Graphische Datenverarbeitung

Graphische Objekte und deren Programmierung

Prof. Dr. Elke Hergenröther

h_da

Wichtige Hinweise:

- Besuchen Sie die Vorlesung!
- Warum?
  - Das Skript und die Folien sind nur eine Zusammenfassung
  - Tafel- und Hörsaalübungen sind wichtige Hilfen für die Klausur
  - Fragen, auch die der anderen Studierenden, helfen Ihnen beim Verständnis

- Bereiten Sie den Stoff vor bzw. nach!
- Warum?
  - Die Klausur ist keine „Auswendiglernklausur"
  - Sie müssen den Stoff wirklich beherrschen, um ihn dann in der Klausur richtig anzuwenden.

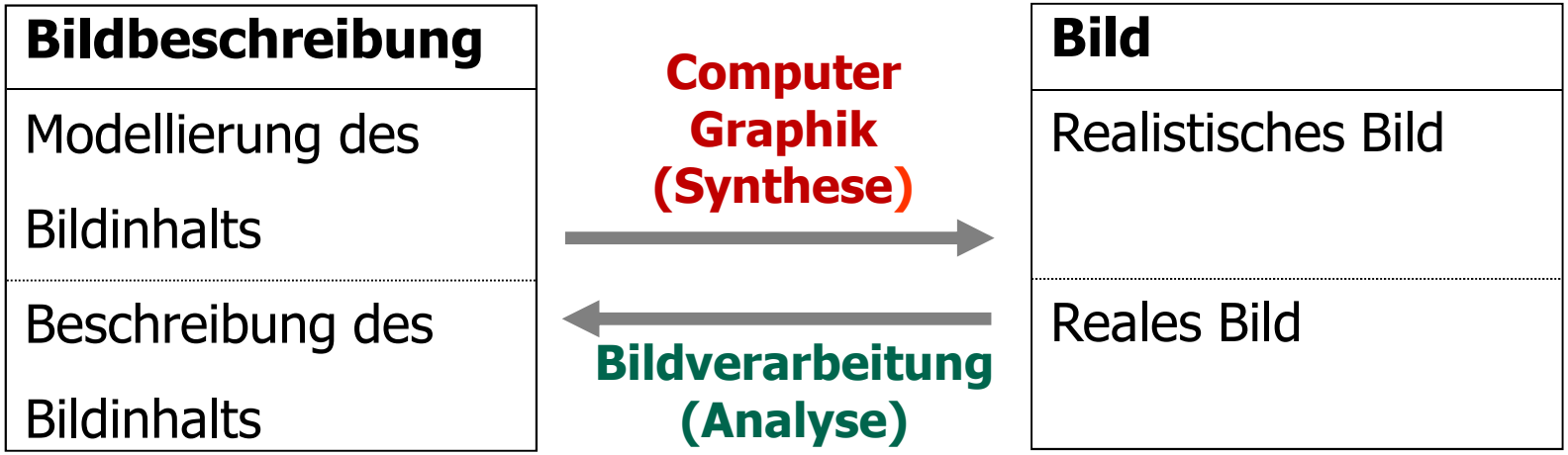
Inhalte der Vorlesung GDV

1. **Computer Graphik**
   - Graphische Programmierung (OpenGL)
   - Koordinatensysteme
   - Visualisierungstechniken
   - Geometrische Transformationen

2. **Bildverarbeitung**
   - Bildbearbeitung (Bild heller machen, etc.)
   - Bildverarbeitung (Bsp. Information aus einem Bild extrahieren)
   - Kompressionsverfahren (Bsp. JPEG)
   - Farbmodelle

3. **Graphik Hardware, etc.:** immer wenn Zeit ist.

Klassische Aufteilung zwischen Computer Graphik und und Bildverarbeitung



Beispiel für Computer Graphik: Ray Tracing

Bildanalyse am Beispiel eines Convolutional Neuronal Network (CNN)

Convolution == Faltung (Begriff aus der Bildverarbeitung)

Bild aus: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

Convolutional Neuronal Network (CNN)

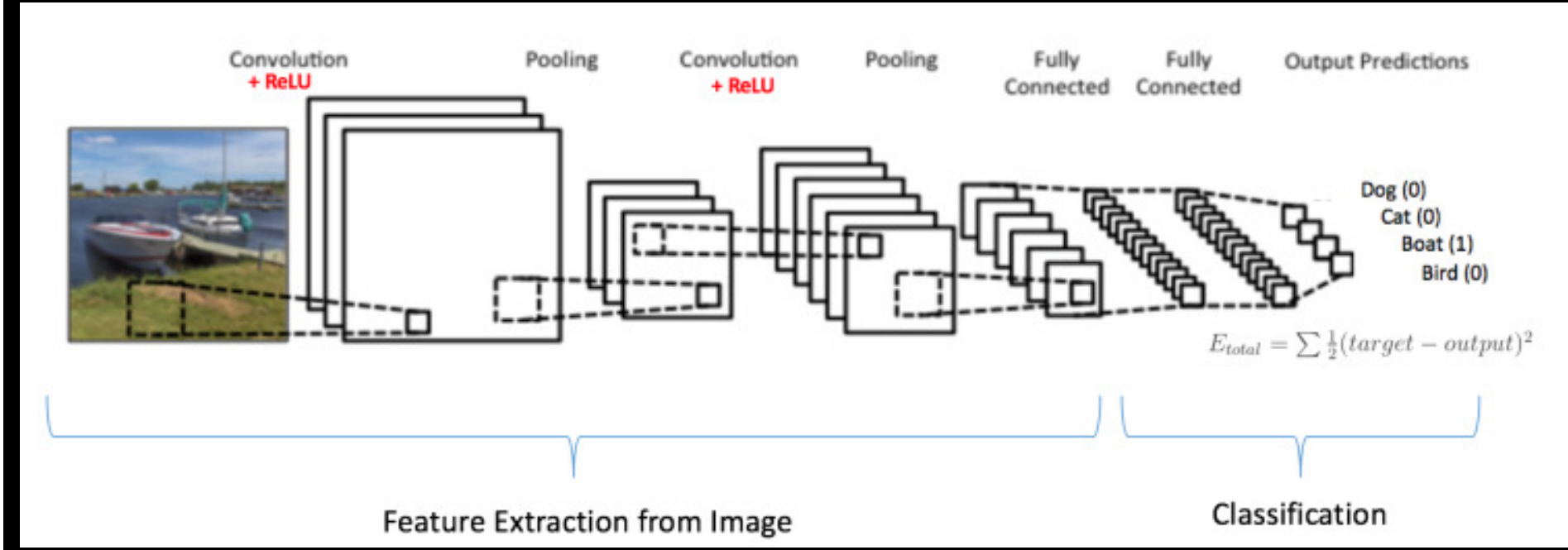


Bild aus: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

Visualisierung der Funktionsweise eines CNN

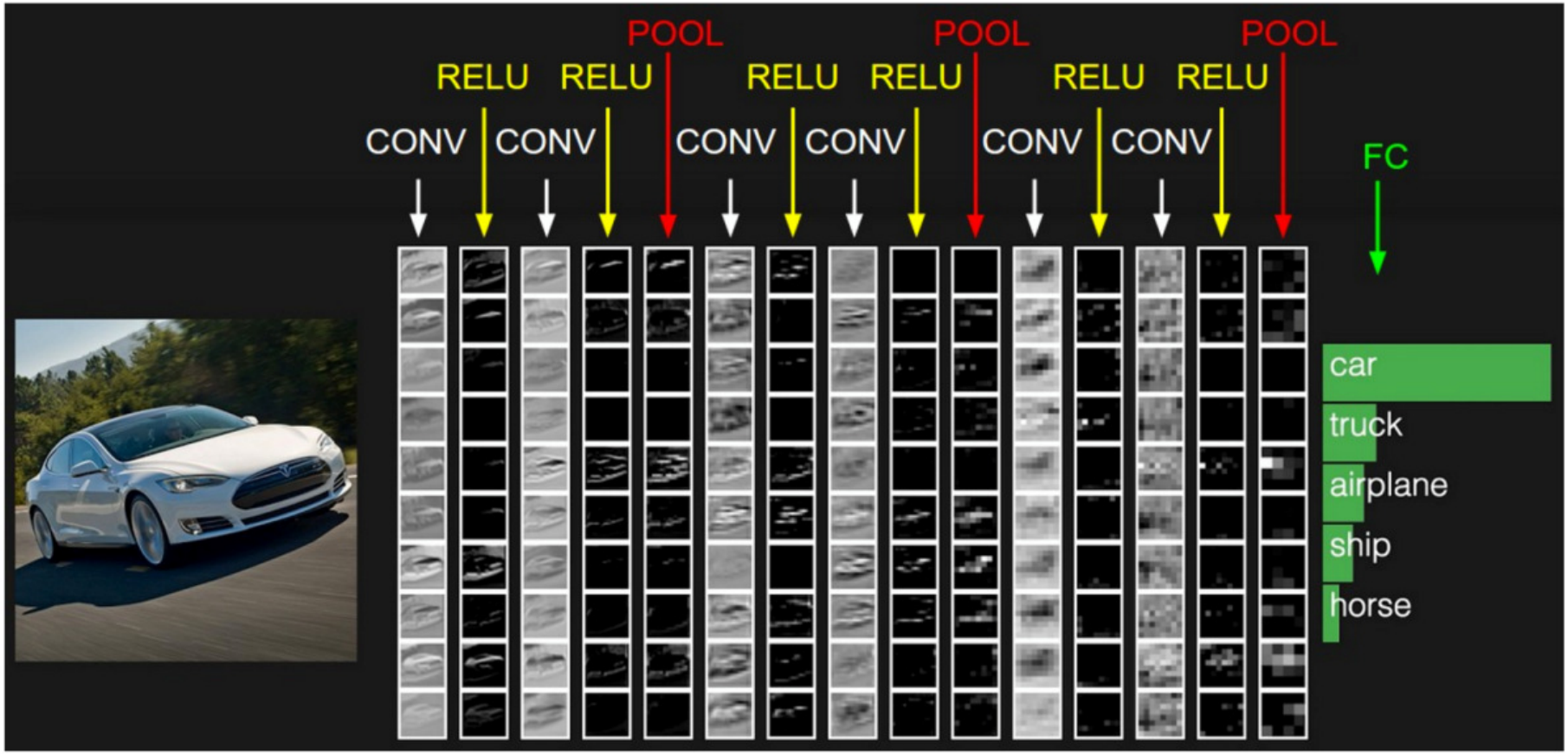
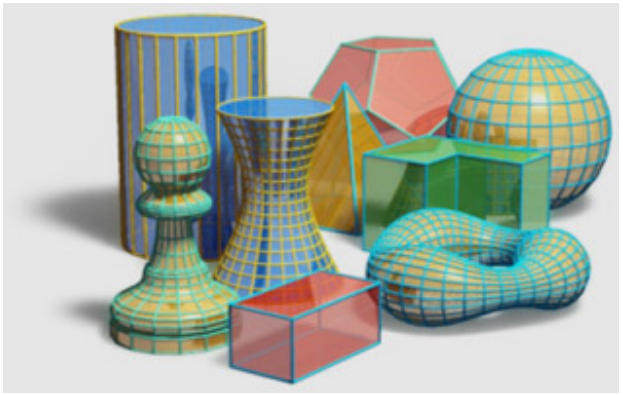
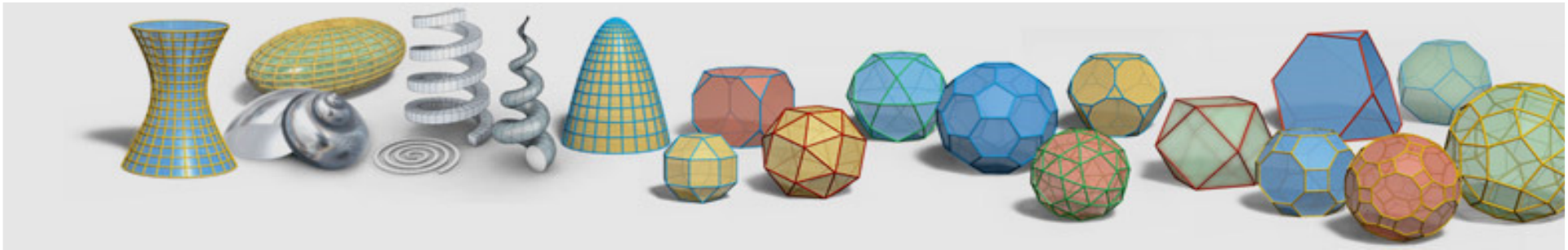


Bild aus: http://cs231n.github.io/convolutional-networks/



Computer Graphik

- Algorithmen zur Erzeugung von Geometrien
- Algorithmen zur „Interaktion“ mit Geometrien
- Algorithmen zur Visualisierung von Geometrien (inkl. Beleuchtungsberechnung, Stereodarstellung, …)
- Shader Technologie
- …



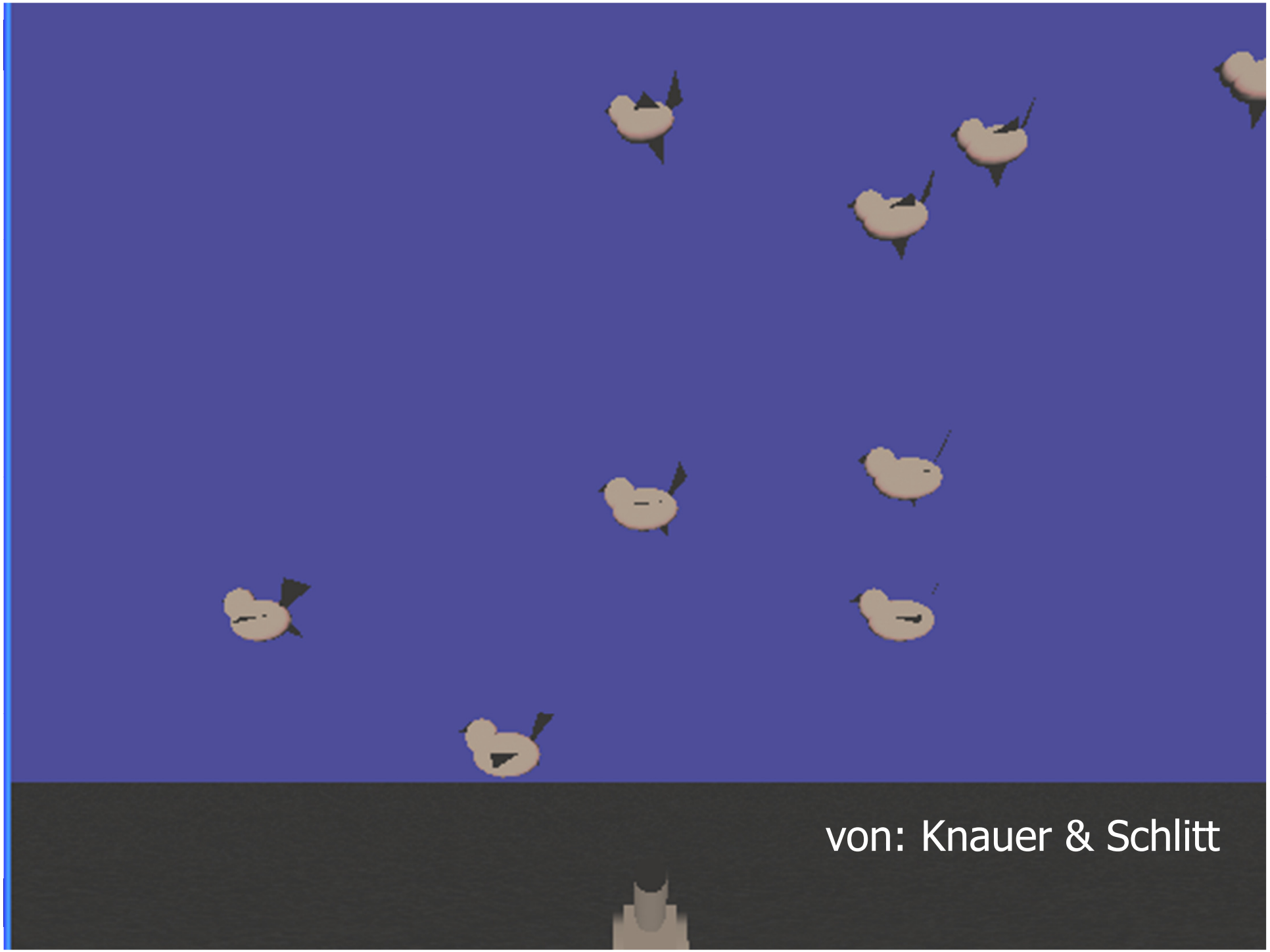
Bilder entnommen aus: http://www.pytha.de/produkt/modeler_1.de.php

