HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
*Fakultät IV
Wirtschaft und
Informatik*

Computergrafik I

Kapitel 4: Transformationen und Szenegraph

Prof. Dr. Ingo Ginkel

Sommersemester 2018

Transformationen und Szenegraph - Kapitelübersicht

- grundlegende Überlegungen / Vektoren vs. Punkte etc.
- Modellierung von Transformationen durch Matrizen
- Grundidee Szenegraph
- Szenegraph - Implementierungsaspekte

Transformationen - grundlegende Überlegungen

- Viele geometrische Objekte können über Punktkoordinaten (d.h. als 3D-Vektoren) beschrieben werden.
  - z.B. ist eine Würfel als Netz definiert durch die Angabe der Koordinaten seiner 8 Eckpunkte und der Konnektivität seiner Faces.
  - oder eine Ellipse durch die Angabe ihres Mittelpunktes und 2 Radien (mit Richtungsinformation! Also auch als 3D Vektor $\Rightarrow$ Skizze).
  - eine Freiform-Kurve durch die Koordinaten seiner Kontrollpunkte.

Beispiel (Definition einer Ellipse)

Transformationen - grundlegende Überlegungen

- Die gewünschte Wirkung z.B. einer Verschiebung ist aber unterschiedlich:
  - Beim Würfel und der Freiform-Kurve werden einfach die Punkte verschoben (Vektor-Addition).
  - Bei der Ellipse soll der Mittelpunkt verschoben, die Radien sollen aber unverändert bleiben.
- Ebenso die gewünschte Wirkung einer Skalierung:
  - Der Mittelpunkt soll unverändert bleiben.
  - aber die Radien sollen entsprechend vergrößert bzw. verkleinert werden.

Transformationen - grundlegende Überlegungen

- Unterscheide zwischen **absoluten Positionen** (Mittelpunkt der Ellipse, Eckpunkte des Würfels) und **relativen Grössen** (Radien, etc.).

- Aber selbst damit ist ein Skalieren nicht unproblematisch.
- Grund: es müssen ggf. absolute Positionen skaliert werden...

Beispiel (Skalieren eines Würfels)

Berechnung einer Translation - Unterscheidung absolut/relativ

- **Erster Ansatz:** Translation komponentenweise berechnen, Unterscheidung zwischen absoluter Position vs. relativer Position explizit.

- ```
  class Position
  {
    private:
      double x,y,z;
      bool isRelative;

    public:
      Position operator + (const Position& arg);
      void operator += (const Position& arg);
  };
  ```

