

# Computergrafik

Jan Fässler

5. Semester (HS 2013)

# Inhaltsverzeichnis

<b>1</b>	<b>Mathematische Grundlagen</b>	<b>1</b>
1.1	Euklidischer Raum . . . . .	1
1.2	Vektoren . . . . .	1
1.2.1	Addition . . . . .	1
1.2.2	skalare Multiplikation . . . . .	1
1.2.3	Betrag . . . . .	2
1.2.4	Einheitsvektor . . . . .	2
1.2.5	Skalarprodukt . . . . .	2
1.2.6	Vektorprodukt . . . . .	2
1.3	Matrizen . . . . .	2
1.3.1	Einheitsmatrix . . . . .	2
1.3.2	Transponierte Matrix . . . . .	3
1.3.3	Multiplikation . . . . .	3
1.4	Homogene Koordinaten . . . . .	3
1.5	2D-Transformationen . . . . .	3
1.5.1	Translation . . . . .	3
1.5.2	Skalierung . . . . .	3
1.5.3	Scherung . . . . .	4
1.5.4	Rotation . . . . .	4
1.6	3D-Transformationen . . . . .	4
1.6.1	Translation . . . . .	4
1.6.2	Skalierung . . . . .	4
1.6.3	Scherung . . . . .	4
1.6.4	Rotation . . . . .	4
1.6.5	Rotation um eine beliebige Achse . . . . .	5
<b>2</b>	<b>Rasteralgorithmen</b>	<b>6</b>
2.1	Inkrementeller Algorithmus . . . . .	6
2.2	Bresenham Algorithmus . . . . .	6
<b>3</b>	<b>Perspektive: 3D auf 2D</b>	<b>7</b>
3.1	Kategorien der Projektionen . . . . .	7
3.2	Parallelprojektion . . . . .	7
3.3	Zentralprojektion . . . . .	7
3.4	Projektionstransformation . . . . .	7
3.4.1	Schritt 1: Scherung H in eine orthogonale Sichtpyramide . . . . .	8
3.4.2	Skalierung S in ein kanonisches Sichtvolumen . . . . .	8
3.4.3	Projektive Transformation N in einen Sichtwürfel . . . . .	8
3.4.4	Perspektivische Division . . . . .	8
3.5	Viewport-Transformation . . . . .	9
<b>4</b>	<b>OpenGL</b>	<b>10</b>
4.1	Was ist OpenGL? . . . . .	10
4.2	Rendering Pipeline . . . . .	10
4.3	Shaders . . . . .	10
<b>5</b>	<b>3D-Objekte</b>	<b>12</b>
5.1	OpenGL-Primitive . . . . .	12
5.2	DrawElements . . . . .	12
5.3	Vertex Buffer Object . . . . .	12
<b>6</b>	<b>Shader</b>	<b>14</b>
6.1	Shader Programme kompilieren, linken & aktivieren . . . . .	14
6.2	einfache Datentypen . . . . .	14
6.3	Typen Qualifizierer . . . . .	14
6.3.1	const . . . . .	14