… und das machen wir im Praktikum

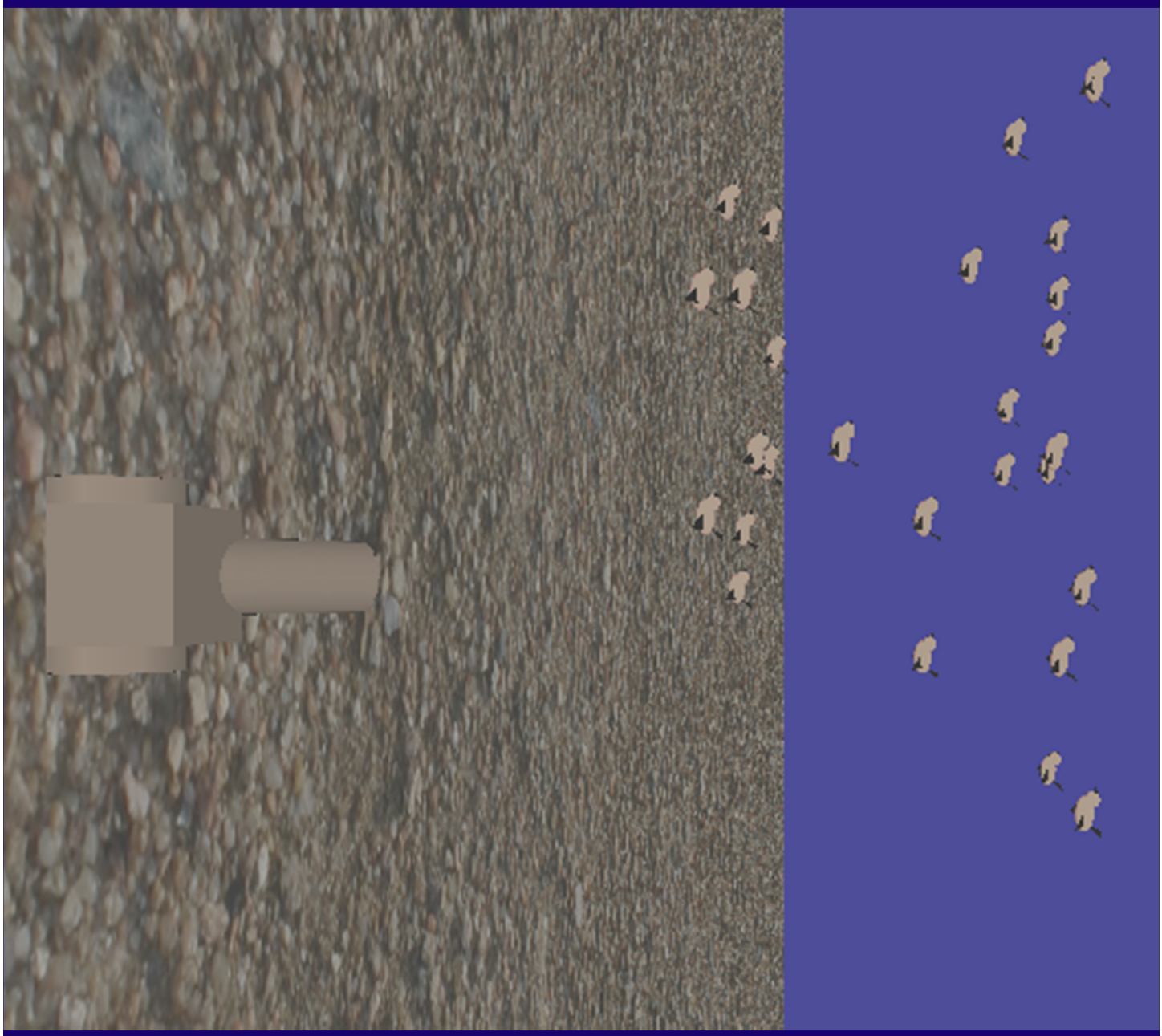
- Teil 1: Graphische Programmierung
  - Ersten drei Termine
  - Besonders gute Arbeiten werden mit bis zu 3,5 Zusatzpunkten belohnt

- Teil 2: Bildberarbeitung: Farbmodelle, Kompression, etc.
  - Nutzen Sie es zur Klausurvorbereitung
  - Bearbeiten der Zusatzaufgaben bringt bis zu 2,5 Zusatzpunkte


Window-Title
von: Fournol & Lüth

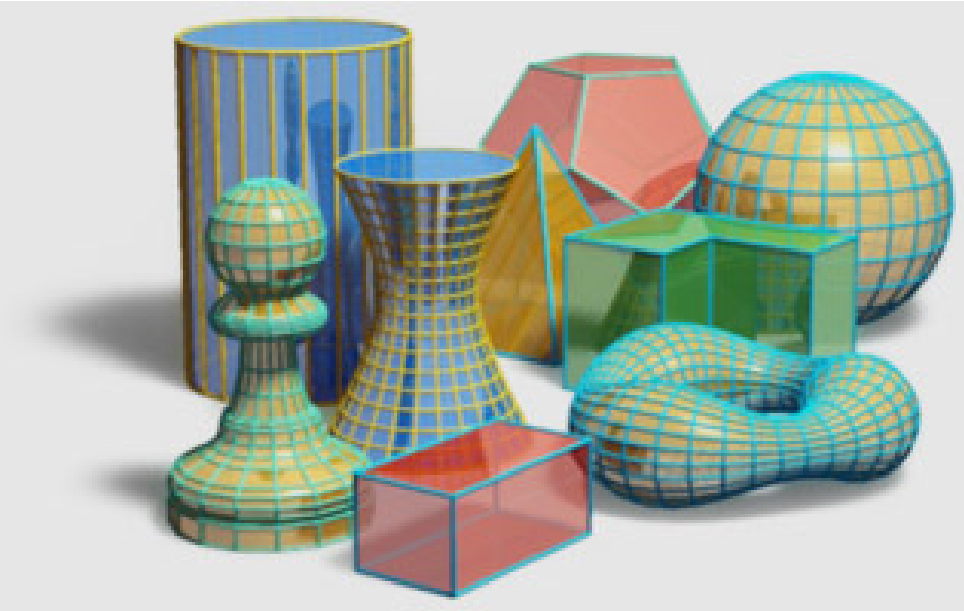

von: Knauer & Schlitt

von: Knauer & Schlitt



Graphische Objekte und graphische Programmierung

1. Schritt: Erzeugung graphischer Objekte



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

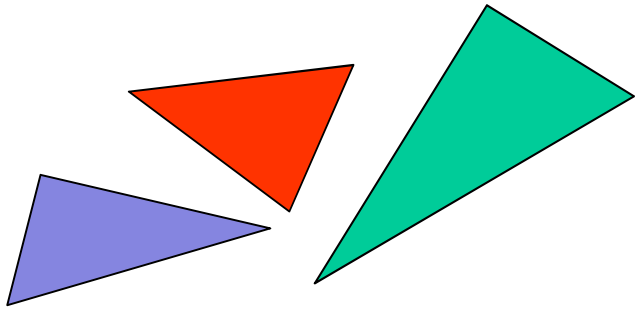
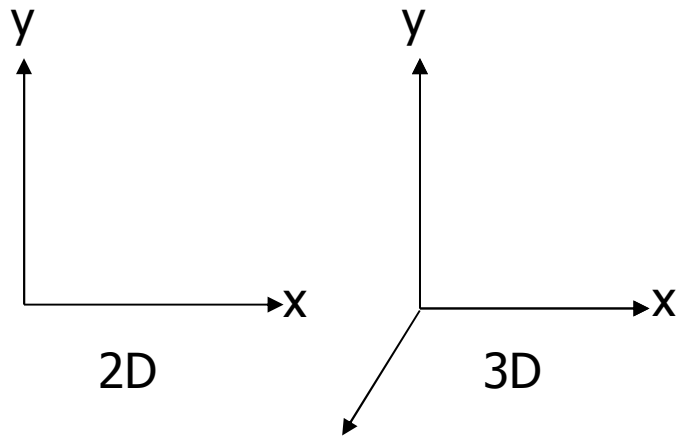
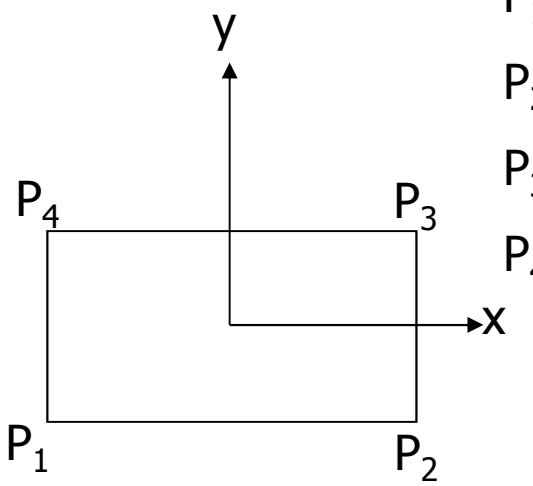


Bild entnommen aus: http://www.pytha.de/produkt/modeler_1.de.php

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



Kartesische Koordinatensysteme



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

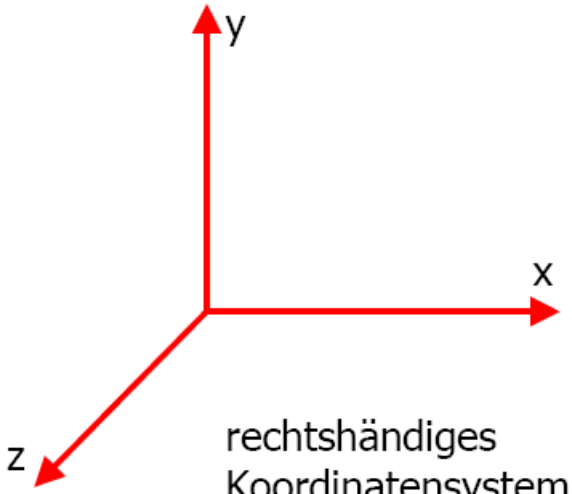
$P_2 = \{ 2.4, -1.2\}$

$P_3 = \{ 2.4, 1.2\}$

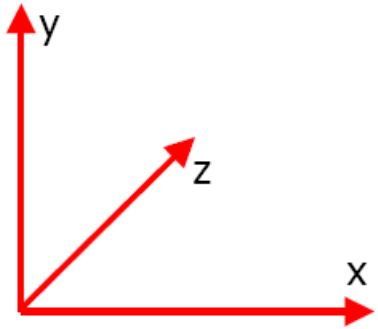
$P_4 = \{-2.4, 1.2\}$

Definition eines Rechtecks in kartesischen Koordinaten

Kartesischen Koordinatensystem

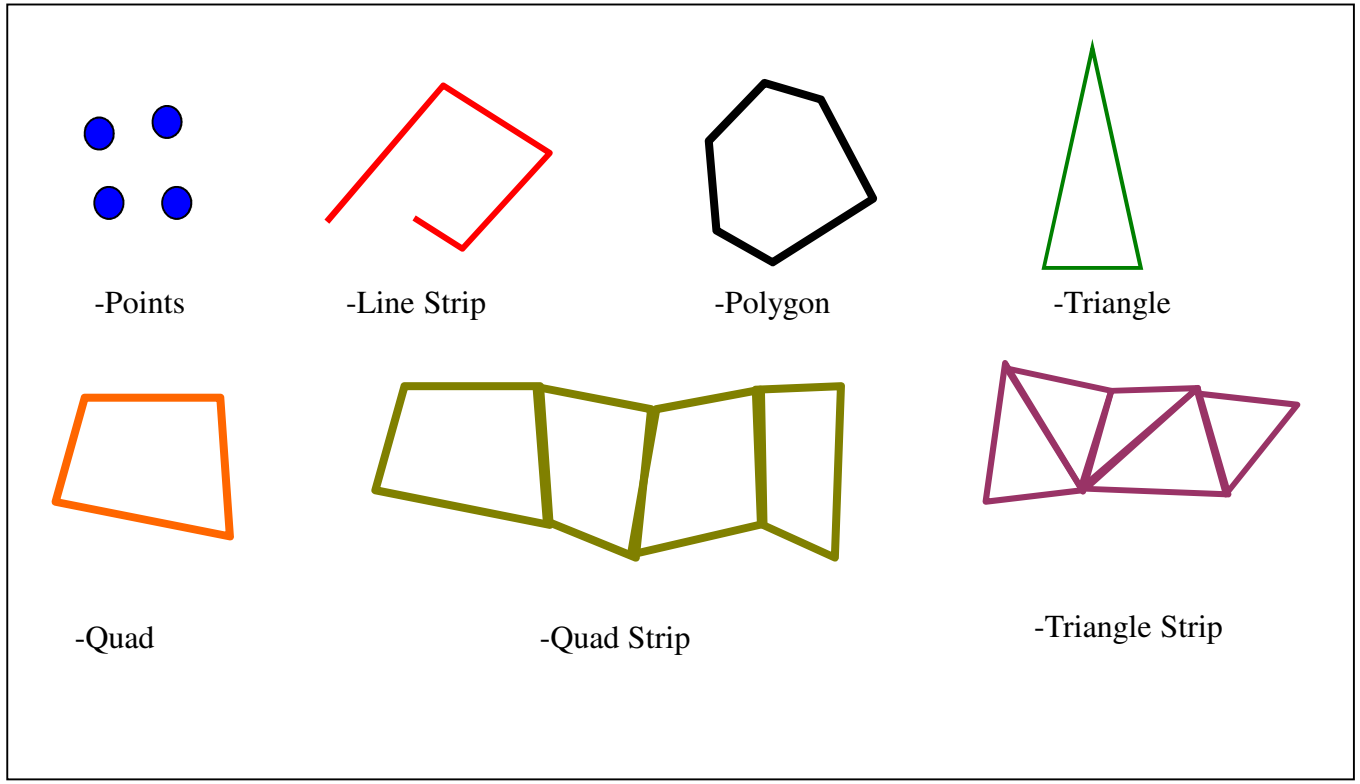


rechtshändiges
Koordinatensystem

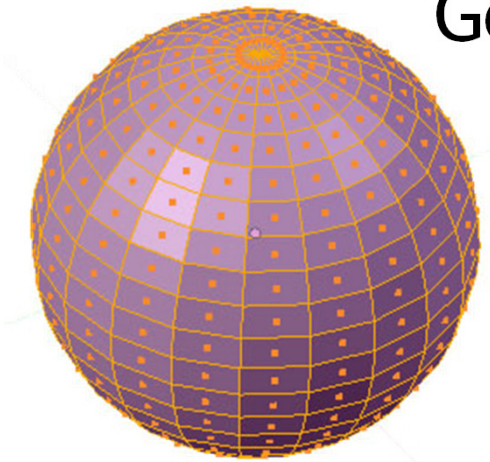


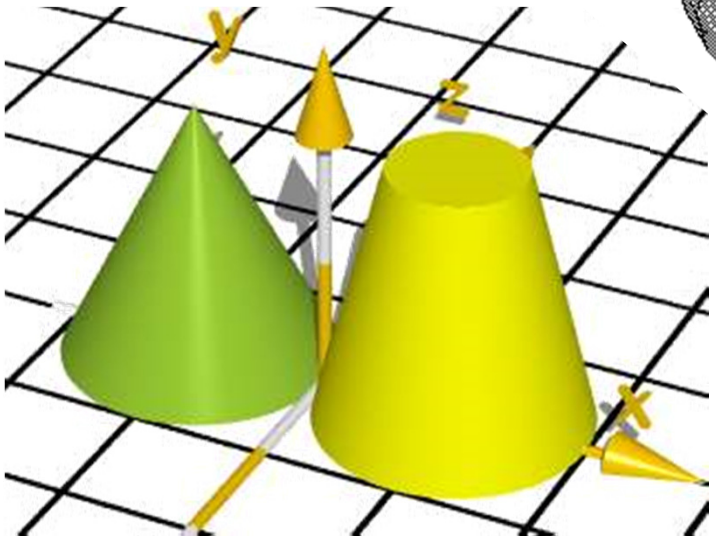
linkshändiges
Koordinatensystem

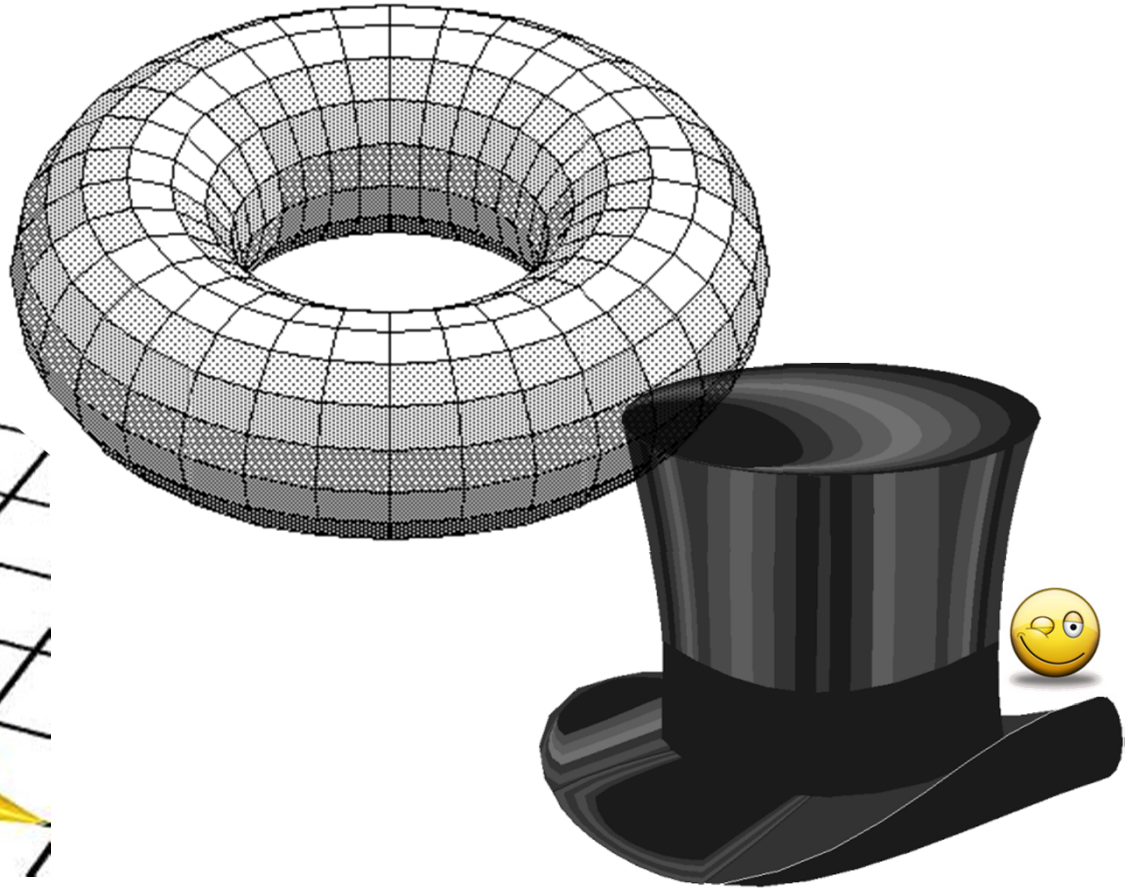
OpenGL verbindet die Punkte zu best. Primitiven



… Hilfs-Bibliotheken stellen komplexere Geometrieen zur Verfügung:

