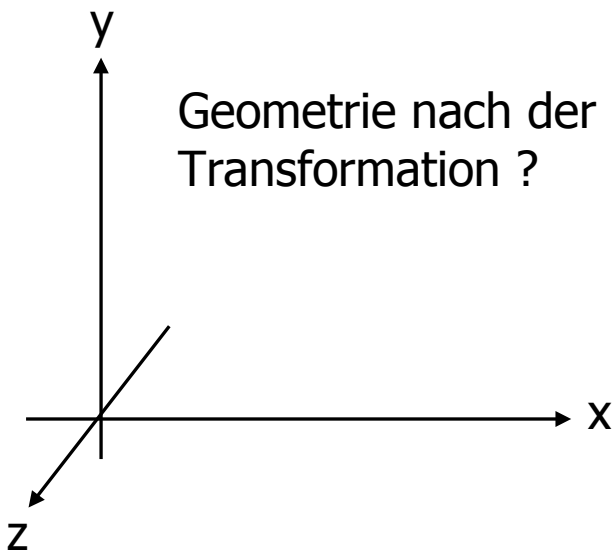


Wie sieht die Geometrie nach der Skalierung mit diesen Faktoren aus?

x = 3 , y = 0.3 und z = 1

Beispiel für eine Skalierung

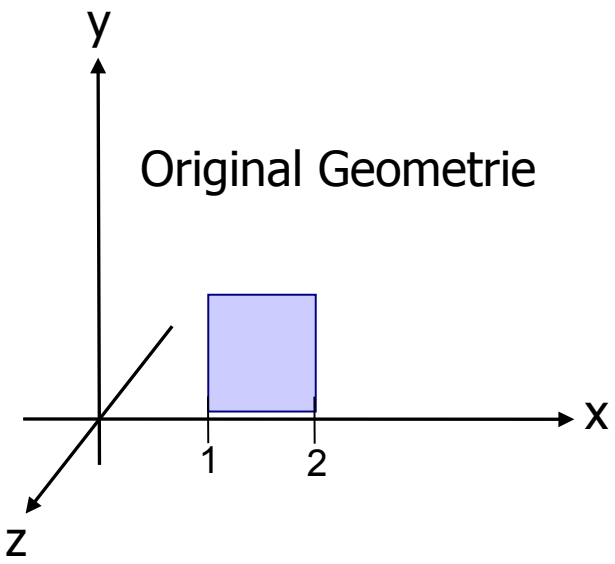


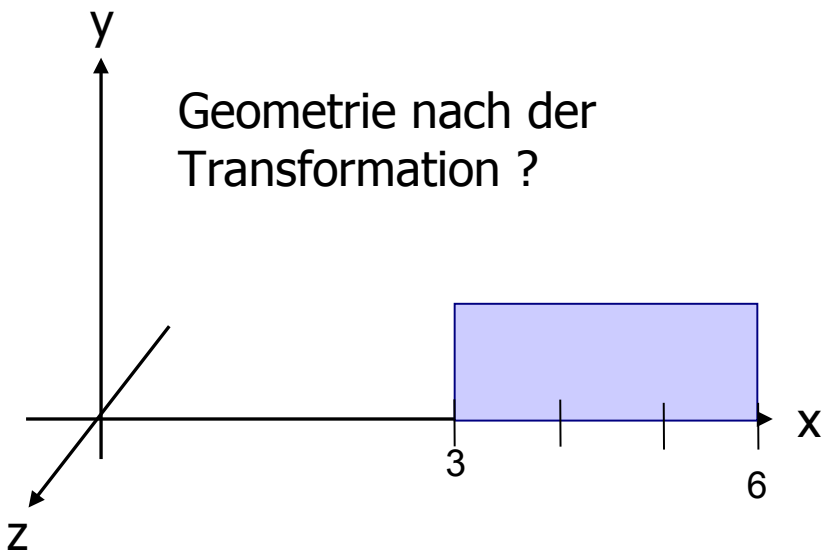


Wie sieht die Geometrie nach der Skalierung mit diesen Faktoren aus?

x = 3 , y = 1 und z = 1

Beispiel für eine Skalierung



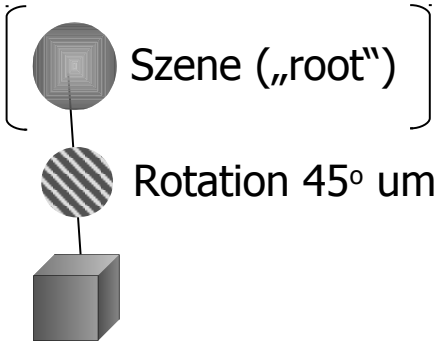


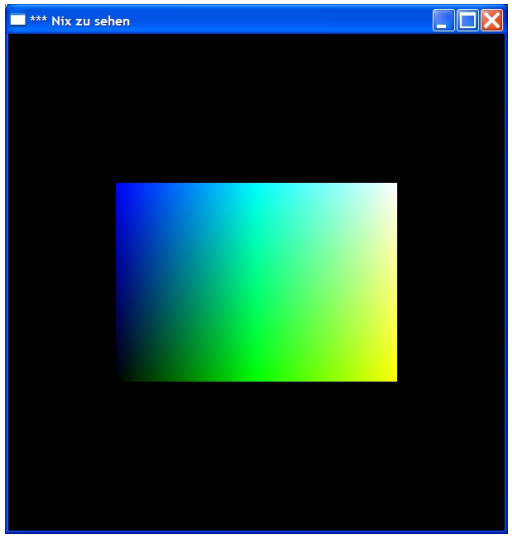
Wie sieht die Geometrie nach der Skalierung mit diesen Faktoren aus?

x = 3 , y = 1 und z = 1

Transformation eines Würfels

```
glRotatef(45., 0., 1., 0.);
Wuerfel(0.8);
```





Transformationen in OpenGL

Matrizenstack: Ermöglicht das temporäre Speichern von Matrizen

Akkumulierte Matrix wird auf den Stack gesichert:
`glPushMatrix();`

Akkumulierte Matrix wird wieder aktiviert mit: `glPopMatrix();`

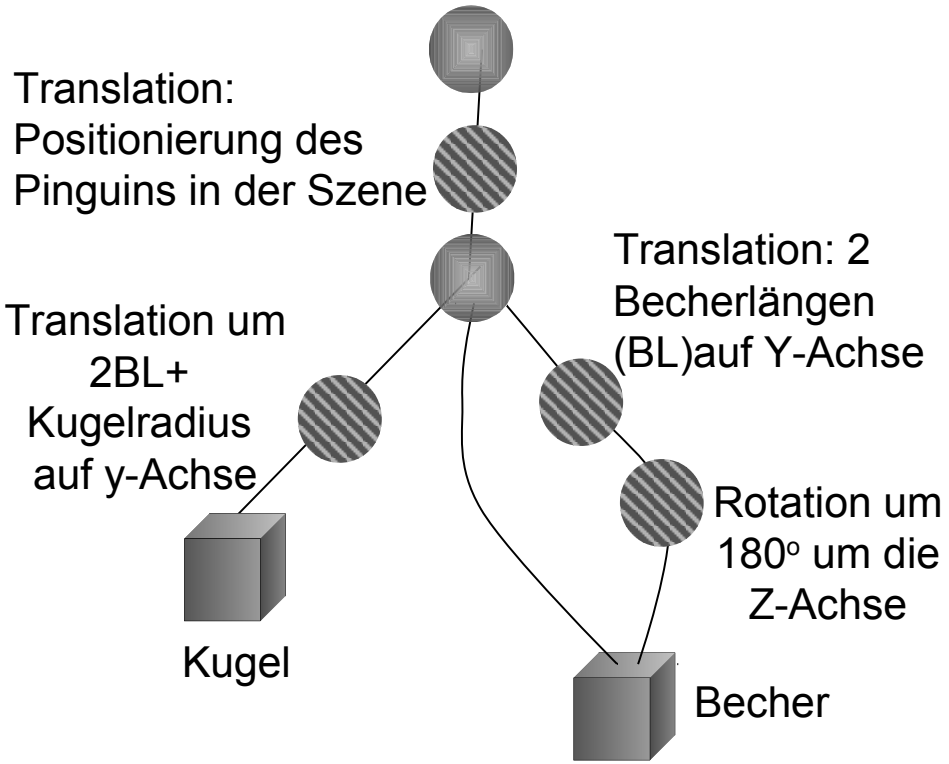
Mit dem Aufruf von `glPopMatrix()` werden alle Änderungen der aktuellen Matrix, die zwischen dem Aufruf `glPushMatrix()` und `glPopMatrix()` gemacht worden sind gelöscht.

Transformationen in OpenGL

```
glLoadIdentity();
glTranslatef(0.3, 0.3, 0); //T1
glPushMatrix();
glTranslatef( -0.25, 0., 0.); //T2
glScalef(0.25, 1., 1.); //S1
Quadrat();
glPopMatrix();
glTranslatef( 0.25, 0., 0.); //T3
glScalef(0.25, 1., 1.); //S2
Quadrat();
```

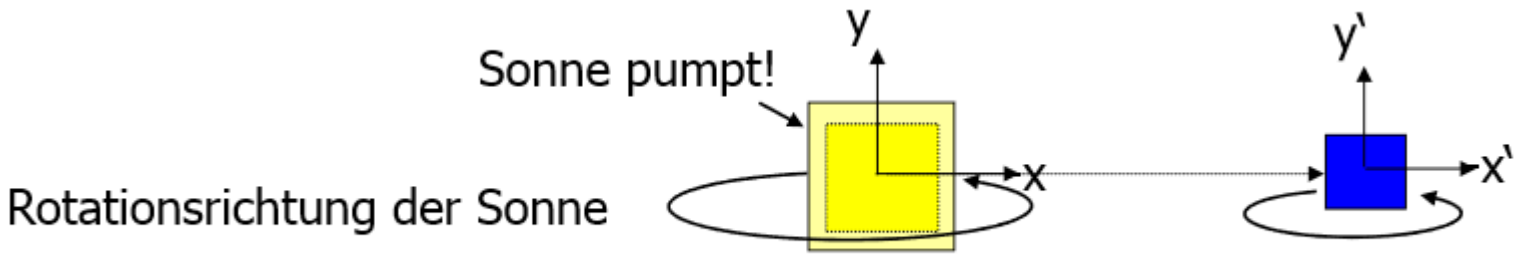
Welche Transformationen werden jeweils auf das Quadrat angewendet?

Probieren Sie den Szenegraphen des Pinguins in OpenGL (mit **`glPushMatrix()`** und **`glPopMatrix()`** zu programmieren …



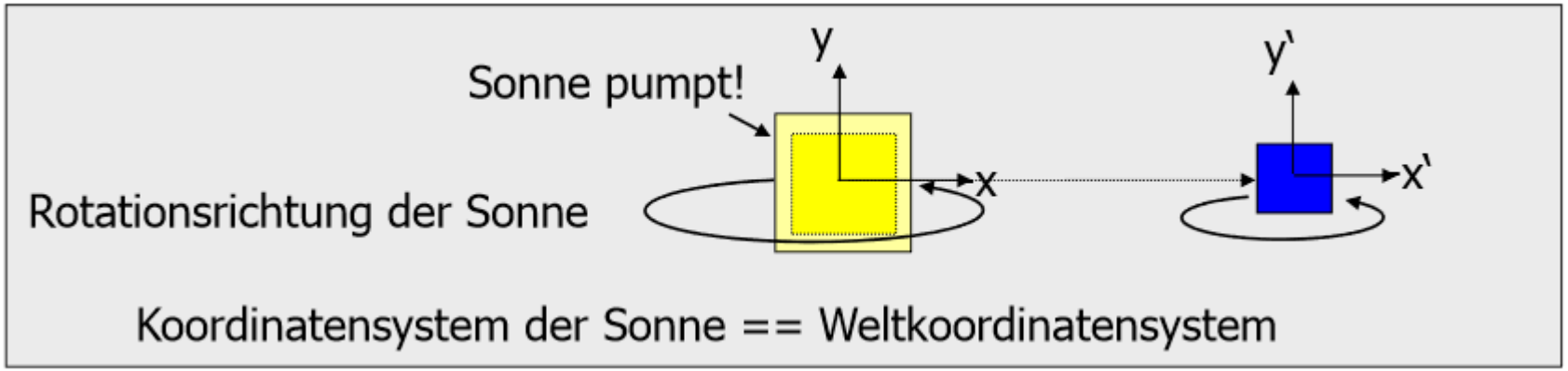
Aufgabe: Miniatursonnensystem

- Sonne rotiert um ihre Y-Achse.
- Während sie rotiert, pumpt sie sich auf und fällt wieder zusammen.
- Die Erde rotiert in einiger Entfernung mit gleicher Winkel-geschwindigkeit (d.h. überstrichener Winkel pro Zeiteinheit ist gleich) um die Y-Achse der Sonne.
- Zusätzlich rotiert die Erde um ihre eigene Y-Achse.
- Erde und Sonne sind eckig ;-)