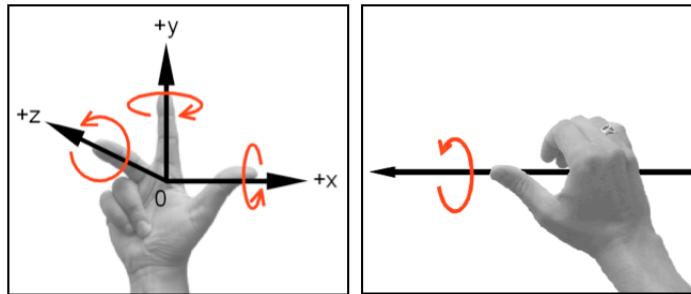
6.3.2	attribute	14
6.3.3	uniform	15
6.3.4	varying	15
<b>7</b>	<b>Beleuchtung</b>	<b>16</b>
7.1	Lichtquellen	16
7.2	Reflexion	16
7.2.1	Ambiente Reflexion	16
7.2.2	Diffuse Reflexion	16
7.2.3	Spiegelnde Reflexion	16
7.2.4	Lichtabnahme (Attenuierung)	17
7.3	Beleuchtungsmodell nach Phong	17
7.4	Erweiterungen im alten OpenGL Beleuchtungsmodell	17
7.5	Blinn's Halbvektor	17
7.6	Schattierung	17
7.6.1	Berechnung pro Dreieck: Flat Shading	17
7.6.2	Berechnung pro Vertex: Gouraud Shading	18
7.6.3	Berechnung pro Pixel: Phong Shading	18
<b>8</b>	<b>Texture Mapping</b>	<b>19</b>
8.1	verschiedene Texturen	19
8.2	Textur Abbildung	19
8.3	Texturfilterung	19
8.3.1	Point Sampling	19
8.3.2	Bilineare Filterung	19
8.3.3	MIP-Mapping	20
8.3.4	Trilineare Filterung	20
8.3.5	Anisotropische Filterung	20
8.4	Bump Mapping	21
8.4.1	Heightfield Bump Mapping	21
8.4.2	Normalmap Bump Mapping	21
8.4.3	Parallax Mapping	21
8.5	Texture Mapping in OpenGL	22
<b>9</b>	<b>Frustum Culling</b>	<b>23</b>
9.1	Ziel	23
9.2	Hüllvolumen	23
<b>10</b>	<b>Alpha Blending</b>	<b>24</b>
10.1	Einleitung	24
10.2	Blendfunktion	24
10.3	Additive Blending	24
10.4	Zeichnungsreihenfolge	24
<b>11</b>	<b>Deferred Shading</b>	<b>25</b>
11.1	Prinzip	25
11.2	Vorteile	25
11.3	Nachteile	25
11.4	OpenGL	25
11.4.1	Framebuffer objects	25
11.4.2	FBO relevant API	25
<b>12</b>	<b>Shadow Rendering</b>	<b>26</b>
12.1	Einleitung	26
12.2	Shadow Volumes	26
12.3	Shadow Maps	26
12.4	Variance Shadow Maps	26
12.5	Cascade Shadow Maps	26

12.6 Vergleich . . . . .	27
--------------------------	----

# 1 Mathematische Grundlagen

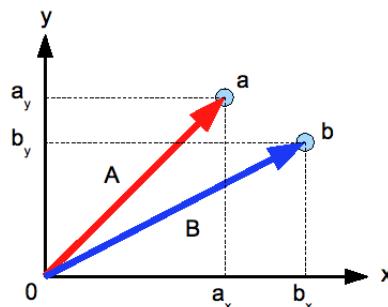
## 1.1 Euklidischer Raum

- OpenGL benutzt das rechtshändige Koordinatensystem. Direct3D oder der Raytracer POVRay benutzen das links-händige System.
- Nehmen Sie die rechte Hand, streckt den Daumen nach rechts, den Zeigefinger nach oben und den Mittelfinger nach vorne.
- Die Finger zeigen dabei in die positiven Richtungen der x-, y- und z-Achse.
- Die Drehwinkel werden im Gegenuhrzeigersinn gemessen.



## 1.2 Vektoren

- Vektoren haben eine Richtung und eine Länge aber KEIN Ort.
- Punkte im Raum können durch Ortsvektoren beschrieben werden.
- Sie werden als n-Tupel, als geordnete Liste von reellen Zahlen beschrieben
- Punkte sind Orte, Vektoren sind Richtungen



### 1.2.1 Addition

$$C = A + B = \begin{bmatrix} a_0 + b_0 \\ a_1 + b_1 \\ \vdots \\ a_{n-1} + b_{n-1} \end{bmatrix} \in \mathbb{R}^n \quad (1)$$

