

Computergrafik II

- Geometriestufe und Transformationen -

Bachelor Medieninformatik
Wintersemester 2012/2013

Prof. Dr.-Ing. Hartmut Schirmacher
<http://schirmacher.beuth-hochschule.de>
hschirmacher@beuth-hochschule.de



- Wiederholung: WebGL, VBOs, Attribute, Indizes
- Geometrie-Stufen und Vertex Shader
- Wozu Transformationen?
- Welche Arten von 3D-Transformationen gibt es?
- Wie hängen Transformationen und Matrizen zusammen?
- Anwendung: wer macht was – Kamera, Szene und Co.

Anhang

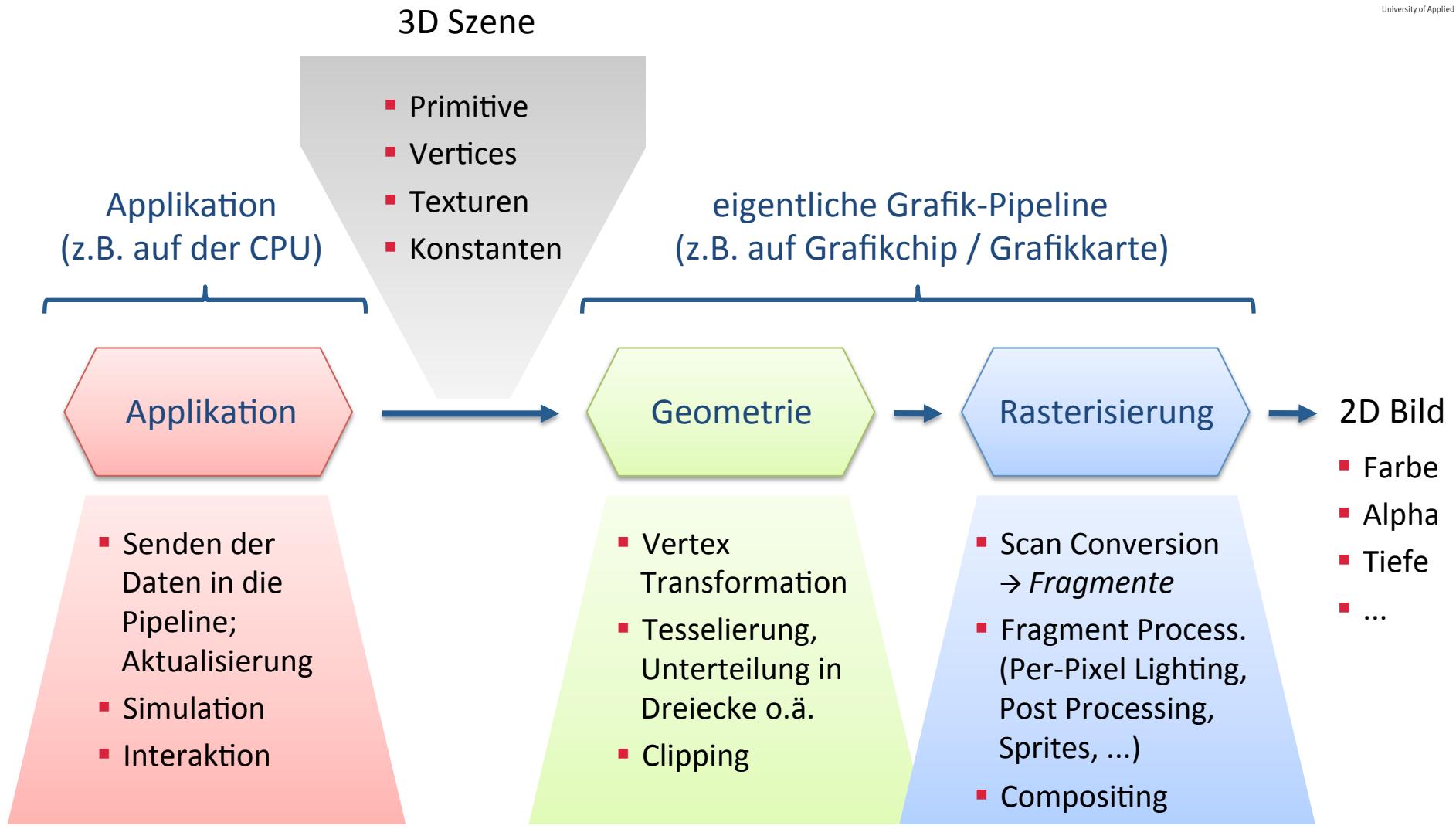
- Entwicklung der Grafik-Hardware
- Einfache Vektorrechnung



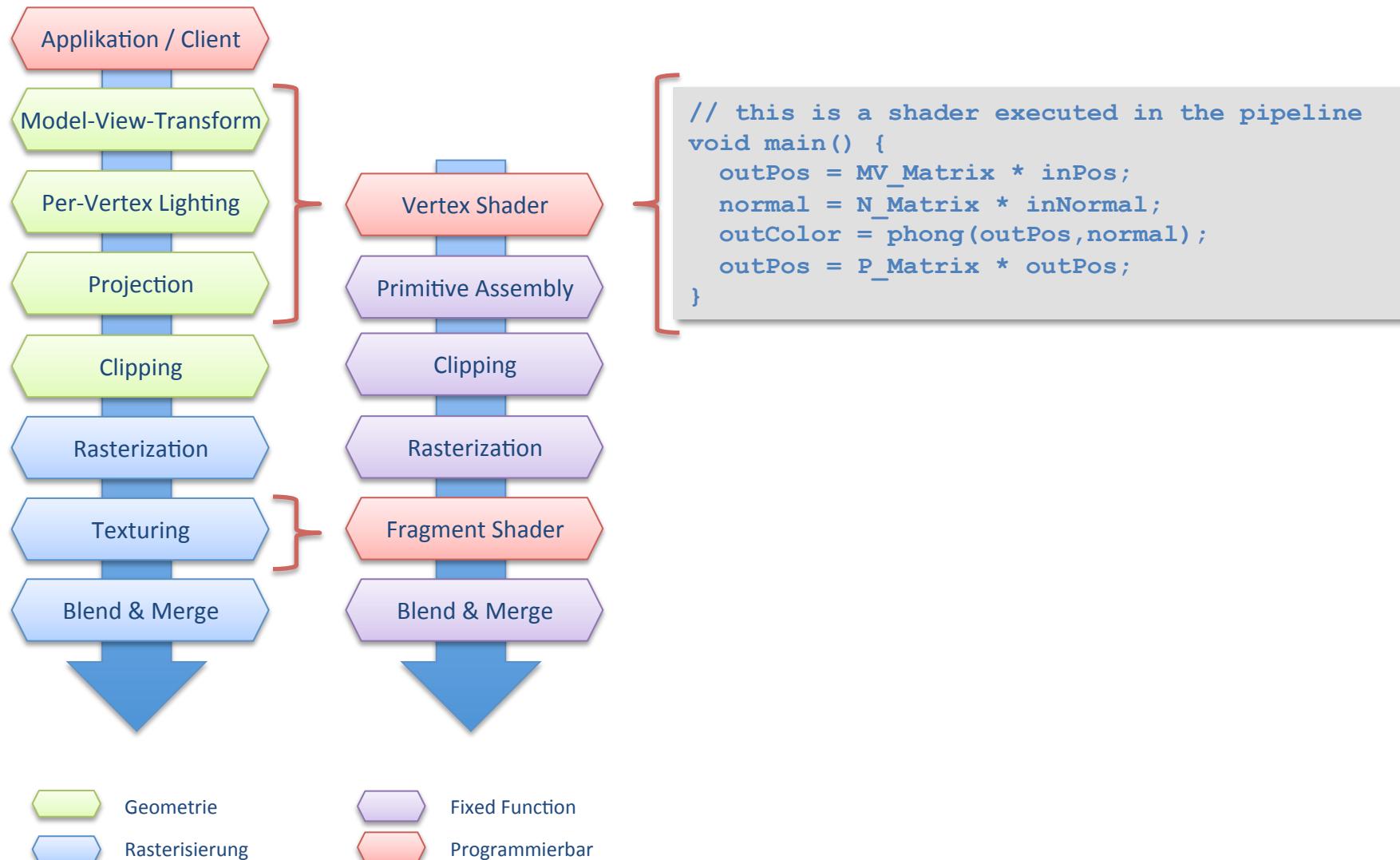
Wiederholung: WebGL, VBOs, Attribute, Indizes



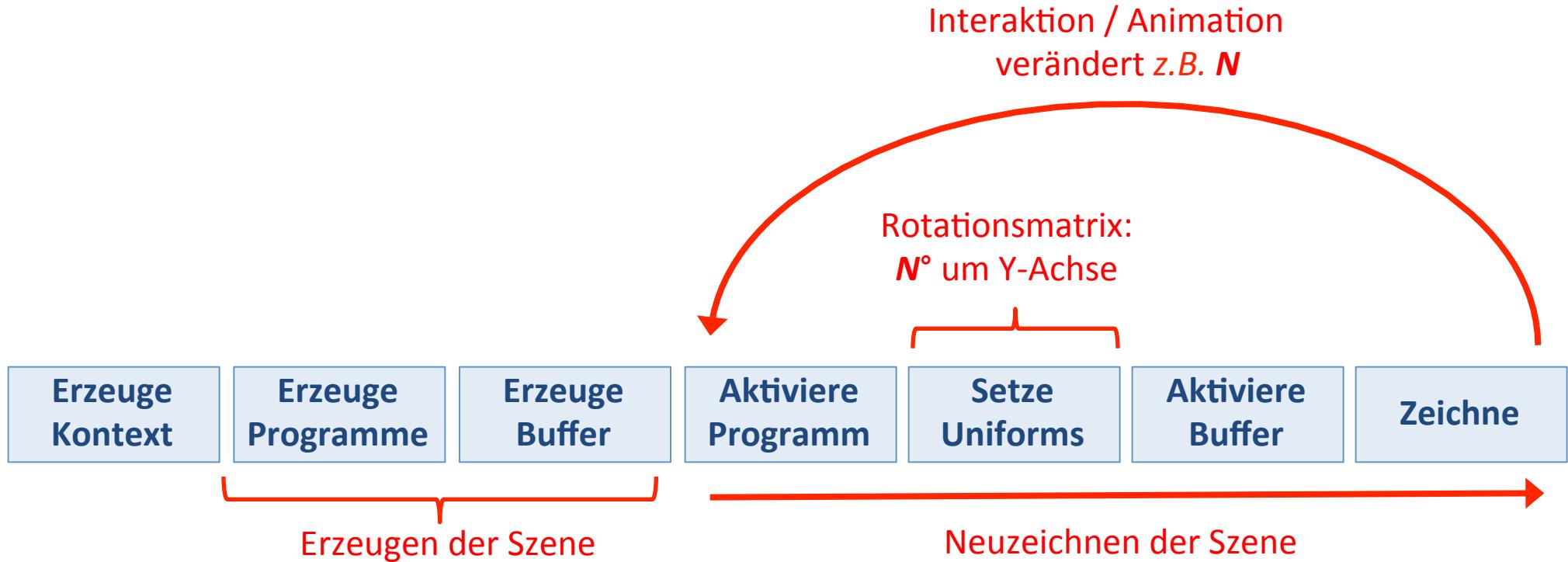
Konzeptionelles Modell der Grafik-Pipeline



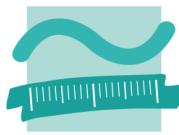
Von der klassischen zur programmierbaren Pipeline



Grundlegende Anatomie eines WebGL-Programms



Vertex Buffer Objects: Attribut-Buffer



Erzeuge
Kontext

Erzeuge
Programme

Erzeuge
Buffer

Aktiviere
Programm

Setze
Uniforms

Aktiviere
Buffer

Zeichne

Vertex-Attribut: potentiell für jeden Vertex verschiedener Wert.
Wird in einem *Vertex Buffer Object (VBO)* gespeichert und in den Speicher der Grafikkarte transferiert.

```
attribute vec3 vertexPosition;  
attribute vec4 vertexColor;  
...  
void main() { ... }
```

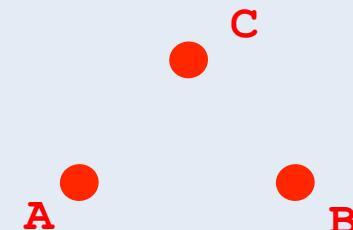
Vertex Shader

```
var coordBuf = new vbo.Attribute(gl, ...);  
var colorBuf = new vbo.Attribute(gl, ...);  
...  
coordBuf.bind(gl, program, "vertexPosition");  
colorBuf.bind(gl, program, "vertexColor");
```



- Vertex Array / Attribut-Buffer anlegen
 - z.B. mit 3 Koordinaten pro Vertex

```
var coords = [ -0.5, -0.5, 0,      // vertex A
               0.5, -0.5, 0,      // vertex B
               0,  0.5, 0 ]; // vertex C
```



```
var coordBuf = new vbo.Attribute(gl, { "numComponents": 3,
                                         "dataType": gl.FLOAT,
                                         "data": coords } );
```



Vertex-Attribute und drawArrays()

- Was kann ich mit einem Koordinaten-Array und drawArrays() erreichen?

```
var coordBuf = vbo.Attribute(...);  
coordBuf.bind(gl, program, "vertexPosition");
```

```
gl.drawArrays(gl.POINTS, 0, 3);
```



```
gl.drawArrays(gl.TRIANGLES, 0, 3);
```



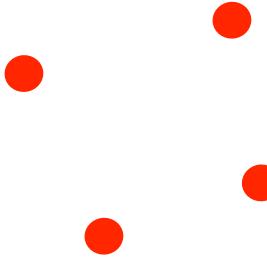
```
gl.drawArrays(gl.LINES, 0, 3);
```



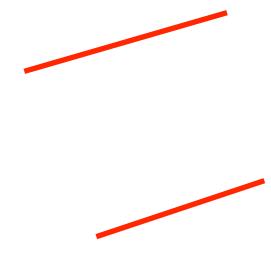
Für jede LINE
werden jeweils
zwei Vertices
benötigt!