- if-Abfrage von `isRelative` in der Implementierung von „+“ bzw. „+=“.
  ```

Berechnung einer Translation - Mathematische Sichtweise

- Mathematisch gesehen handelt es sich bei der Translation um eine Vektor-Addition.
  - Ein Punkt $\mathbf{P}$ mit Koordinaten $\begin{bmatrix} x \\ y \end{bmatrix}$ wird um $\mathbf{T} = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}$ verschoben.
  - Es entsteht ein neuer Punkt $\mathbf{P}'$ mit $\mathbf{P}' = \mathbf{P} + \mathbf{T}$
  - In Vektor-Schreibweise:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} x + t_1 \\ y + t_2 \end{bmatrix}$$

- Die Tatsache ob es sich um eine relative oder absolute Koordinate handelt, ist nicht in der Berechnung abzulesen, sondern muss separat mit-geloggt werden.

Transformationen - 2D Rotation um den Ursprung

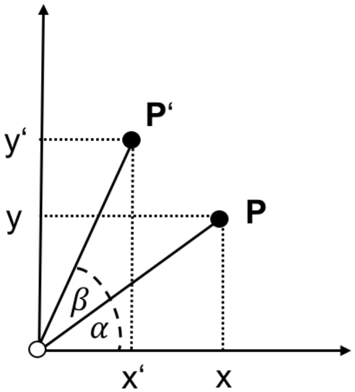
- Rotiere Punkt **P** an die Position **P‘**.

$$x = l \cdot \cos\alpha$$
$$y = l \cdot \sin\alpha$$

$$x' = l \cdot \cos(\alpha + \beta)$$
$$y' = l \cdot \sin(\alpha + \beta)$$

$$\begin{aligned} x' &= l \cdot \cos(\alpha + \beta) = l \cdot (\cos\alpha \cdot \cos\beta - \sin\alpha \cdot \sin\beta) \\ &= x \cdot \cos\beta - y \cdot \sin\beta \end{aligned}$$

$$\begin{aligned} y' &= l \cdot \sin(\alpha + \beta) = l \cdot (\sin\alpha \cdot \cos\beta + \cos\alpha \cdot \sin\beta) \\ &= y \cdot \cos\beta + x \cdot \sin\beta \end{aligned}$$



Transformationen - 2D Rotation um den Ursprung

- **Zusammenfassend:**

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

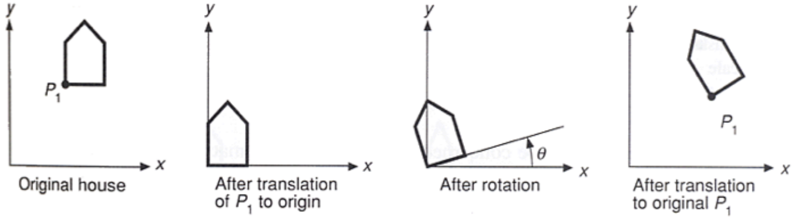
- In Matrix-Vektor Schreibweise: $\mathbf{P}' = R(\beta) \cdot \mathbf{P}$ mit der Rotationsmatrix

$$\begin{bmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{bmatrix}$$

- Die Matrix $R(\beta)$ dereht die Punkte um den Winkel β in mathematisch positiver Richtung (gegen den Uhrzeigersinn).

Transformationen - Rotationen um einen beliebigen Punkt

- Rotation um einen beliebigen Punkt $\mathbf{P}_1$
  - (1) Translation von $\mathbf{P}_1$ in den Ursprung.
  - (2) Rotation um den Ursprung
  - (3) Translation von $\mathbf{P}_1$ in die ursprüngliche Position



- **Bemerkung:** Die Matrizenmultiplikation ist nicht kommutativ, d.h. die Reihenfolge der Matrizen muss der Reihenfolge der Operationen entsprechen.

Transformationen - Komposition von Transformationen

- Die Hintereinanderausführung von Rotation, Translation und Skalierung führt auf die Abbildungsgleichung $\mathbf{P}' = S \cdot (T + R \cdot \mathbf{P})$.
- Müssenen mehrere solcher Transformationen hintereinander ausgeführt werden, so stört die Addition in der der obigen Gleichung:
- Es müsste jede Matrix-Vektor-Multiplikation sequentiell hintereinmander ausgeführt werden, das Zwischenergebnis wird für die Vektor-Addition nötig
- viel einfacher wäre es nur Matrizen zu haben, diese könnte man vor Berechnung der neuen Punktposition in eine Matrix zusammenfassen.
- Gewünscht wäre also folgende Struktur:

$$\mathbf{P}' = M_n \cdot \ldots \cdot M_3 \cdot M_2 \cdot M_1 \cdot \mathbf{P})$$

Transformationen - Komposition von Transformationen

- Ausnutzung des Assoziativität spart viele Berechnungsschritte:

Multiplikationen:

$$3 \cdot 3 \cdot 3 \cdot (\#Trans - 1) + (3 \cdot 3 \cdot \#Punkte) \overset{?}{<>} \#Trans(3 \cdot 3 \cdot \#Punkte)$$

$$3 \cdot 3 \cdot 3 \cdot (10 - 1) + 9 \cdot 1Mio << 90Mio$$