### 1.2.2 skalare Multiplikation

$$s * A = \begin{bmatrix} s * a_0 \\ s * a_1 \\ \vdots \\ s * a_{n-1} \end{bmatrix} \in \mathbb{R}^n \quad (2)$$

### 1.2.3 Betrag

$$\text{Länge} = |A| = \sqrt{A * A} = \sqrt{a_0^2 + a_1^2 + \dots + a_{n-1}^2} \quad (3)$$

### 1.2.4 Einheitsvektor

Ein Vektor mit der Länge 1 wird Einheitsvektor bezeichnet

$$E = \frac{1}{|A|} * A = \frac{1}{\sqrt{a_0^2 + a_1^2 + \dots + a_{n-1}^2}} * \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \quad (4)$$

### 1.2.5 Skalarprodukt

$$A \bullet B = |A| * |B| + \cos(\alpha) \quad (5)$$

$$\cos(\alpha) = \frac{A \bullet B}{|A| * |B|} \quad (6)$$

$$\cos(\alpha) = \frac{p}{|A|} \quad (7)$$

$$\text{Projektion} = p = \cos(\alpha) * |A| = \frac{A \bullet B}{|B|} \quad (8)$$

### 1.2.6 Vektorprodukt

Der resultierende Vektor N steht senkrecht auf den Vektoren A und B

$$N = A \times B = \begin{bmatrix} a_1 * b_2 - a_2 * b_1 \\ a_2 * b_0 - a_0 * b_2 \\ a_0 * b_1 - a_1 * b_0 \end{bmatrix} \in \mathbb{R}^n \quad (9)$$

## 1.3 Matrizen

Eine Matrix ist ein rechteckiges Schema von reellen Zahlen mit  $n_x$  Zeilen und  $n_y$  Spalten

$$A_{n_x \times n_y} = \begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n_y-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n_y-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_x-1,0} & a_{n_x-1,1} & \dots & a_{n_x-1,n_y-1} \end{bmatrix} \quad (10)$$

### 1.3.1 Einheitsmatrix

Wird ein Vektor damit transformiert resultiert daraus derselbe Vektor

$$I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \quad (11)$$

### 1.3.2 Transponierte Matrix

Bei der transponierten Matrix sind die Spalten und Zeilen vertauscht

$$A_{nx \times ny}^T = \begin{bmatrix} a_{0,0} & a_{1,0} & \dots & a_{nx-1,0} \\ a_{0,1} & a_{1,1} & \dots & a_{nx-1,1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{0,ny-1} & a_{1,ny-1} & \dots & a_{nx-1,ny-1} \end{bmatrix} \quad (12)$$

### 1.3.3 Multiplikation

$$C_{m \times p} = A_{m \times n} * B_{n \times p} \quad (13)$$

$$= \begin{bmatrix} a_{0,0} & \dots & a_{a,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \dots & a_{m-1,n-1} \end{bmatrix} * \begin{bmatrix} a_{0,0} & \dots & a_{a,p-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \dots & a_{n-1,p-1} \end{bmatrix} \quad (14)$$

$$= \begin{bmatrix} \sum_{i=0}^{n-1} a_{0,i} * b_{i,0} & \dots & \sum_{i=0}^{n-1} a_{0,i} * b_{i,p-1} \\ \vdots & \ddots & \vdots \\ \sum_{i=0}^{m-1} a_{0,i} * b_{i,0} & \dots & \sum_{i=0}^{m-1} a_{0,i} * b_{i,p-1} \end{bmatrix} \quad (15)$$

## 1.4 Homogene Koordinaten

Man kann Transformationen einfacher handhaben und miteinander kombinieren, wenn sie sich alle einheitlich, durch eine Multiplikation beschreiben lassen. Dies lässt sich durch sogenannte homogene Koordinaten erreichen. In der homogenen Darstellung werden die Koordinaten des Punktes P um eine weitere Komponente w ergänzt.