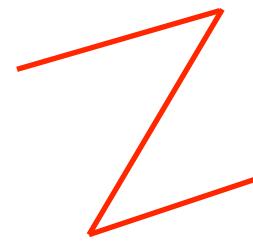
Was kann man mit vier Vertices zeichnen?



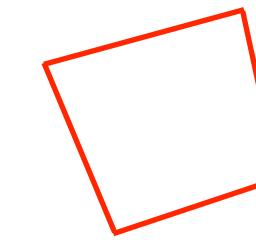
vier Punkte



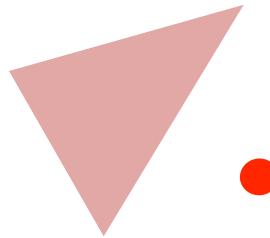
zwei Linien



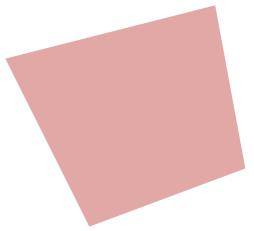
drei Linien



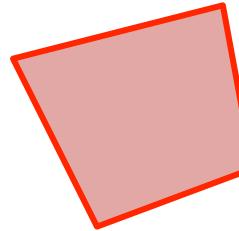
vier Linien



ein Dreieck und ein Punkt



zwei Dreiecke



zwei Dreiecke
und vier Linien

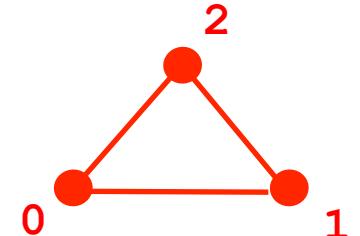
...



OpenGL Element Buffer (a.k.a. Index Buffer)

- Attribut-Buffer definiert die Vertices:

```
var coords = [ -0.5, -0.5, 0, ... ]  
var coordBuf = vbo.Attribute(..., coords);
```



- Index-Buffer definiert zusätzlich die Konnektivität:

```
var lines = [ 0, 1, 1, 2, 2, 0];  
var lineBuf = new vbo.Indices(gl, {"indices": lines});
```

- Zeichnen: drawElements() anstelle von drawArrays()

```
coordBuf.bind(gl, program, "vertexPosition");  
lineBuf.bind(gl);  
gl.drawElements(gl.LINES, 6, gl.UNSIGNED_SHORT, 0);
```



Woraus besteht ein Modell? models/triangle.js

```
// constructor, takes WebGL context object as argument
var Triangle = function(gl) {

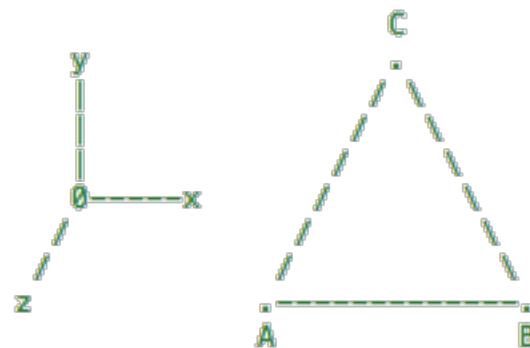
    // generate vertex coordinates and store in an array
    var coords = [ -0.5, -0.5, 0, // coordinates of A
                  0.5, -0.5, 0, // coordinates of B
                  0, 0.5, 0 // coordinates of C
                ];

    // create vertex buffer object (VBO) for the coordinates
    this.coordsBuffer = new vbo.Attribute(gl, { "numComponents": 3,
                                                "dataType": gl.FLOAT,
                                                "data": coords
                                              });
};

// draw triangle as points
Triangle.prototype.draw = function(gl, program) {

    // bind the attribute buffers
    this.coordsBuffer.bind(gl, program, "vertexPosition");

    // connect the vertices with triangles
    gl.drawArrays(gl.POINTS, 0, this.coordsBuffer.numVertices());
};
```



Erzeuge Kontext

Erzeuge Programme

Erzeuge Buffer

Aktiviere Programm

Setze Uniforms

Aktiviere Buffer

Zeichne

Konstruktor
des Modells

draw()
des Modells

Welche Aufgaben hat die Szene?

Szene **erzeugt** benötigte
Programme und Modelle

Szene bestimmt für jedes
Modell, **wo und mit welchem**
Shader es gezeichnet wird

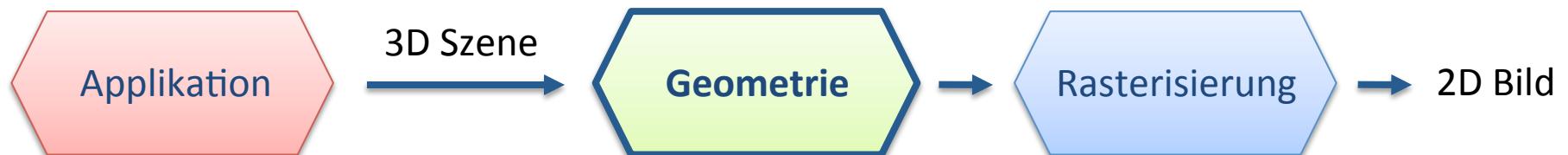
Szene ruft nacheinander
draw() von jedem
Modell auf



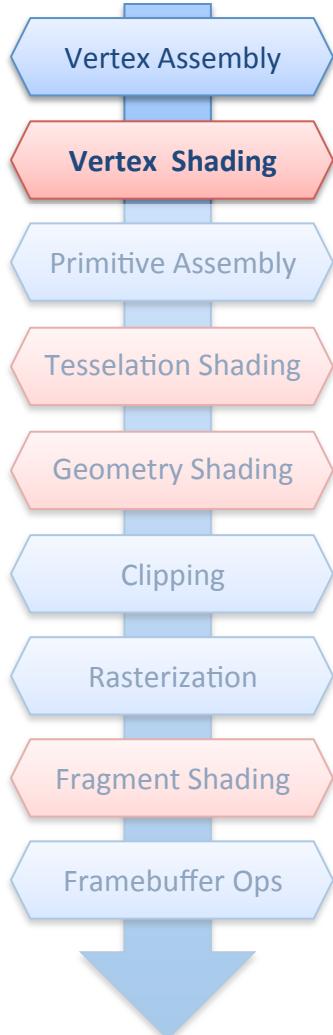
Siehe `main.js` im Übungsframework



Die Geometrie-Stufen der Grafikpipeline



drawArrays () oder
drawElements ()

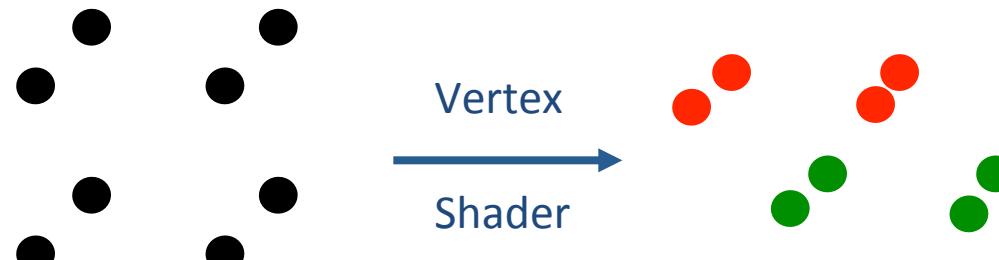


■ Vertex Assembly

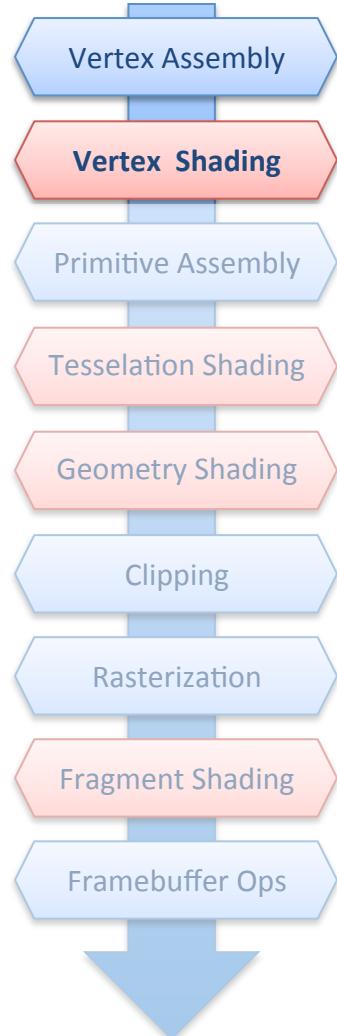
- Suche zu einem einzelnen Vertex alle **Attribute** aus potentiell mehreren *Vertex Buffers* zusammen

■ Vertex Shader

- **Transformation des Vertex in Clip-Koordinaten**
(z.B. je nach aktueller Kamera / Szenen-Transformation)
- Generierung oder Transformation weiterer Vertex-Attribute
(z.B. Normalen, Texturkoordinaten, Farben)
- Beleuchtungsberechnung per Vertex (heutzutage selten)



drawArrays () oder
drawElements ()

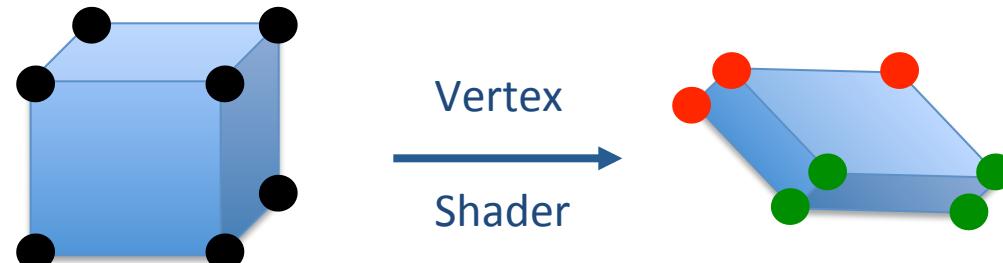


■ Vertex Assembly

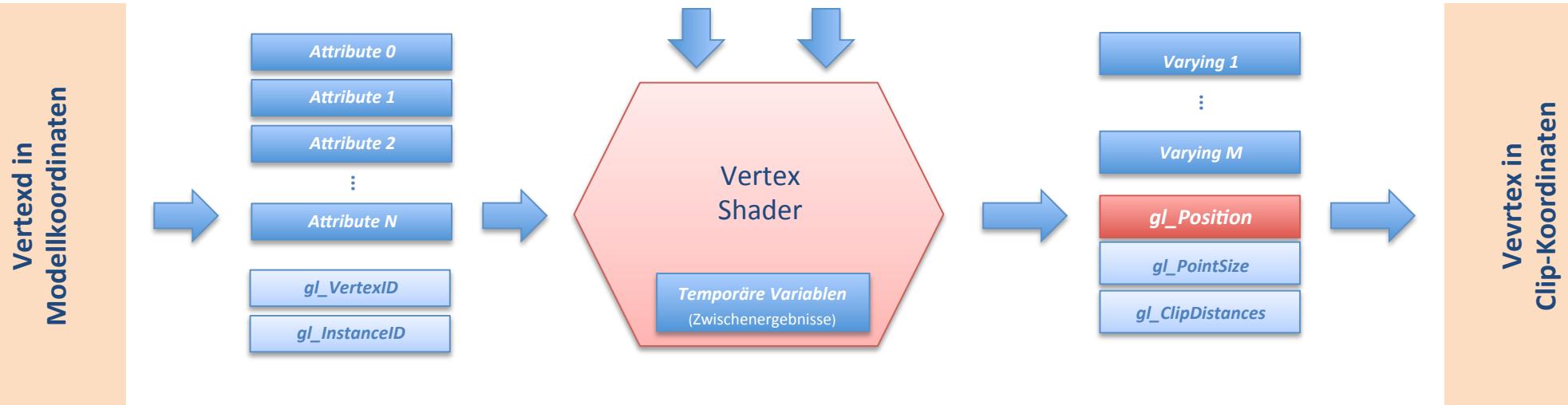
- Suche zu einem einzelnen Vertex alle **Attribute** aus potentiell mehreren *Vertex Buffers* zusammen

■ Vertex Shader

- **Transformation des Vertex in Clip-Koordinaten**
(z.B. je nach aktueller Kamera / Szenen-Transformation)
- Generierung oder Transformation weiterer Vertex-Attribute
(z.B. Normalen, Texturkoordinaten, Farben)
- Beleuchtungsberechnung per Vertex (heutzutage selten)



Ein- und Ausgabe des Vertex Shaders



- Aufruf des Vertex Shaders
 - Massiv parallel für jeden Vertex, der mittels OpenGL erzeugt wird
 - Der Vertex-Shader kennt nur einen einzelnen Vertex, keine Nachbarschaft
 - Ist ein flexibles Rechenprogramm mit Bedingungen, Unterfunktionen und temporären Variablen

- Eingabewerte
 - Per-Vertex-Informationen (attribute)
 - Globale Variablen (uniform) und Texturen (sampler)
- Ausgabewerte:
 - **gl_Position**: fest eingebaute Variable zur Festlegung der Vertex-Position in Clip-Koordinaten
 - Varying-Variablen → Eingabe für weitere Shader



Beispiel: Transformation im Vertex-Shader

attribute: Vertex-Attribut, erhält Daten aus einem Attribut-Buffer (VBO)
Die Position ist potentiell für jeden Vertex verschieden.

```
attribute vec3 vertexPosition;  
uniform mat4 modelViewMatrix;  
uniform mat4 projectionMatrix;
```

```
void main() {
```

```
    gl_Position = projectionMatrix * modelViewMatrix *  
                 vec4(vertexPosition, 1.0);
```

```
}
```

Zwei Matrizen werden als globale Variablen (**uniform**) übergeben. Sie sind für alle Vertices während eines `gl.draw*`-Befehls konstant.