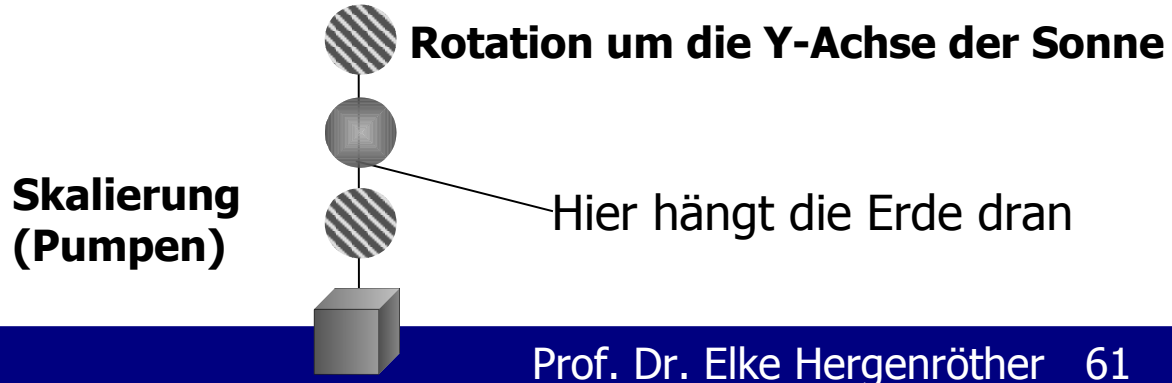


Koordinatensystem der Sonne == Weltkoordinatensystem

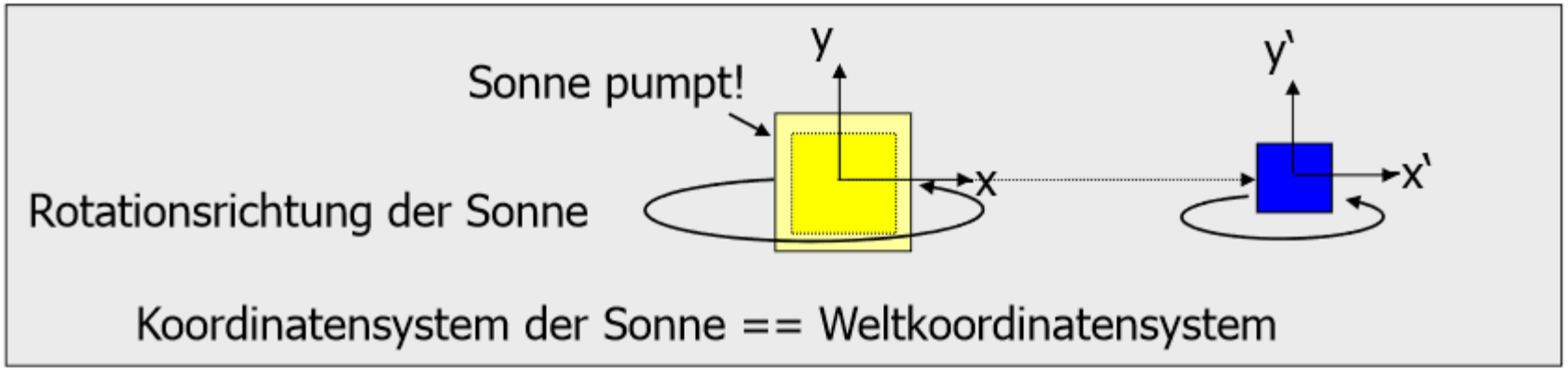
Szenengraph des Miniatursonnensystems



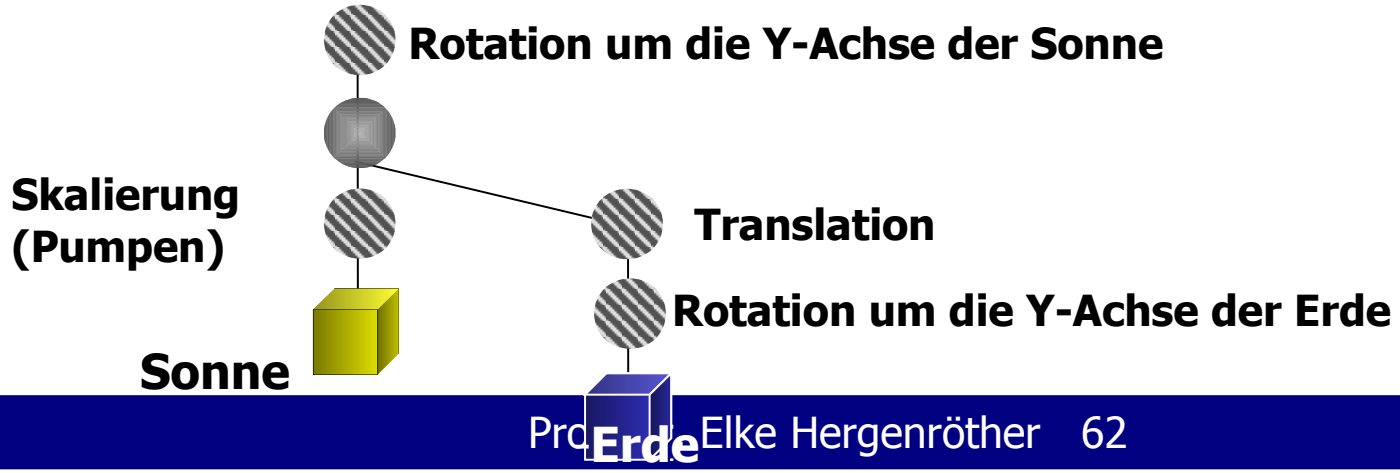
- Sonne und Erde rotieren mit gleicher Winkelgeschwindigkeit um die Y-Achse der Sonne
- Aber nur die Sonne pumpt sich auf und fällt zusammen



Szenengraph des Miniatursonnensystem

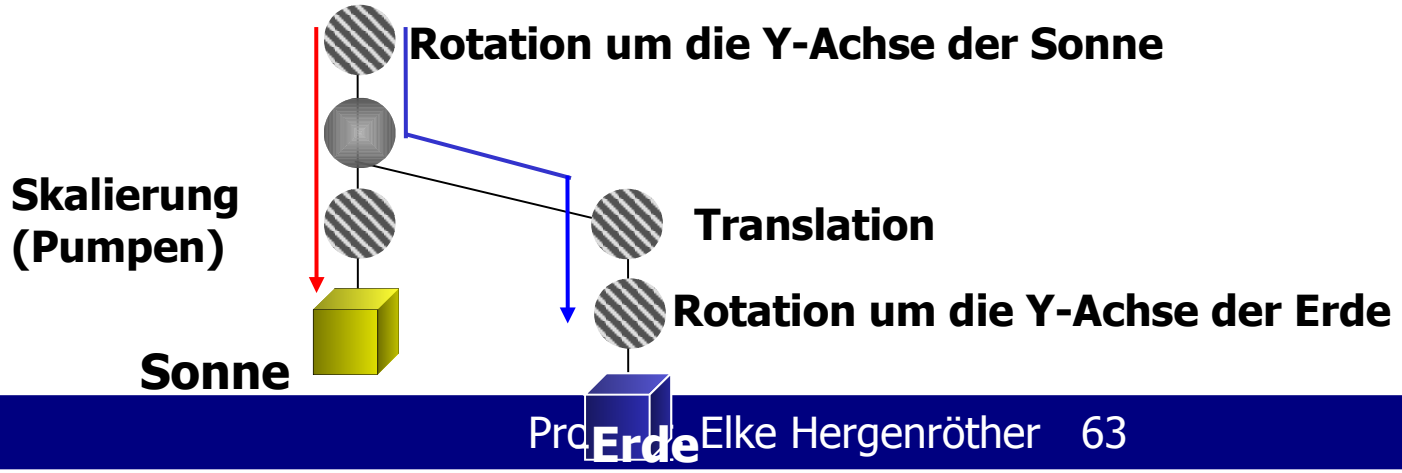


- Erde rotiert zusätzlich um ihre eigne Y-Achse



Prinzipielle Codierung des Szenengraphs

```
//Gruppenknoten zeigen an, wann die akkumulierte Matrix auf
//dem Stack gesichert werden muss.
glLoadIdentity();
glRotatef(fRotSonne, 0., 1., 0.);
glPushMatrix(); //Matrix wird auf den Stack gesichert
    glScalef(fX, fY, fZ); // hier wird gepumpt
    Wuerfel ( 0.5);
glPopMatrix(); //Matrix wird wieder aktiviert
glPushMatrix();
    glTranslatef( 0.5, 0., 0.);
    glRotatef( fRotErde, 0., 1., 0.);
    Wuerfel ( 0.2);
glPopMatrix();
```

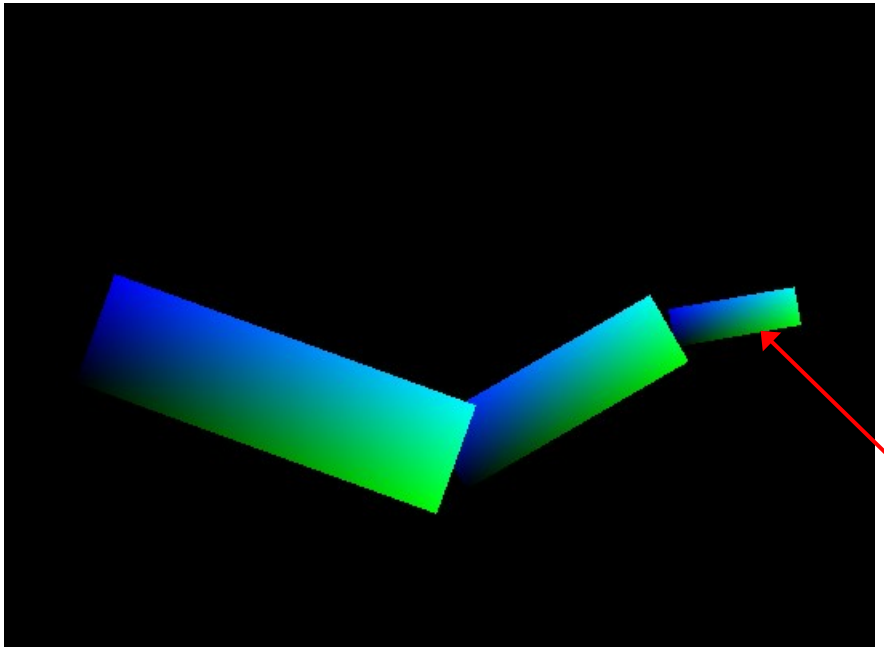


```
void Animate ()
{
   static float fRotSonne = 0., fRotErde=0;
   static float fX=1, fY=1, fZ=1;
   glLoadIdentity();
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   // Puffer loeschen
   glRotatef(fRotSonne+= 5., 0., 1., 0.);
   glPushMatrix(); //Matrix wird auf den Stack gesichert
   if( fX == 1)   { fX=1.12; fY=1.12; fZ=1.12;}
   else    {fX=1; fY=1; fZ=1;}
   glScalef(fX, fY, fZ);
   Wuerfel ( 0.5);
   glPopMatrix(); //Matrix wird wieder aktiviert
   glTranslatef( 0.5, 0., 0.);
   glRotatef( fRotErde+= 25., 0., 1., 0.);
   Wuerfel ( 0.2);
   glFlush();
   Sleep(200);
}
```