Additionen: $3 \cdot 3 \cdot 2 \cdot (\#Trans - 1) + (3 \cdot 2 \cdot \#Punkte) \overset{?}{<>} \#Trans(3 \cdot 2 \cdot \#Punkte)$

$$3 \cdot 3 \cdot 2 \cdot (10 - 1) + 6 \cdot 1Mio << 60Mio$$

- **Bemerkung:** Üblicherweise werden die Matrix-Matrix Multiplikationen CPU-seitig ausgeführt (Festlegung Positionen, Orientierung und Größe der Objekte), die finalen Matrix-Vektor-Multiplikationen (viel mehr Op's!) werden auf der Grafik-Karte berechnet.

Transformationen - Punkte vs. Vektoren

Definition 4.1 (Punkte vs. Vektoren)

- *Punkte bezeichnen. absolute Positionen im Raum*
- *Vektoren bezeichnen die Verschiebung von Punkten bzw. Richtungen zwischen Punkten im Raum.*
- *Vektoren sind als invariant unter Verschiebungen des Koordinatenursprungs.*

- Damit sind Aussagen über absolute und relative Größen unabhängig vom aktuellen Ort (= Koordinatensystem) möglich.

Noch offen:

- automatische Unterscheidung von Punkten und Vektoren ohne Sonderfall-Behandlung / if-Abfragen.

Homogene Koordinaten

Definition 4.2 (Homogene Koordinaten)

Das Quadrupel $[x, y, z, \lambda]^T$ stellt die so genannten homogenen Koordinaten des **Punktes** $[x/\lambda, y/\lambda, z/\lambda]^T \in \mathbb{R}^3$ *dar.*

Bemerkungen:

- Es gibt also unendlich viele Darstellungen $[x, y, z, \lambda]^T$, (mit $\lambda \neq 0$) ein und desselben **Punktes** $[x/\lambda, y/\lambda, z/\lambda]^T \in \mathbb{R}^3$.
- Wir verwenden meist die so genannte Standard-Darstellung mit $\lambda = 1$
  - Ausnahme: Erstelle eine Matrix um eine Projektion auszuführen: $\Rightarrow$ später.
- die Definition gilt analog für $[x, y, \lambda]^T$ und $[x/\lambda, y/\lambda]^T \in \mathbb{R}^2$
- Den bisher ausgeschlossenen Fall $\lambda = 0$ kann man nutzen um die homogenen Koordinaten eines **Vektors** $[x, y, z]^T \in \mathbb{R}^3$ zu definieren als: $[x, y, z, 0]^T$

Homogene Koordinaten - Vektor-Berechnungen

- **Frage:** Welchen Nutzen bringt diese Definition für die Translation als Matrix und die **automatische** Unterscheidung von Punkten und Vektoren bzw. deren Verhalten bei Transformationen?
- Die Differenz zweier Punkte ist ein Vektor: $\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} - \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 - x_2 \\ y_1 - y_2 \\ 0 \end{bmatrix}$
- Ebenso ist die Differenz oder Summe zweier Vektoren ein Vektor.
- Addition von Punkt und Vektor ergibt einen Punkt $\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \\ 1 \end{bmatrix}$
- Die Addition zweier Punkte macht keinen Sinn, da die Zusatzkoordinate im Ergebnis weder 0 noch 1 wäre.

Homogene Koordinaten - 3D Transformationen - Translation

- Translation eines Punktes $\begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$ um den Vektor $\begin{bmatrix} t_1 & t_2 & t_3 \end{bmatrix}^T$:

$$\begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_1 \\ y + t_2 \\ z + t_3 \\ 1 \end{bmatrix}$$

- **Bemerkung:** Ein Vektor ist invariant bezüglich Translation:

Homogene Koordinaten - 3D Transformationen - Skalierung

- Skalierung eines Punktes $\begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$ um die Faktoren s_1, s_2, s_3

$$\begin{bmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot s_1 \\ y \cdot s_2 \\ z \cdot s_3 \\ 1 \end{bmatrix}$$

- **Bemerkung 1:** Obwohl ein Punkt eine absolute Größe ist, kann sie also skaliert werden (Anwendung: z.B. Würfel).
- **Bemerkung 2:** Ein Vektor ist **nicht** invariant bezüglich Skalierung:

Homogene Koordinaten - Rotationen - zunächst 2D

- Rotation eines Punktes $\begin{bmatrix} x & y & 1 \end{bmatrix}^T$ um den Winkel φ um den Ursprung

$$\begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\varphi - y\sin\varphi \\ x\sin\varphi + y\cos\varphi \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

- Rotation von $\begin{bmatrix} x & y & 1 \end{bmatrix}^T$ um den Winkel φ um den Punkt $\begin{bmatrix} P_x & P_y \end{bmatrix}^T$

$$\begin{bmatrix} 1 & 0 & P_x \\ 0 & 1 & P_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -P_x \\ 0 & 1 & -P_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

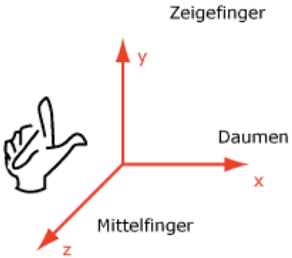
- **Bemerkung:** Auswewrtungsreihenfolge von Rechts nach Links!

Homogene Koordinaten - 3D Rotationen

- Alle Rotationen erfolgen im mathematisch positiven Sinn (d.h. gegen den Uhrzeigersinn)
- Der Betrachter „sitzt“ dabei auf der Rtotationsachse und schaut in Richtung Ursprung des Koordinatensystems.
- Wir betrachten zunächst die Rotationen um die einzelnen Koordinatenachsen um den Winkel φ

$\Rightarrow$ Transformationsmatrizen $R_x(\varphi)$, $R_y(\varphi)$, $R_z(\varphi)$

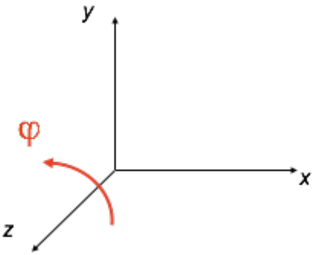
Rechtshändiges Koordinatensystem



Homogene Koordinaten - 3D Rotation um die z-Achse

- Rotation eines Punktes $\begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$ um den Winkel φ um die z-Achse

$$\underbrace{\begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{=:R_z(\varphi)} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\varphi - y\sin\varphi \\ x\sin\varphi + y\cos\varphi \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z \\ 1 \end{bmatrix}$$

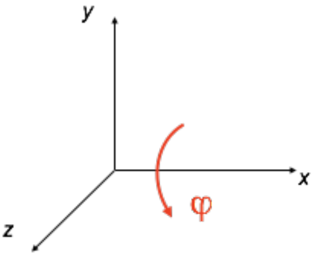


- **Bemerkung:** Drehung um den Winkel φ um die z-Achse entspricht dem 2D-Fall, wobei die z-Koordinate konstant bleibt.

Homogene Koordinaten - 3D Rotation um die z-Achse

- Rotation eines Punktes $\begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$ um den Winkel φ um die x-Achse

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{=:R_x(\varphi)} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y\cos\varphi - z\sin\varphi \\ y\sin\varphi + z\cos\varphi \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y' \\ z' \\ 1 \end{bmatrix}$$

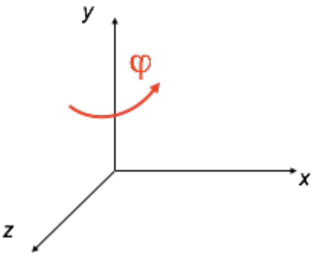


- **Bemerkung:** Drehung um den Winkel φ um die x-Achse entspricht dem 2D-Fall, wobei die x-Koordinate konstant bleibt.

Homogene Koordinaten - 3D Rotation um die z-Achse

- Rotation eines Punktes $\begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$ um den Winkel φ um die y-Achse

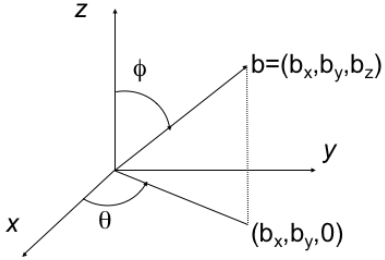
$$\underbrace{\begin{bmatrix} \cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{=:R_y(\varphi)} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} z\sin\varphi + x\cos\varphi \\ y \\ z\cos\varphi - x\sin\varphi \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y \\ z' \\ 1 \end{bmatrix}$$



- **Bemerkung:** Drehung um den Winkel φ um die y-Achse entspricht dem 2D-Fall, wobei die y-Koordinate konstant bleibt.

3D Rotation um eine beliebige Achse durch den Ursprung

- Jede Rotation um eine beliebige Achse kann aus Rotationen um die einzelnen Koordinatenachsen zusammengesetzt werden.
- **Leonard Euler (1707-1783):** Darstellung von $\mathbf{b} = b_x, b_y, b_z$ durch ψ und θ, zusätzliche Rotation um diese Achse beschreibt die Lage eines Objektes im Raum