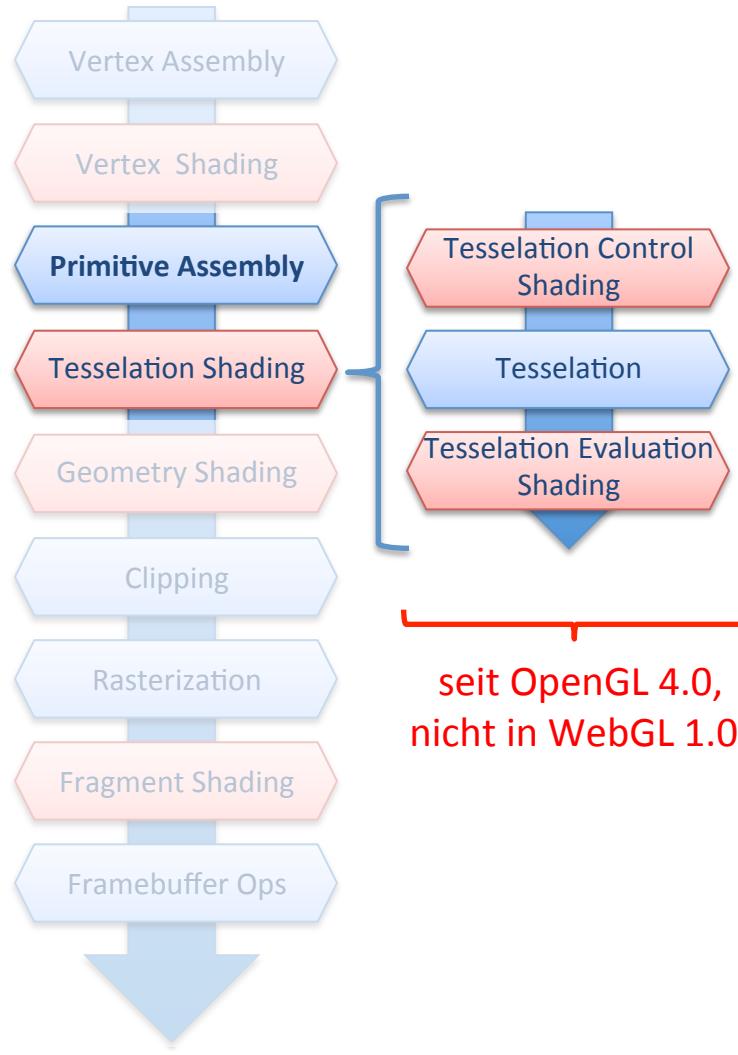
Wandelt den `vec3` in einen `vec4` mit $w = 1$ um.

Matrix-Multiplikationen liest man von rechts nach links:
wende erst `modelViewMatrix`, dann `projectionMatrix` an.

Ziel: `gl_Position` muss nach Ausführung des Shaders die Vertex-Position in *Clip-Koordinaten* enthalten.



Primitive Assembly und Tesselation Shading



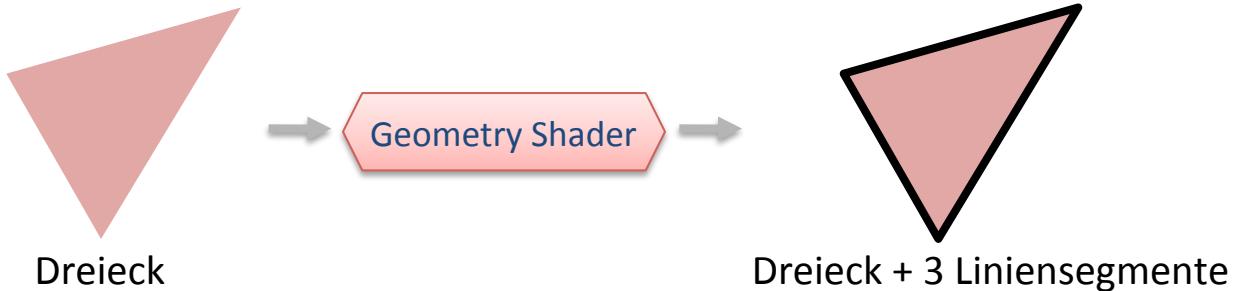
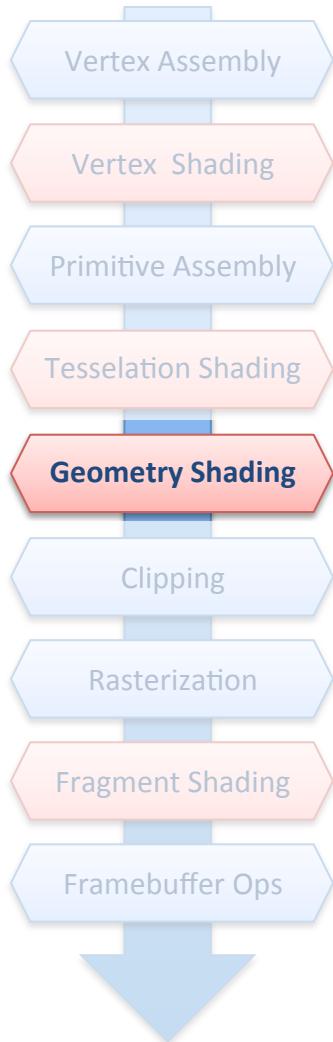
Tesselation Shader Demo
Dynamische Unterteilung,
Zielform: Kugel
Quelle: Philip Rideout, prideout.net

- Diese Stufen operieren nicht auf einzelnen Vertices, sondern auf *Primitiven*
- Primitive Assembly sammelt alle Vertex-Daten zu einem *Primitiv* zusammen (Linie, Dreieck, ...)
- Fasse N Vertices zu einem *Patch* zusammen.
 - Z.B. N = 16 für einen Bézier-Patch
- Tess. Shader hat Zugriff auf alle Vertices in Patch
- Tess. Shader entscheidet dynamisch, ob Patch in Unter-Patches unterteilt wird
 - je nach Abstand Auge-Dreieck,
 - auf der Silhouette stärker als innen,
 - ...

<http://prideout.net/blog/?p=48>

Geometry Shading

nicht in WebGL 1.0!

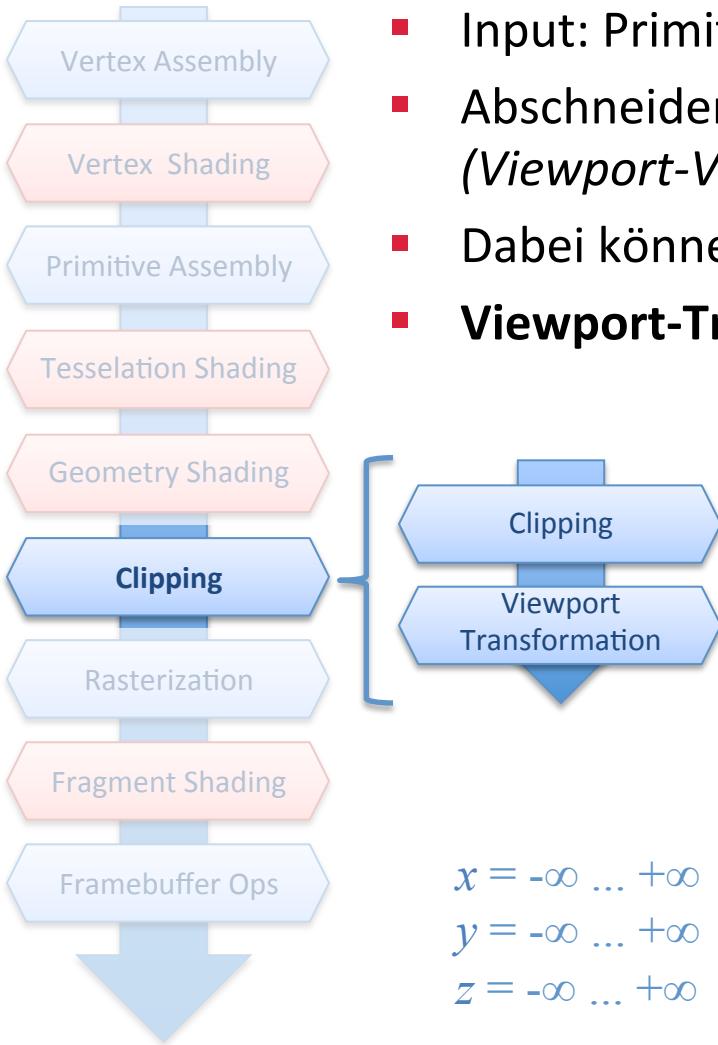


- **Input**
 - einzelne Primitive (Punkte, Linien, Dreiecke)
- **Output**
 - beliebig viele neue Primitive eines bestimmten Typs (Punkt, Linie, oder Dreieck)
- **Arbeitsweise**
 - Lege fest, welchen Primitivtyp der G-Shader erzeugen soll (z.B. LINE)
 - Erzeuge neue Vertices → und somit neue Primitive
 - Shader hat Zugriff auf alle Vertices eines Primitivs *sowie die direkten Nachbarn*

G-Shader wurden vor den Tesselierungs-Shadern eingeführt; T-Shader sind in vielen Fällen effizienter bei der Erzeugung neuer Geometrie



Clipping und Viewport-Transformation



- Input: Primitive in *Clip-Koordinaten*
- Abschneiden der Primitive an den **Grenzen des Viewports** (*Viewport-Volumen, alle Koordinaten von -1 ... 1*)
- Dabei können neue Primitive / Vertices entstehen!
- **Viewport-Transformation** in Fenster-Koordinaten

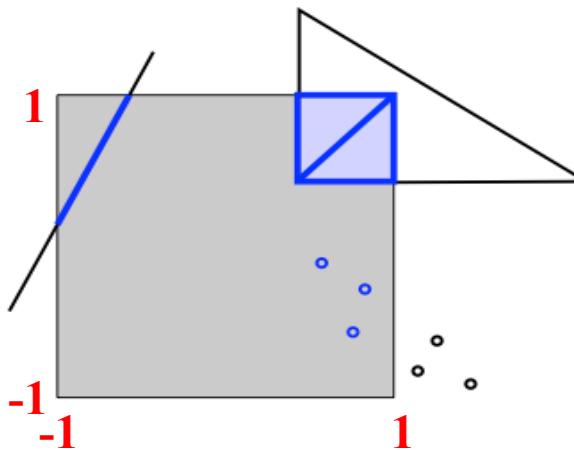


Abb. mit freundlicher Genehmigung von Henrik Tramberend, Beuth Hochschule für Technik Berlin

Matrizen und Transformationen

„is this the dreaded math chapter?“

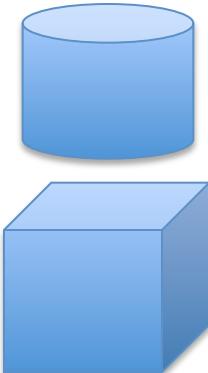


Wozu Transformationen?