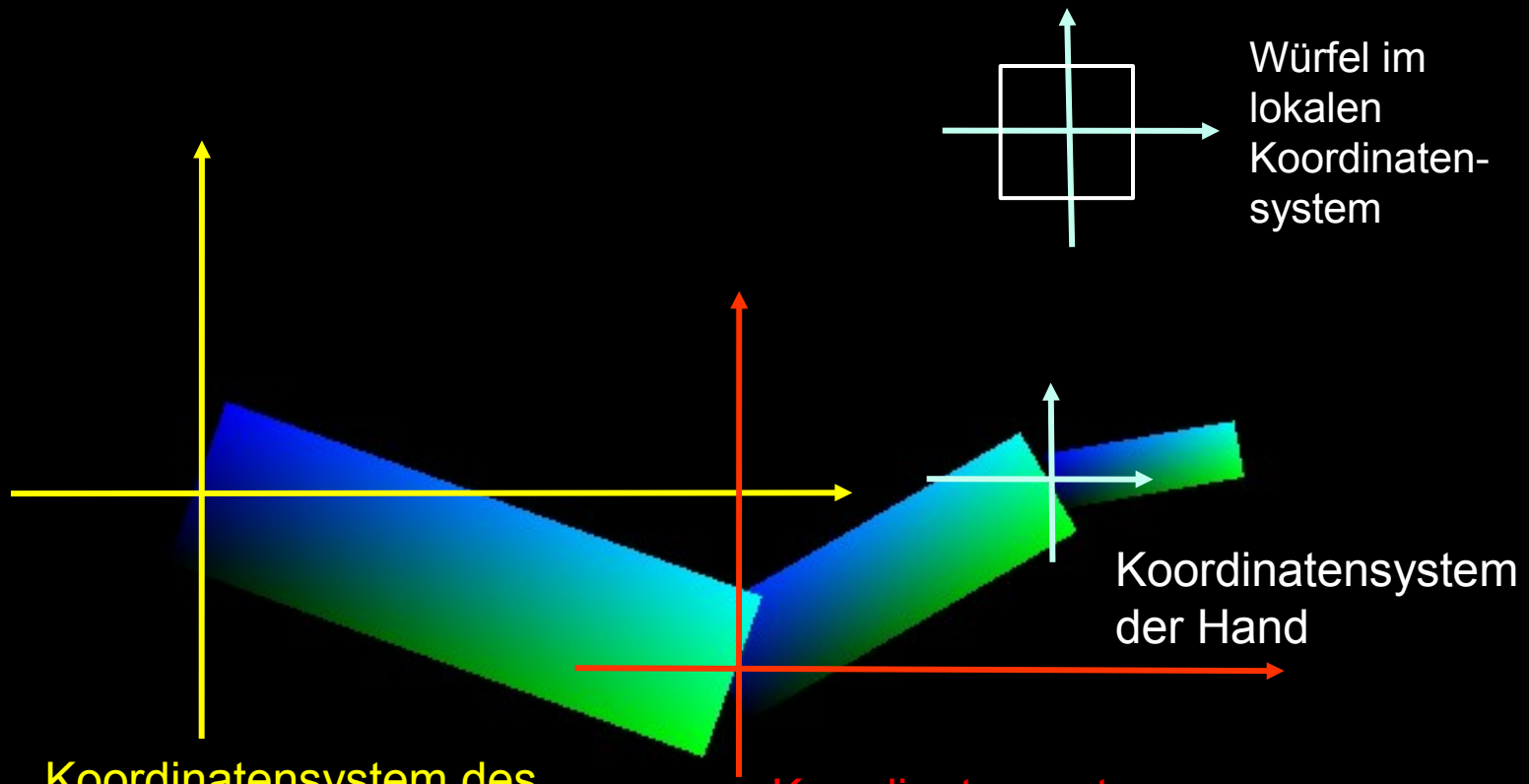
-Miniatursonnensystem

Wiederholung: Szenengraph, Push- und Pop-Matrix am Beispiel des Roboterarms:

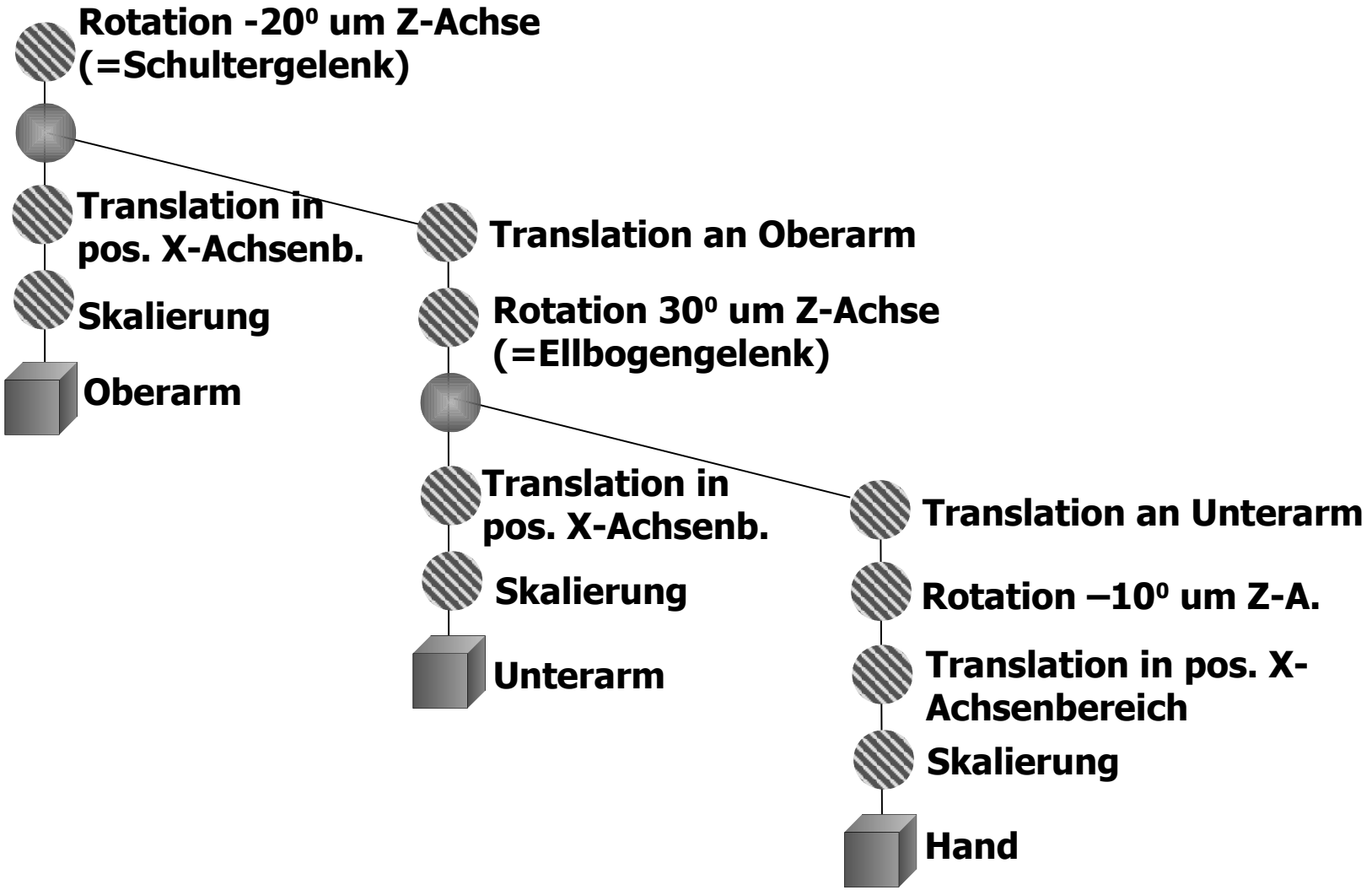
Ausgehend von unserem Würfel, soll dieser Roboterarm erstellt werden:



Konstruktionsprinzip des Roboterarms



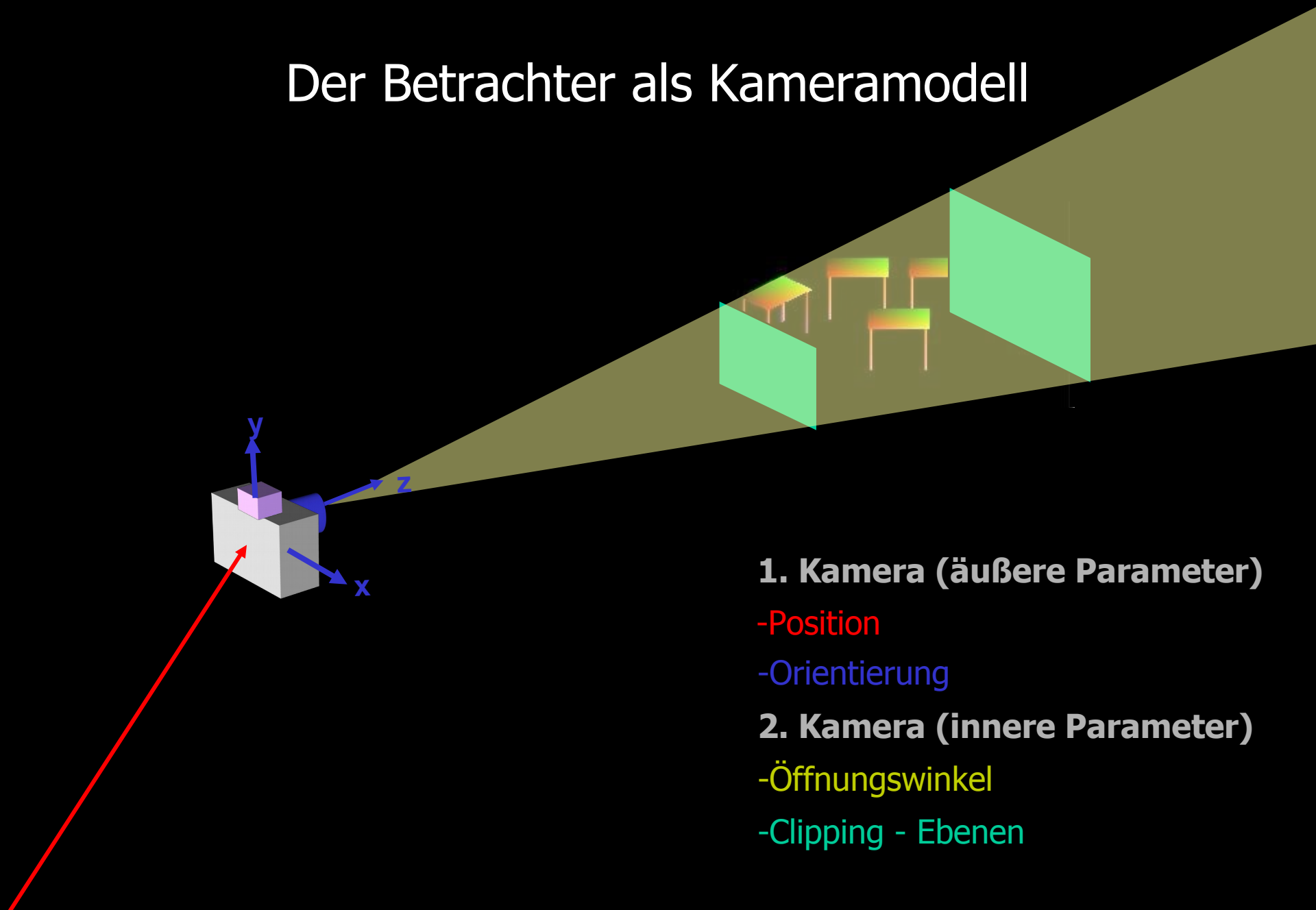
Szenengraph des Roboterarms



Erzeugung graphischer Objekte

- Virtuelle Kamera
  - Innere und äußere Kameraparameter
  - Projektion der 3D Szene in den 2D Raum