$$P = \begin{bmatrix} x \\ y \\ w \end{bmatrix} \quad (16)$$

Mit homogenen Koordinaten gibt es unendlich viele Repräsentationen eines Punktes. Die Normalisierung wird mit der Division durch w erreicht. Um diesen Schritt zu umgehen, benutzt man deshalb die Standarddarstellung mit  $w = 1.0$ . Ist die Komponente  $w = 0$ , so liegt der repräsentierte Punkt im Unendlichen (Division durch 0!). Seine Koordinaten legen dann lediglich die Richtung fest, in der der Punkt im Unendlichen liegt.

## 1.5 2D-Transformationen

### 1.5.1 Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (17)$$

### 1.5.2 Skalierung

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (18)$$

### 1.5.3 Scherung

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & h_{xy} & 0 \\ h_{xy} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (19)$$

### 1.5.4 Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) & 0 \\ \sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (20)$$

## 1.6 3D-Transformationen

### 1.6.1 Translation

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (21)$$

### 1.6.2 Skalierung

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (22)$$

### 1.6.3 Scherung

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & h_{xy} & h_{xz} & 0 \\ h_{xy} & 1 & h_{yz} & 0 \\ h_{zx} & h_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (23)$$

### 1.6.4 Rotation

Rotation um die x-Achse:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\varphi) & -\sin(\varphi) & 0 \\ 0 & \sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (24)$$

Rotation um die y-Achse:

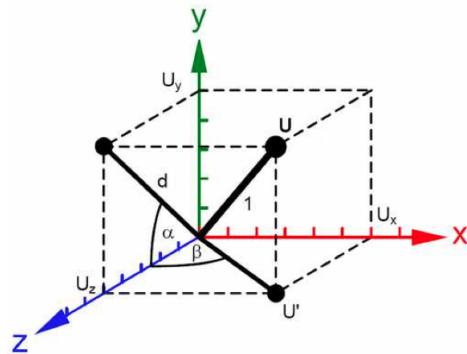
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\varphi) & 0 & \sin(\varphi) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\varphi) & 0 & \cos(\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (25)$$

Rotation um die z-Achse:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) & 0 & 0 \\ \sin(\varphi) & \cos(\varphi) & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (26)$$

### 1.6.5 Rotation um eine beliebige Achse

1. Drehen um  $\alpha$  um die x-Achse
2. Drehen um  $-\beta$  um die y-Achse
3. Drehen um  $\varphi$  um die z-Achse
4. Raum zurückdrehen:  $\beta$  um y-Achse,  $-\alpha$  um x Achse



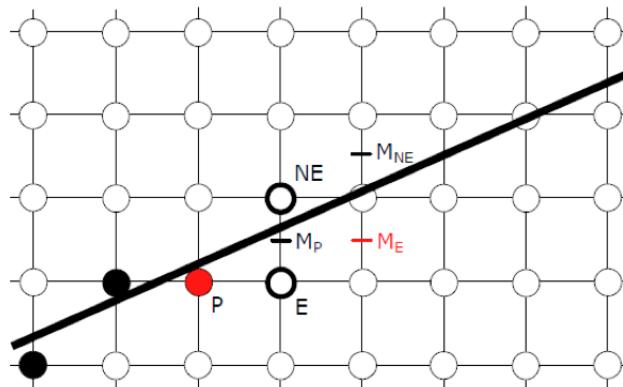
## 2 Rasteralgorithmen

### 2.1 Inkrementeller Algorithmus

Listing 1: Inkrementeller Algorithmus

```
1 void LineIncrement(int x0, int y0, int x1, int y1) {
    int dy=y1-y0;
    int dx = x1 - x0;
    float t = 0.5f; // offset
    drawPoint(x0, y0);
6   if (Abs(dx) > Abs(dy)) {
        float m = (float) dy / (float) dx;
        t = t + y0; // Initialisieren von y
        dx = (dx < 0) ? -1 : 1;
        m = m * dx; // Teilstieigung berechnen
11  while (x0 != x1) {
            x0 = x0+dx;
            t = t + m; // Steigung zu y addieren
            drawPoint(x0, (int) t); // Runden durch Casting
        } else {...}
16 }
}
```