- **Zunächst Sonderfall:** Erzeuge Rotationsmatrix $R_G(\alpha)$ wobei die Drehachse G durch den Ursprung geht und von einem Vektor $\mathbf{b} = \begin{bmatrix} b_x & b_y & b_z \end{bmatrix}^T$ mit $\|\mathbf{b}\| = 1$ generiert wird.

3D Rotation um eine beliebige Achse durch den Ursprung - Schritt 1

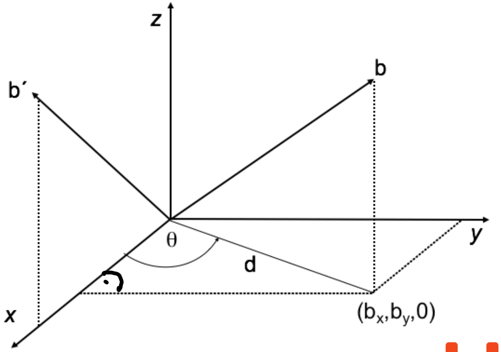
Schritt 1: Drehe den Vektor **b** in die *zx*-Ebene.

- Dazu Rotation $R_z(-\theta)$ mit Winkel $-\theta$ um die z-Achse
- Für die Rotationsmatrix $R_z(-\theta)$ werden die Werte $\cos(-\psi)$ und $\sin(-\theta)$ benötigt.
- Berechne diese direkt:

$d = \sqrt{b_x^2 + b_y^2}$ und damit

$$\cos(-\theta) = \cos(\theta) = \frac{b_x}{d},$$

$$\sin(-\theta) = -\sin(\theta) = -\frac{b_y}{d}$$



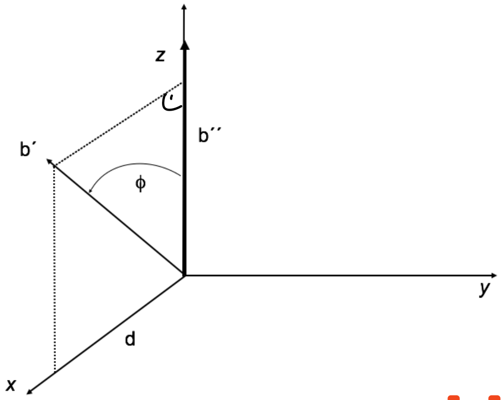
3D Rotation um eine beliebige Achse durch den Ursprung - Schritt 2

Schritt 2: Drehe den resultierenden Vektor $\mathbf{b}'$ auf die z-Achse.

- Dazu Rotation $R_y(-\phi)$ mit Winkel $-\phi$ um die y-Achse
- Für die Rotationsmatrix $R_y(-\phi)$ werden die Werte $\cos(-\phi)$ und $\sin(-\phi)$ benötigt.
- Berechne diese direkt:

$$\cos(-\phi) = \cos(\phi) = \frac{b_z}{1} = b_z,$$

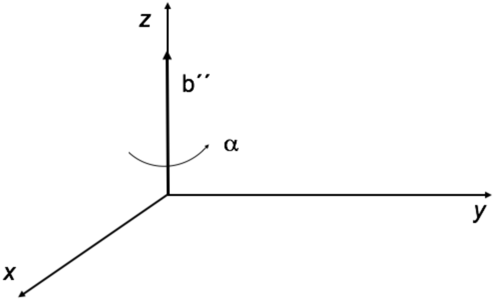
$$\sin(-\phi) = -\sin(\phi) = -\frac{d}{1} = -d$$



3D Rotation um eine beliebige Achse durch den Ursprung - Schritt 3

Schritt 3: Drehe den resultierenden Vektor $\mathbf{b}''$ um die z-Achse um den Winkel α.

- Also Rotation $R_z(-\alpha)$ mit Winkel α um die z-Achse
- hier jetzt $\cos\alpha$ und $\sin\alpha$ berechnen.



- Nach dieser Rotation müssen die Hilfs-Rotationen aus den Schritten 1 und 2 wieder rückgängig gemacht werden.
- Basierend auf der verwendeten Reihenfolge heißt diese Vorgehensweise zyz-Konvention.

3D Rotation um eine beliebige Achse - Gesamtergebnis

- **Rotation um eine beliebige Gerade durch den Ursprung:**

  Verknüpfung aller Einzel-Rotationen:

  $$R_b(\alpha) = R_z(\theta) \cdot R_y(\phi) \cdot R_z(\alpha) \cdot R_y(-\phi) \cdot R_z(-\theta)$$

- **Rotation um eine allgemeine Gerade:**

  $G : a + \lambda \mathbf{b}, \lambda \in \mathbb{R}, \|\mathbf{b}\| = 1, a = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix}^T.$

  Hier ist vor und nach der obigen Rotation noch eine entsprechende Translation einzufügen, also:

  $$R_G(\alpha) = T(a_x, a_y, a_z) \cdot R_z(\theta) \cdot R_y(\phi) \cdot R_z(\alpha) \cdot R_y(-\phi) \cdot R_z(-\theta) \cdot T(-a_x, -a_y, -a_z)$$

Szenegraph - Idee

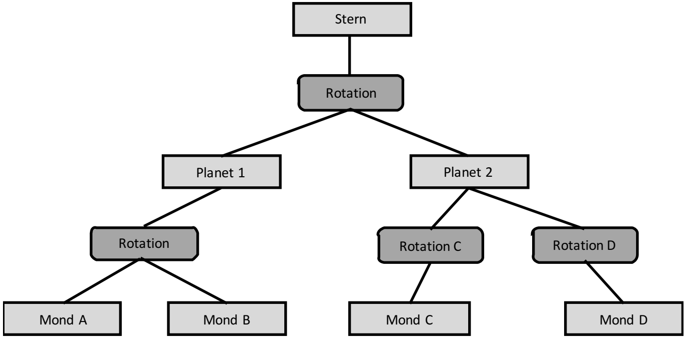
Implementierungsstand bisher:

- ein einzelnes Objekt wird mit einer Transformationsmatrix versehen und gezeichnet.
    - Zur Darstellung von z.B. Normalen werden weitere Objekte benutzt, die mit derselben Transformationsmatrix gezeichnet werden.
    - Für mehrere Objekte schon umständlich, verschiedene Transformationsmatrizen für verschiedene Objekte und Objekt-Gruppen (Normalen)

Wünschenswert: Mechanismus für Objektgruppen und zur Modellierung von Hierarchien bei Transformationen

- **Beispiel:** Planeten rotieren um einen Stern, zusätzlich um ihre eigene Achse, Monde rotieren um die Planeten. $\Rightarrow$ Verschiedene Rotationsachsen um die rotiert wird.
- **Hierarchie:** Mond-Rotationen (d.h. die konkrete Position eines Mondes) ist von der eigenen Rotation um den Planeten und von dessen Rotation um den Stern abhängig

Szenegraph - Beispiel



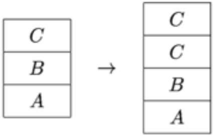
Abarbeitung:

```
zeichne den Stern
speichere die Aktuelle Matrix
    wende eine Rotation an
    zeichne Planet 1
    speichere die Aktuelle Matrix
        wende eine Rotation an
        zeichne Mond A
        zeichne Mond B
    "vergesse" die letzte Rotation
    zeichne Planet 2
    speichere die Aktuelle Matrix
        wende eine Rotation an
        zeichne Mond C
    "vergesse" die letzte Rotation
    speichere die Aktuelle Matrix
        wende eine andere Rotation an
        zeichne Mond D
    "vergesse" die letzte Rotation
"vergesse" die vorletzte Rotation
```

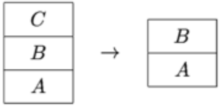
- **Annahme:** Jede Transformation kann den aktuellen Wert der Transformations-Matrix speichern bevor die eigene Transformation angewendet wird.

Szenegraph - Abarbeitung mit Matrix-Stack

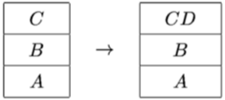
- Nutze einen Stack um das Anwenden, „Merken“ und „Vergessen“ von Transformationen zu realisieren.
  - „Merke“ entspricht dem Kopieren und erneutem Auflegen der obersten Matrix vom Stack: `void pushMatrix()`.



  - „Vergesse“ entspricht dem entfernen der obersten Matrix vom Stack: `void popMatrix()`



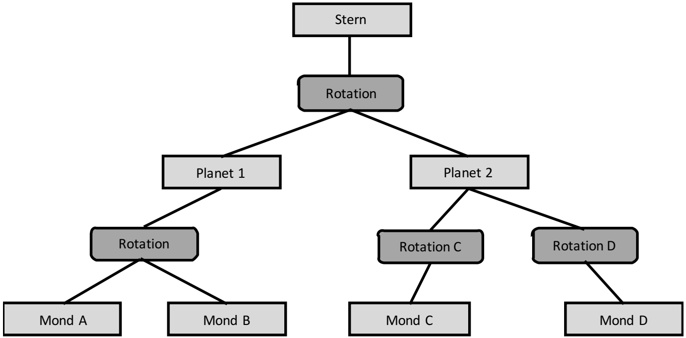
  - Anwenden einer Transformation D entspricht der Multiplikation der obersten Matrix mit einer Matrix D, die die entsprechende Operation ausführt.



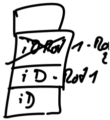
- Beim Rendering eines Objekts: übergebe immer die aktuell oberste Matrix auf dem Stack zum Zeichnen an den Renderer.

Szenegraph - Beispiel - Abarbeitung mit Stack

- **Annahme:** Initial besteht der Stack *S* aus einer Matrix, die die Transformation des Sterns enthält.

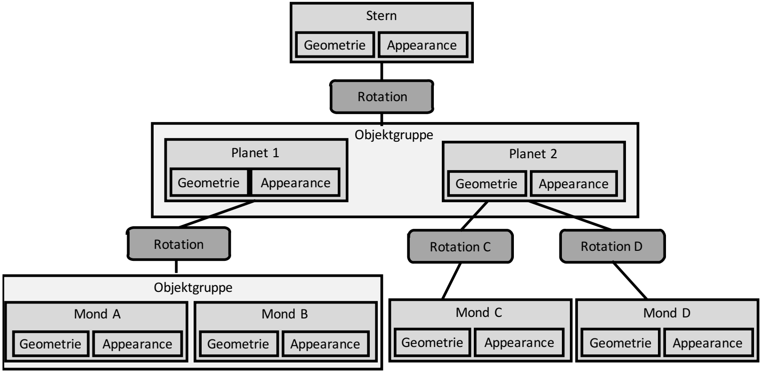


Abarbeitung:



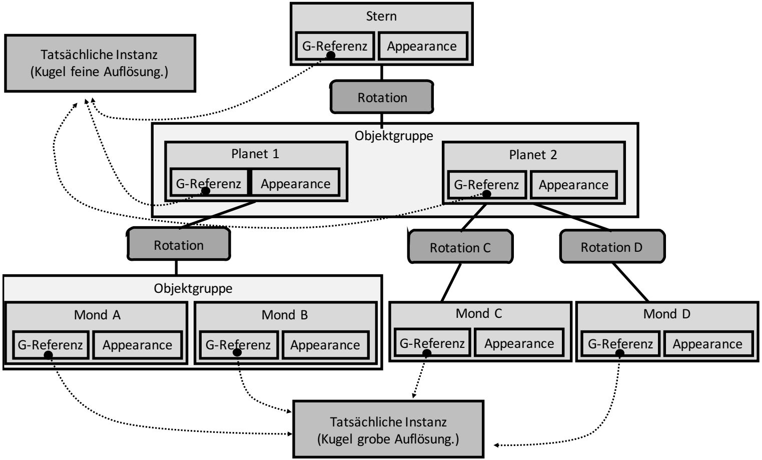