Der Betrachter als Kameramodell



1. Kamera (äußere Parameter)

-Position

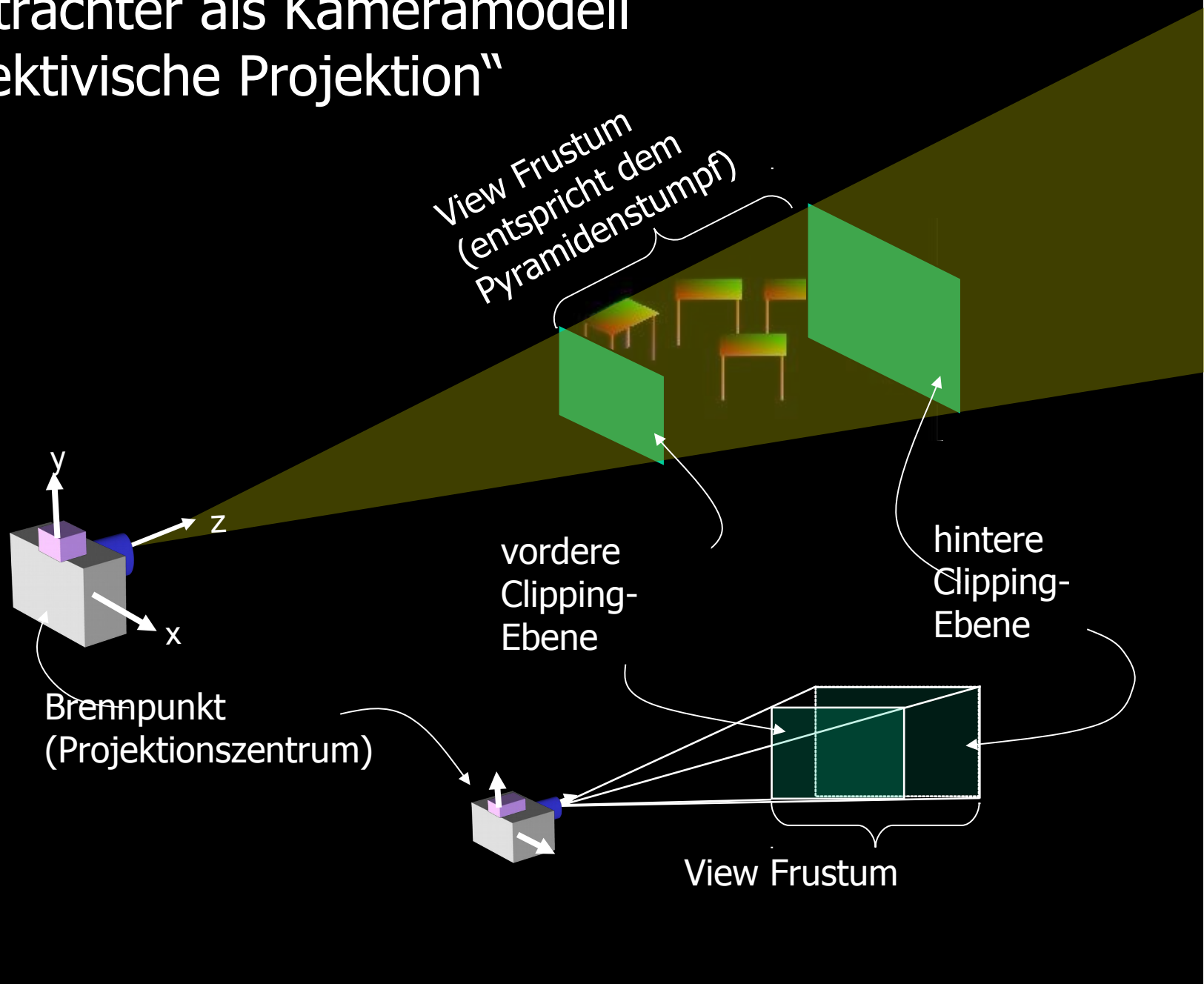
-Orientierung

2. Kamera (innere Parameter)

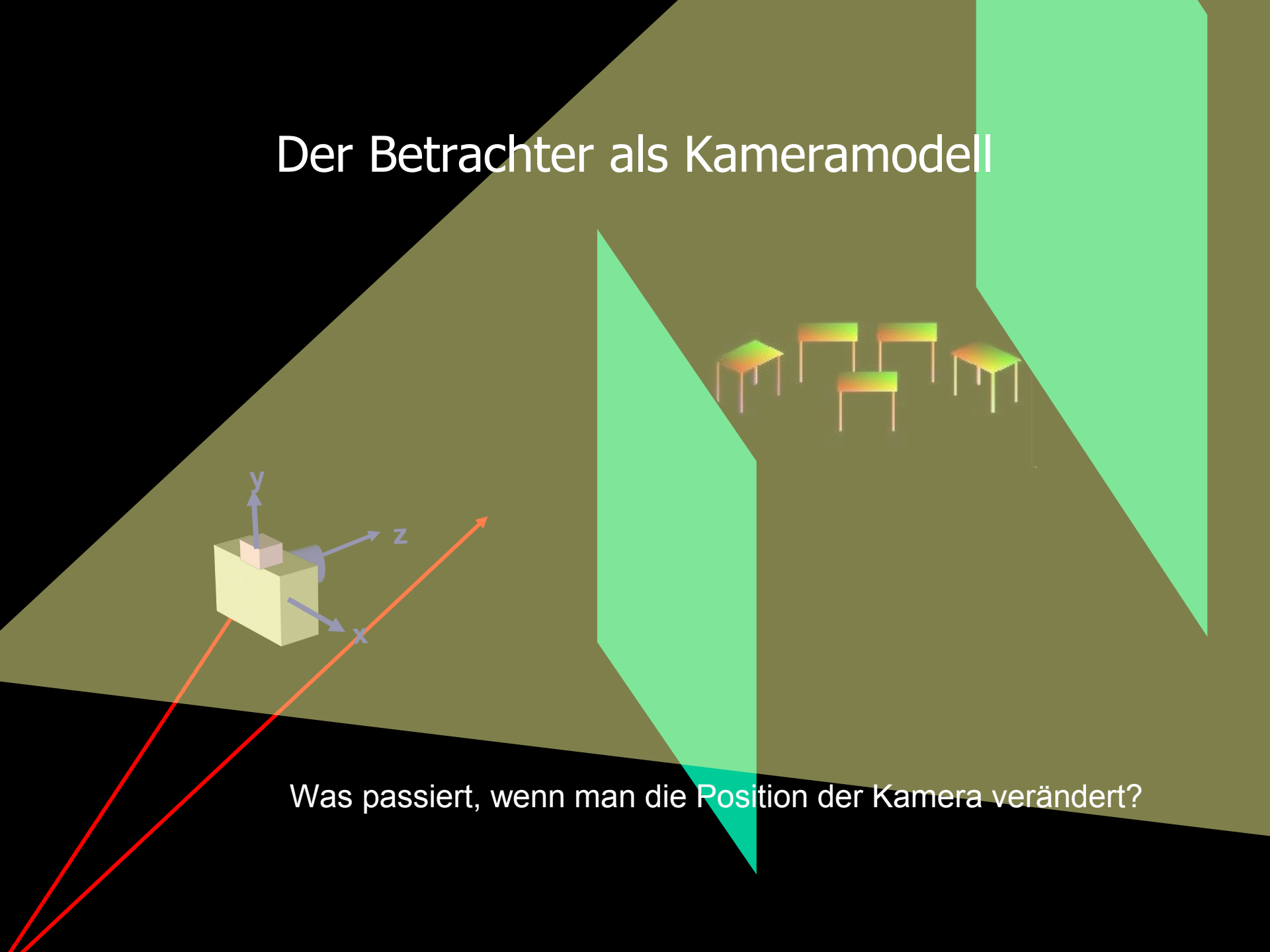
-Öffnungswinkel

-Clipping - Ebenen

Der Betrachter als Kameramodell „Perspektivische Projektion“

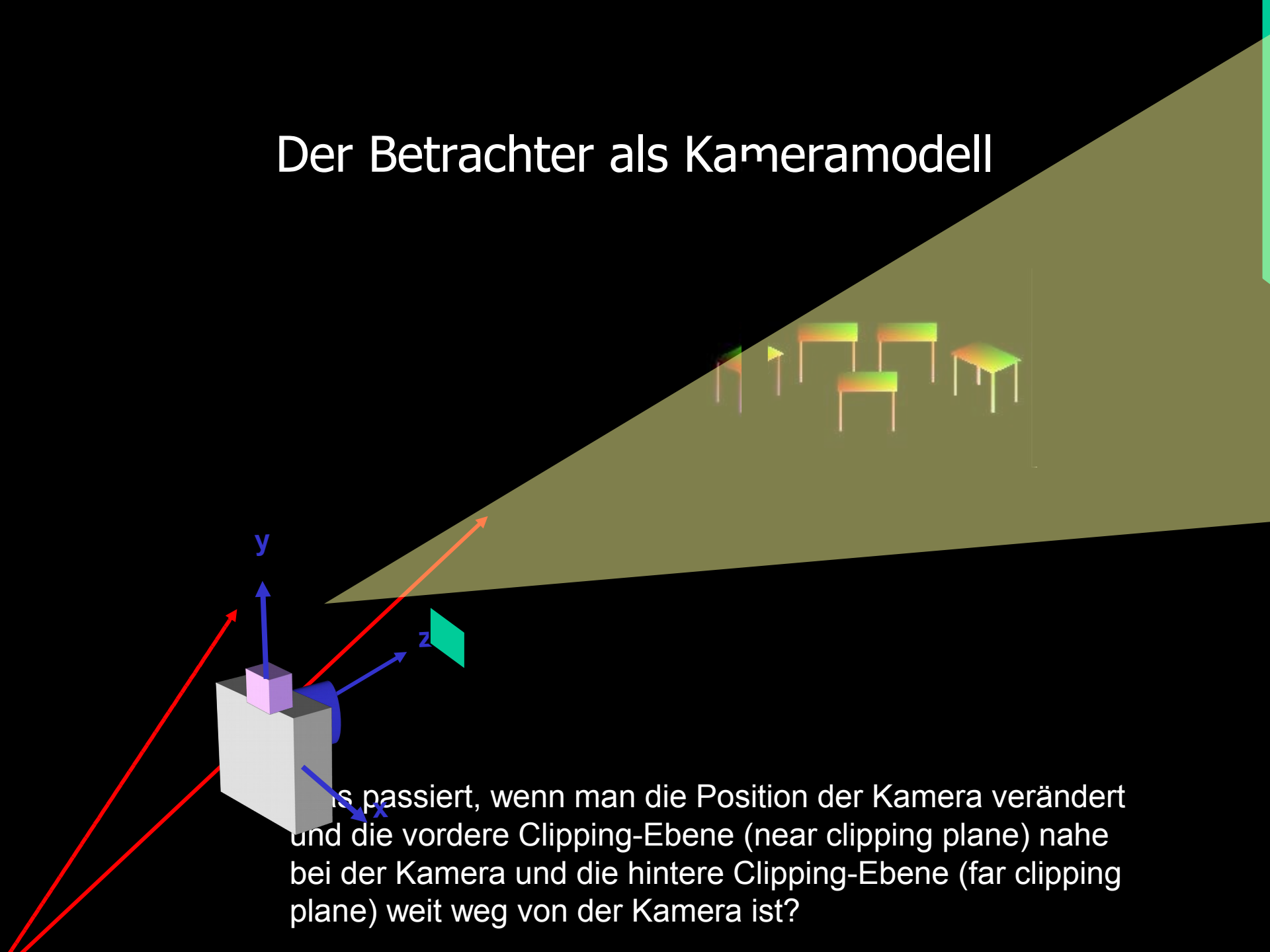


Der Betrachter als Kameramodell



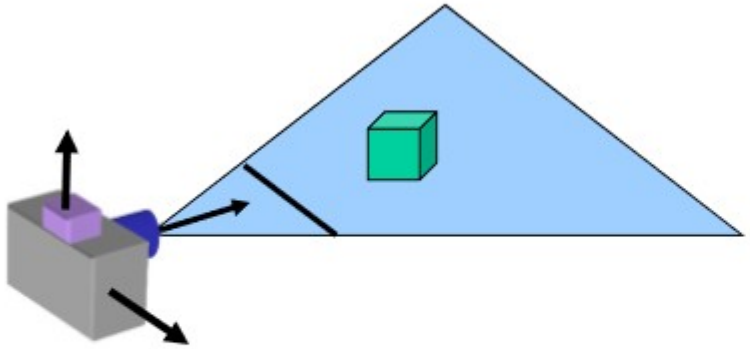
Was passiert, wenn man die Position der Kamera verändert?

Der Betrachter als Kameramodell

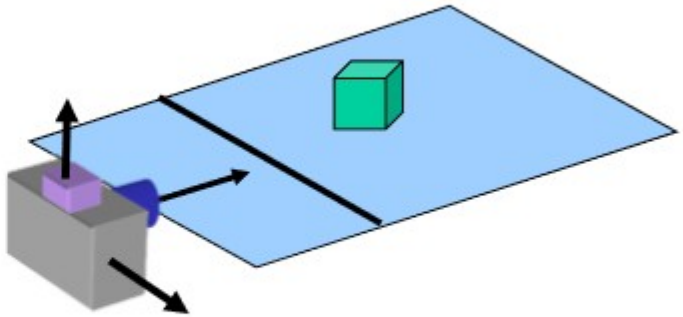


s passiert, wenn man die Position der Kamera verändert und die vordere Clipping-Ebene (near clipping plane) nahe bei der Kamera und die hintere Clipping-Ebene (far clipping plane) weit weg von der Kamera ist?

Projektionsmöglichkeiten



Perspektivische Projektion

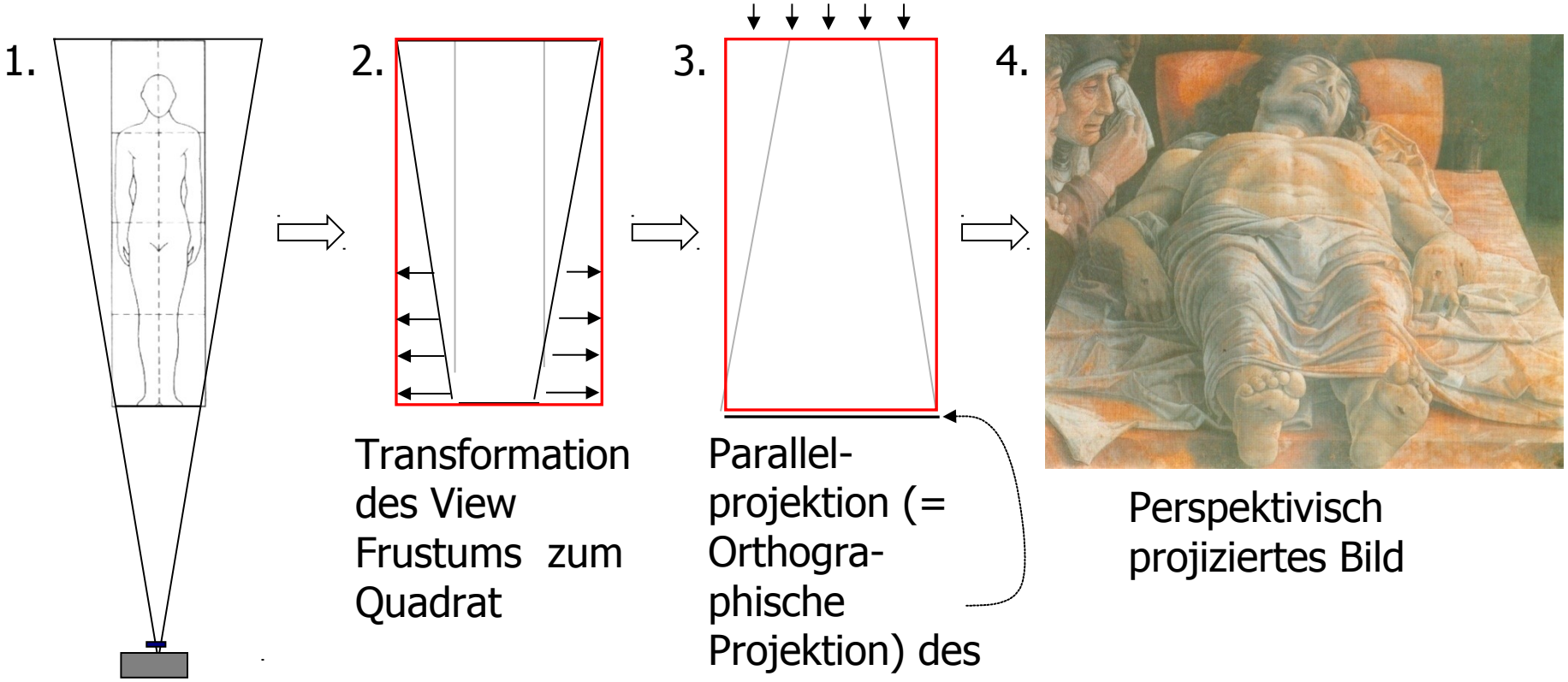


Parallel- bzw. orthographishe Projektion

Perspektivische Projektion

- „Natürliche Darstellung“
- Am meisten verwendete Projektionsart
- Sehstrahlen treffen sich in einem Fluchtpunkt