### 2.2 Bresenham Algorithmus



Listing 2: Bresenham Algorithmus

```
void Line_Bresenham(int x0, int y0, int x1, int y1) {
    int dy =y1-y0;
3   int dx = x1 - x0;
    int deltaE = (dy<<1) - (dx<<1);
    int deltaNE = (dy<<1);

    int stepx, stepy;
8   if(dy<0){dy=-dy; stepy=-1;} else {stepy=1;}
    if(dx<0){dx=-dx; stepx=-1;} else {stepx=1;

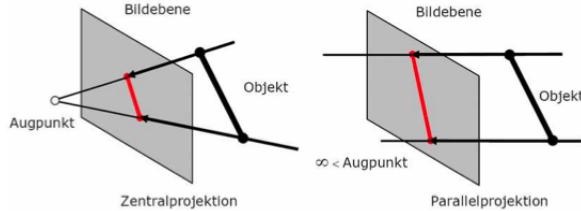
    drawPoint(x0, y0);

13  if (dx > dy) {
        int D = (dy<<1) - dx;
        while (x0 != x1) {
            if(D<0) { D=D+deltaE; }
            else { y0=y0+stepy; D = D + deltaNE; }
18        x0 = x0 + stepx;
            drawPoint(x0, y0);
        } else {...}
    }
}
```

## 3 Perspektive: 3D auf 2D

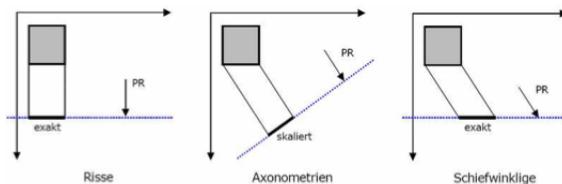
### 3.1 Kategorien der Projektionen

Es gibt zwei Arten von Projektionen: perspektivische und parallele. Bei Parallelprojektionen ist der Blickpunkt unendlich weit entfernt, wodurch sich parallele Projektionsstrahlen ergeben. Im Gegensatz dazu führen die Projektionsstrahlen bei Zentralprojektionen ins Zentrum, in den Blickpunkt.



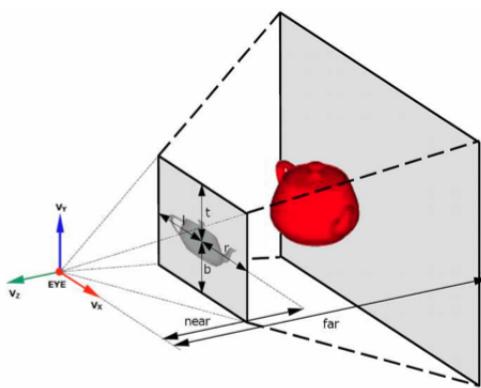
### 3.2 Parallelprojektion

Bei Parallelprojektionen sind die Projektionsstrahlen parallel zueinander. Sie können gegen die Projektionsebene schief oder senkrecht (orthogonal) stehen. Parallelprojektionen werden vor allem in technischen Zeichnungen verwendet, um die Tiefeninformationen ablesbar zu halten. Im Gegensatz zur Perspektive sind weiter entfernte Objekte nicht kleiner als nahe Objekte.