```
render (Stern,S.top())
S.pushMatrix();
    S.Top()*=Rot1;
    render (Planet1,S.top());
    S.pushMatrix();
        S.Top()*=Rot2;
        render (Mond A,S.top());
        render (Mond B,S.top())
    S.popMatrix();
    render (Planet2,S.top());
    S.pushMatrix();
        S.Top()*=RotC;
        render (Mond C,S.top())
    S.popMatrix();
    S.pushMatrix();
        S.Top()*=RotD;
        render (Mond D,S.top())
    S.popMatrix();
S.popMatrix();
```

Szenegraph - Erweiterung: Geometrie vs. Apperance



- Trennung von Geometrie und Erscheinung
- An jedes Objekt können Eigenschaften angeheftet werden:
  z.B. Farbe, Rauheit der Oberfläche, Reflexionseigenschaften, Texturen, ...
- Objektgruppen für gleiche Transformation von mehreren Objekten.

Szenegraph - Erweiterung: Geometrie vs. Apperance



- Nutze Geometrie-Referenzen
- Zeichne also ein und dieselbe Instanz mit verschiedenen Transformationen und verschiedenem Aussehen.
- Nur einmal Initialisierung des Objektes auf Grafik-Karte nötig!
- **Hier:** Beispielhaft zwei verschiedene Instanzen für eine Kugel, insgesamt aber 7x gezeichnet.

Szenegraph - Implementierungsaspekte

- Entwerfe gundlegende Struktur für Klassen, die einen Szeneraphen realisieren
- Hier: **eine** von vielen Möglichkeiten das umzusetzen