„Berechnung" der perspektivischen Projektion

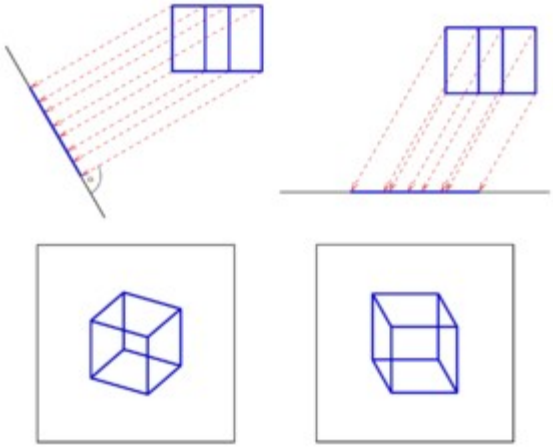


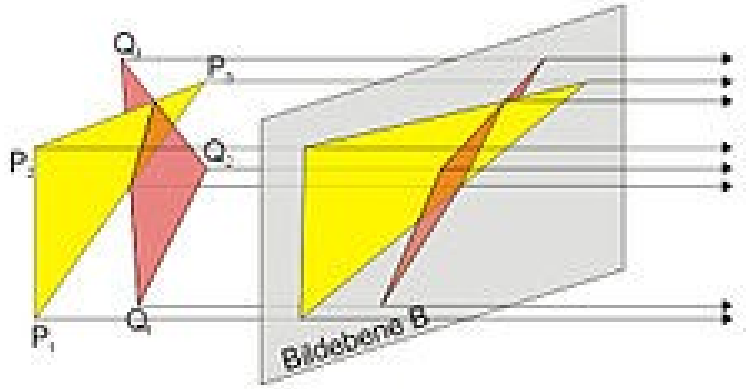
Projektionsverfahren

Setzen des Volumenraums und Berechnung der Projektionsmatrix geschieht durch:

Parallel Projektion (orthographische Projektion):

```
glOrtho(GLdouble left, GLdouble right,
  GLdouble bottom, GLdouble top,
  GLdouble near, GLdouble far);
```



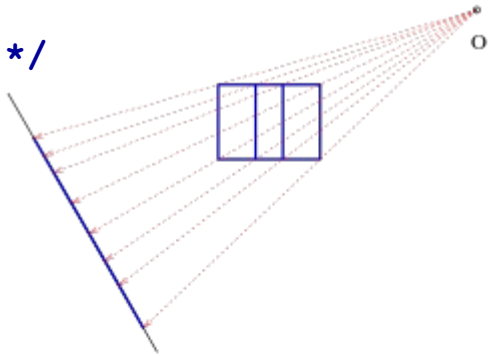


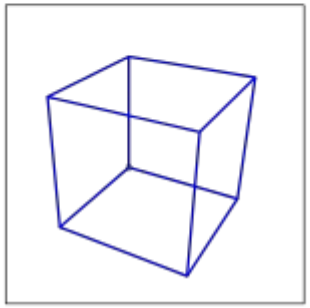
Projektionsverfahren

Setzen des Volumenraums und Berechnung der Projektionsmatrix geschieht durch:

Perspektivische Projektion:

```
gluPerspective( GLdouble fovy, /* vertikaler Öffnungswinkel */
 GLdouble aspect,    /* Breiten zu Höhenverhältnis */
 Gldouble near,      /* minimaler und */
 Gldouble far );     /* maximaler Abstand */
                     /* von der Kamera zum Frustum */
```





Der sich bewegende Betrachter

- Es gibt keinen expliziten Unterschied zwischen Kamera und Betrachter.
- Es bleibt dem Programmierer überlassen, ob der Betrachter sich um das Objekt bewegt oder ob sich das Objekt um den Betrachter bewegt. (Auswahl : Anwendungsbedingt)
- D.h. die Betrachtertransformation wird auch zu den Transformationen hinzugezählt.

Simulation des bewegten Betrachters in OpenGL

- OpenGL: Rechtshändiges Koordinatensystem
- Standardmäßig guckt die Kamera von Nullpunkt entlang der negativen Z-Achse.
- Um die Betrachterposition zu ändern, gibt es den folgenden GLUT-Befehl:

```
gluLookAt( Gldouble eyex, Gldouble eyey, Gldouble eyez,     /*Betrachterpos.*/
  Gldouble centerx, Gldouble centery, Gldouble centerz,     /*Blickziel*/
  Gldouble upx, Gldouble upy, Gldouble upz );               /*Oben-Betrachter*/
```

- gluLookAt setzt die äußeren Kameraparameter
- Transformationen des Betrachters werden in der Matrix GL_MODELVIEW gespeichert

Graphische Objekte und ihre Erzeugung

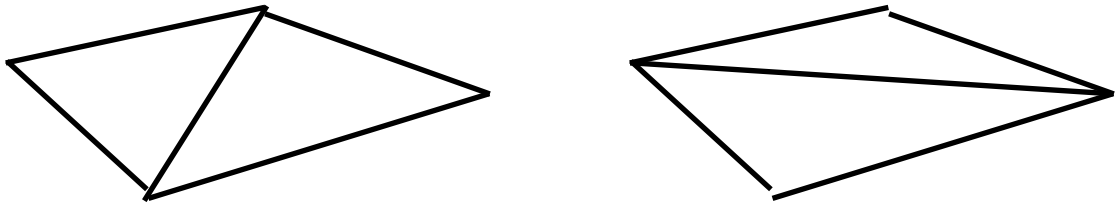
Koordinatensysteme der geometrischen Daten:

- Kartesisches Koordinatensystem
- Polarkoordinatensystem
- Zylinderkoordinatensystem
- Kugelkoordinatensystem

Geometrische Beschreibung der Objekte

Polygonale Darstellung:

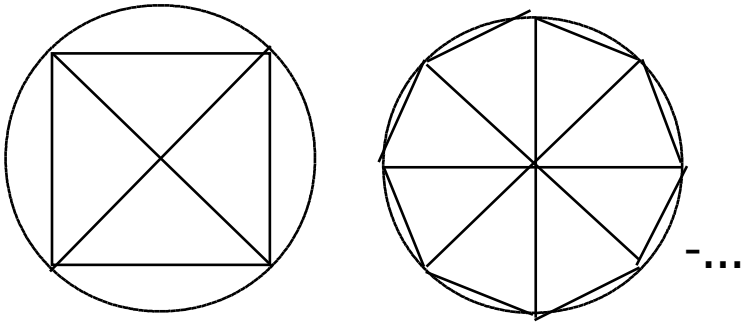
- Polygone (mehr als 3 Kanten) müssen eben sein.
- 4 Punkte können bereits eine unebene Fläche bilden.



Geometrische Beschreibung der Objekte

Alle Formen der darzustellenden Objekte werden durch Dreieckgitter angenähert!

Problembeschreibung am Beispiel der Modellierung einer runden Tischplatte:



Form des Tischplatte wird angenähert durch 4, 8, ... usw. Dreiecke

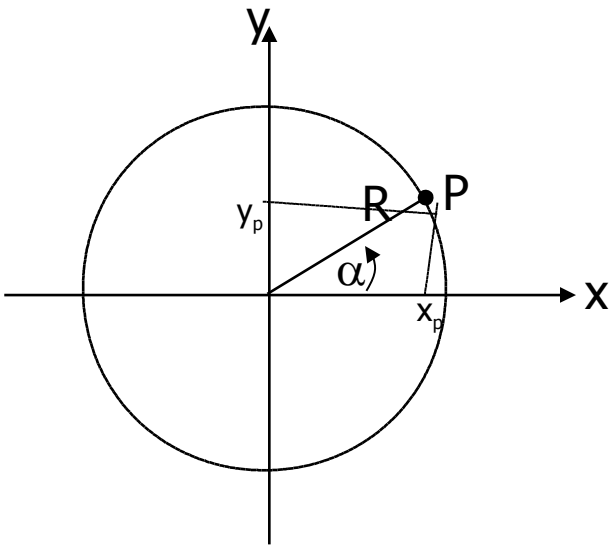
Besser: Kreise nicht durch einzelne Dreiecke definieren sondern durch eine Funktion beschreiben ....

Geometrische Beschreibung der Objekte

Definition der x- und y-Werte eines Kreis durch Funktionen:

$x_p = R * \cos(\alpha)$

$y_p = R * \sin(\alpha)$ mit $(0 <= \alpha <= 2\pi)$



Annahme: Mittelpunkt des Kreises liegt im Ursprung

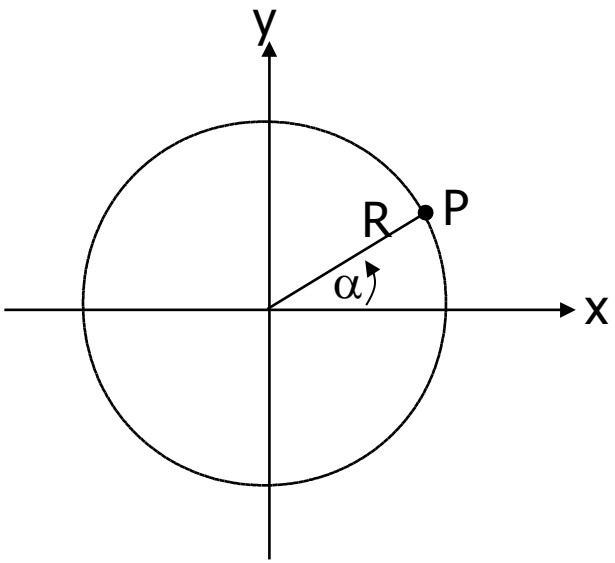
Geometrische Beschreibung der Objekte

Fazit:

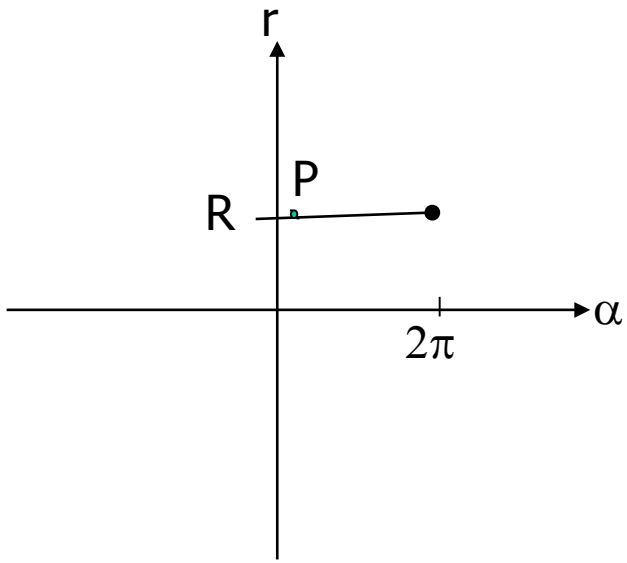
- Zur eindeutige Beschreibung der Kreisform genügt die Angabe des Radius (wenn der Mittelpunkt im Ursprung liegt)
- Mit der Angabe eines Winkelintervalls kann man nicht nur einen Kreis sondern auch einen Halbkreis, Viertelkreis und bei variablem Radius noch viele weitere zweidimensionale Geometrien definieren.