### 3.3 Zentralprojektion

Durch die Zusammenführung der Projektionsstrahlen im Projektionszentrum entsteht eine optische Tiefenwirkung. Das Sichtvolumen (Englisch für View Frustum) entspricht bei der perspektivischen Projektion einem Pyramidenstumpf. Es wird wie das orthogonale Sichtvolumen durch die Parameter  $t=\text{top}$ ,  $l=\text{left}$ ,  $r=\text{right}$ ,  $b=\text{bottom}$ ,  $n=\text{near}$  und  $f=\text{far}$  definiert:



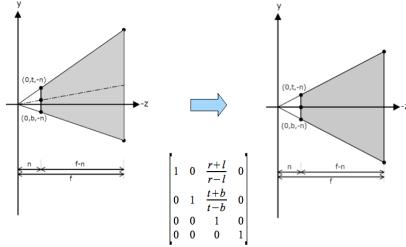
### 3.4 Projektionstransformation

Wir können View-Koordinaten nicht auf die Near-Clipping Plane projizieren. Ein zusätzlicher Schritt, die perspektivische Normalisierung, ist notwendig. Die perspektivische Transformation kann in vier Schritte zerlegen werden:

1. Scherung H in eine orthogonale Sichtpyramide
2. Skalierung S in ein kanonisches Sichtvolumen
3. Projektive Transformation N in einen Sichtwürfel
4. Perspektivische Division

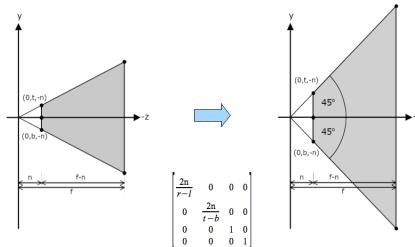
### 3.4.1 Schritt 1: Scherung H in eine orthogonale Sichtpyramide

Ist die Blickrichtung (Mittellinie der Pyramide) nicht parallel zur negativen z-Achse und somit nicht rechtwinklig zur Projektionsfläche, so muss sie zuerst mit einer Scherung H mit der z-Achse in Übereinstimmung gebracht werden.

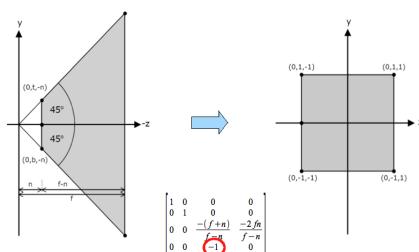


### 3.4.2 Skalierung S in ein kanonisches Sichtvolumen

Stimmt die Blickrichtung mit der negativen z-Achse überein, so wird zur Vereinfachung der perspektivischen Normalisierung im nächsten Schritt mit einer Skalierung S das Sichtvolumen in eine Sichtpyramide transformiert. Die Skalierung betrifft ebenfalls nur die x- und y-Koordinaten.



### 3.4.3 Projektive Transformation N in einen Sichtwürfel



### 3.4.4 Perspektivische Division

Die perspektivische Projektion transformiert die View-Koordinaten in homogene sogenannte Clip-Koordinaten ( $C_X, C_Y, C_Z, C_W$ ). Mit diesen Koordinaten wird in einem nächsten Zwischenschritt das 3D-Clipping durchgeführt. Polygone, die aus der Projektionsebene hinausragen, werden abgeschnitten.

$$\begin{bmatrix} C_x \\ C_y \\ C_z \\ C_w \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix}$$