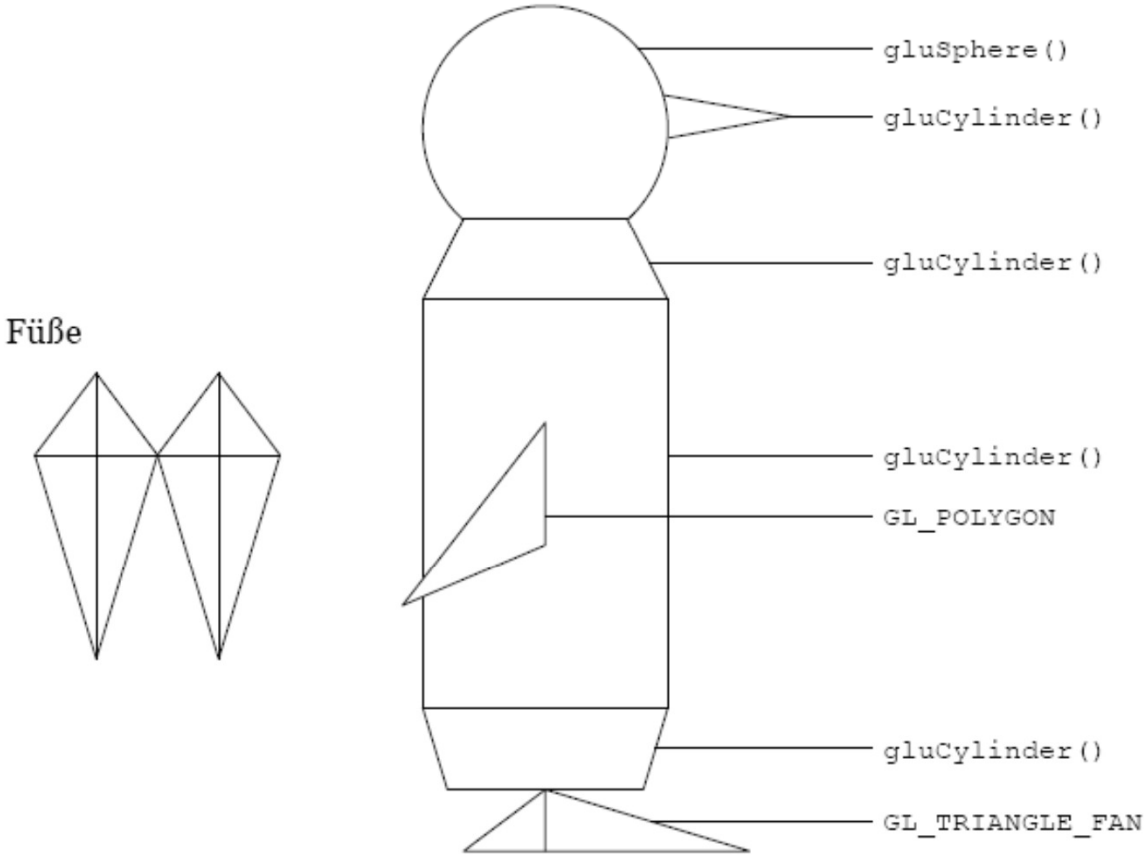




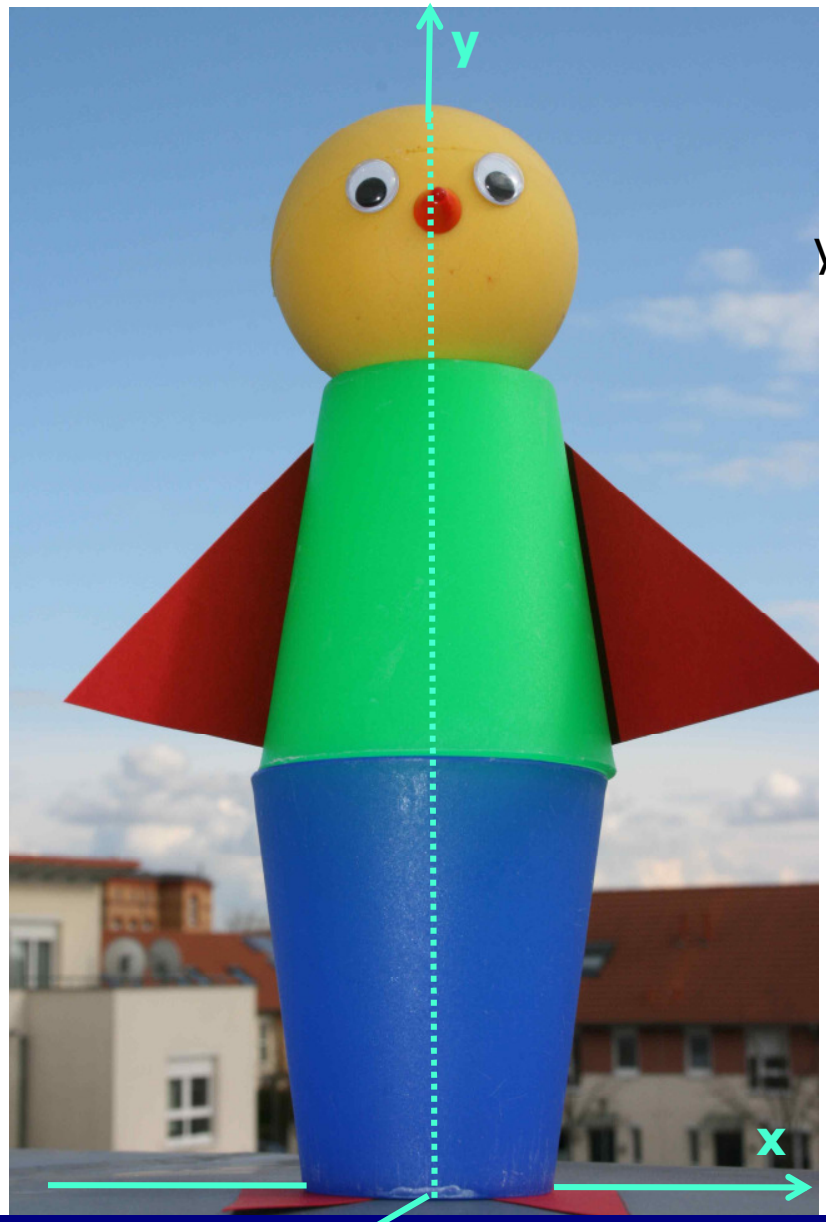


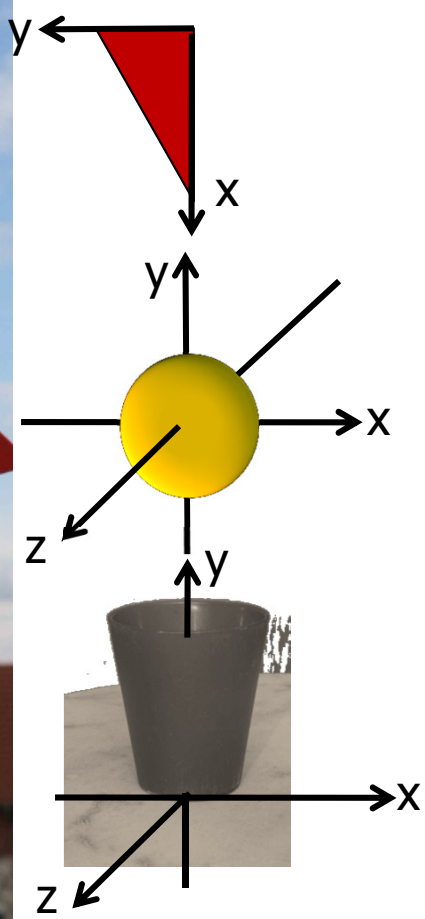
Erzeugen einer Geometrie am Beispiel eines Pinguins
(von Dominik Paulus)

Pinguin - Aufbau



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

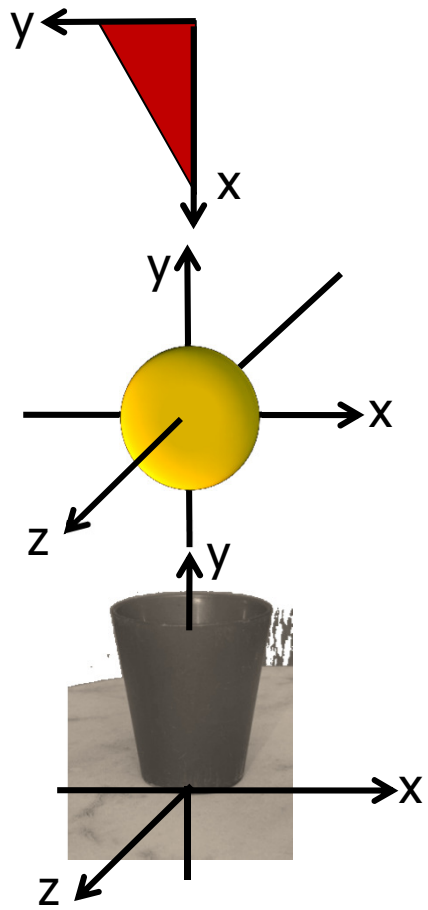
Was fehlt noch zur vollständigen Modellierung des Pinguins?

- 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 die Primitive im lokalen Koordinatensystem „manipulieren“, um einen Pinguin zu konstruieren?

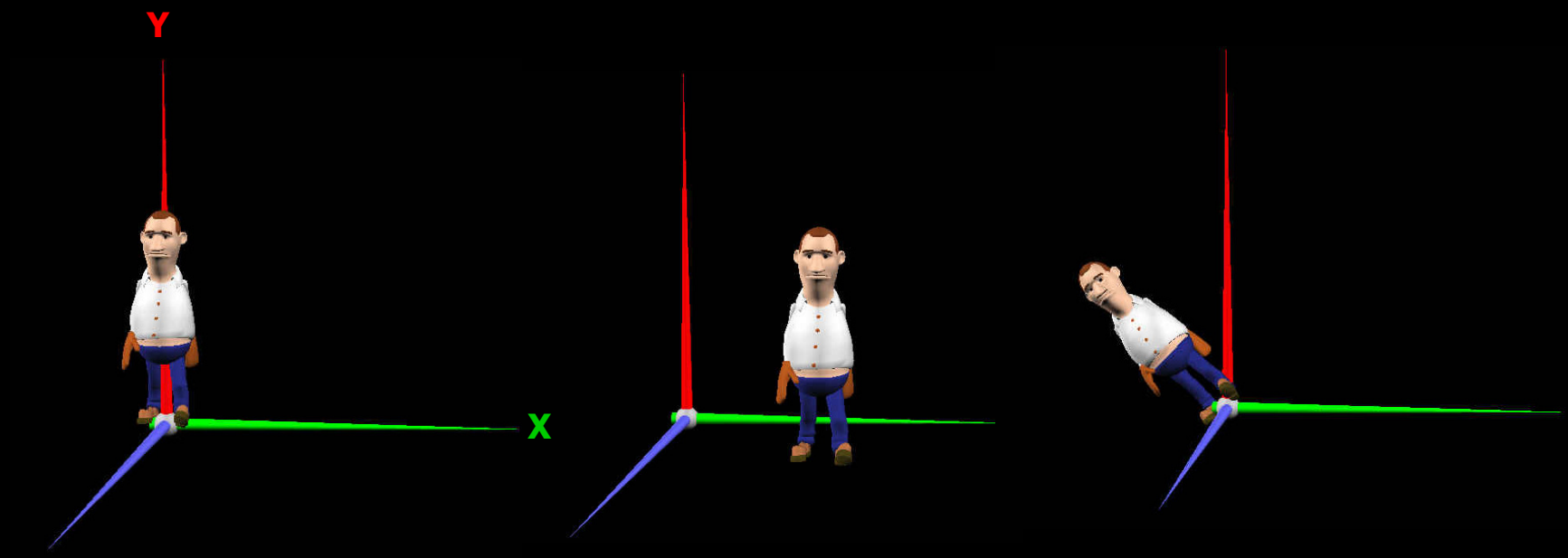


Man kann sie:

- 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);

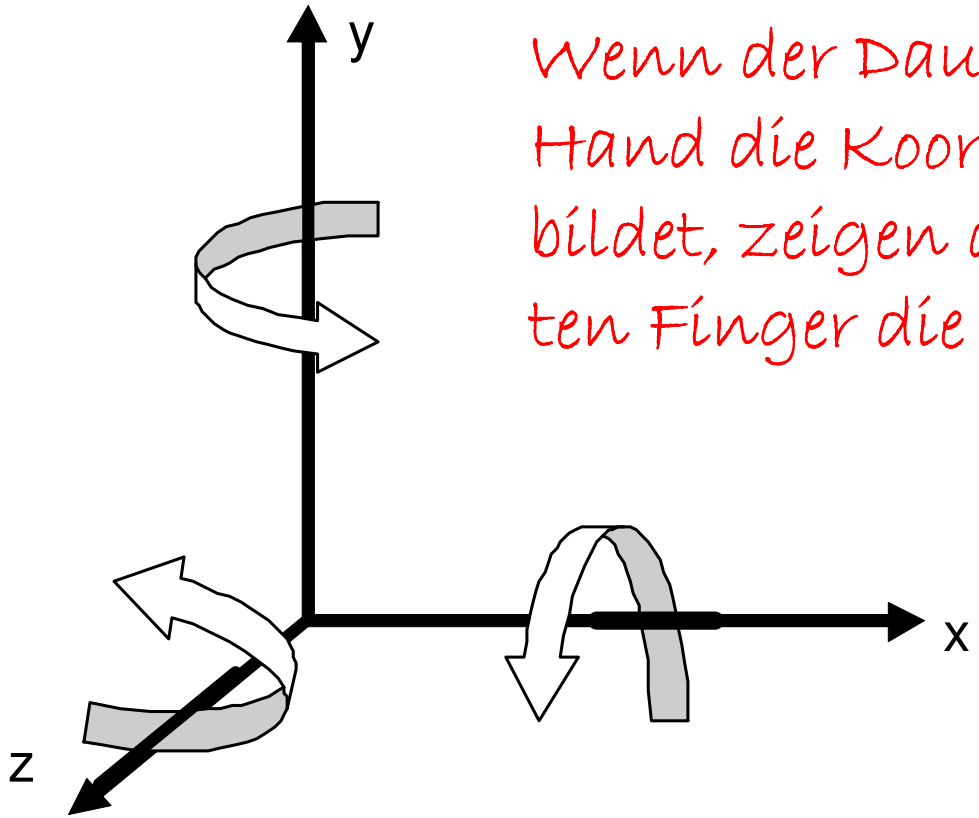
Rotation um die Z-Achse

glRotatef(Winkel, x, y, z);

Mit x, y, z: Rotationsachse

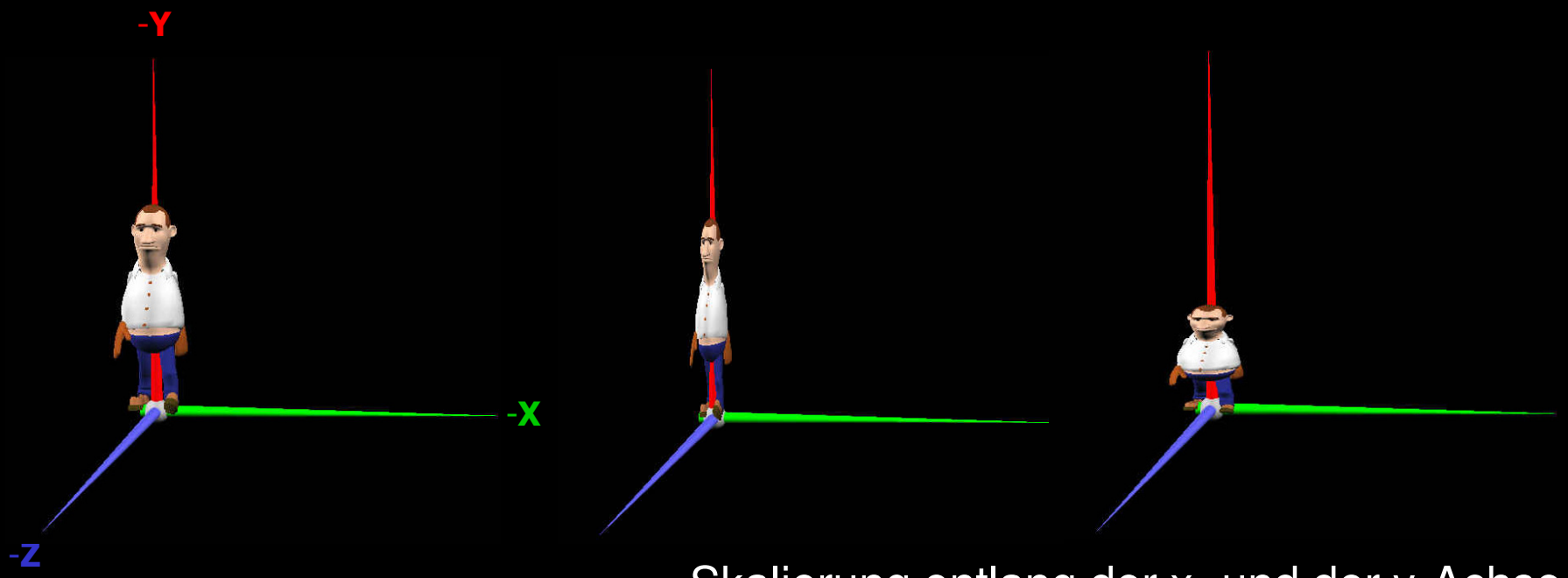
Um welche Achse wird die Geometrie bei der Rotation gedreht?

Demonstration des mathematisch positiven Drehsinns



Wenn der Daume der rechten Hand die Koordinatenachse bildet, zeigen die angewinkelten Finger die Drehrichtung an.