⇨ Polarkoordinatensystem

Geometrische Beschreibung der Objekte



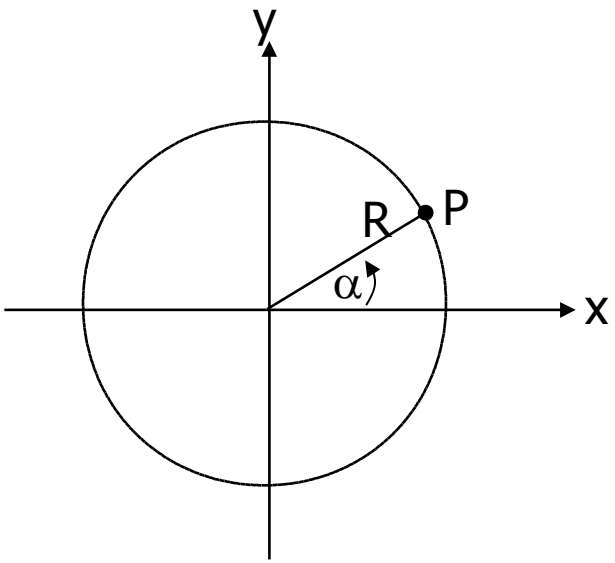
Kartesisches Koordinatensystem



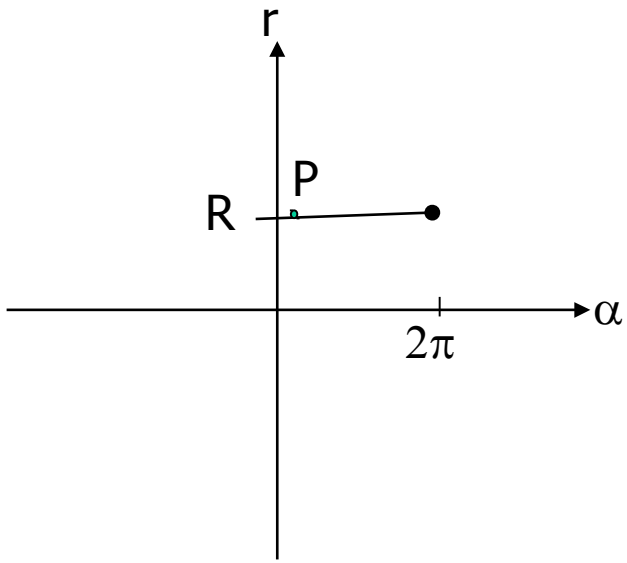
Polarkoordinatensystem

Beschreibung des Objektes erfolgt nicht in x- und y-Koordinaten sondern in r- (Radius) und α-Koordinaten (Winkel)

Geometrische Beschreibung der Objekte



Kartesisches Koordinatensystem



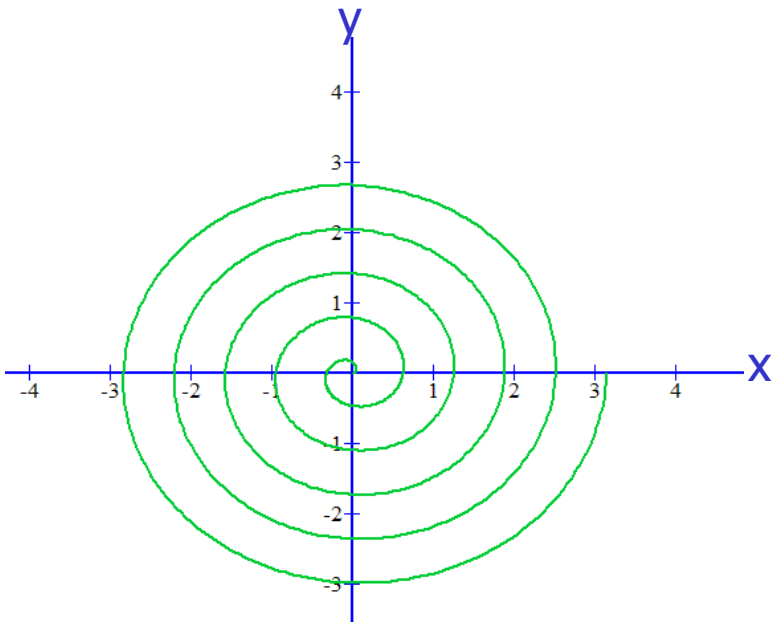
Polarkoordinatensystem

Umrechnung vom Polarkoordinatensystem ins Kartesische Koordinatensystem

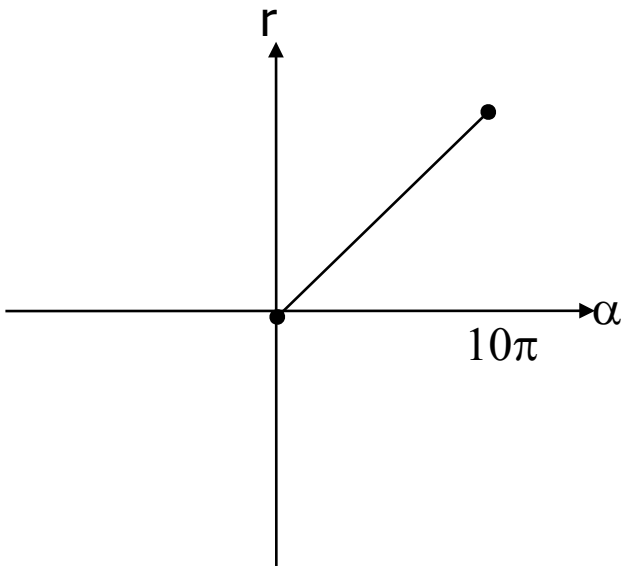
$x = R * \cos \alpha \qquad y = R * \sin \alpha$

Geometrische Beschreibung der Objekte

Polarkoordinaten werden zur Beschreibung von Objekten verwendet, die durch die Drehung entlang einer Achse erzeugt werden können: Bsp. spiralförmige Objekte

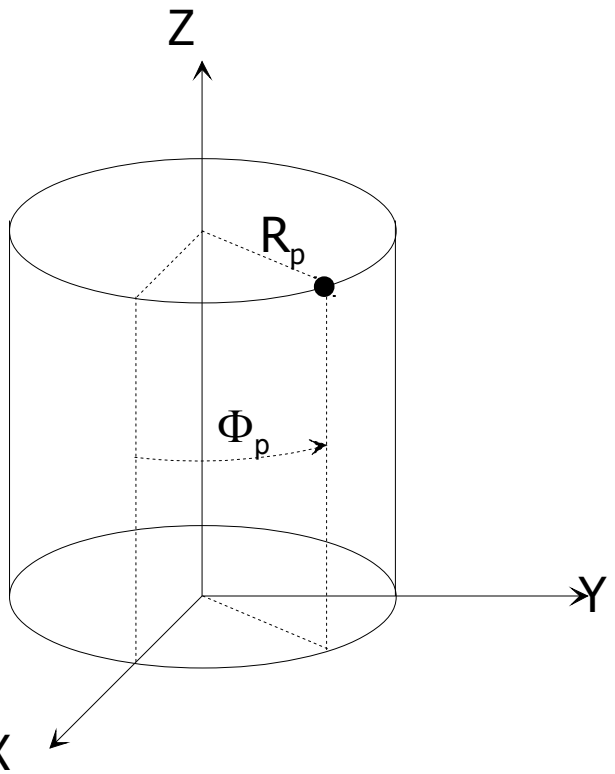


Kartesisches Koordinatensystem

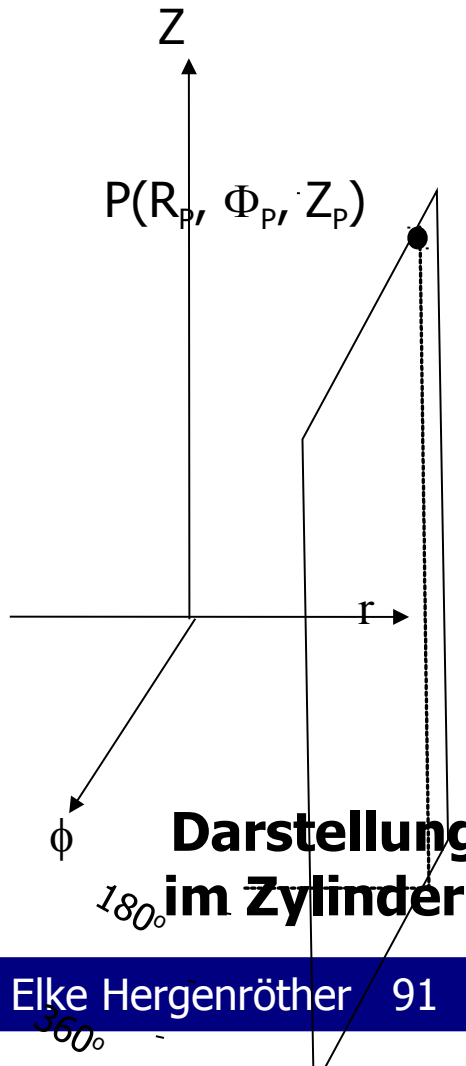


Polarkoordinatensystem

Geometrische Beschreibung der Objekte



Kartesisches Koordinatensystem



Darstellung des Zylindermantels im Zylinderkoordinatensystem

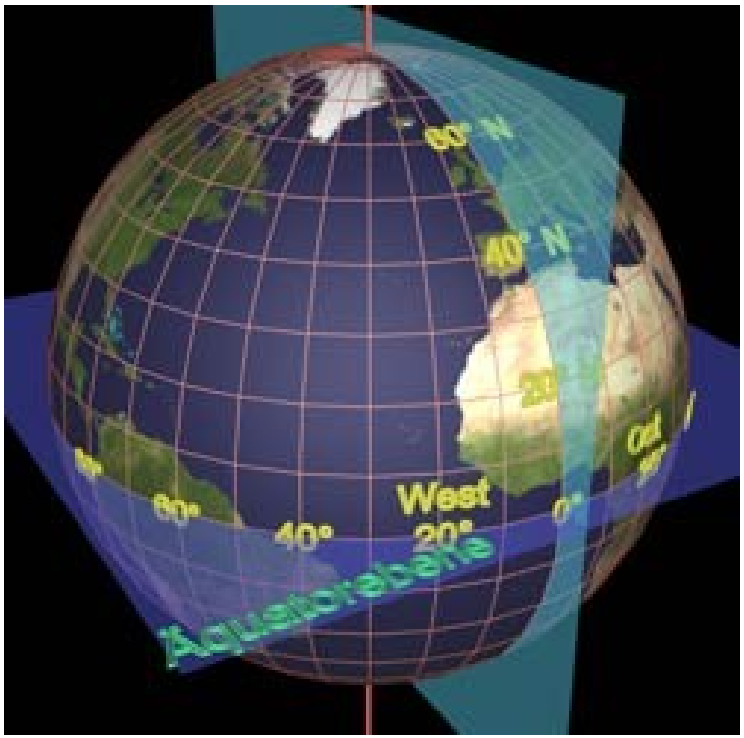
Umrechnung vom Zylinderkoordinaten-system ins Kartesische Koord.

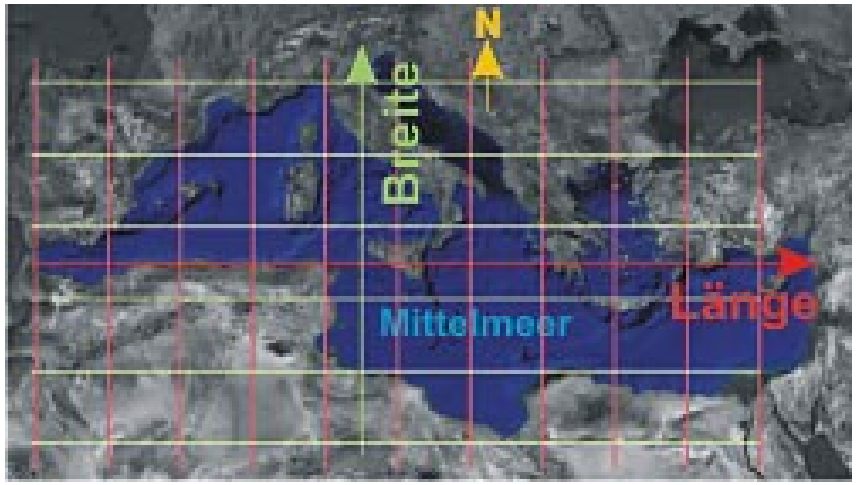
$x = R * \cos \Phi$

$y = R * \sin \Phi$

$z = z$

Motivation Kugelkoordinatensystem

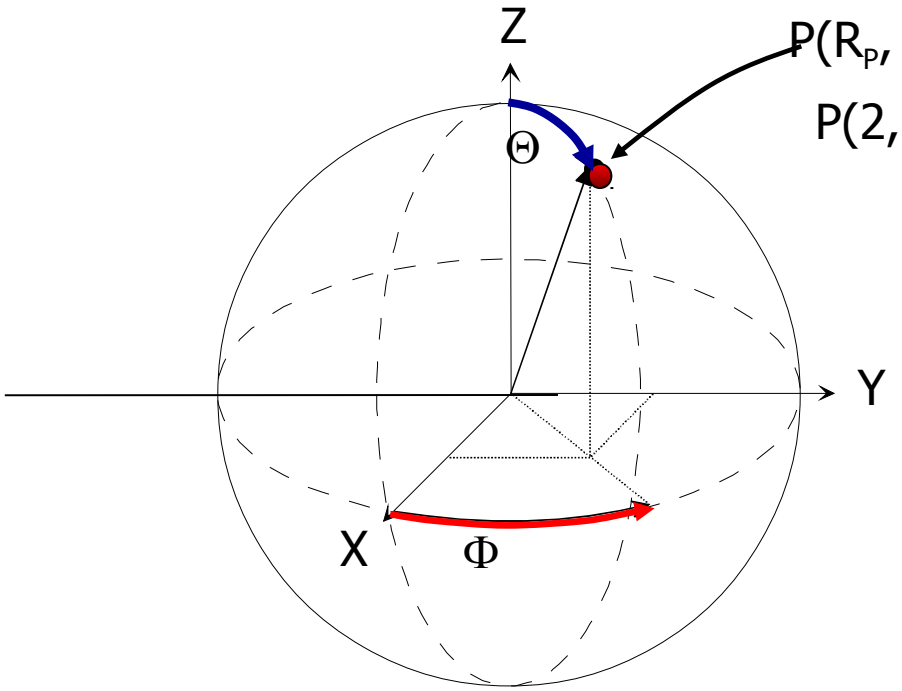




Wieviele Parameter benötigt man um einen Punkt im Kugelkoordinaten-system eindeutig bestimmen zu können?