```
class CgScenegraphEntity {

private:
    std::vector<CgBaseRenderableObject*> list_of_objects;
    glm::mat4 m_current_transformation;

    CgAppearance m_apperarance;

    CgScenegraphEntity* m_parent;
    std::vector <CgScenegraphEntity*> m_children;

public:
    // entsprechende get/set Methoden

};
```

```
class CgAppearance {

private:
    glm::vec3 object_color;
    glm::vec3 difuse_material;
    ...

public:
    // entsprechende
    // get/set Methoden

};
```

- eine `CgScenegraphEntity` ist also immer potentiell eine Gruppe von Objekten
- Speicherverwaltung für die Geometrie-Objekte: $\Rightarrow$ `CgSceneControl`!

Szenegraph - Implementierungsaspekte

- `render`-Mechanismus muss jetzt den Baum ab-iterieren und die jeweils passende Matrix für den Render-Aufruf erzeugen
- Zur Übersichtlichkeit: Rendering der Szene nicht direkt im Controller (`CgSceneControl`) sondern in `CgScenegraph`:

```
class CgScenegraph {

private:
    CgScenegraphEntity* m_root_node;
    stl::stack <glm::mat4> m_modelview_matrix_stack;

    void pushMatrix() {m_modelview_matrix_stack.push_back(m_modelview_matrix_stack.back());}
    void popMatrix() {m_modelview_matrix_stack.pop_back();}
    void applyTransform(glm::mat4 arg) {m_modelview_matrix_stack.back()*=arg;}

public:
     void render(CgBaseRenderer* renderer);

     // get/set Methoden, Konstruktor, Destruktor, ...

};
```

Szenegraph - Implementierungsaspekte

Erweiterungsmöglichkeit A:

- Automatische Selektion der passenden Auflösung für ein geometrisches Objekt durch `CgSceneContol`
- Dazu Objektreferenzen durch Anfrage bei einer Factory ersetzen
  - Diese Factory erzeugt und verwaltet die Objekte
  - entscheidet selbst ob eine Auflösung neu erzeugt wird
  - gibt das Objekt in der passenden Auflösung zurück
- Trennung von Struktur (= Anordnung der Objekte) und Instanzen für verschiedene Auflösungen
- zusätzliche Infos notwendig: Kameraposition, Projektionsart etc. um spätere Größe der Objekte in der Darstellung abschätzen zu können

Szenegraph - Implementierungsaspekte

Erweiterungsmöglichkeit B:

- Automatische Selektion der passenden Auflösung für ein geometrisches Objekt durch `CgScenegraphEntity` selbst
- Dazu Objektgruppen erweitern mit einem ObjectSelect-Mechanismus
  - In der Geometrie-Objekt-Liste von `CgScenegraphEntity` werden verschiedene Objekte in verschiedenen Auflösungen hinterlegt.
  - Es wird jedoch die Liste der Objekte beim Zeichnen nicht ab-iteriert, sondern eines der Objekte ausgewählt.
  - Diese Objekte sind alle auf der Grafik-Karte in entsprechenden Buffern initialisiert, über die ID wird der jeweils passende Buffer ausgewählt.
- auch hier zusätzliche Infos notwendig: Kameraposition, Projektionsart etc. um spätere Größe der Objekte in der Darstellung abschätzen zu können