Transformation im 3D-Raum: Skalierung



Skalierung entlang der x- und der y-Achse

glScalef(x, y, z);

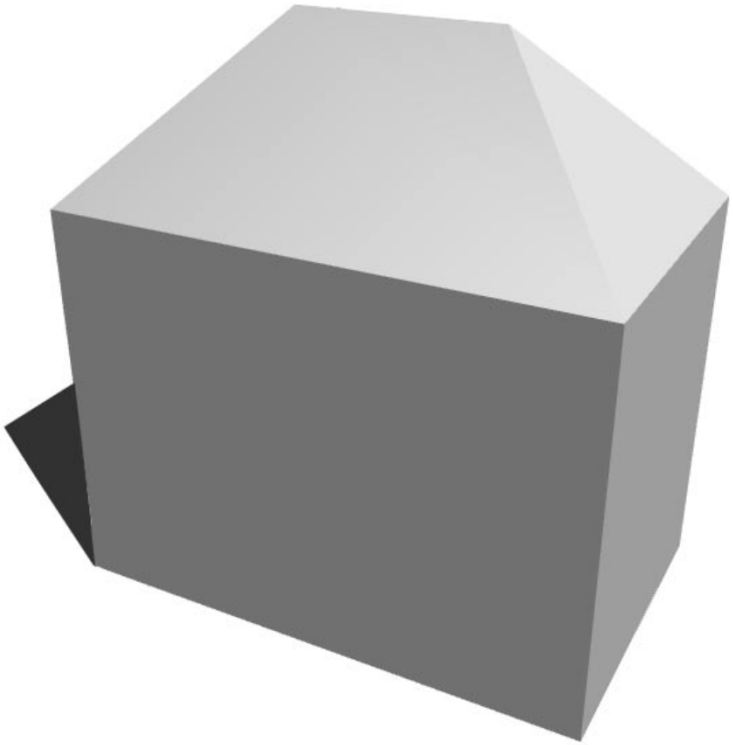
Wie verhält sich die Figur bei der Skalierung, wenn sie nicht am Nullpunkt definiert wurde?

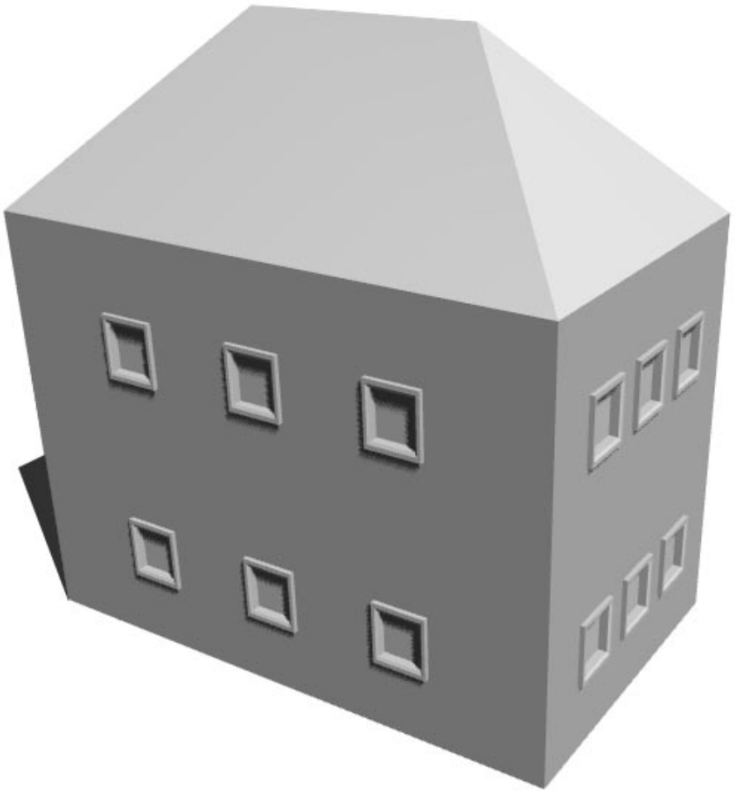
Sven Janusch „Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden“

Anno 1701, Related Designs Software GmbH

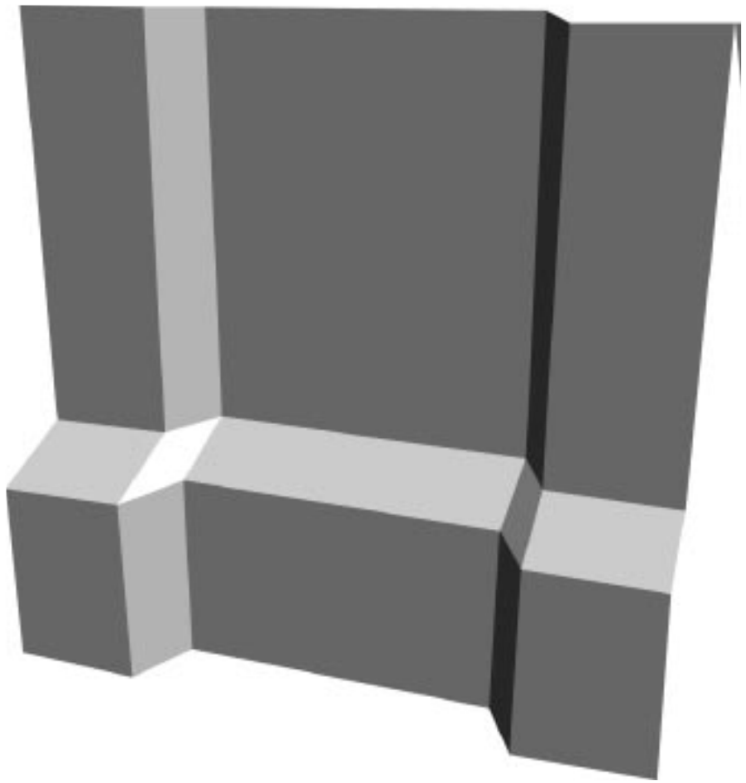
Sven Janusch „Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden“

Sven Janusch „Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden“

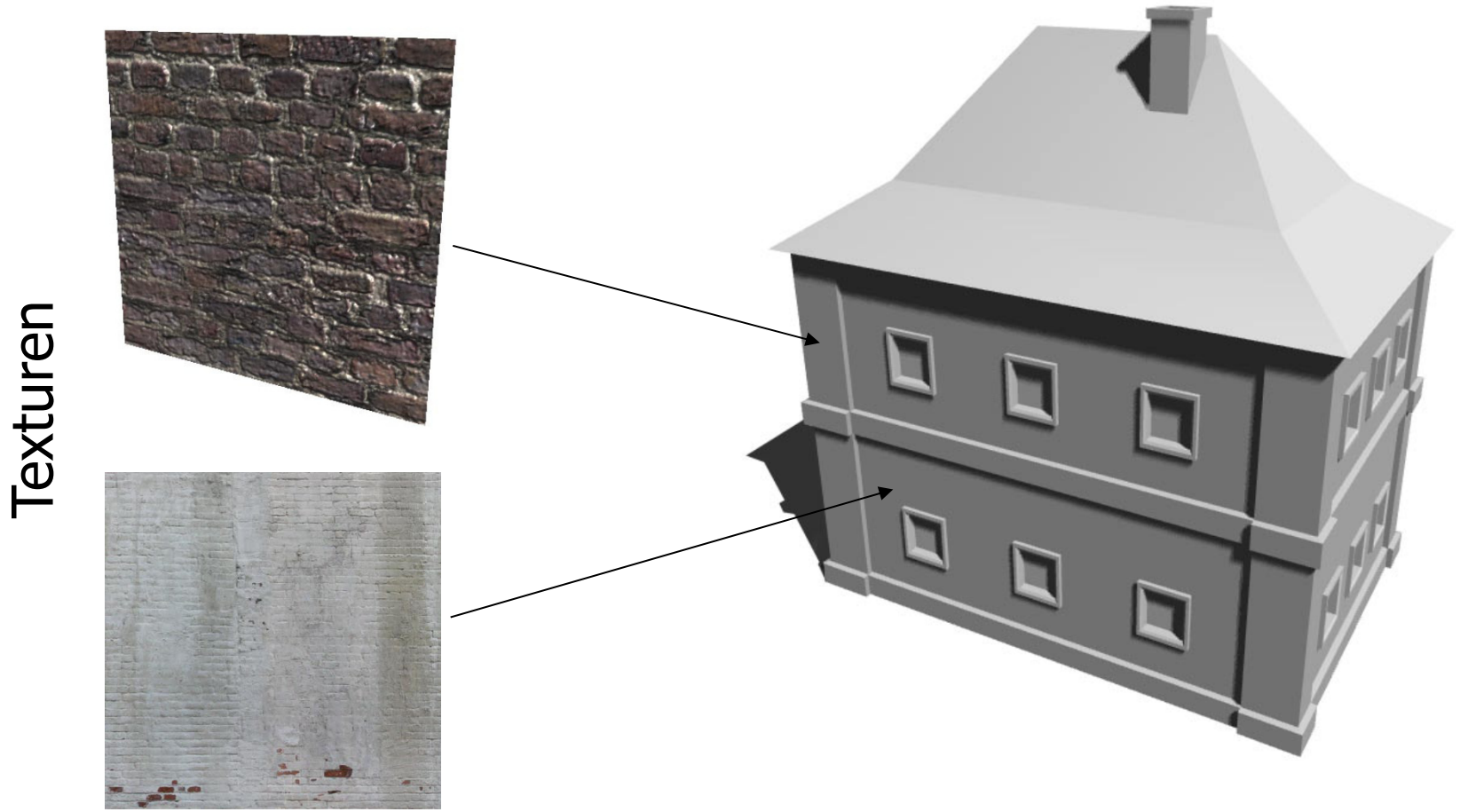




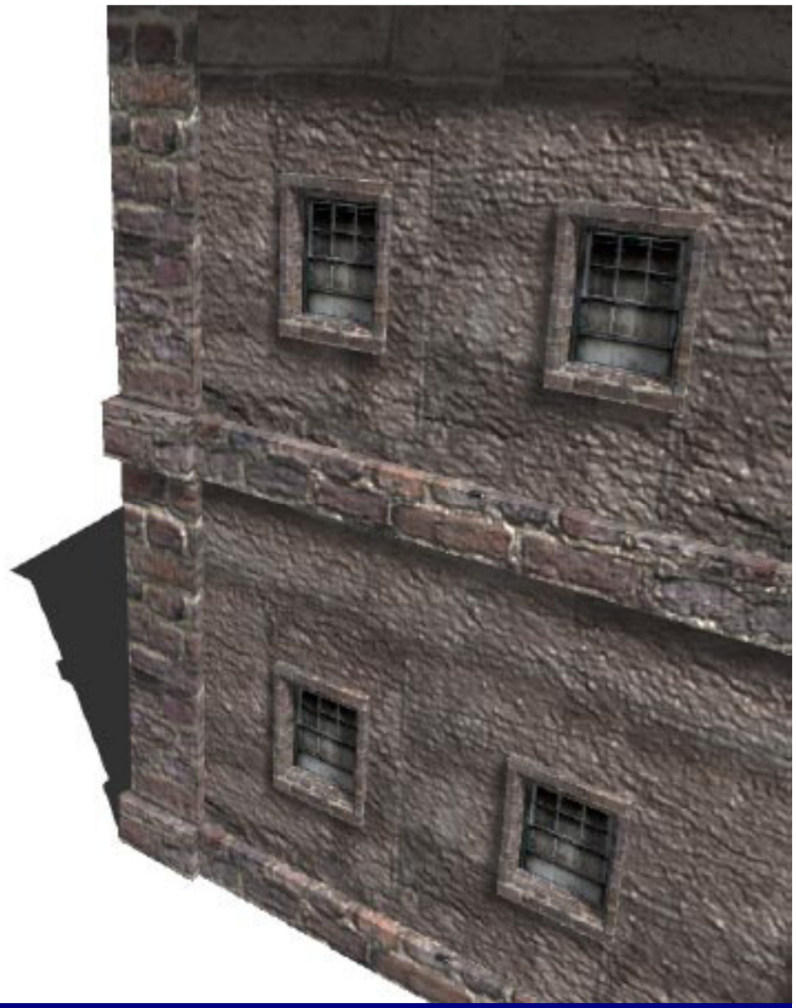
Sven Janusch „Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden“



Sven Janusch „Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden“



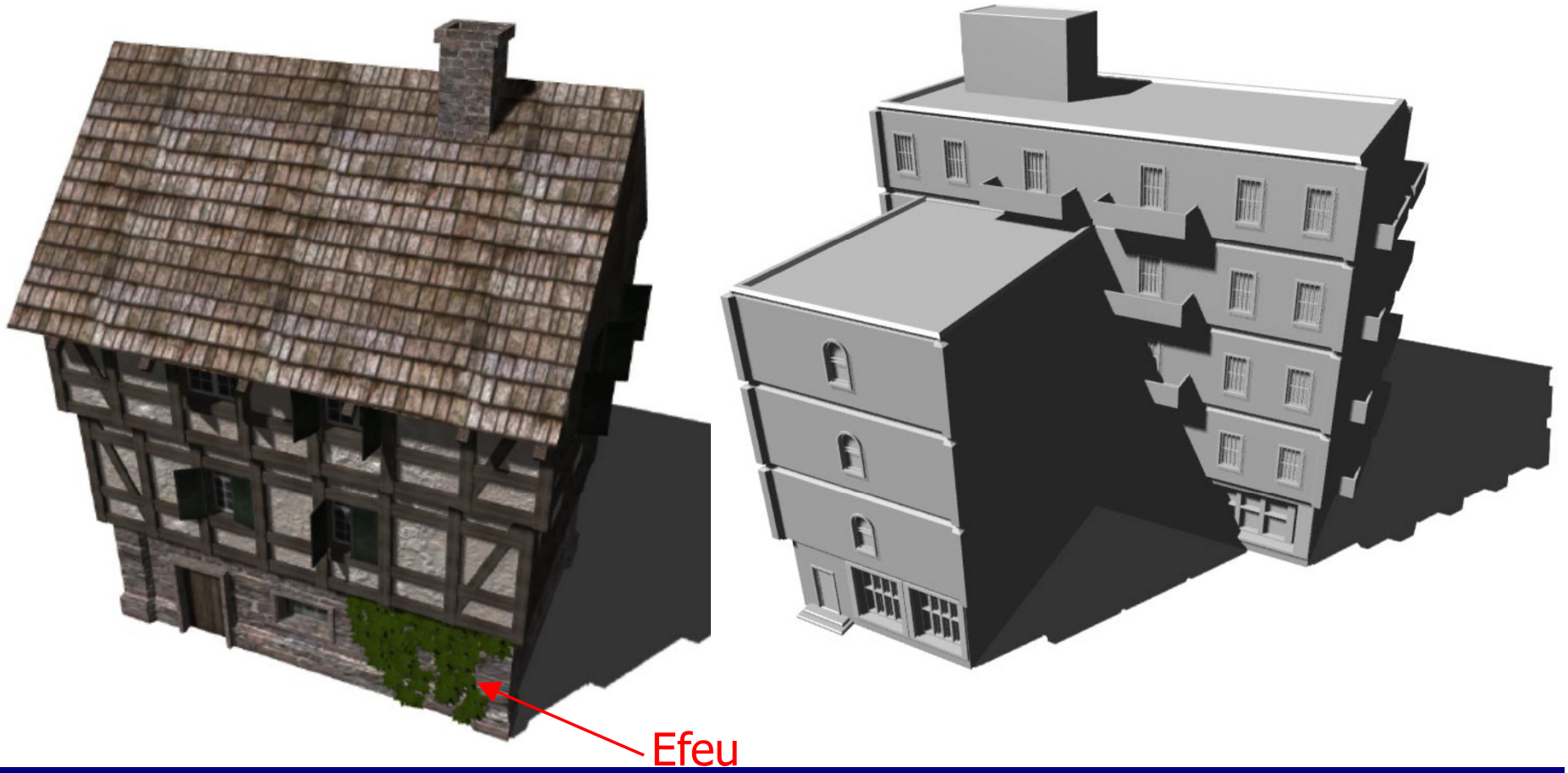
Sven Janusch „Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden“



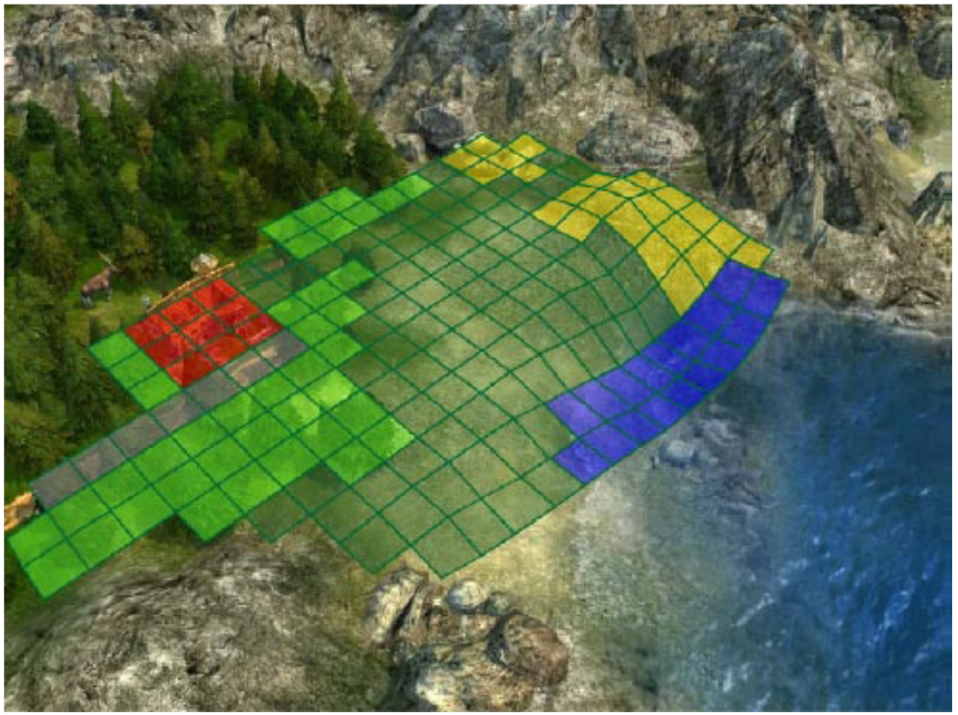
Sven Janusch „Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden“