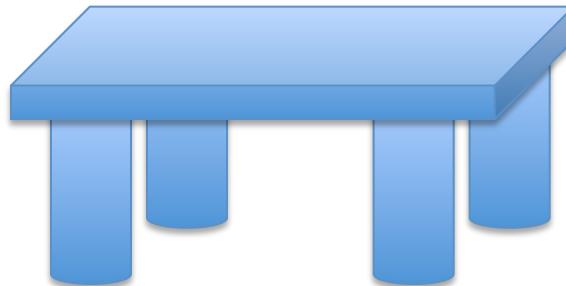
- Platzieren mehrerer Objekte in einer Szene
- Animation und Deformation von Objekten
- Transformation der gesamte Szene
- Positionierung der Kamera
- Öffnungswinkel / „Objektiv“ der Kamera

} Model-View-Transformation

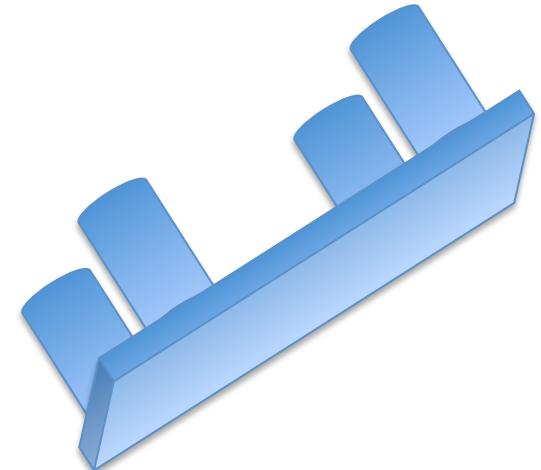
} Projektion



1 Zylinder + 1 Würfel



Würfel und Zylinder instanziert,
positioniert und deformiert



Aus Sicht der
virtuellen Kamera



- Lineare Abbildung F : Für Vektoren \mathbf{u}, \mathbf{v} und einen Skalar s gilt:

- Additivität: $F(\mathbf{u} + \mathbf{v}) = F(\mathbf{u}) + F(\mathbf{v})$
- Homogenität: $F(s \cdot \mathbf{u}) = s \cdot F(\mathbf{u})$

- x' , y' und z' werden als lineare Kombinationen von x , y und z berechnet:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = F(x, y, z) = \begin{pmatrix} a_{11}x + a_{12}y + a_{13}z \\ a_{21}x + a_{22}y + a_{23}z \\ a_{31}x + a_{32}y + a_{33}z \end{pmatrix} \quad (1)$$

- Daher ist die Matrix-Schreibweise anwendbar:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = F(x, y, z) = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{A}_{(3,3)} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Multiplikation Matrix-Vektor
ergibt wieder Gleichung (1)

- Verkürzte Schreibweise mit Vektor $\mathbf{u} = (u_x, u_y, u_z)^T$:

$$\mathbf{u}' = F(\mathbf{u}) = \mathbf{A}\mathbf{u}$$



- **Matrix A** mit m Zeilen und n Spalten
 - Element a_{ij} ist in Zeile i , Spalte j

$$\mathbf{A}_{(m,n)} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

- **Transponierte \mathbf{A}^T**
 - Vertausche Zeilen mit Spalten

$$\mathbf{A}_{(m,n)}^T = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}$$

- Beispiel: $\mathbf{A} = \begin{bmatrix} 8 \\ 23 \\ 14 \\ 1 \end{bmatrix} \equiv \mathbf{A}^T = \begin{bmatrix} 8 \\ 23 \\ 14 \\ 1 \end{bmatrix}^T = [8 \ 23 \ 14 \ 1]$



■ Addition zweier Matrizen

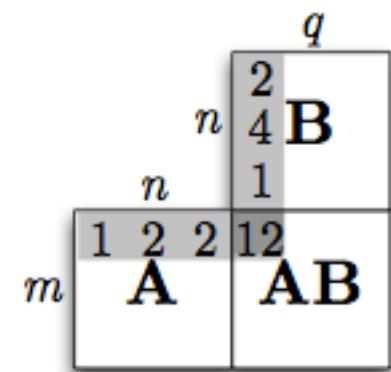
- Elementweise Addition $(\mathbf{A} + \mathbf{B})_{ij} = a_{ij} + b_{ij}$
- Eigenschaften:
 - $(\mathbf{A} + \mathbf{B}) + \mathbf{C} = \mathbf{A} + (\mathbf{B} + \mathbf{C})$ (assoziativ)
 - $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$ (kommutativ)

■ Multiplikation zweier Matrizen

- Berechnung:

$$\mathbf{C}_{(m,q)} = \mathbf{A}_{(m,n)} \mathbf{B}_{(n,q)} \quad \Rightarrow \quad C_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

- Zeilen- und Spaltenanzahl müssen kompatibel sein
- „Zeile der ersten Matrix, Spalte der zweiten Matrix“
- Eigenschaften:
 - $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$ (assoziativ)
 - $\mathbf{AB} \neq \mathbf{BA}$ (nicht kommutativ!)



Falk'sches Schema

Abb. mit freundlicher Genehmigung von Henrik Tramberend, Beuth Hochschule für Technik Berlin



- **I: Identität bzgl. der Multiplikation**

- $\mathbf{AI} = \mathbf{IA} = \mathbf{A}$

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

- **\mathbf{A}^{-1} : Inverse bzgl. der Multiplikation**

- Wenn für $\mathbf{A}_{(m,m)}$ gilt: $\mathbf{P} = \mathbf{AQ}$
 - gibt es dann ein $\mathbf{A}^{-1}_{(m,m)}$ mit $\mathbf{Q} = \mathbf{A}^{-1}\mathbf{P}$?
 - Wenn ja, dann ist \mathbf{A}^{-1} die **Inverse von A**

Berechnung: 1/Determinante * Adjunkte
ca. 95 Mults + 49 Adds in `glmatrix.js`

Einsetzen: $\mathbf{Q} = \mathbf{A}^{-1}\mathbf{P} = \mathbf{A}^{-1}(\mathbf{AQ}) = (\mathbf{A}^{-1}\mathbf{A})\mathbf{Q}$

→ also ist $(\mathbf{A}^{-1}\mathbf{A}) = \mathbf{I}$, also $(\mathbf{A}^{-1}\mathbf{A}) = (\mathbf{AA}^{-1}) = \mathbf{I}$



Beispiele für 3D-Transformationen

- Skalierung
- Spiegelung
- Scherung
- Rotation

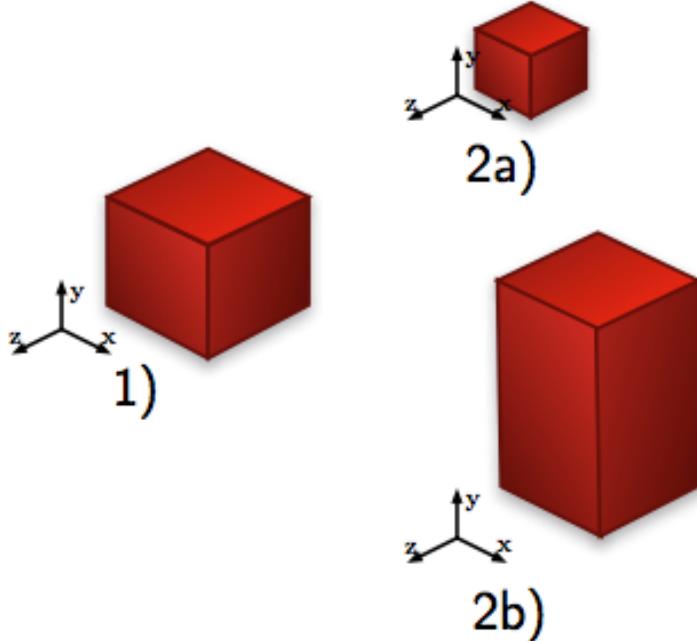


Wiederholen:

- Berechnung der Operatoren
- Darstellung in 3x3-Matrixform



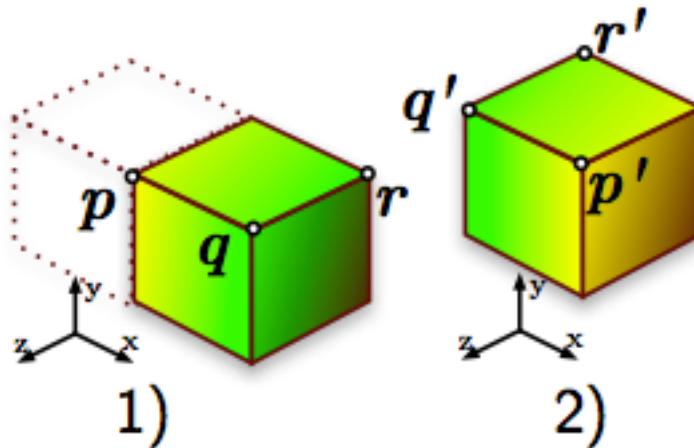
Skalierung in y-Richtung



- Berechnung
- Matrixform
- Inverse

Abb. mit freundlicher Genehmigung von Henrik Tramberend, Beuth Hochschule für Technik Berlin

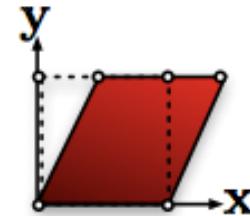
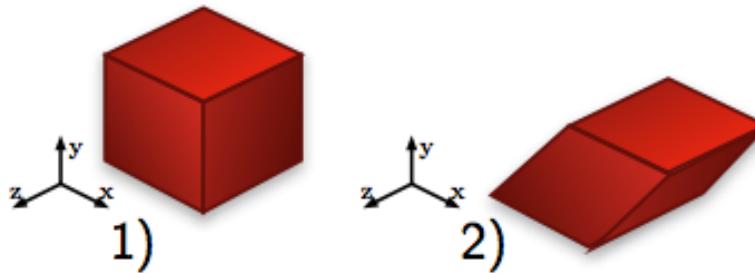




- Berechnung
- Matrixform
- Inverse

Abb. mit freundlicher Genehmigung von Henrik Tramberend, Beuth Hochschule für Technik Berlin

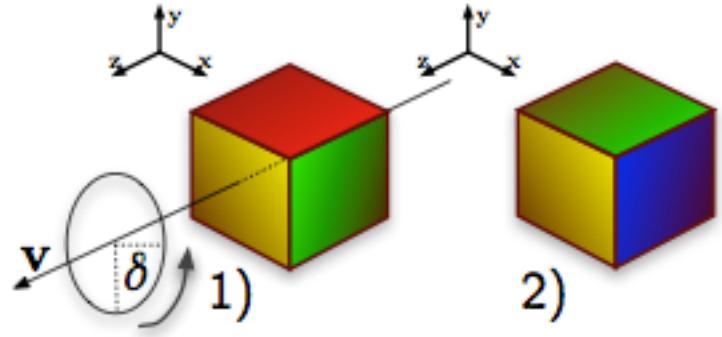




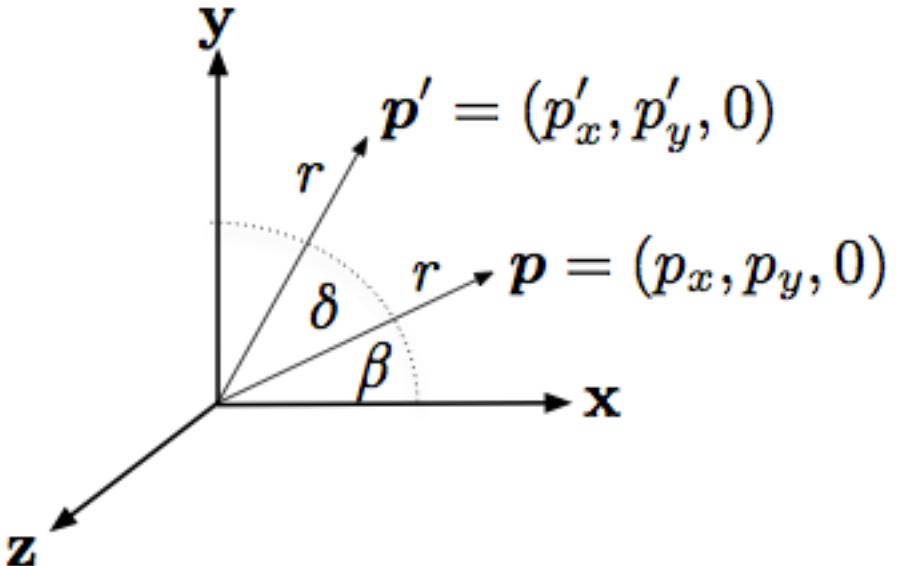
- Berechnung
- Matrixform
- Inverse

Abb. mit freundlicher Genehmigung von Henrik Tramberend, Beuth Hochschule für Technik Berlin





- Berechnung
- Matrixform
- Inverse



Hilfreiche Formeln: zusammengesetzte Winkel

$$\begin{aligned}\sin(\alpha + \beta) &= \sin(\alpha) \cos(\beta) + \cos(\alpha) \sin(\beta) \\ \cos(\alpha + \beta) &= \cos(\alpha) \cos(\beta) - \sin(\alpha) \sin(\beta)\end{aligned}$$

Abb. mit freundlicher Genehmigung von Henrik Tramberend, Beuth Hochschule für Technik Berlin



- Rotation
- Skalierung
- Spiegelung
- Scherung

- Translation



Wiederholen:
- Berechnung der Operatoren
- Darstellung in 3x3-Matrixform

Was unterscheidet die Translation
von den anderen Transformationen?

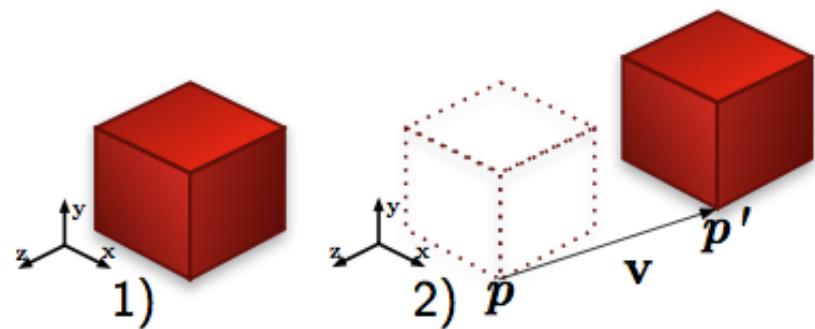


Abb. mit freundlicher Genehmigung von Henrik Tramberend, Beuth Hochschule für Technik Berlin



- Erweitere die lineare Abbildung um einen additiven Term:

$$F(x, y, z) = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Eine lineare Abbildung mit zusätzlicher Translation wird *affine Abbildung* genannt.