Geometrische Beschreibung der Objekte

Vom kartesischen Koordinatensystem ins Kugelkoordinatensystem



$P(R_P, \Phi_P, \Theta_P) =$

$P(2, 45^0, 20^0)$

$R_P \in R$

$\Phi_P \in [0, 2\pi]$

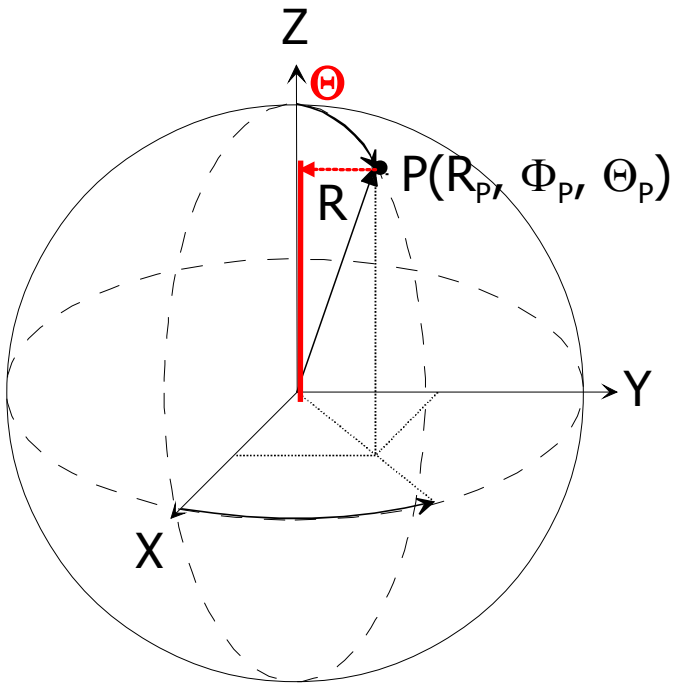
$\Theta_P \in [0, \pi]$, warum?

Wie sieht die Kugeloberfläche im Kugelkoordinatensystem aus?

Geometrische Beschreibung der Objekte

Umrechnung vom Kugelkoordinaten-system ins kartesische Koordinatensystem:

z = R * cos Θ



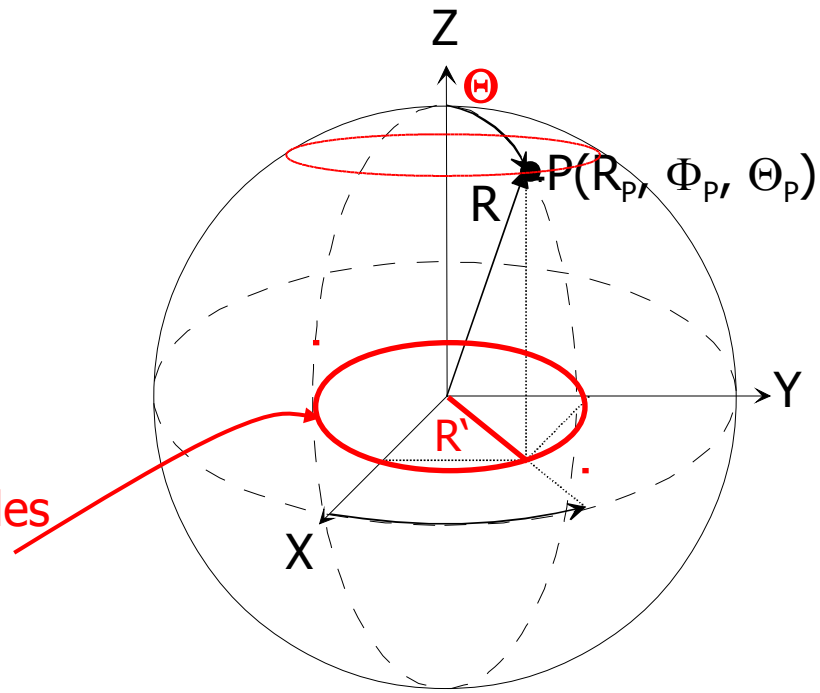
Geometrische Beschreibung der Objekte

Umrechnung vom Kugelkoordinatensystem ins kartesische Koordinatensystem:

$z = R * \cos \Theta$

Hilfsradius zur Bestimmung des Breitenkreises :

$R' = R * \sin \Theta$



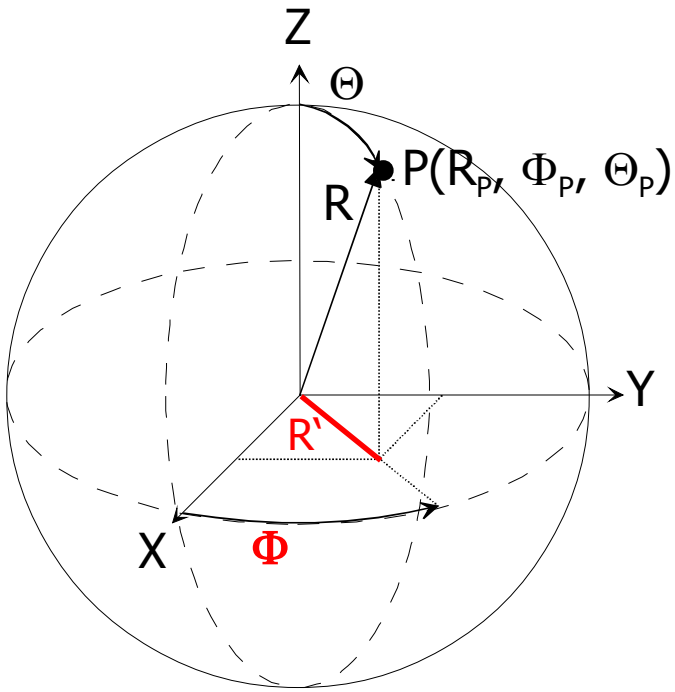
Geometrische Beschreibung der Objekte

Umrechnung vom Kugelkoordinatensystem ins kartesische Koordinatensystem:

x = R‘ * cos Φ

y = R‘ * sin Φ

z = R * cos Θ



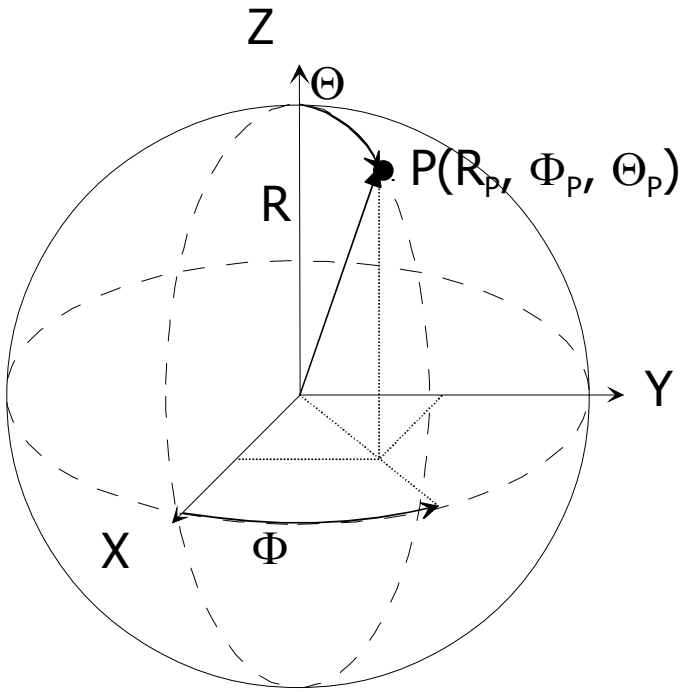
Geometrische Beschreibung der Objekte

Umrechnung vom Kugelkoordinatensystem ins kartesische Koordinatensystem:

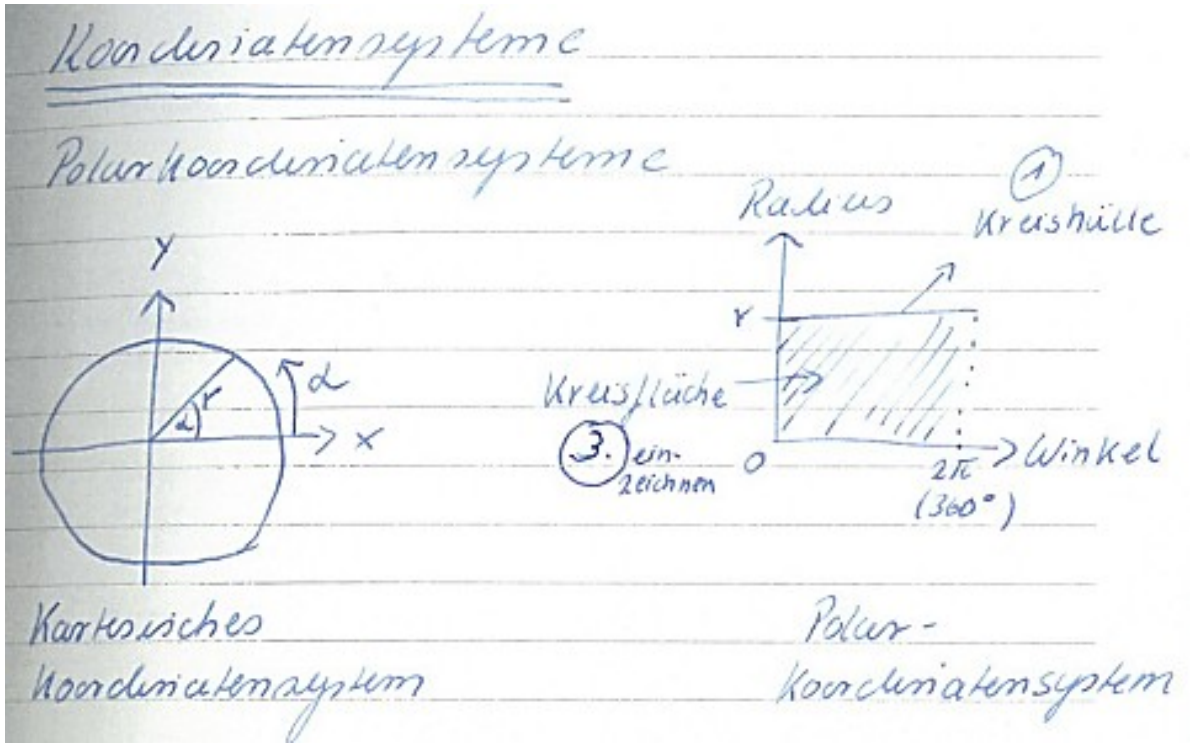
$x = R * \sin \Theta * \cos \Phi$

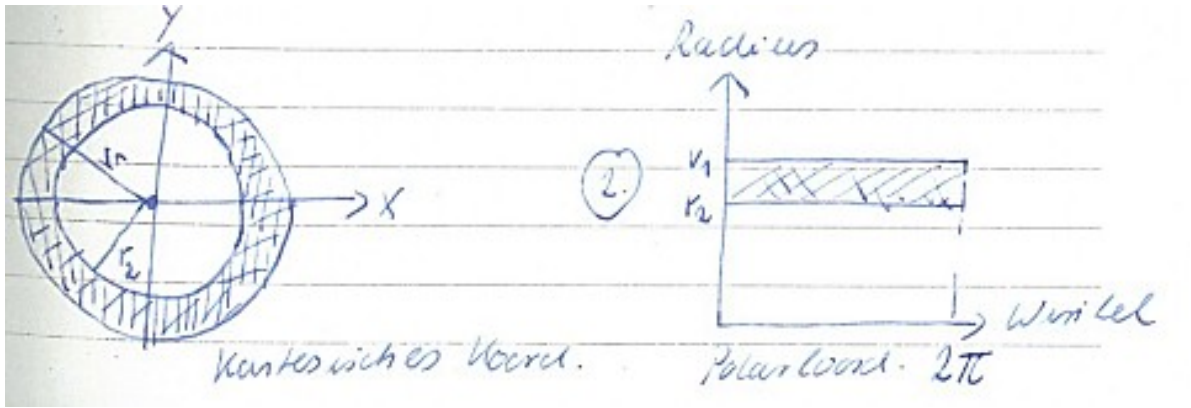
$y = R * \sin \Theta * \sin \Phi$

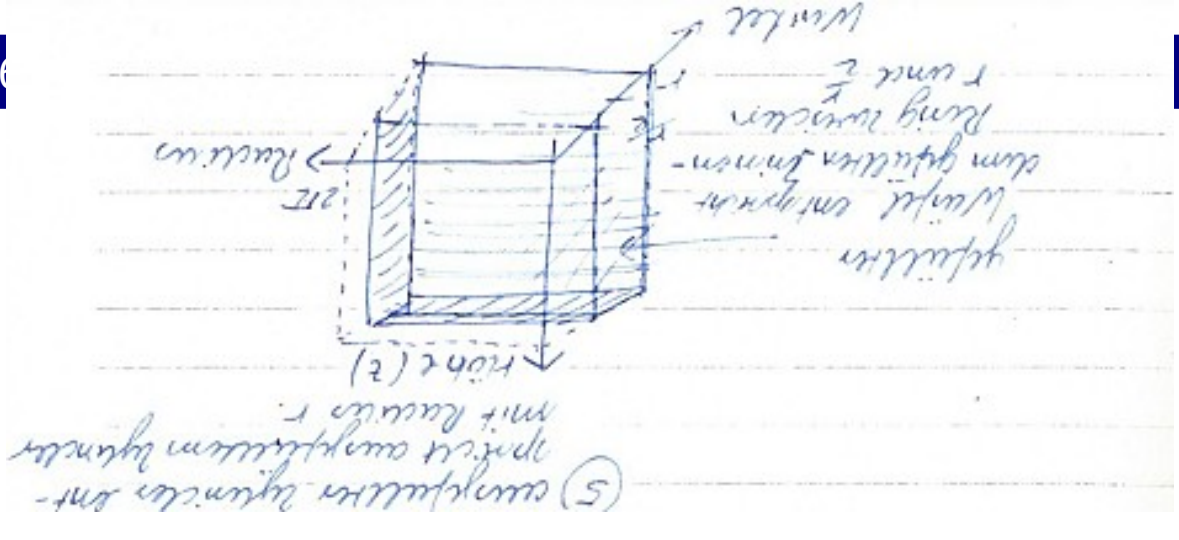
$z = R * \cos \Theta$

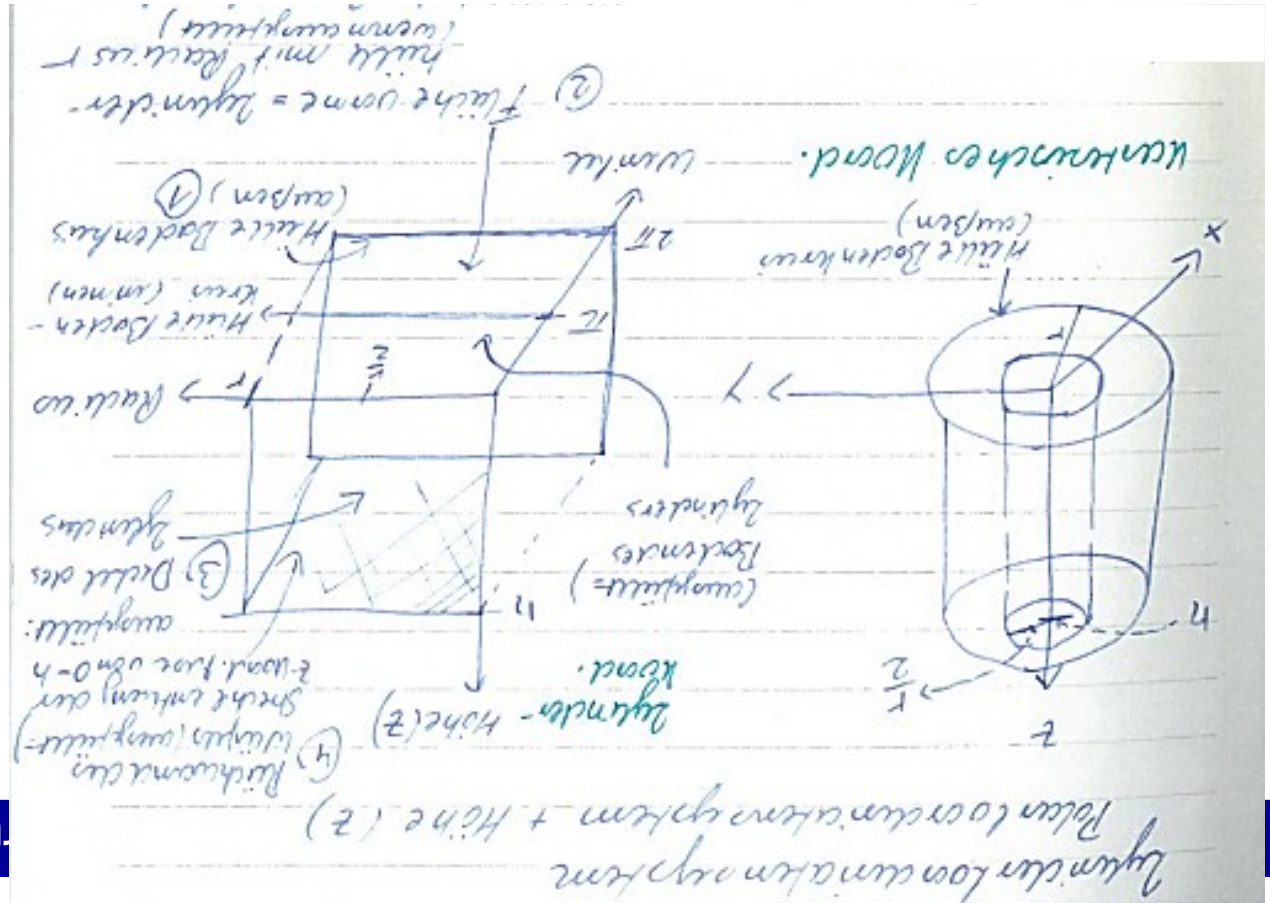


Polar koor dinat ensy stem - Notiz en

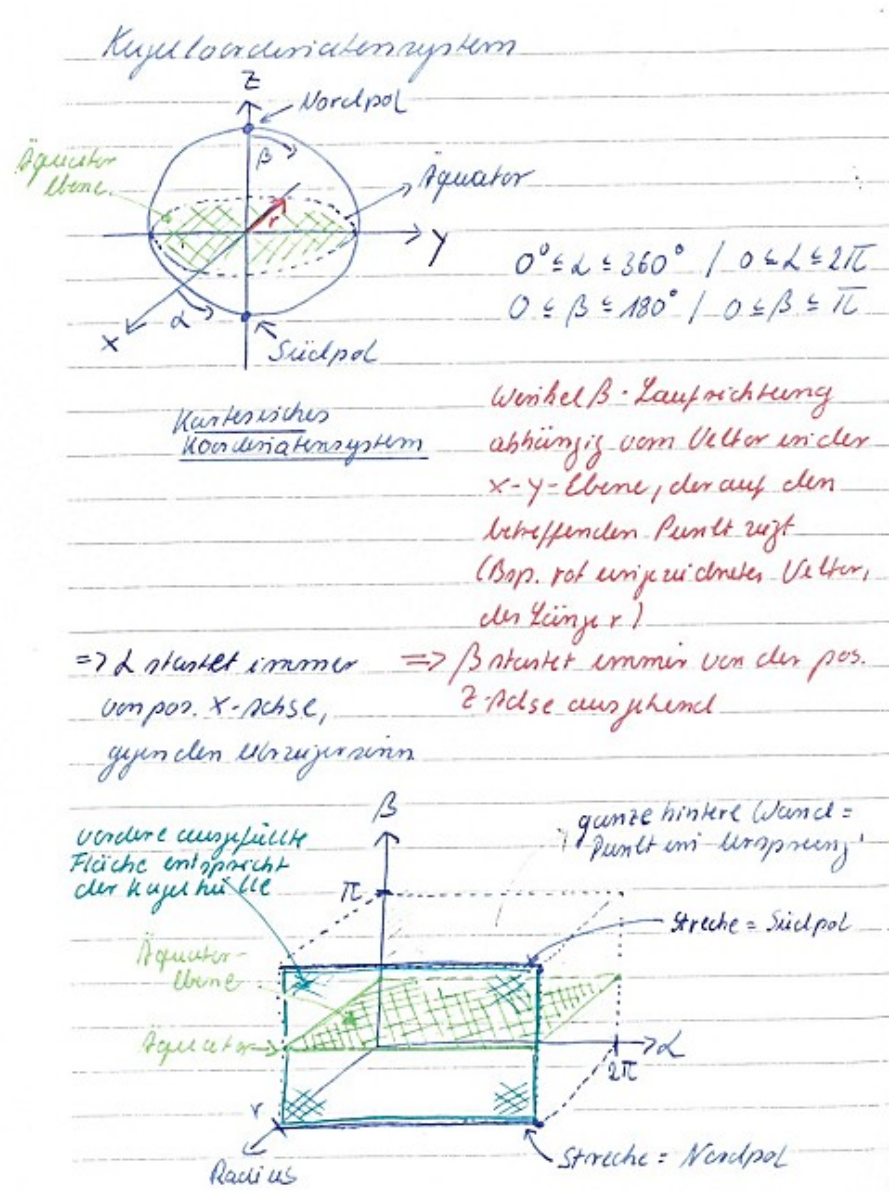


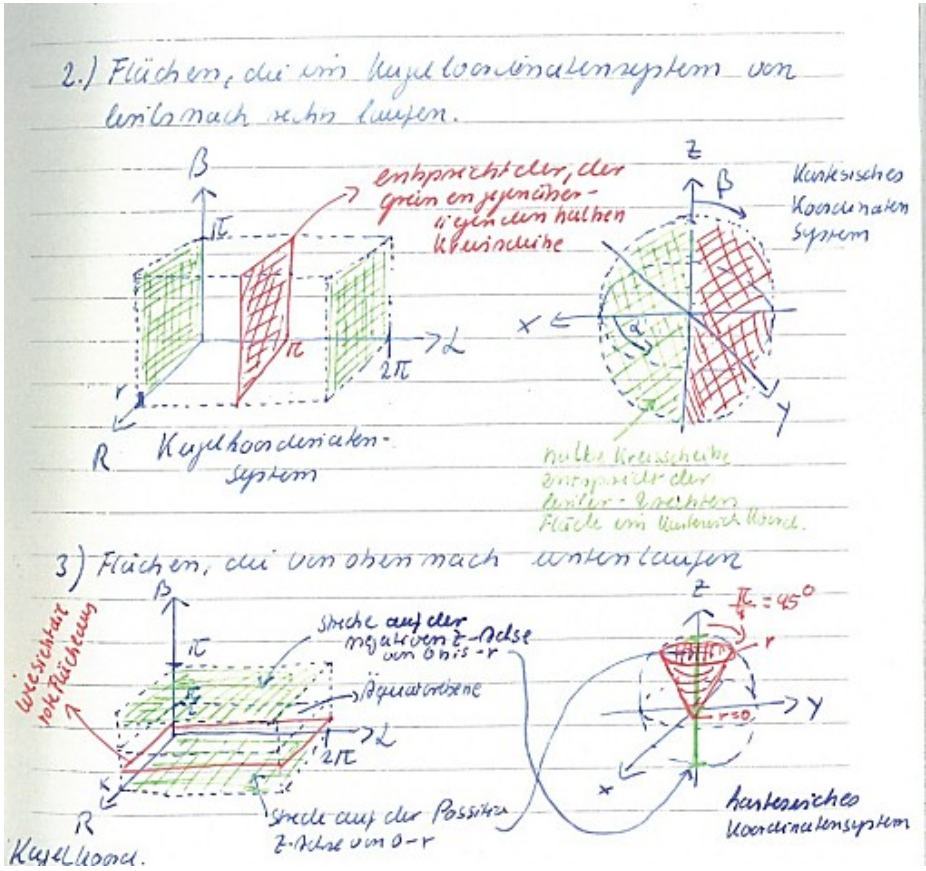






Zylinderkoordinatensystem - Notizen





Notizen zum Kugelkoordinatensystem