$$C_x = \frac{2n}{r-l} V_x + \frac{r+l}{r-l} V_z$$

$$C_y = \frac{2n}{t-b} V_y + \frac{t+b}{t-b} V_z$$

$$C_z = \frac{-(f+n)}{f-n} V_z + \frac{-2fn}{f-n}$$

$$C_w = -V_z$$

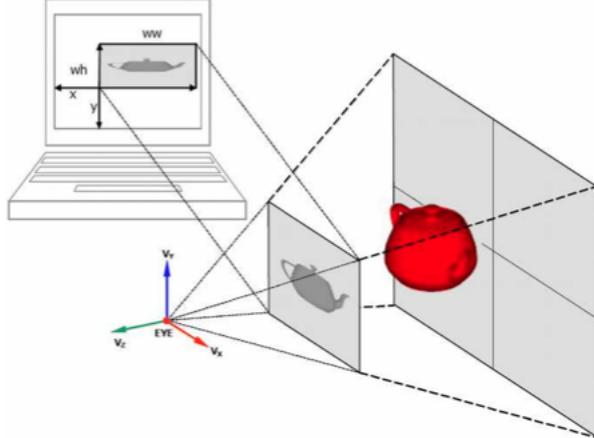
Um die gewünschten normalisierten Device-Koordinaten (NDC) ( $D_X, D_Y, D_Z$ ) zu erhalten, muss nach dem Clipping die Division durch  $C_W$  durchgeführt werden. Dieser Schritt wird auch perspektivische Division genannt:

$$\begin{bmatrix} D_x \\ D_y \\ D_z \\ 1 \end{bmatrix} = \begin{bmatrix} C_x/C_w \\ C_y/C_w \\ C_z/C_w \\ C_w/C_w \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} V_x + \frac{r+l}{r-l} V_z / -V_z \\ \frac{2n}{t-b} V_y + \frac{t+b}{t-b} V_z / -V_z \\ \frac{-(f+n)}{f-n} V_z + \frac{-2fn}{f-n} / -V_z \\ 1 \end{bmatrix} \quad D_x, D_y, D_z \in [-1, 1]$$

### 3.5 Viewport-Transformation

Wir sind mit der Projektionstransformation noch nicht ganz am Ende angelangt. Es fehlt noch die Umrechnung von den normalisierten Device-Koordinaten ( $D_X, D_Y, D_Z$ ) der virtuellen Projektionsebene zu den Fensterkoordinaten ( $S_X, S_Y$ ) und dem Tiefenbufferwert  $S_Z$  zwischen  $n$  und  $f$ . Für ein Fenster mit den Fensterkoordinaten von  $[x, y]$  unten-links bis  $[ww, wh]$  oben-rechts und einem Tiefenbuffer mit Werten zwischen  $n$  und  $f$  (bei OpenGL normalerweise 0-1) ergäbe sich eine Viewport-Matrix wie folgt:

$$\begin{bmatrix} S_x \in [x, ww] \\ S_y \in [y, wh] \\ S_z \in [n, f] \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{ww}{2} & 0 & 0 & \frac{x+ww}{2} \\ 0 & \frac{wh}{2} & 0 & \frac{y+wh}{2} \\ 0 & 0 & \frac{f-n}{2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (27)$$