Sven Janusch „Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden“

Sven Janusch „Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden“

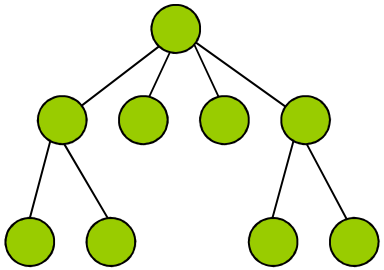


Sven Janusch „Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden“

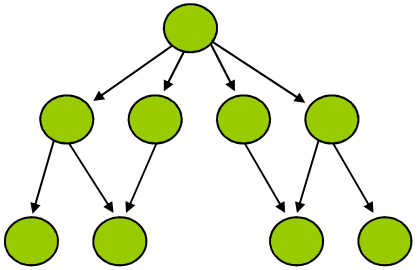


Der Szenengraph

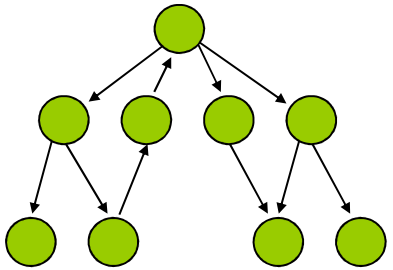
Unterschied Baum und Graph



Baum



gerichteter
zyklenfreier Graph



gerichteter
nicht-zyklenfreier Graph

Szenengraph

Szenengraph besteht aus mindestens 3 Knotentypen:

Gruppen

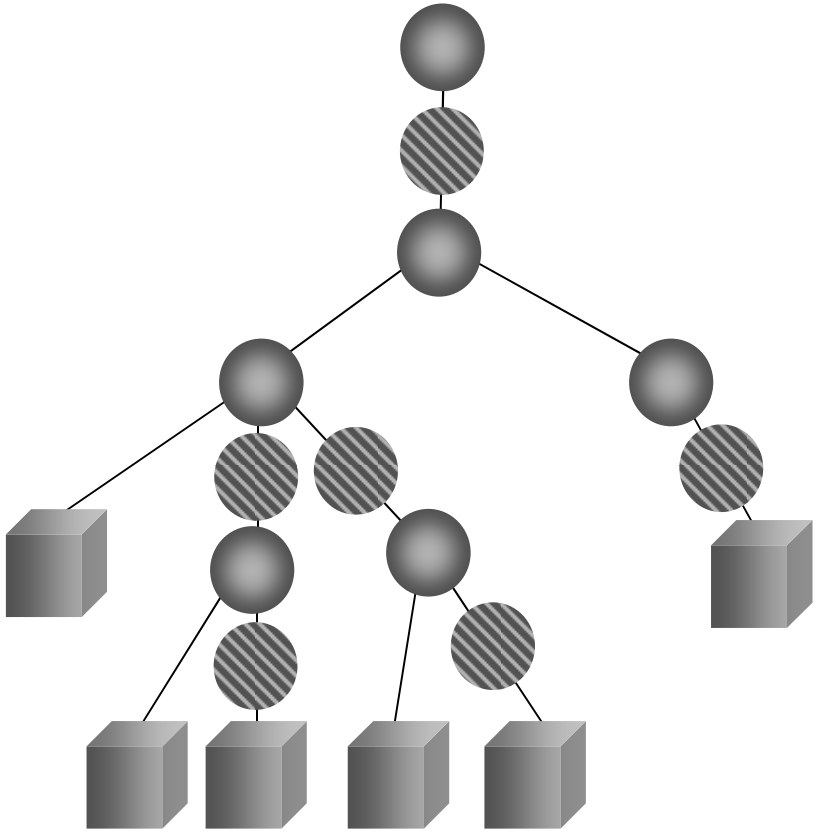
Geometrien (inkl. Materialeigenschaften)

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**
- heterogen
- Geometrie / darstellbare Primitive in den Blättern

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

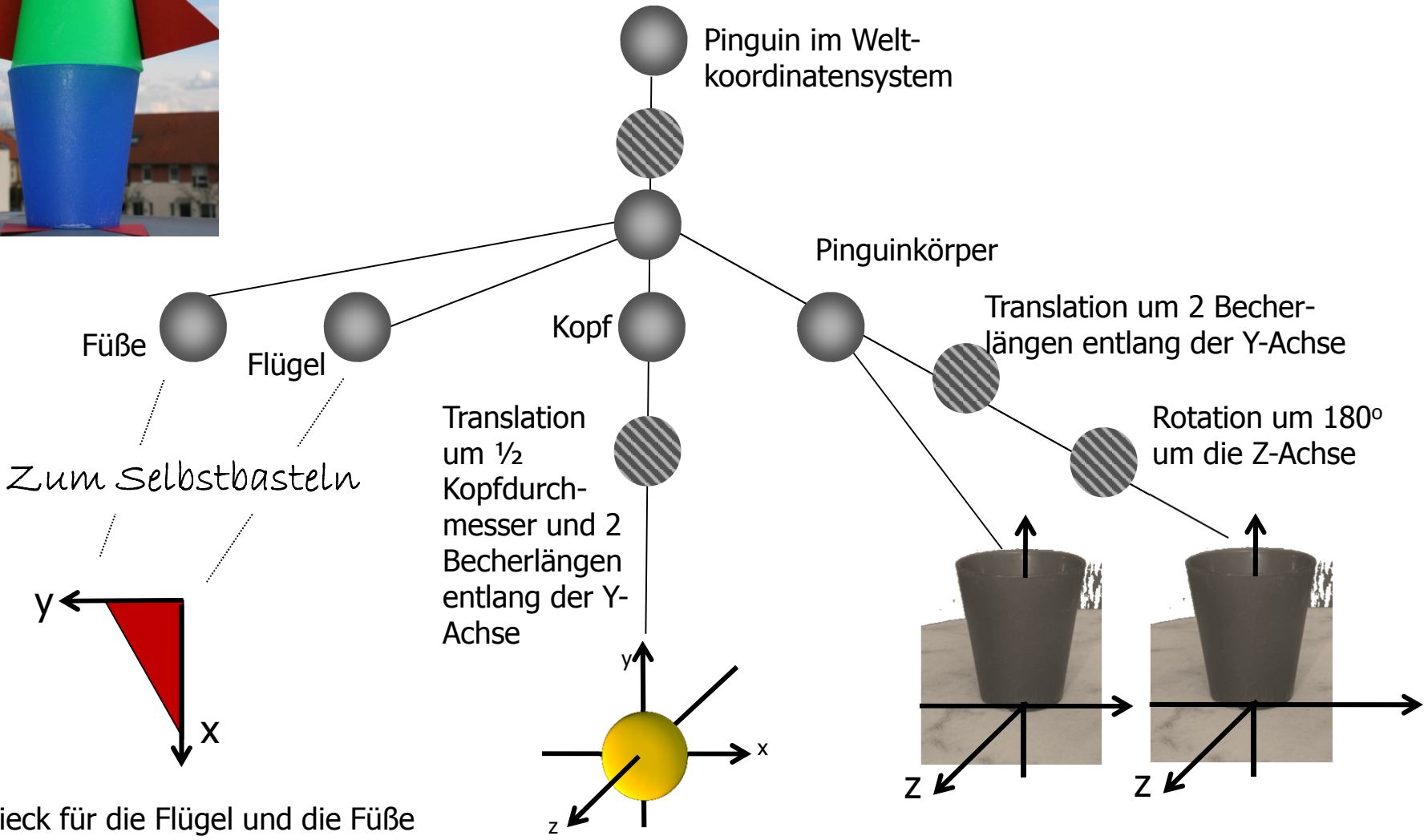
(Sehen wir dann beim Pinguin)

Erzeugung des Pinguins mit Hilfe eines Szenengraphen...

Folgende Primitive stehen zur Verfügung:

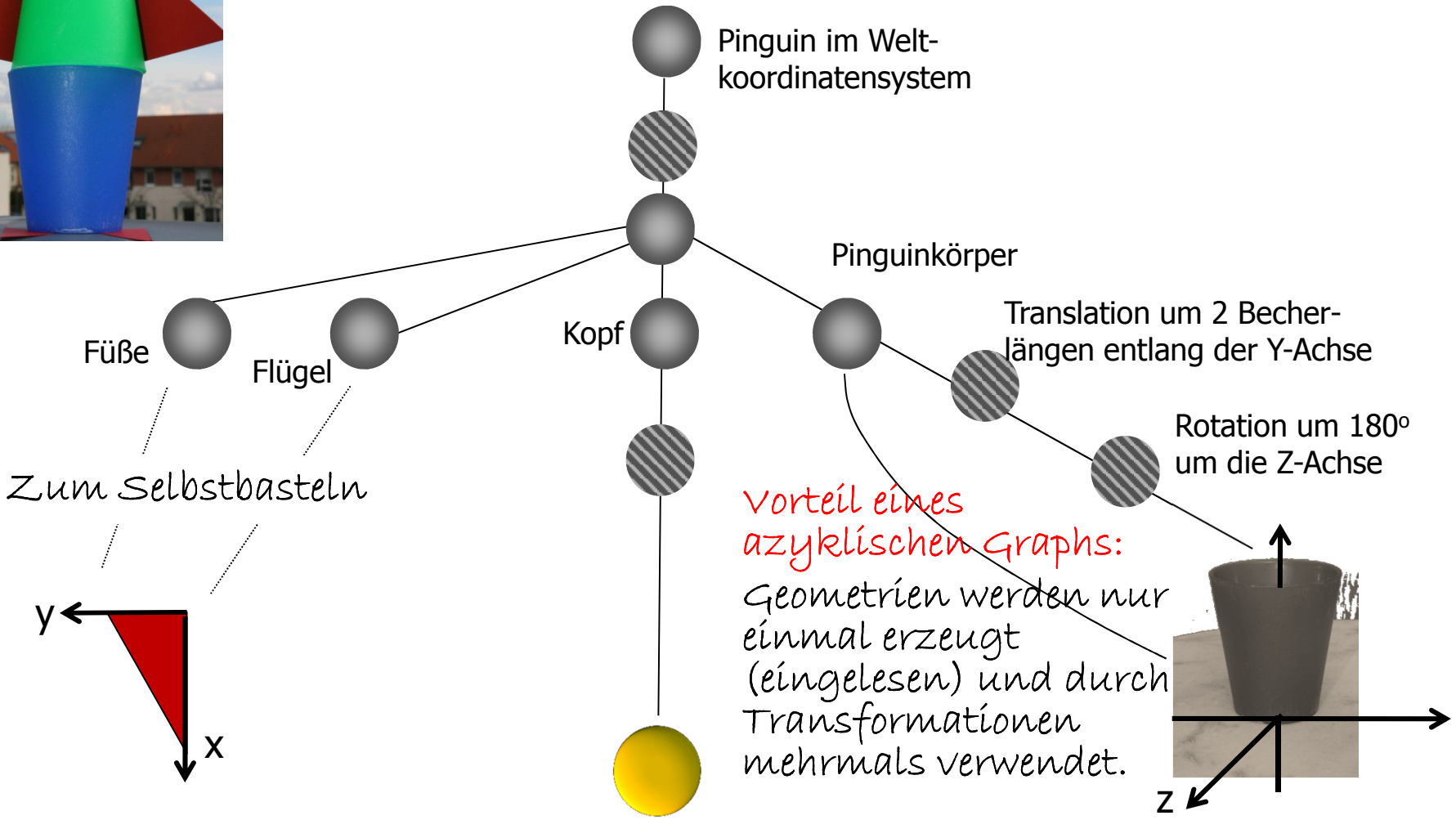
- Becher mit der Öffnung nach oben
- Ball (bereits mit Augen und Nase)
- [Ein Dreiecke]

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();
```

1. OpenGL Programm

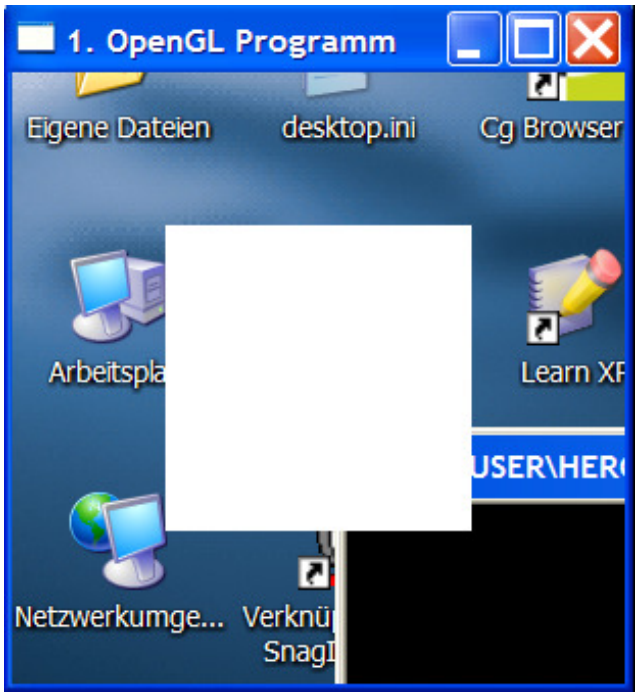
```
#include <GL/glut.h> //GLUT .h-Datei, lädt auch GL .h-Dateien

void display() //Zeichenfunktion
{
   glBegin( GL_POLYGON );
      glVertex3f( -0.5, -0.5, 0);
      glVertex3f(  0.5, -0.5, 0);
      glVertex3f(  0.5,  0.5, 0);
      glVertex3f( -0.5,  0.5, 0);
   glEnd();
   glFlush(); //Buffer leeren
}

int main(int argc, char **argv)
{
   glutInit(&argc, argv); // GLUT initialisieren
   glutInitDisplayMode( GLUT_RGB ); // Fenster Konfiguration
   glutCreateWindow("1. OpenGL Programm"); // Fenster Erzeugung
   glutDisplayFunc(display); // Zeichenfunktion bekannt machen
   glutMainLoop();
   return 0;
}
```

Welche Farbe hat das hier erzeugte Polygon?

Welche Hintergrundfarbe hat das hier erzeugte Fenster?

Ausgabe des 1. OpenGL Programms:

Keine Farbe und der Hintergrund ist nicht gesetzt:

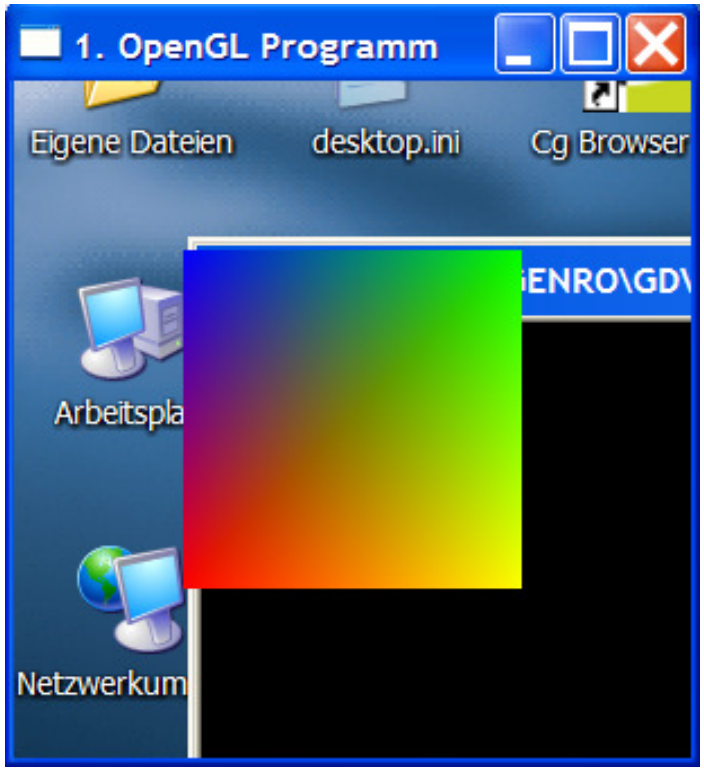


Erweiterung des 1. OpenGL Programm