- Schreibweise mittels **4x4-Matrix**:

$$F(x, y, z, w) = \begin{bmatrix} a_{11} & a_{12} & a_{13} & t_x \\ a_{21} & a_{22} & a_{23} & t_y \\ a_{31} & a_{32} & a_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Rotation, Translation
Skalierung,
...

Zeige:
Für welches w liefert diese Matrix die gewünschte affine Abbildung ?



- Was ist die Bedeutung der w -Koordinate?
 - Könnte man Sie für die erwähnten Operationen nicht einfach immer ignorieren? (Also die Berechnung der 4. Komponente sparen?)
 - Kann jedoch auch sehr elegant zur Unterscheidung von Punkten und Richtungen verwendet werden. Vereinbarung:
 - **Positionsvektor (Punkt):** $w = 1$
 - **Richtungsvektor:** $w = 0$
 - Nun transformiert dieselbe Matrix *Punkte* anders als *Richtungen*:
 - Translation eines Punktes: **Punkt wird verschoben**
 - Translation einer Richtung: **Richtung bleibt unverändert**
 - Bei der perspektivischen Projektion hat w noch weitere Bedeutung
 - Bei der Projektion entstehen w -Koordinaten mit beliebigen Werten
 - Danach alle Koordinaten des resultierenden Vektors wieder durch w teilen („Homogenisierung“)

ja, kann man im Prinzip!

nachrechnen!

(später)



- Subtraktion Punkt – Punkt

$$\begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} - \begin{bmatrix} q_x \\ q_y \\ q_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x - q_x \\ p_y - q_y \\ p_z - q_z \\ 0 \end{bmatrix}$$

- Addition Richtung + Richtung

$$\begin{bmatrix} u_x \\ u_y \\ u_z \\ 0 \end{bmatrix} + \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix} = \begin{bmatrix} u_x + v_x \\ u_y + v_y \\ u_z + v_z \\ 0 \end{bmatrix}$$

- Addition Punkt + Richtung

$$\begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} + \begin{bmatrix} u_x \\ u_y \\ u_z \\ 0 \end{bmatrix} = \begin{bmatrix} p_x + u_x \\ p_y + u_y \\ p_z + u_z \\ 1 \end{bmatrix}$$

- Multiplikation Skalar • Richtung

$$s \begin{bmatrix} u_x \\ u_y \\ u_z \\ 0 \end{bmatrix} = \begin{bmatrix} su_x \\ su_y \\ su_z \\ 0 \end{bmatrix}$$



Rechnen mit Vektoren in homogenen Koordinaten (2)

- Skalarprodukt zweier Richtungsvektoren

$$\begin{bmatrix} u_x \\ u_y \\ u_z \\ 0 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix} = u_x v_x + u_y v_y + u_z v_z + 0$$

- Andere Operationen (z.B. Addition Punkt+Punkt) sind eigentlich nicht definiert!



Vektoren in der Praxis: 3 vs. 4 Komponenten

- Implementierung Vektoren / Matrizen
 - Vektoren werden oft nur mit 3 Komponenten gespeichert

```
var posInEyeSpace = mat4.multiplyVec3(mvMatrix4x4, pos);
```

gl-matrix.js: Implizite Umwandlung bei der Multiplikation ($w := 1$)

```
p = modelViewMatrix * vec4(vertexPosition, 1.0);
```

Beispiel im Vertex Shader: explizite Umwandlung von `vec3` nach `vec4`



- Matrixaddition und Multiplikation: assoziativ? kommutativ?
- Matrix-Vektor-Multiplikation: was steht wo?
- Wie sieht eine Identitätsmatrix aus?
- Welche Arten von Transformationen können mit einer 3×3 -Matrix dargestellt werden?
- Wodurch unterscheidet sich die Translation von den anderen Transformationen?
- Wie unterscheidet man bei homogenen Koordinaten Positions- von Richtungsvektoren?
- Für welche Kombinationen von Punkten und Richtungen sind Addition und Subtraktion sinnvoll / erlaubt ?

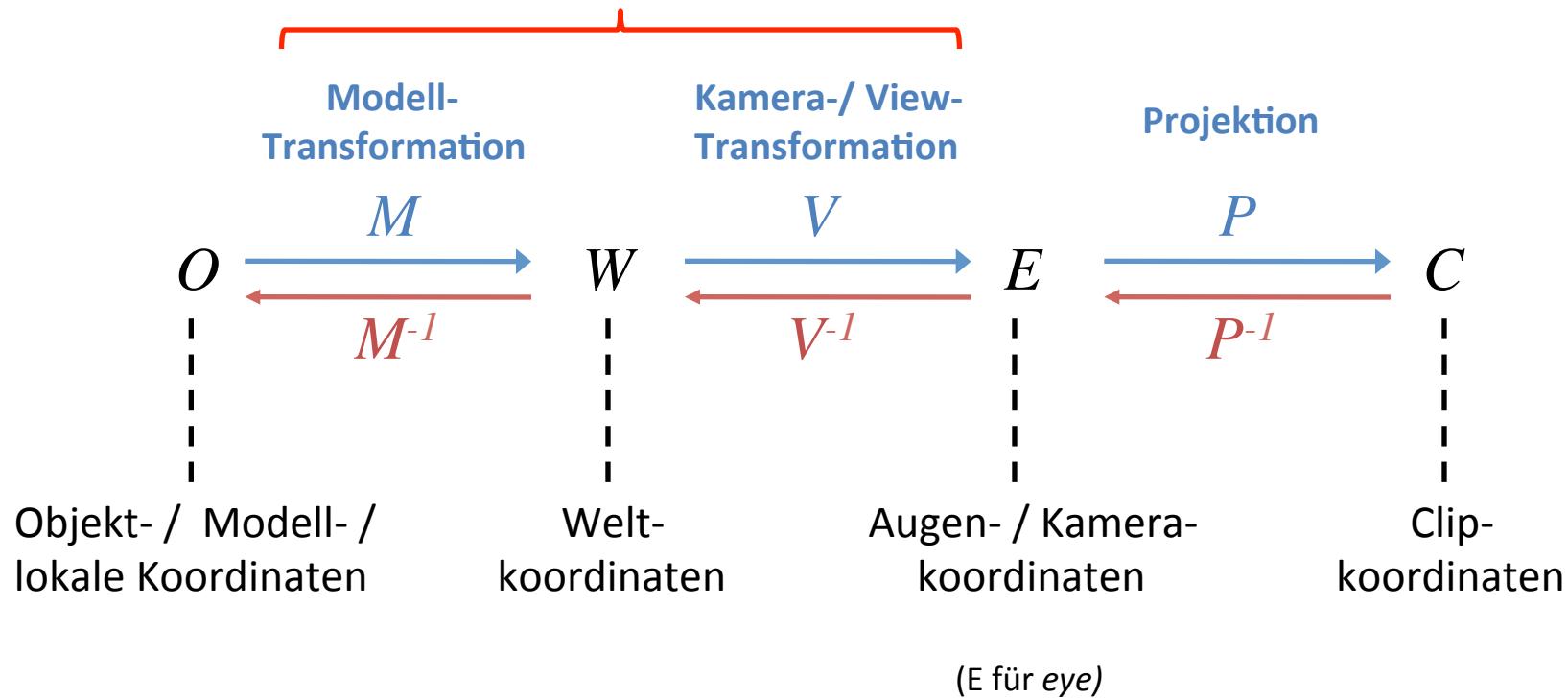


Die Transformationskette der Geometriestufe



Die Transformationskette von Modell- in Clipkoordinaten

Wird häufig zur *Model-View-Matrix*
zusammengefasst



Die Transformationskette von Modell- in Clipkoordinaten

Matrix-Kette

$$\mathbf{p}' = \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M} \cdot \mathbf{p}$$

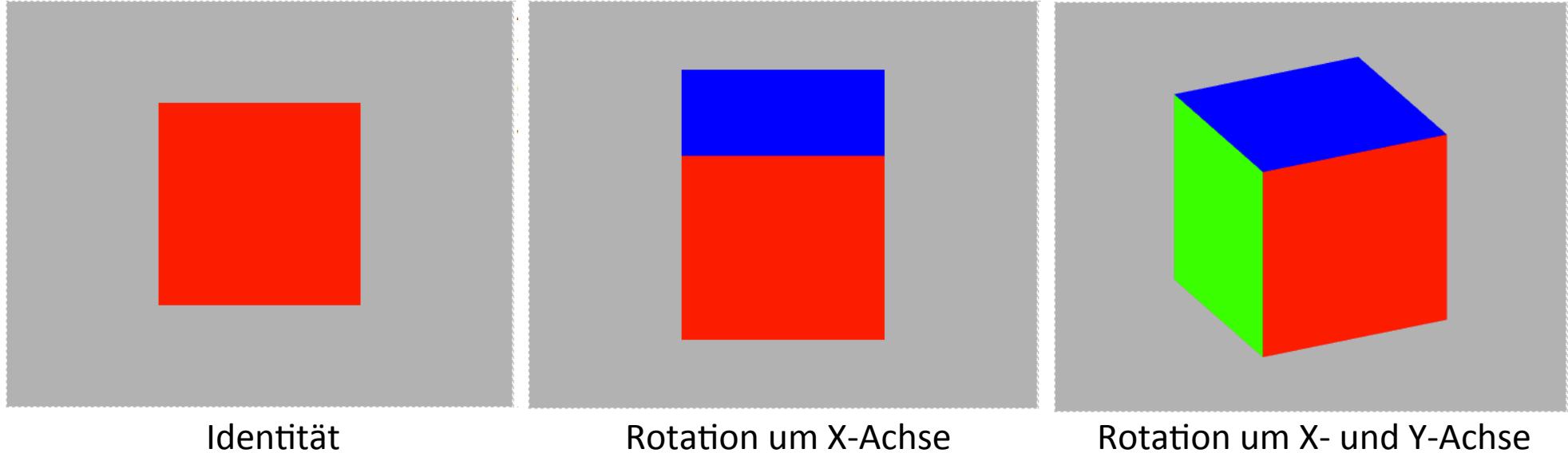
↓ ↓
Vertex in Vertex in
Clip-Koordinaten Modellkoordinaten

■ Achtung Reihenfolge!

- Der zu transformierende Vertex steht rechts
- Die Transformationen werden der Reihe nach von links multipliziert
- → von rechts nach links lesen
- Mehr hierzu bei „zusammengesetzte Transformationen“

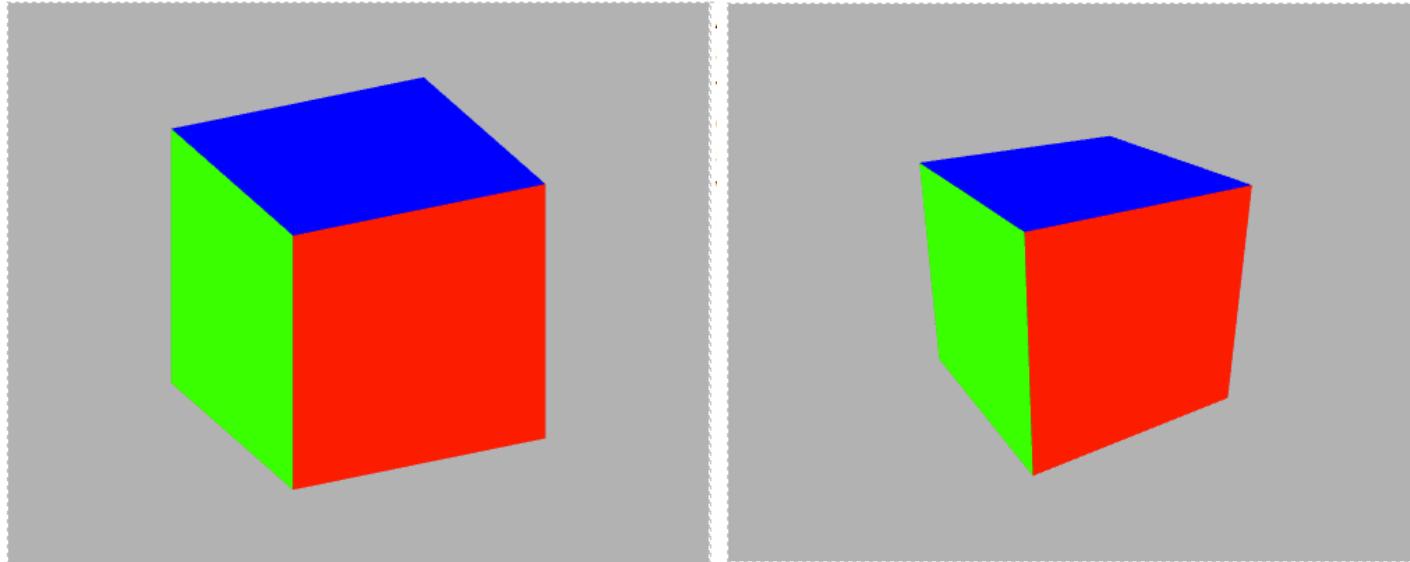


Effekt der Model-View-Transformationsmatrix



- Wer rotiert hier eigentlich, die Kamera oder die Szene?
 - Die Modelview-Matrix ist ***relative Transformation*** zwischen zwei Koordinatensystemen
 - Rotation der Szene entspricht *entgegengesetzter* Rotation der Kamera, die eine Transformation ist die ***Inverse*** der anderen