```
#include <GL/glut.h>

void display()
{
   glBegin( GL_POLYGON );
      // State- Machine: Wenn nur der erste Farbwert angeben wird haben
      // alle folgenden Eckpunkte den gleichen Farbwert
      glColor4f( 1., 0., 0., 1.);
      glVertex3f( -0.5, -0.5, 0);
      glColor4f( 1., 1., 0., 1.); // Gelb
      glVertex3f(  0.5, -0.5, 0);
      glColor4f( 0., 1., 0., 1.);
      glVertex3f(  0.5,  0.5, 0);
      glColor4f( 0., 0., 1., 1.);
      glVertex3f( -0.5,  0.5, 0);
   glEnd();
   glFlush(); //Buffer leeren
}

int main(int argc, char **argv)
{
   ... // wie vorher
}
```

Ausgabe des 1. OpenGL Programm

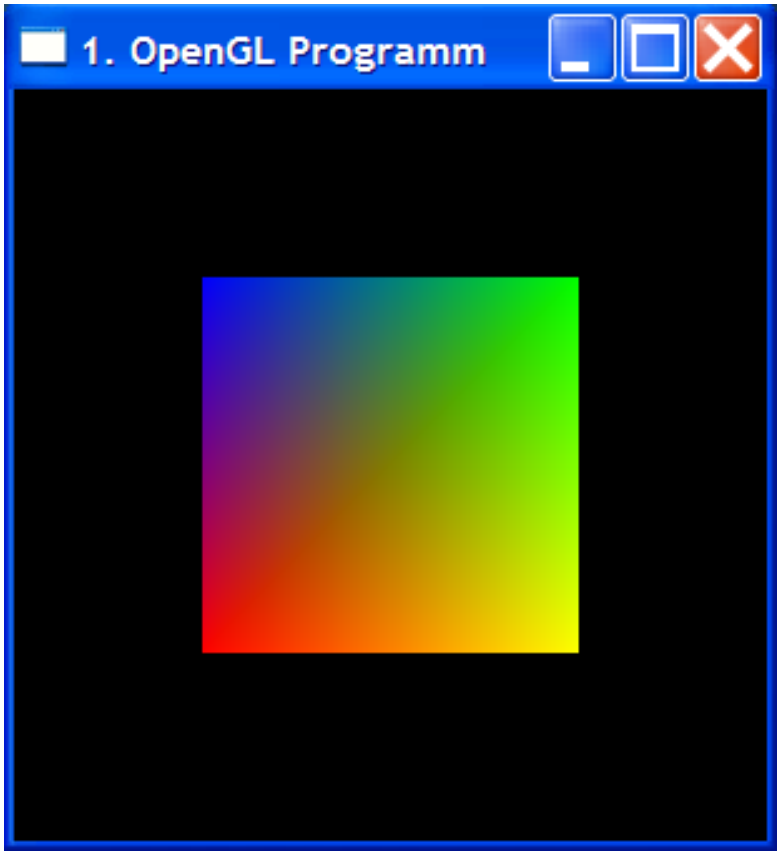
Der Hintergrund ist noch nicht gelöscht worden:



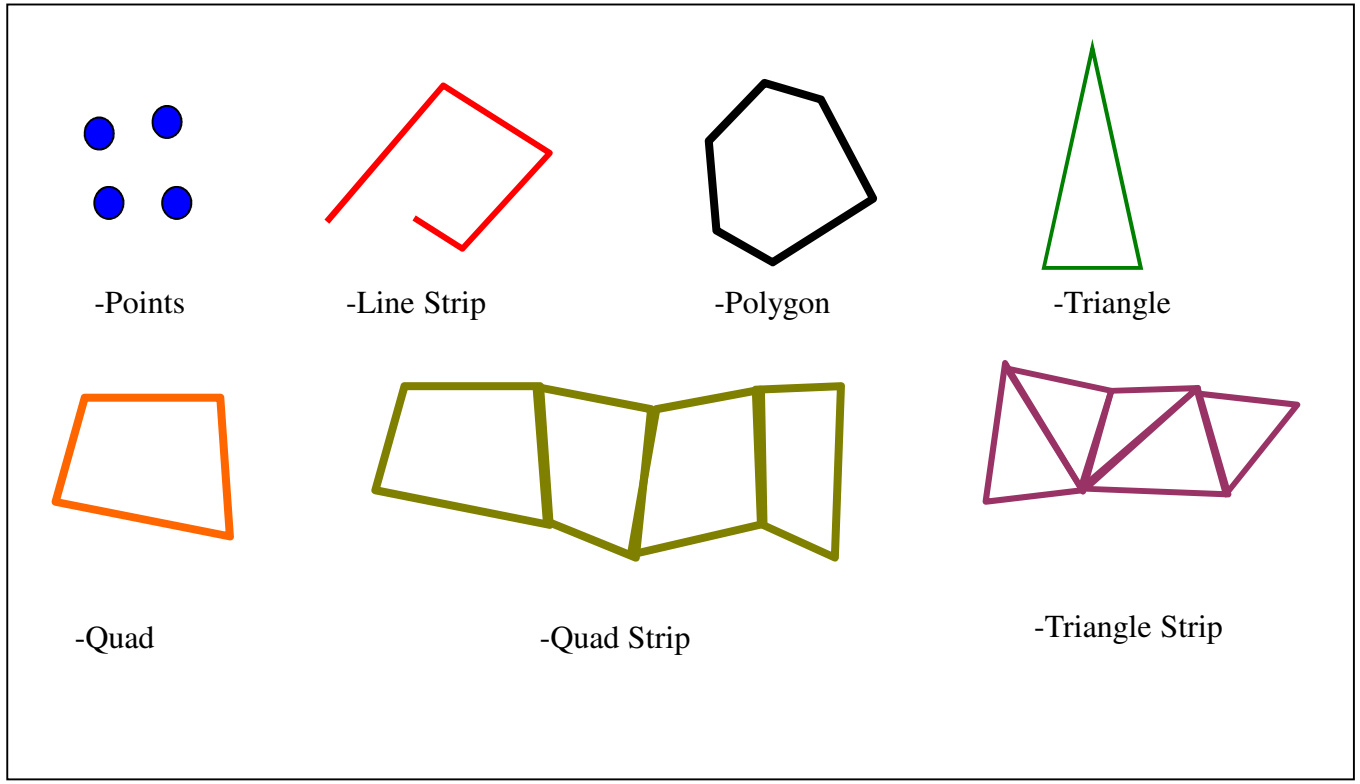
Erweiterung des 1. OpenGL Programm

```
#include <GL/glut.h>

void display()
{
   glClear(GL_COLOR_BUFFER_BIT ); //löschen des Hintergrunds

   glBegin( GL_POLYGON );
      // State- Machine: Wenn nur der erste Farbwert angeben wird haben
      // alle folgenden Eckpunkte den gleichen Farbwert
      glColor4f( 1., 0., 0., 1.);
      glVertex3f( -0.5, -0.5, 0);
      glColor4f( 1., 1., 0., 1.); // Gelb
      glVertex3f(  0.5, -0.5, 0);
      glColor4f( 0., 1., 0., 1.);
      glVertex3f(  0.5,  0.5, 0);
      glColor4f( 0., 0., 1., 1.);
      glVertex3f( -0.5,  0.5, 0);
   glEnd();
   glFlush(); //Buffer leeren
}

int main(int argc, char **argv) // wie vorher
...
```

Ausgabe des 1. OpenGL Programm



OpenGL Primitive

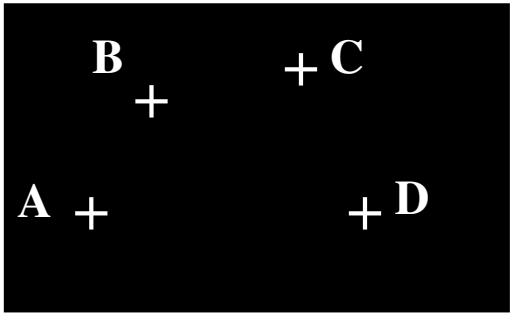


Definition der OpenGL Primitive

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

Beispiel:

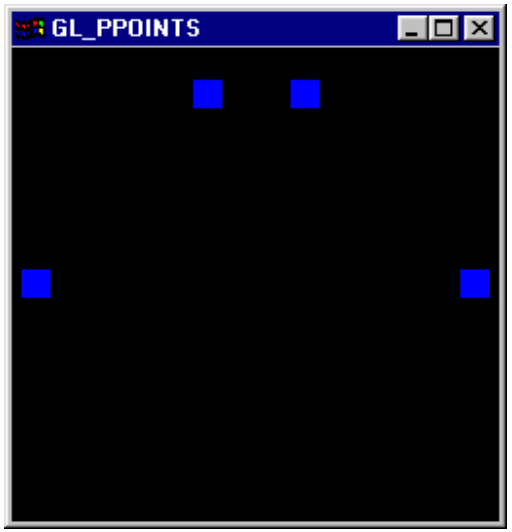
```
glBegin ( GL_POINTS );
   glVertex3f ( xA, yA, zA );
   glVertex3f ( xB, yB, zB );
   glVertex3f ( xC, yC, zC );
   glVertex3f ( xD, yD, zD );
glEnd ();
```



Achtung:
Dies ist nicht die Bildschirm-ausgabe. Wegen der Kleinheit 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 ( );
```

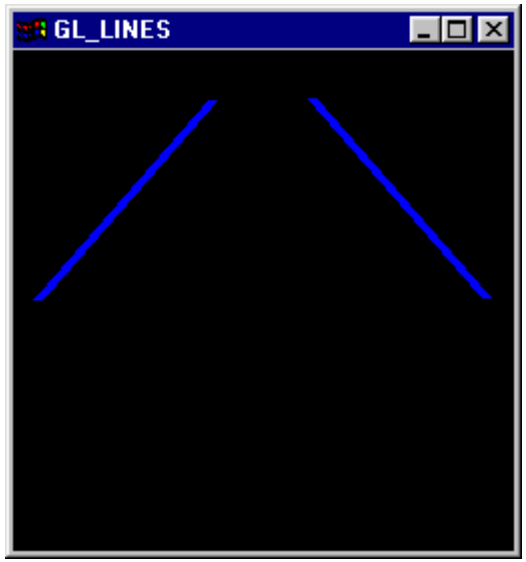


OpenGL ist eine State-Machine:

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

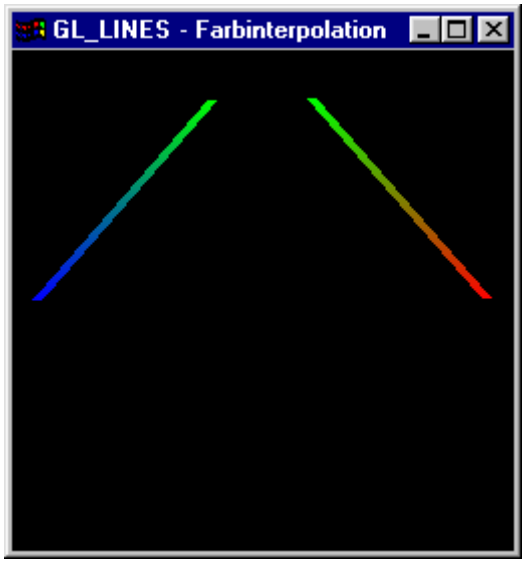
OpenGL Primitiv: Linie

```
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 ( );
```



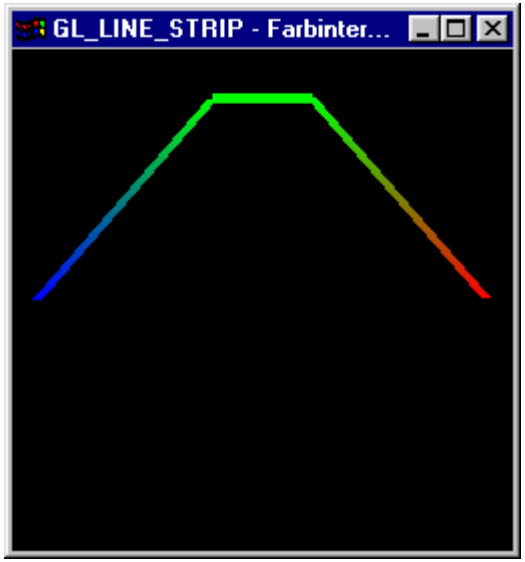
OpenGL Primitiv: Linie

```
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 ( );
```



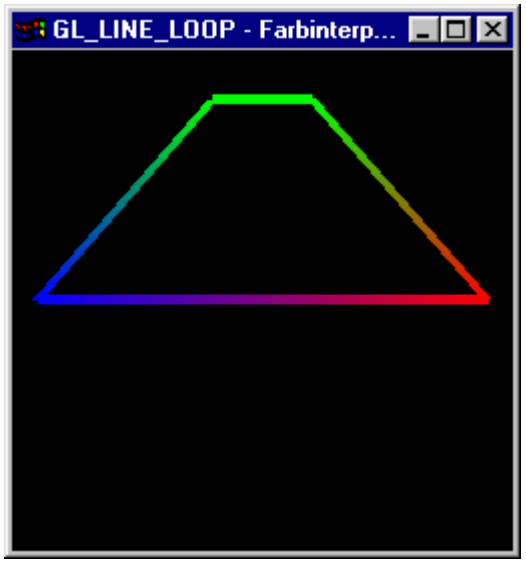
OpenGL Primitiv: Linienzug

```
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 ( );
```



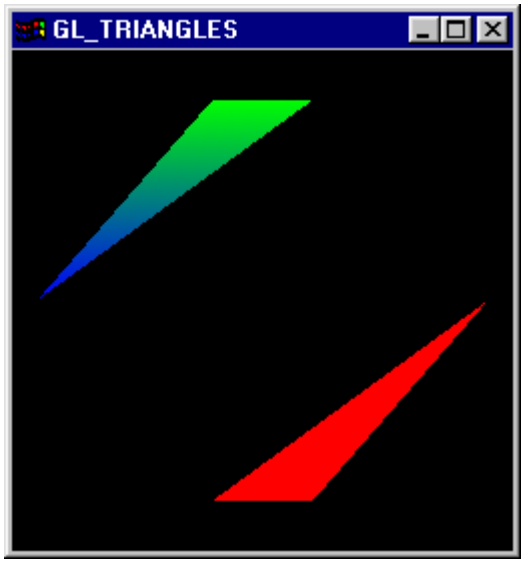
OpenGL Primitiv: Geschlossener Linienzug

```
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 ( );
```



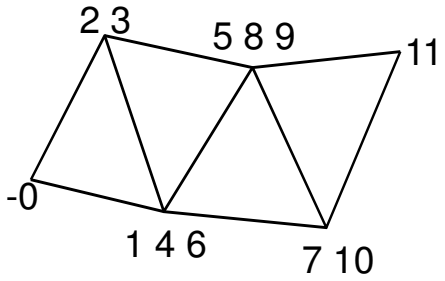
OpenGL Primitiv: Dreieck

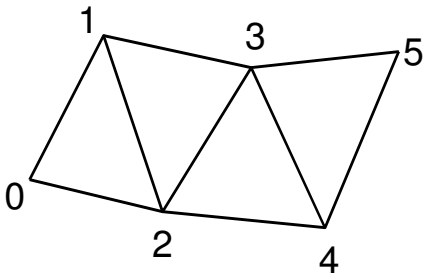
```
glBegin(GL_TRIANGLES);
   glColor4f ( 0.0f, 0.0f, 1.0f, 1.0f );
   glVertex3f(-0.9f, 0.0f, 0.0f );
   glColor4f ( 0.0f, 1.0f, 0.0f, 1.0f );
   glVertex3f(-0.2f, 0.8f, 0.0f );
   glVertex3f(+0.2f, 0.8f, 0.0f );
   glColor4f ( 1.0f, 0.0f, 0.0f, 1.0f );
   glVertex3f(+0.9f, 0.0f, 0.0f );
   glVertex3f(+0.2f,-0.8f,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



OpenGL Primitiv: Dreiecksstreifen
(Triangle Strip)

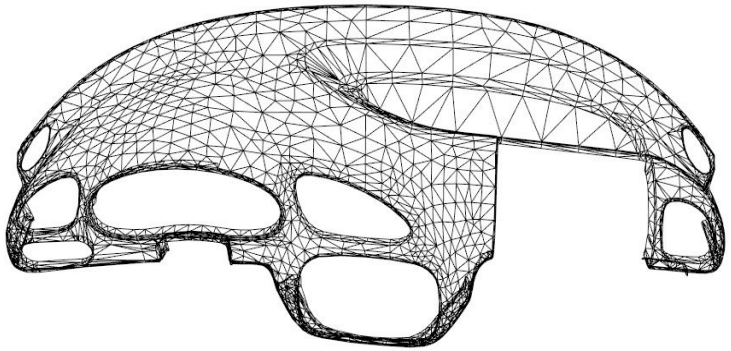
- Ziel ist es, möglichst wenig Elemente anzulegen.
- Eckpunkte können durch zusammenhängende Strips recycelt werden.



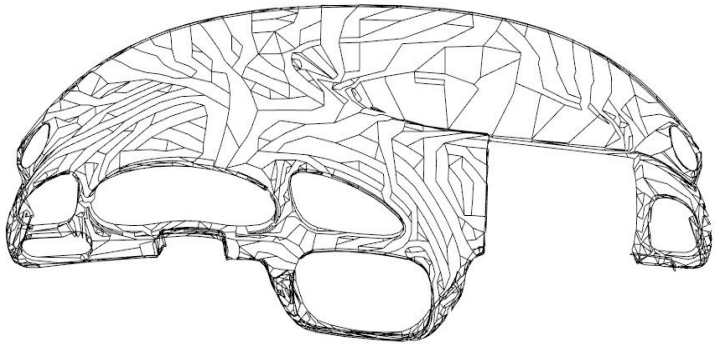


Anzahl der Punkte ohne Streifen- und mit Streifenbildung

OpenGL Primitiv: Dreiecksstreifen



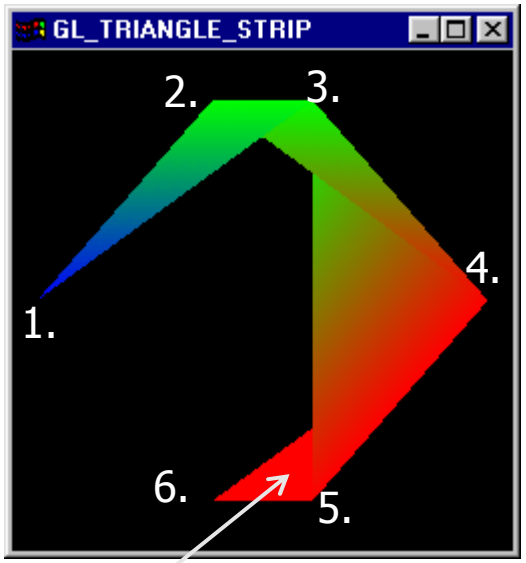
4320 Dreiecke
12960 Eckpunkte



905 Strips
6127 Eckpunkte

OpenGL Primitiv: Dreiecksstreifen

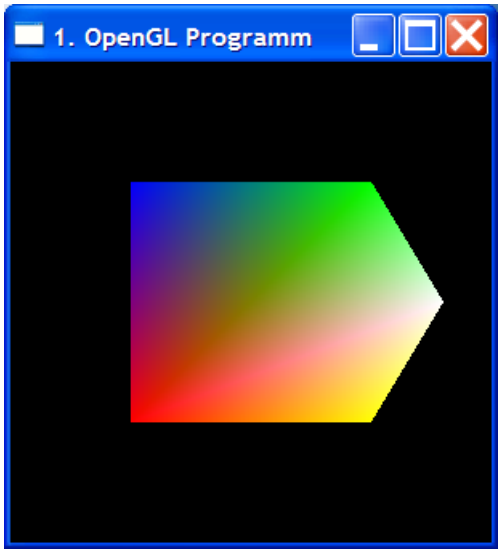
```
glBegin(GL_TRIANGLE_STRIP);
1.  glColor4f ( 0.0, 0.0, 1.0, 1.0f );
    glVertex3f(-0.9, 0.0, 0.0 );
2.  glColor4f ( 0.0, 1.0, 0.0, 1.0f );
3.  glVertex3f(-0.2, 0.8, 0.0 );
    glVertex3f( 0.2, 0.8, 0.0 );
4.  glColor4f ( 1.0, 0.0, 0.0, 1.0f );
5.  glVertex3f( 0.9, 0.0, 0.0 );
    glVertex3f( 0.2,-0.8,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