Orthographische Projektion

Perspektivische Projektion

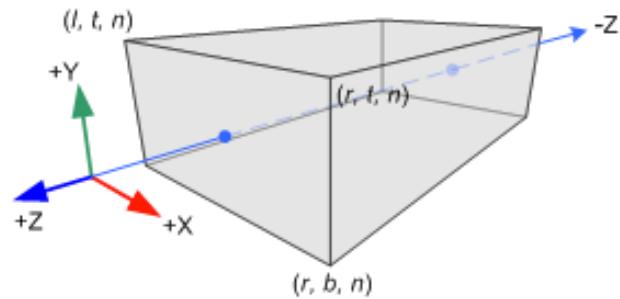
- Orthographische Projektion
 - Parallelle Linien bleiben parallel
- Perspektivische Projektion
 - „Schräge“ Projektion mit einem wählbaren Öffnungswinkel
 - Stürzende Linien, ferne Objekte erscheinen kleiner



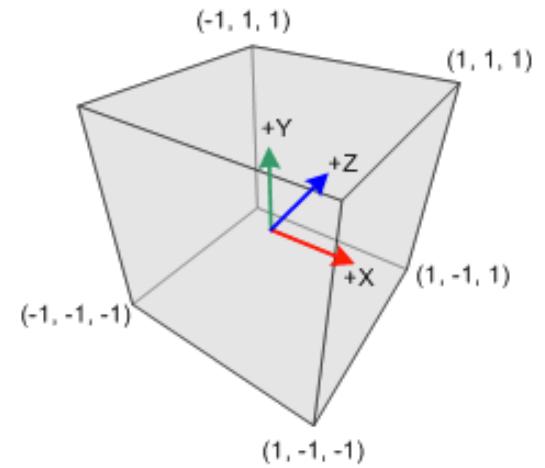
Orthographische Projektion

Illustrationen von Song Ho Ahn, www.songho.ca

Kamera-Koordinaten



Clip-Koordinaten



$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Projektionsmatrix

In gl-matrix.js:

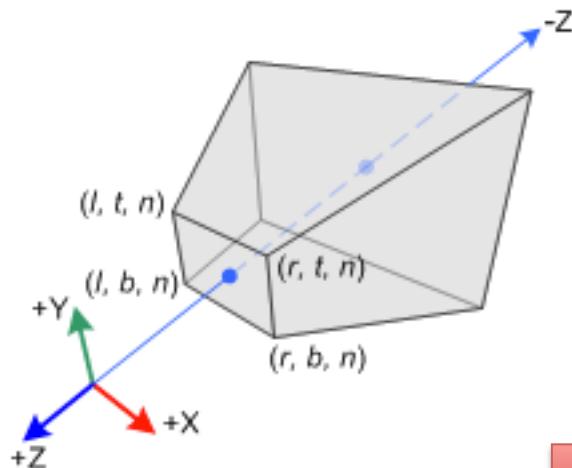
```
var pOrtho= mat4.ortho(left, right, top, bottom, near, far);
```



Perspektivische Projektion

Illustrationen von Song Ho Ahn, www.songho.ca

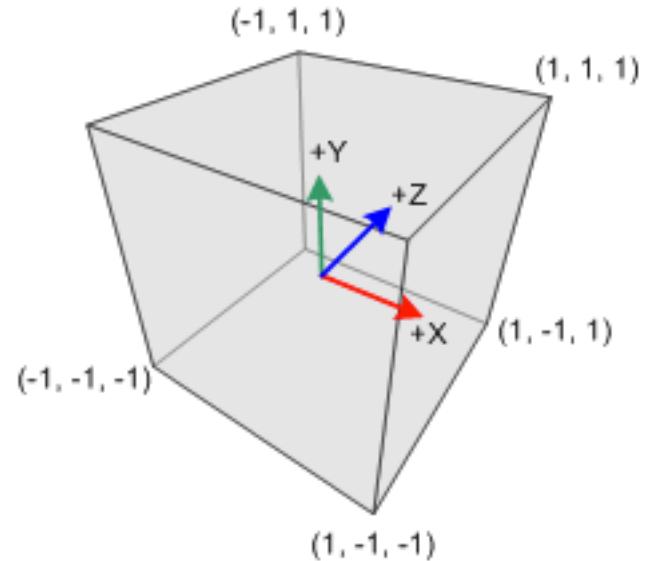
Kamera-Koordinaten



$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Projektionsmatrix

Clip-Koordinaten



In gl-matrix.js:

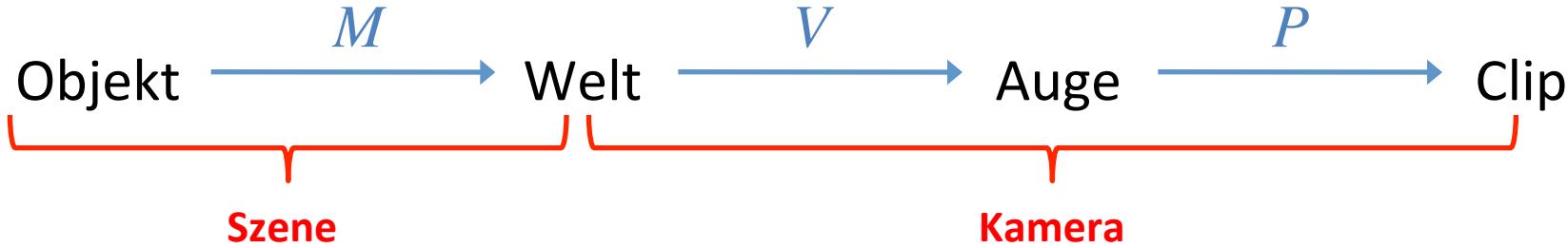
```
var pProj = mat4.perspective(fovy, aspect, near, far);
```



Praktische Anwendung der Transformationskette



Wer ist für welche Transformation verantwortlich?



- Wo befindet sich welches Objekt in der Welt?
- Wie ist die Welt insgesamt transformiert?
- Wo befindet sich die Kamera relativ zur Welt?
- Welche Art Projektion („Objektiv“) wird verwendet?

Je nach Animations- / Interaktionsmodell wird die Kamera oder die Szene manipuliert, daher oftmals redundante Transformationen.



Zusammenarbeit von Kamera und Szene



- Bei der Interaktion können Kamera und Szene getrennt manipuliert werden

```
if(<key 'w' pressed>)
    camera.transform.translate( [0,0, delta_z] );
```

(Pseudocode)

- Unmittelbar vor dem Rendering des Objekts wird die vollständige Model-View-Matrix zusammengesetzt:

```
modelView = invert(camera.transform) * scene.transform;
```

(Pseudocode)



$$\mathbf{p}' = \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M} \cdot \mathbf{p}$$

Die Transformation der Kamera innerhalb der Welt
ist die **Inverse** der Viewing-Transformation



Matrizen erzeugen und manipulieren mit gl-matrix.js

- Erzeuge neue 4x4 Identitätsmatrix:

```
var matrix = mat4.identity();
```

- Kopiere Matrix in ein neues Matrix-Objekt

```
var copy = mat4.create(matrix);
```

- Füge zu `matrix` eine Translation um **(-3,2,1)** hinzu:

```
mat4.translate(matrix, [-3,2,1] );
```

- Füge zu `matrix` eine Rotation um **25°** um die X-Achse hinzu:

```
mat4.rotate(matrix, 25 * Math.PI/180, [1,0,0] );
```

- Invertiere Matrix (verändert Matrix)

```
mat4.inverse(matrix);
```



Matrizen erzeugen und manipulieren mit gl-matrix.js

- Multipliziere zwei Matrizen (verändert erste Matrix):

```
mat4.multiply(matrix, matrix2);
```

- Erzeuge neue orthographische Projektionsmatrix:

```
var pOrtho = mat4.ortho(left, right, top, bottom, near, far);
```

Darzustellender Ausschnitt in X-, Y- und Z-Richtung

- Erzeuge neue perspektivische Projektionsmatrix:

```
var pProj = mat4.perspective(fovy, aspect, near, far);
```



gl-matrix.js enthält weitere Typen wie `vec3` und `mat3`.



Werte von Matrizen im Shader / Programm setzen

In der Applikation (vgl. main.js):

```
var setTransforms = function(program, modelView, projection) {  
  
    program.use();  
  
    program.setUniform("projectionMatrix", "mat4", projection);  
    program.setUniform("modelViewMatrix", "mat4", modelView);  
  
};
```

Zugehöriger Vertex Shader:

```
attribute vec3 vertexPosition;  
uniform mat4 modelViewMatrix;  
uniform mat4 projectionMatrix;  
  
void main() {  
    gl_Position = projectionMatrix * modelViewMatrix *  
                 vec4(vertexPosition, 1.0);  
}
```



Anhang: Grafikpipeline und -Hardware



Von Grafik-Computern zur programmierbaren GPU

- 1981 – 200x: Silicon Graphics (SGI) Grafik-Workstations
 - 1987 SGI "4D" Reihe → verbreitetster Rechner für Animation
 - 1992 MIPS R4000, Einsatz einer der ersten 64-bit-Prozessoren
- 1996: 3DFX Voodoo, erste PC Grafikkarte
- 1999: Transform & Lighting (T&L) Unit
 - NVIDIA GeForce 256 "the world's first GPU"; Pipeline auf einem einzigen Chip, erste T&L-Einheit für den Consumer-Markt
- 2001: Erste Vertex Programs fester Länge
 - NVIDIA GeForce 3
- 2003: Vertex und Fragment Shader variabler Länge
 - NVIDIA GeForce FX, ATI Radeon 9500/9800, 3DLabs WildCat VP
- 2006: Hardware Tessellation
 - 2001: Erste Tessellation-Hardware in ATI Radeon 8500, fand aber keine große Verwendung / Verbreitung
 - 2005/2006: Xbox 360, ATI Radeon HD2000
 - 2009 NVIDIA GeForce 400 Serie
 - Verbreitung durch DirectX 11 und OpenGL 4.1

1999
15 Mio Dreiecke/Sek.
480 Mio Pixel/Sek.

2010
100.000 Mio Vertices/Sek.
40.000 Mio Pixel/Sek.



SGI Onyx Hochleistungsrechner unterstützte Multiprozessoren und mehrere InfiniteReality Grafik-Pipes



SGI RM11 Raster Manager mit 1 GB onboard Speicher



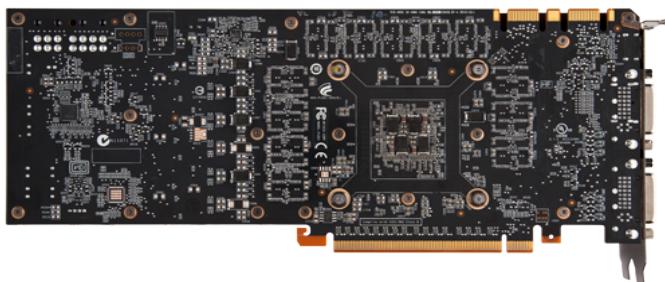
NVidia GeForce3 Ti 200 GPU



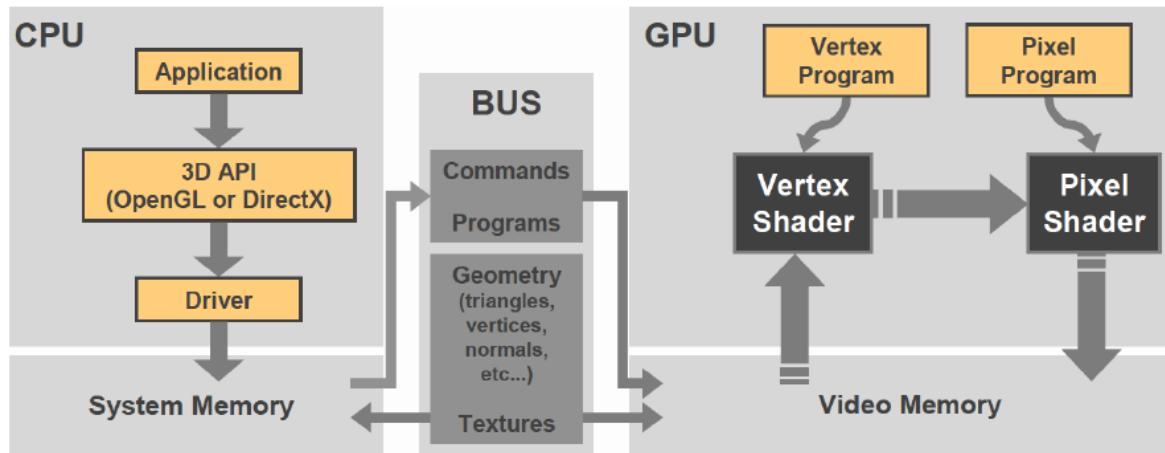
Beispiel: (vor kurzer Zeit noch) aktuelle Grafikkarte

Quellen: nvidia.de, gpureview.com

NVIDIA GeForce GTX 580



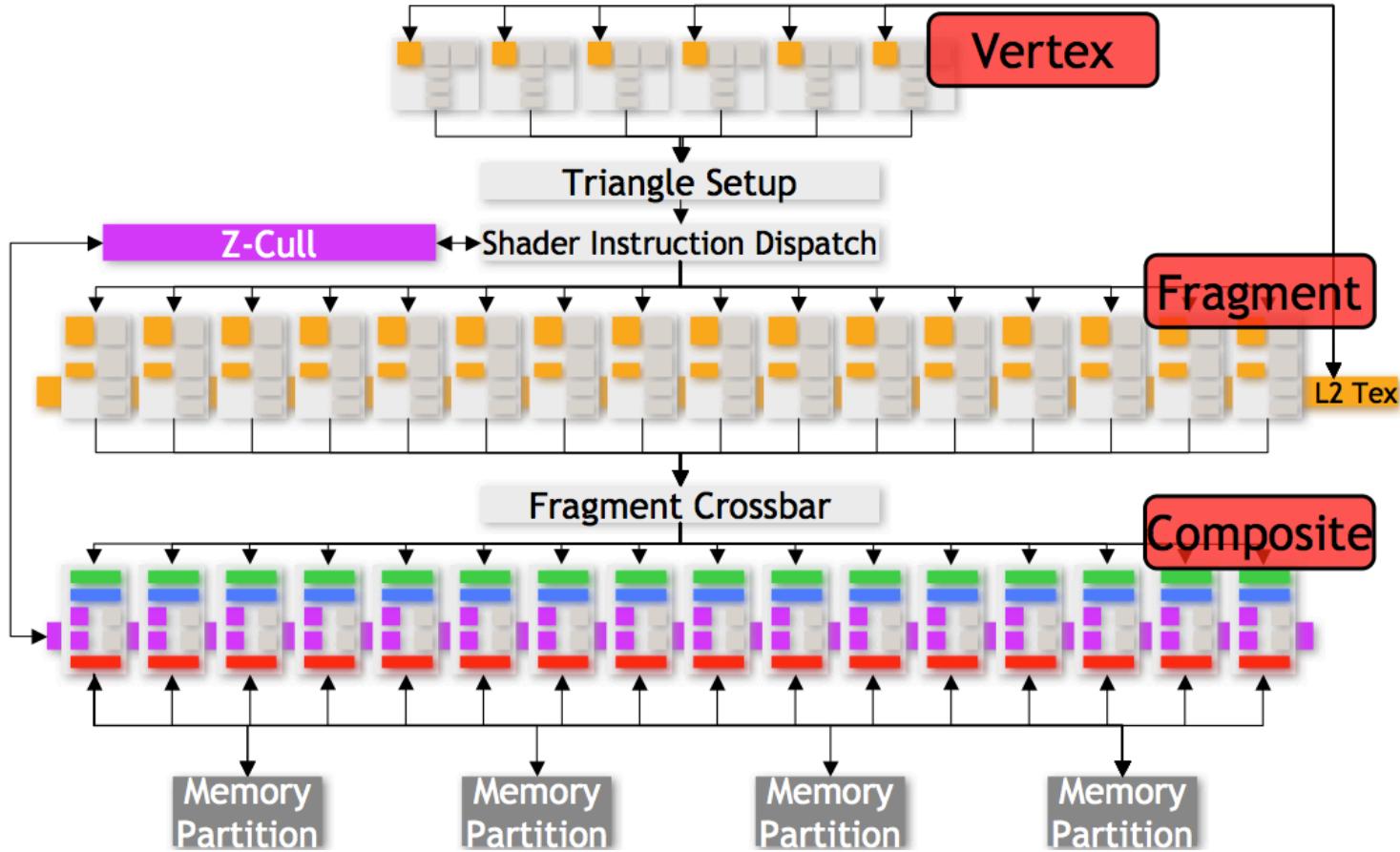
Model	GeForce GTX 580
Manufacturer	NVIDIA
GPU	GTX 500
Release Date	2010-11-09
Core Clock	772 MHz
Memory Clock	2004 MHz (4008 DDR)
Memory Bandwidth	192.384 GB/sec
Shader Operations	395264 MOperations/sec
Pixel Fill Rate	37056 MPixels/sec
Texture Fill Rate	49408 MTexels/sec
Vertex Operations	98816 MVertices/sec
Framebuffer	1536 MB
DirectX Compliance	11.0
OpenGL Compliance:	3.2
PS/VS Version	5.0/5.0
Fragment Pipelines	512
Vertex Pipelines	512
Texture Units	64



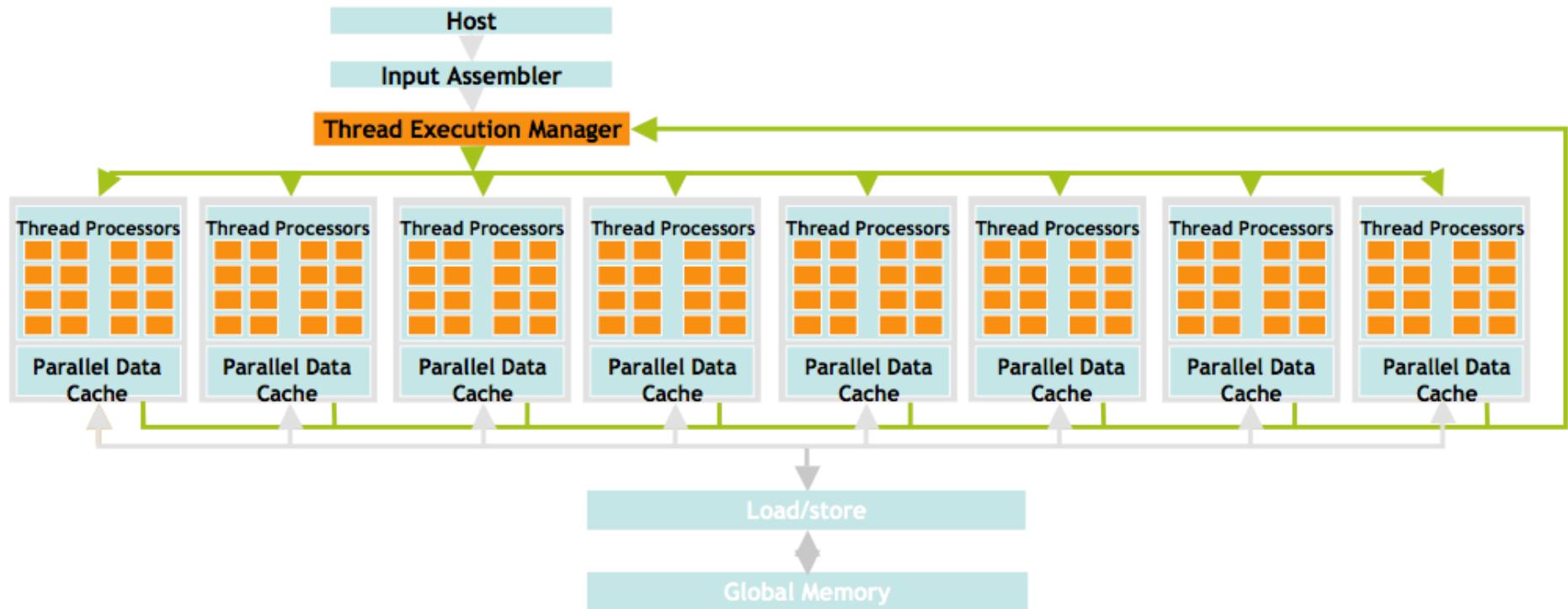
- Applikation „installiert“ Shader-Programme direkt auf der GPU
- Applikation / CPU transferiert Nutzdaten zur GPU
 - Geometrische Primitive, Vertex-Daten, globale Daten
- Grafikkarte fungiert als Streaming-Maschine auf den Nutzdaten
 - Shader-Programme werden massiv parallel für alle Primitive ausgeführt, z.B.
 - Alle Vertices (Vertex Shader)
 - Alle Patches / Dreiecke (Tessellation & Geometry Shader)
 - Alle Pixel / Fragmente (Pixel/Fragment Shader)
 - Hochperformanter Zugriff auf On-Board Grafikspeicher



Quelle: *Nick Triantos, NVIDIA*

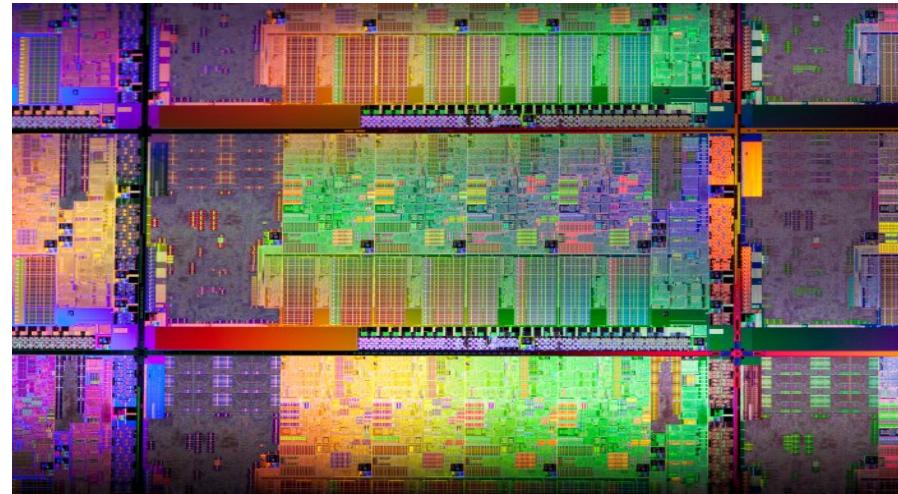
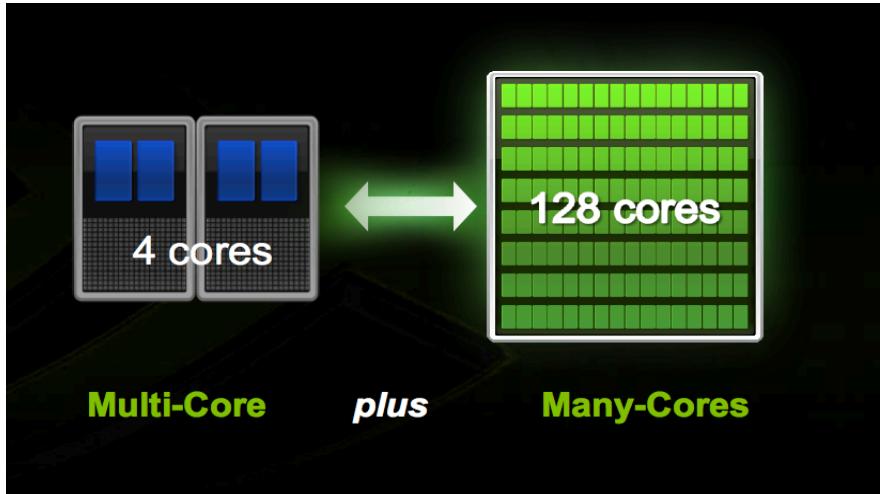


Quelle: Ian Buck, NVIDIA



Die Hardware ist inzwischen eine Maschine zur parallelen Ausführung massiv vieler Shaderkerne.
Die Hardware ist für alle Shadertypen gleich (Unified Shader Model); die Steuerlogik sorgt dafür,
Daß jeder Shadertyp die richtigen Eingabedaten erhält.





- GPU entwickelt sich
 - vom Grafik-Spezialprozessor
 - zur Allzweck - SIMD Computing Engine
 - *GPGPU: general purpose GPU programming*
- Zusammenspiel von CPU und GPU
 - Wer übernimmt welche Aufgaben?
 - Wie minimiert man den Datenverkehr?
- CPU + GPU zusammen auf einem Chip
 - Schnellerer Datenaustausch
 - Mehr *Leistung pro Watt*
 - Weniger Platzbedarf (mobile Endgeräte)
 - Preiswertere Produktion

Links: nvidia.com. Rechts: Intel Sandy Bridge Wafer, Intel/heise.de



Anhang: Einfache Vektoralgebra



Typ	Darstellung	Beispiel
Winkel	klein, griechisch	α, β, γ
Skalarwert	klein, lateinisch, kursiv	a, b, c
Vektor (Richtung oder Punkt)	klein, lateinisch, fett	a, b, c, sv, n
Matrix	groß, lateinisch, fett	M, T
Skalarprodukt (dot product)	Punkt	$\mathbf{a} \bullet \mathbf{b}$
Kreuzprodukt (cross product)	Kreuz	$\mathbf{a} \times \mathbf{b}$
Matrix-Determinante	senkrechte Linien	$ M $
Vektornorm / Vektorlänge	senkrechte Linien	$ n $

Empfohlene Literatur:

- John Vince, *Mathematics for Computer Graphics*, Springer Verlag.
- Peter Shirley, *Fundamentals of Computer Graphics*, AK Peters.



- Vektor im kartesischen dreidimensionalen Raum

- Ein Vektor besitzt drei Komponenten, die i.a. mit x , y und z gekennzeichnet werden.