Transformationen in OpenGL

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

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

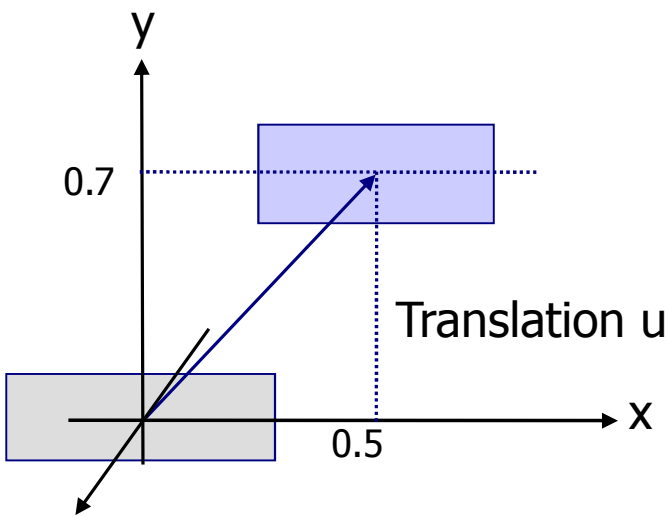
Rotationen: *Geometrie wird um den angegebenen Winkel (gegen den Uhrzeigersinn) um die Achse (x, y, z) rotiert.*

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

Skalierung:

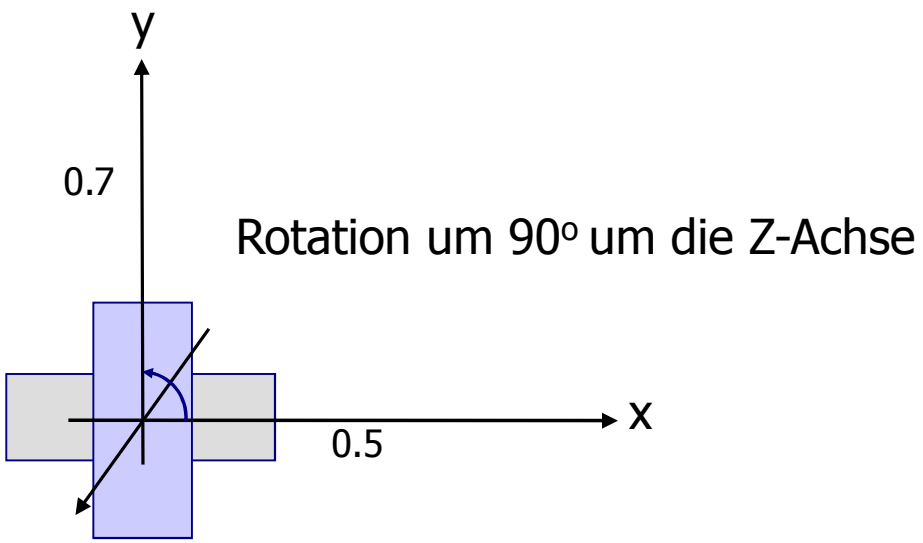
```
glScalef( x, y, z);
```

Beispiel für eine Translation

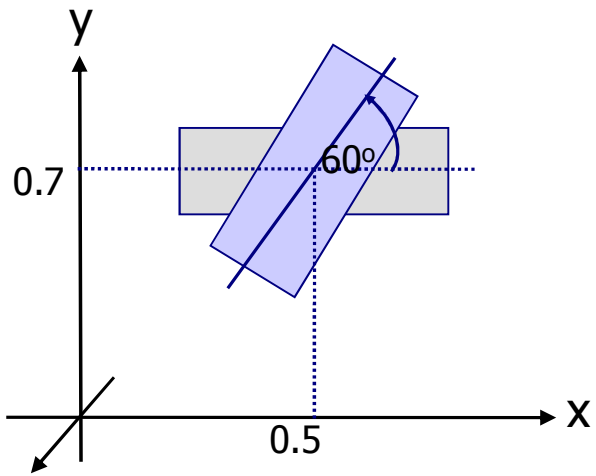


Translation um den Vektor (0.5, 0.7, 0.)

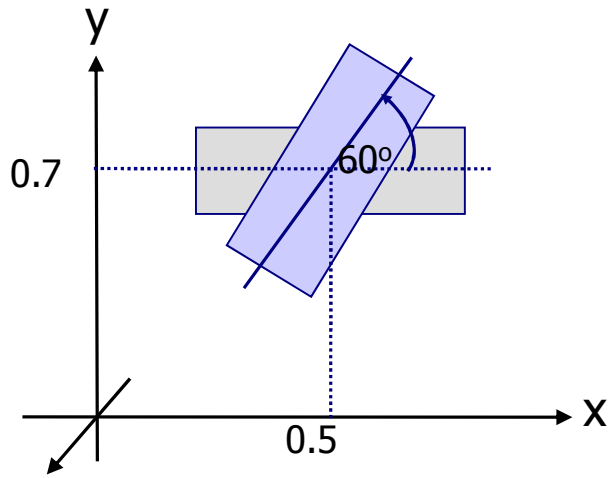
Beispiel für eine Rotation



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

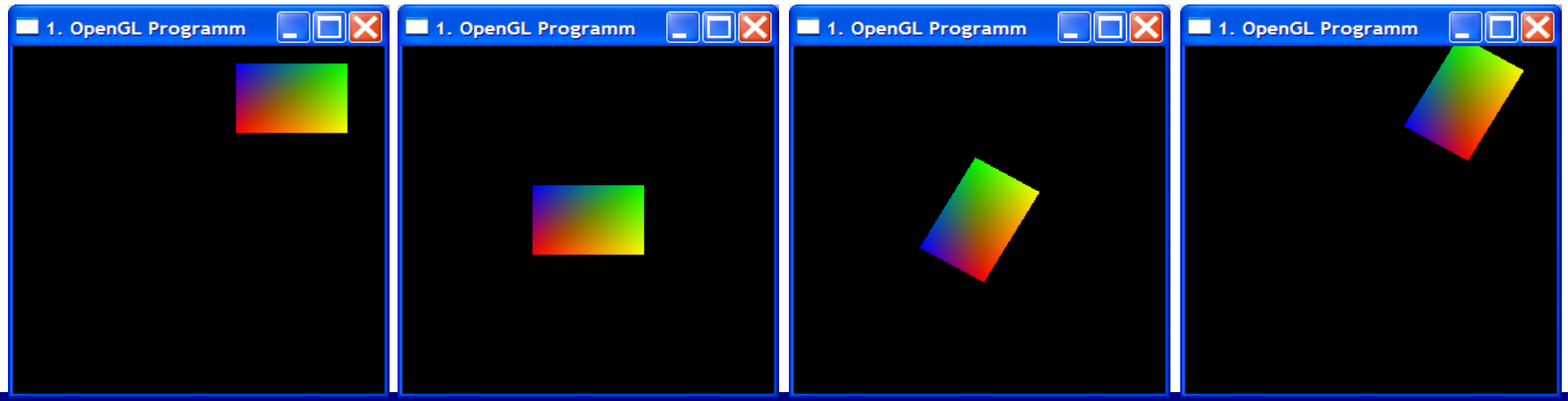
Reihenfolge in der die Transformationen angewendet werden ist wichtig!

Quadrat soll um 60° gedreht werden:

Transformationen werden in **umgekehrter** Reihenfolge angegeben:

Reihenfolge in der, die Transformationen auf das Quadrat angewendet werden:

```
glTranslatef( 0.5, 0.7, 0.);// Rücktransformat.
glRotatef( 60., 0., 0., 1.);
glTranslatef( -0.5, -0.7, 0.);//in den Ursprung
Quadrat();
```



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 <u>nicht</u> dasselbe Ergebnis, wie b.)

a.)
$$\begin{bmatrix}3\\2\\3\\1\end{bmatrix} = \begin{bmatrix}1&0&0&2\\0&1&0&3\\0&0&1&0\\0&0&0&1\end{bmatrix} \cdot \begin{bmatrix}0&1&0&0\\-1&0&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} \cdot \begin{bmatrix}1\\2\\3\\1\end{bmatrix}$$

b.)
$$\begin{bmatrix}5\\-3\\3\\1\end{bmatrix} = \begin{bmatrix}0&1&0&0\\-1&0&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} \cdot \begin{bmatrix}1&0&0&2\\0&1&0&3\\0&0&1&0\\0&0&0&1\end{bmatrix} \cdot \begin{bmatrix}1\\2\\3\\1\end{bmatrix}$$

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‘:

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

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

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).

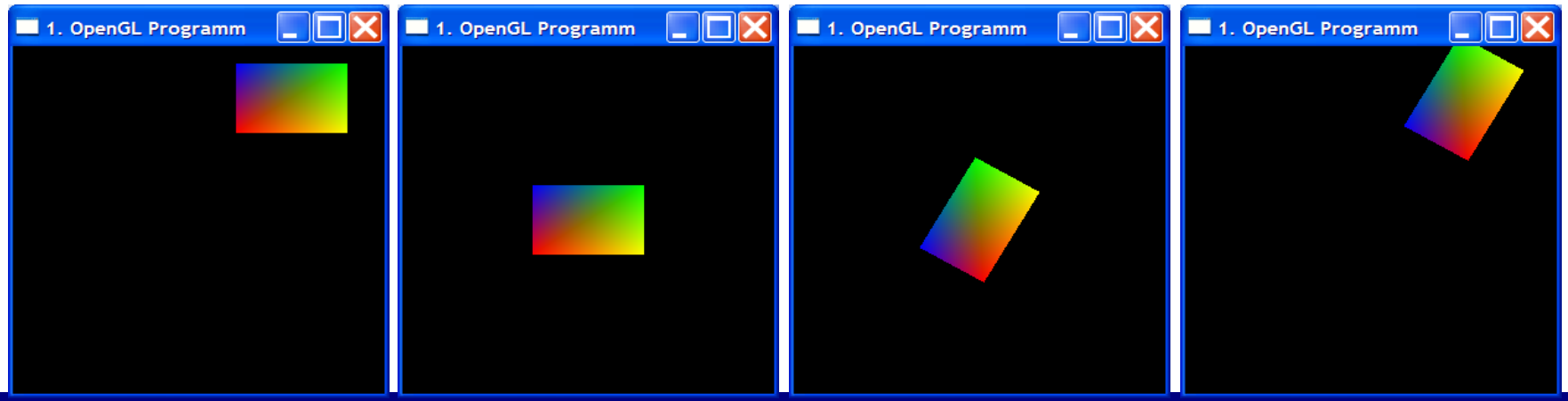
Quadrat soll um 60° gedreht werden:

Transformationen werden in **umgekehrter** Reihenfolge angegeben:

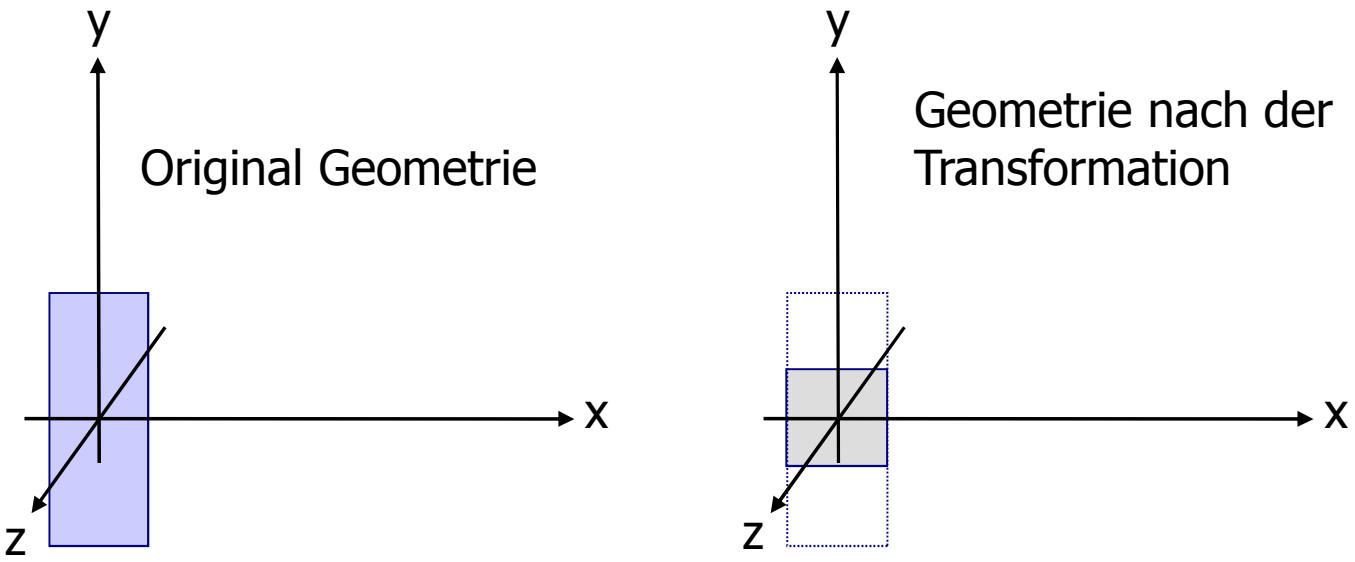
Reihenfolge in der, die Transformationen auf die akkumulierte Matrix M multipliziert werden:

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

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



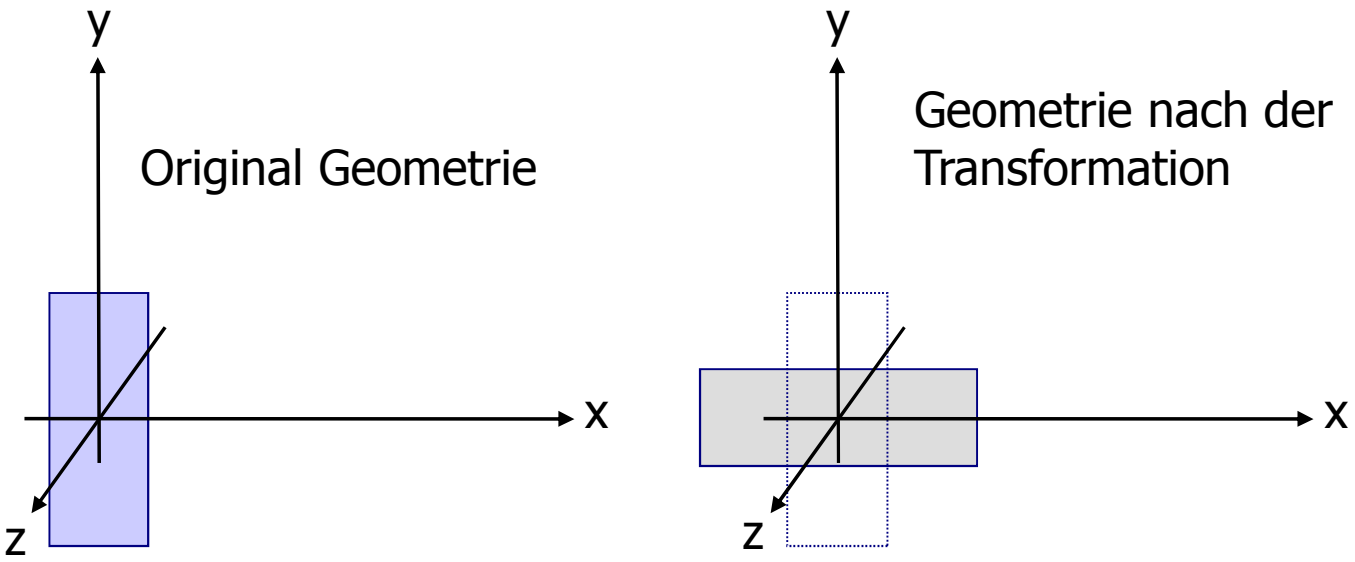
Beispiel für eine Skalierung



Skalierung um die Faktoren:

x = 1, y = ? und z = ?