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = (v_x \ v_y \ v_z)^T$$

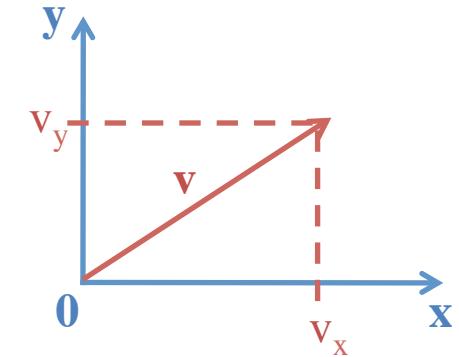
- Später: Homogene Koordinaten mit zusätzlicher w -Komponente.

- Formale Addition und Subtraktion

- Wird komponentenweise durchgeführt

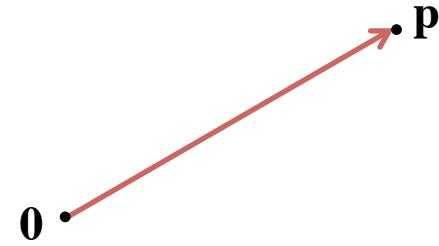
$$\mathbf{p} \pm \mathbf{q} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \pm \begin{pmatrix} q_x \\ q_y \\ q_z \end{pmatrix} = \begin{pmatrix} p_x \pm q_x \\ p_y \pm q_y \\ p_z \pm q_z \end{pmatrix}$$

- Was bedeuten die Vektoren und Operationen geometrisch?



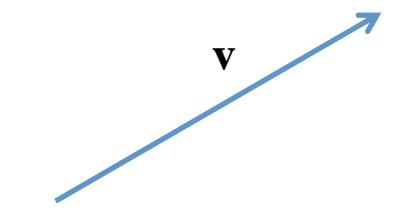
- Interpretation 1: Der Vektor als Positionsvektor \mathbf{p}

- Repräsentiert Koordinaten des Punktes \mathbf{p}
 - Relativ zum Ursprung $\mathbf{0}$ des Koord.-Systems
 - Wird auch „gebundener Vektor“
(bound vector) genannt



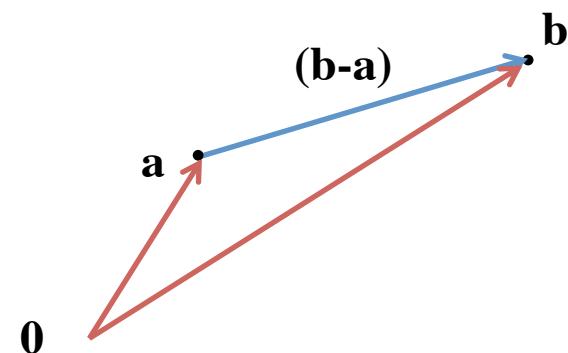
- Interpretation 2: Richtungsvektor \mathbf{v}

- Repräsentiert eine Richtung
 - Kann frei verschoben werden
 - Wird auch „freier Vektor“
(free vector) genannt



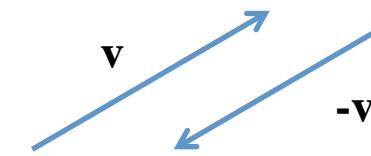
- Richtung = Differenz zweier Positionen

- Richtungsvektor $(\mathbf{b}-\mathbf{a})$ zeigt von \mathbf{a} nach \mathbf{b}

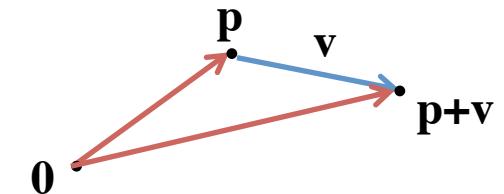


Rechnen mit Positionen und Richtungen

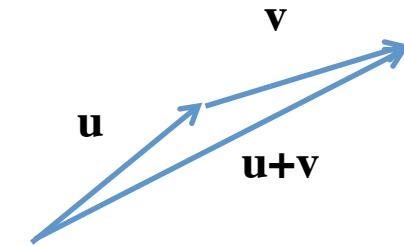
- Negieren einer Richtung $u = -v$
 - Der Pfeilspitze des Vektors wird umgekehrt.



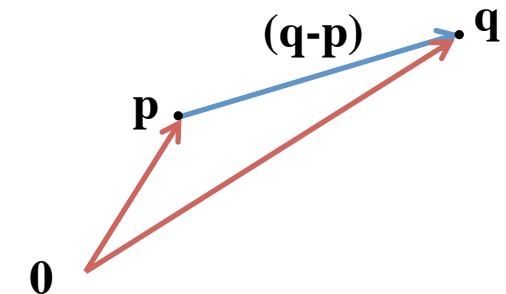
- Addition Punkt p + Richtung v
 - Gehe von einem festen Punkt aus in eine bestimmte Richtung.
 - Ergibt einen neuen Positionsvektor.



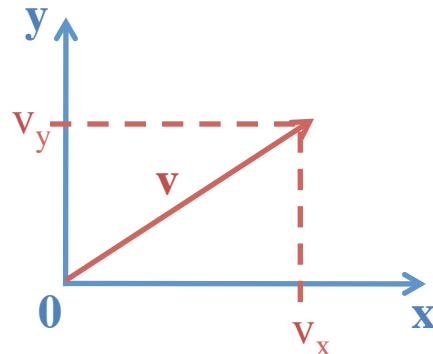
- Addition Richtung u + Richtung v
 - Aneinanderhängen der zwei Vektorpfeile
 - ergibt einen neuen Richtungsvektor.



- Subtraktion Punkt q – Punkt p
 - ergibt einen Richtungsvektor von p nach q .



Länge / Norm $|\mathbf{v}|$ eines Vektors \mathbf{v}



- Siehe Abbildung: im kartesischen Koordinatensystem ist die Länge des Vektors die der Hypotenuse.
- In 2D: $|\mathbf{v}| = \sqrt{v_x^2 + v_y^2}$
- In 3D: $|\mathbf{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$

- Normalisierung:
 - Bringe die Länge des Vektors auf 1 (Einheitsvektor).
 - Wie: teile alle Komponenten durch die Länge des Vektors:

$$\mathbf{v}' = \frac{1}{|\mathbf{v}|} \mathbf{v}$$



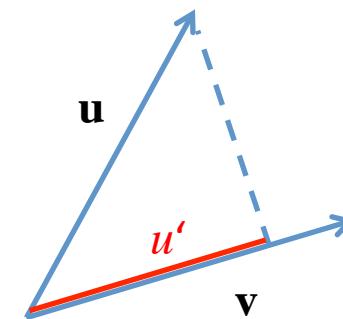
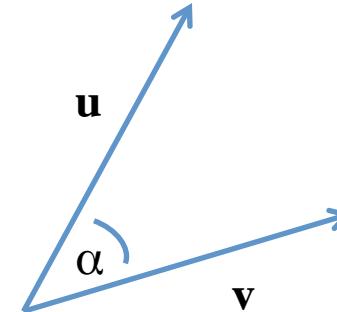
■ Skalarprodukt (SP)

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| \cdot |\mathbf{v}| \cos \alpha$$

- Länge der beiden Vektoren • Kosinus des eingeschlossenen Winkels
- Vektoren normalisiert \rightarrow SP ergibt Kosinus
- Vektoren normalisiert und parallel \rightarrow SP = 1
- Vektoren orthogonal \rightarrow SP = 0
- Winkel $> 90^\circ \rightarrow$ SP < 0
- SP eines Vektors mit sich selbst: $\mathbf{x} \cdot \mathbf{x} = |\mathbf{x}|^2$

■ Projektion auf Vektor

- Ist \mathbf{v} ein Einheitsvektor, so ist das SP die Länge der Projektion von \mathbf{u} auf \mathbf{v} .



Wie berechne ich das Skalarprodukt?

- Definition in \mathbb{R}^n

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=0}^{n-1} u_i v_i$$

- Definition in \mathbb{R}^3

$$\mathbf{u} \cdot \mathbf{v} = \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = u_x v_x + u_y v_y + u_z v_z$$



- Kreuzprodukt $\mathbf{w} = \mathbf{u} \times \mathbf{v}$ zweier Vektoren

$$|\mathbf{w}| = |\mathbf{u}| \cdot |\mathbf{v}| \sin \alpha$$

- Eigenschaften

- $\mathbf{w} \perp \mathbf{u}$
- $\mathbf{w} \perp \mathbf{v}$
- $\mathbf{u}, \mathbf{v}, \mathbf{w}$ spannen ein rechtshändiges Koordinatensystem auf.

- Berechnung:

$$\mathbf{w} = \mathbf{u} \times \mathbf{v} = \begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix}$$

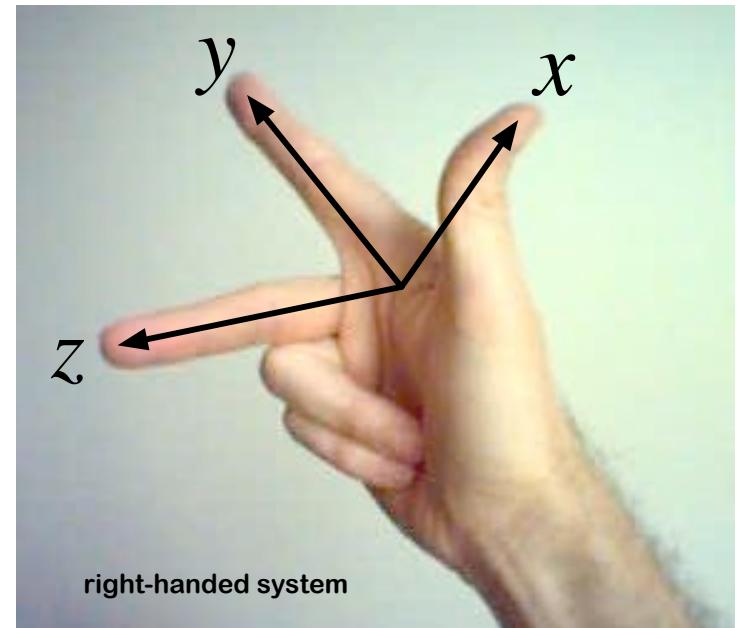


Abb. mit freundlicher Genehmigung von Henrik Tramberend, Beuth Hochschule für Technik Berlin



■ Normalenvektor

- \mathbf{w} ist orthogonal zu \mathbf{u} und \mathbf{v}
- Also ist \mathbf{w} eine Normale zu der von \mathbf{u} und \mathbf{v} aufgespannten Ebene
- Bilden $|\mathbf{u}|$ und $|\mathbf{v}|$ ein Orthonormal-
system (orthogonal und $|\bullet|=1$)
bilden, ist auch $|\mathbf{w}| = 1$

■ Fläche

- Wegen $|\mathbf{w}| = |\mathbf{u}| \cdot |\mathbf{v}| \sin \alpha$
ist $|\mathbf{w}|$ die Fläche des von
 \mathbf{u} und \mathbf{v} aufgespannten
Parallelogramms

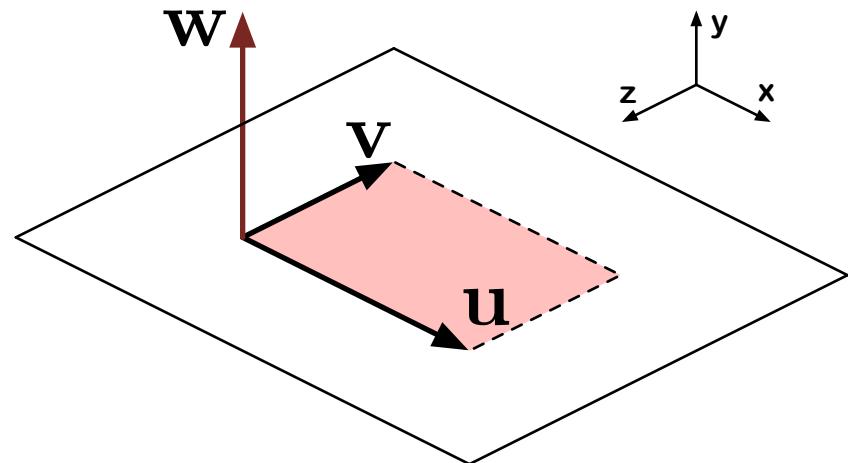
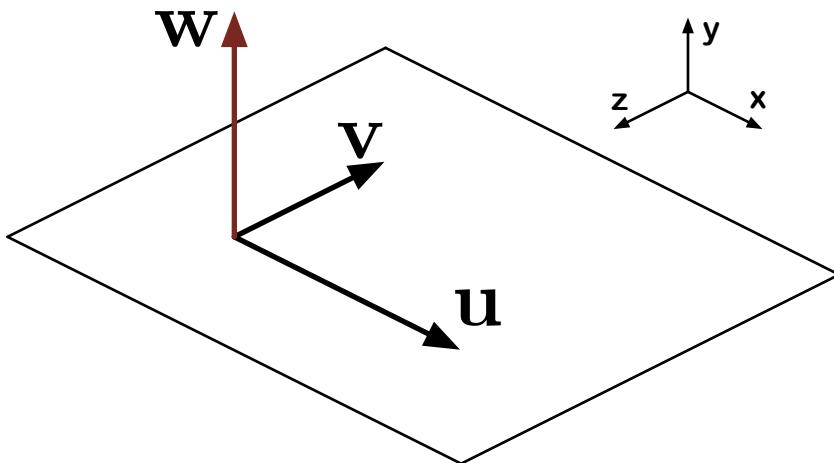


Abb. mit freundlicher Genehmigung von Henrik Tramberend, Beuth Hochschule für Technik Berlin