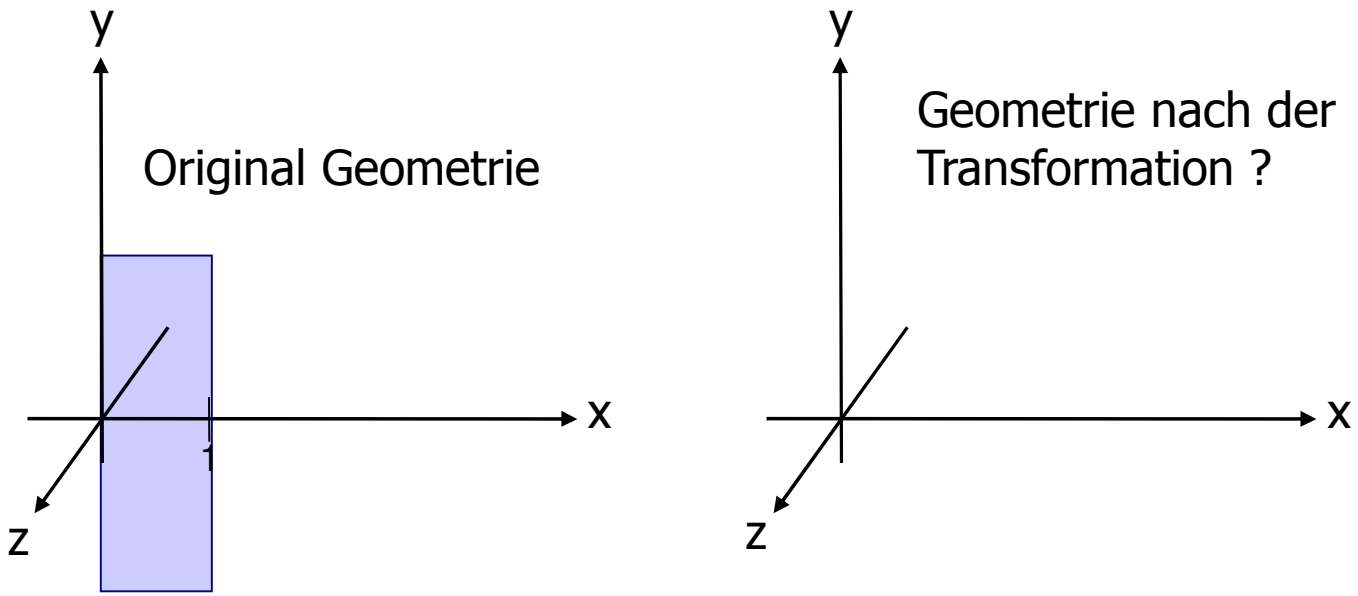
Beispiel für eine Skalierung



Skalierung um die Faktoren:

x = ?, y = ? und z = 1

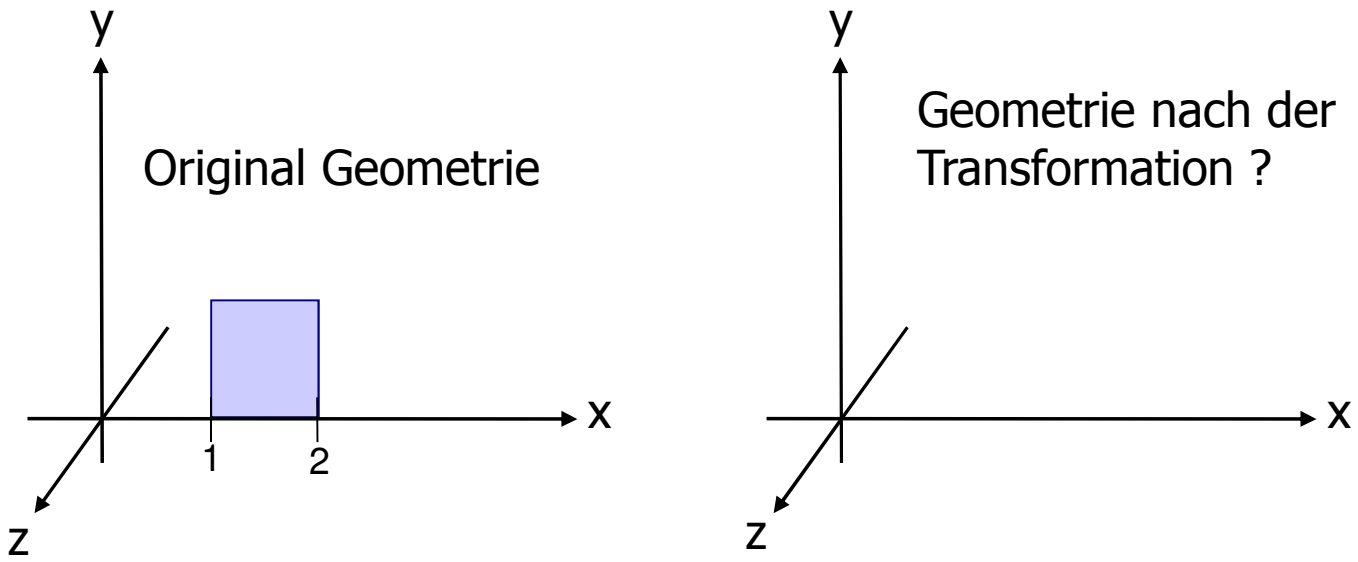
Beispiel für eine Skalierung



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

x = 3 , y = 0.3 und z = 1

Beispiel für eine Skalierung

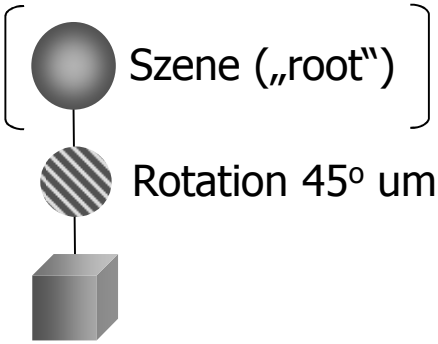


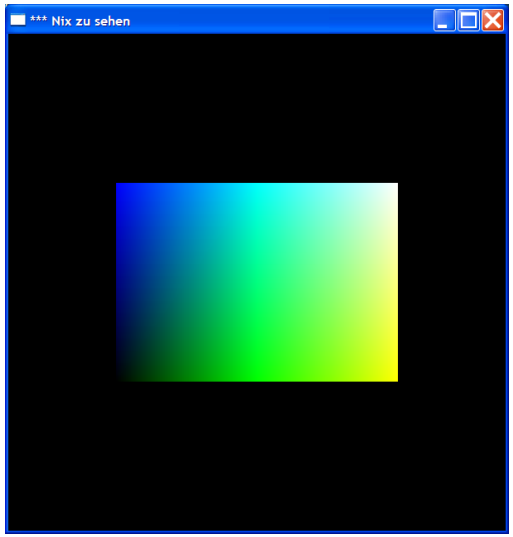
Wie sieht die Geometrie nach der Skalierung mit diesen Faktoren aus?

x = 3 , y = 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();`

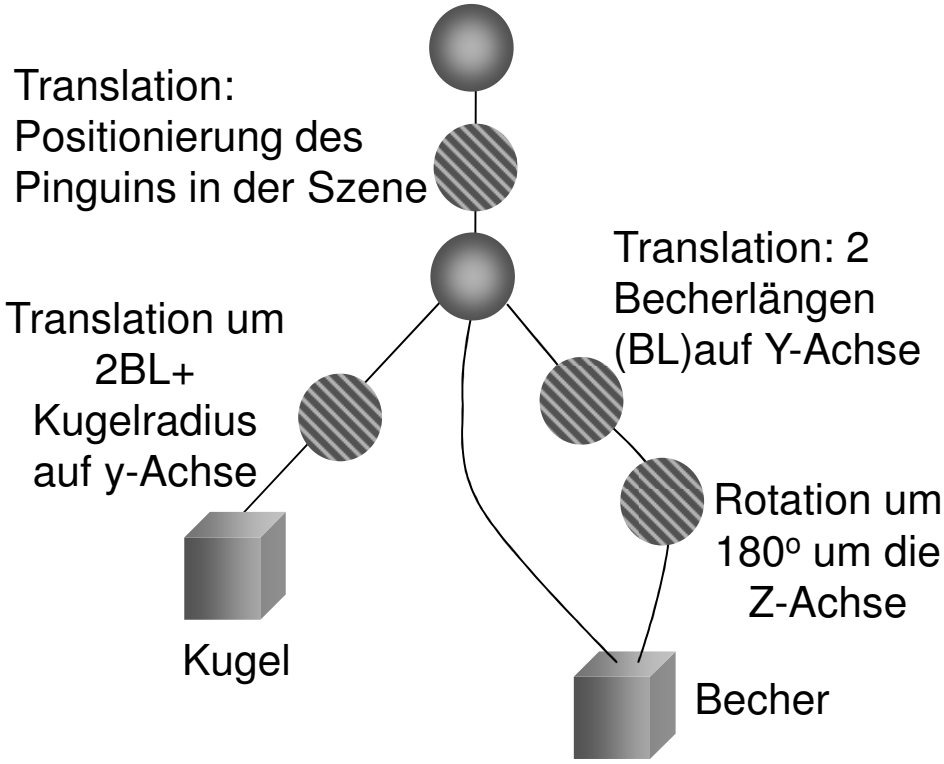
Mit dem Aufruf von `glPopMatrix()` werden alle Änderungen der aktuellen Matrix, die zwischen dem Aufruf `glPushMatrix()` und `glPopMatrix()` gemacht worden sind gelöscht.

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();
```

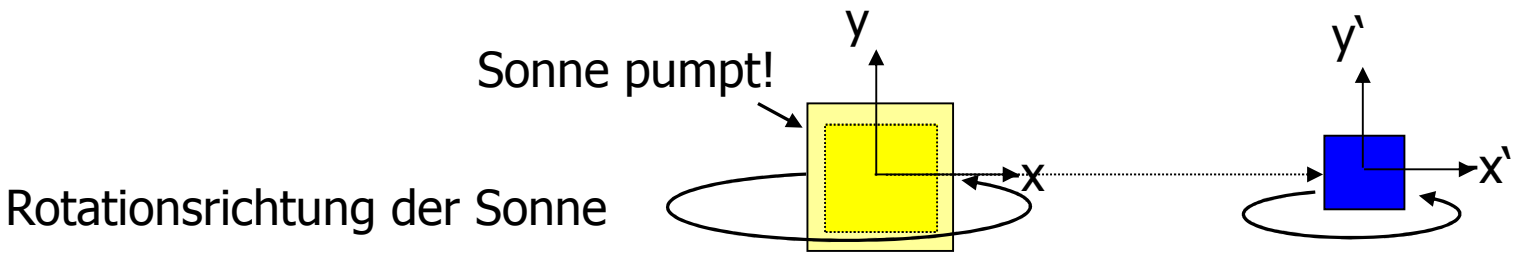
Welche Transformationen werden jeweils auf das Quadrat angewendet?

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



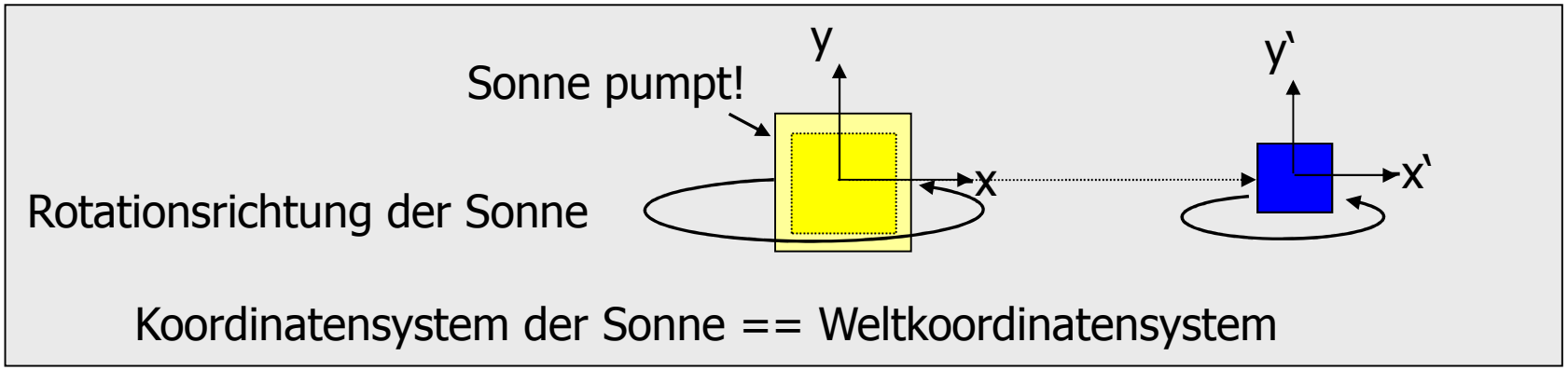
Aufgabe: Miniatursonnensystem

- 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 ;-)

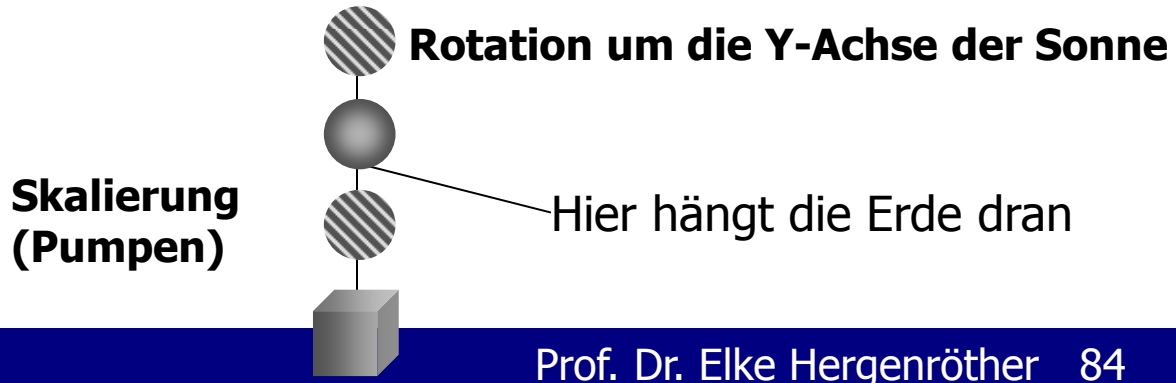


Koordinatensystem der Sonne == Weltkoordinatensystem

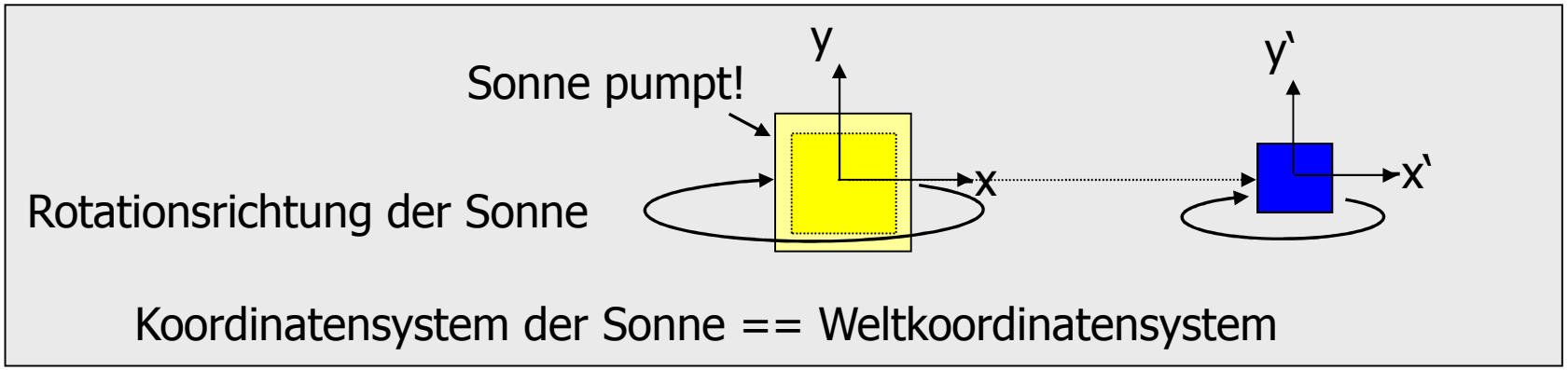
Szenengraph des Miniatursonnensystems



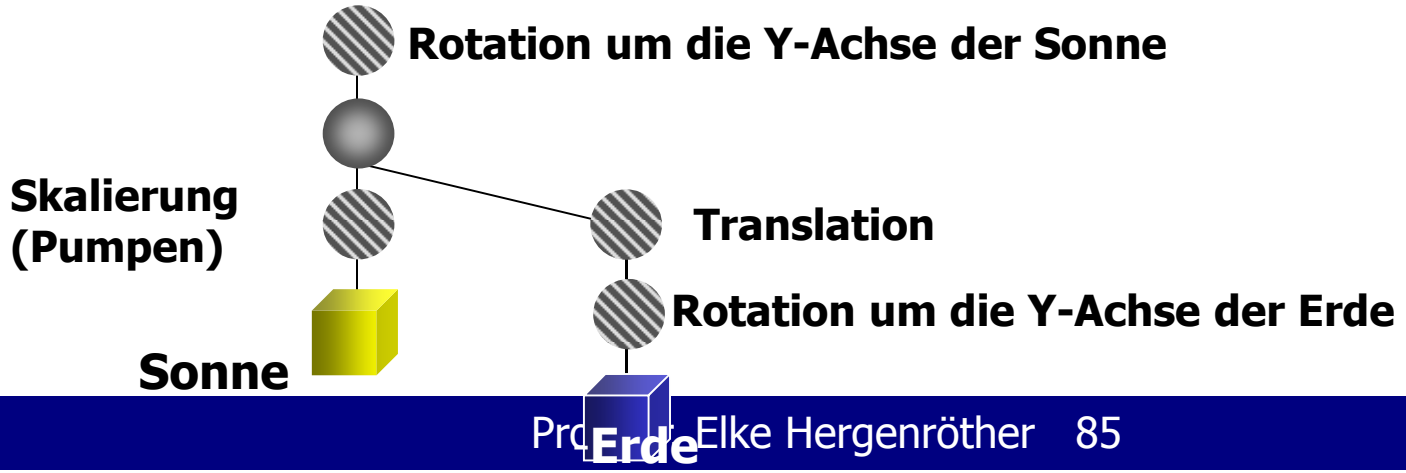
- Sonne und Erde rotieren mit gleicher Winkelgeschwindigkeit um die Y-Achse der Sonne
- Aber nur die Sonne pumpt sich auf und fällt zusammen



Szenengraph des Miniatursonnensystem

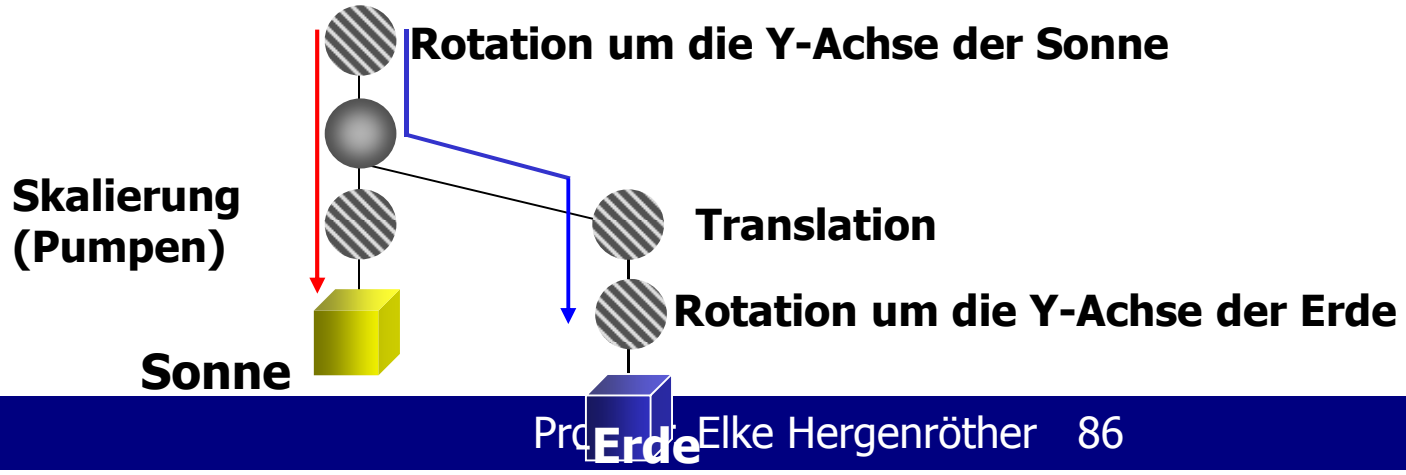


- Erde rotiert zusätzlich um ihre eigne Y-Achse



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