## 4 OpenGL

### 4.1 Was ist OpenGL?

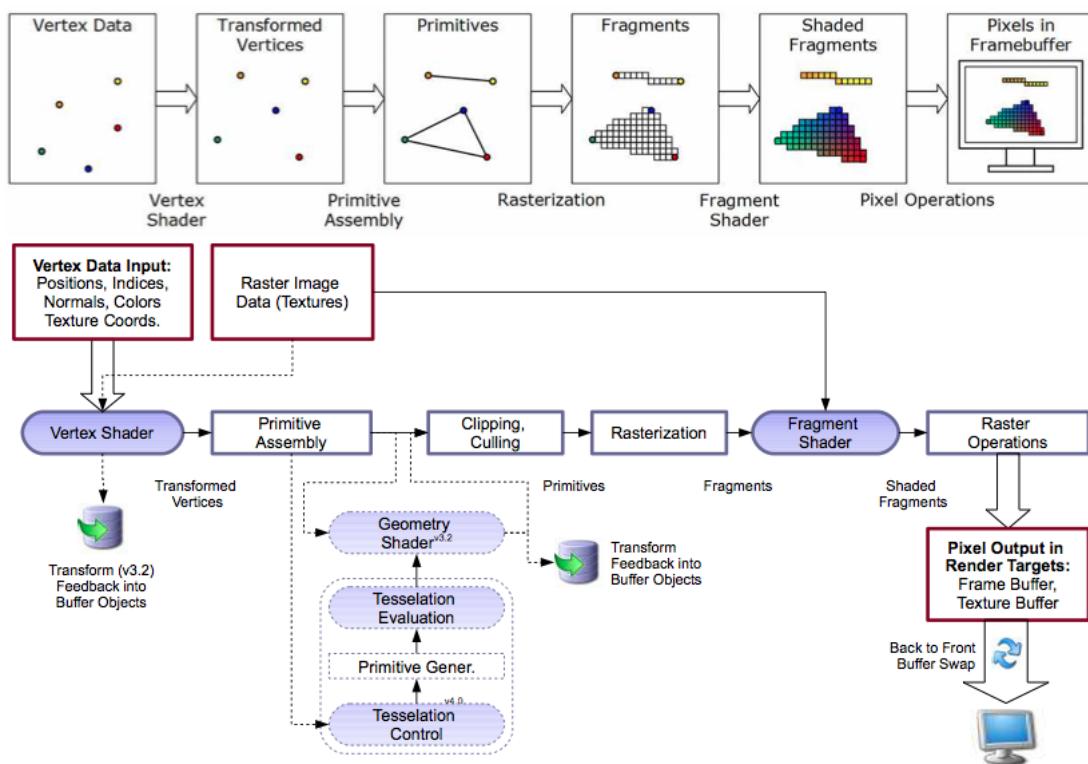
OpenGL ist ein Software-Interface zur Grafik-Hardware. Das Ziel ist die Echtzeitdarstellung (**Realtime Rendering**) von **2D- und 3D-Objekten**. Die Objekte werden durch Raumpunkte (**Vertizes**) definiert und können durch eine beliebige Transformation auf ein Ausgaberechteck (**Viewport**) projiziert werden. Die Farbe von jedem Pixel im Ausgabebild kann durch Berechnung und/oder durch Belegung mit Rasterbildern (**Texture Mapping**) bestimmt werden.

OpenGL bietet keine Funktionalität für die Anbindung an ein GUI oder andere betriebssystemabhängige Dienste. Diese werden von den jeweiligen Betriebssystemen bereitgestellt. OpenGL bietet nur primitive geometrische Objekte (Punkte, Linien und Dreiecke) an.

OpenGL ist eine Zustandsmaschine (**State Machine**). Zu jedem Zeitpunkt des Rendering-Prozesses herrscht ein gültiger Zustand. Wird ein Zustand durch ein Ereignis verändert, gilt dieser für alle nachfolgenden Objekte so lange, bis der Zustand wieder verändert wird. Das Setzen der aktuellen Farbe bestimmt z. B. den Zustand der Farbe. Dieser gilt fortan für alle folgenden Objekte, bis der Farbzustand wieder verändert wird. Jede Zustandsvariable hat einen Defaultwert. Der aktuelle Wert eines Zustandes kann mit einer `glGet*`() Funktion abgefragt werden. Bestimmte Zustände können aktiviert oder deaktiviert werden mit `glEnable` und `glDisable`.

Das OpenGL API funktioniert nach dem Client/Server Modell, wobei der Client die Applikation und die OpenGL-Implementation der Server ist.

### 4.2 Rendering Pipeline



### 4.3 Shaders

Bei der Initialisierung in `onInit` werden der Quellcode des Vertex Shaders und des Fragment Shaders mit `loadShader` geladen und mit `buildShader` und `buildProgram` kompiliert und zu einem Programm gelinkt.

Listing 3: Minimaler Vertex Shader

```
attribute vec4 a_position; // Vertex position attribute
uniform vec4 u_color; // uniform color
uniform mat4 u_mvpMatrix; // = projection * modelView
varying vec4 v_color; // Resulting color per vertex

void main(void) {
    v_color = u_color; // pass color for interpolation
    gl_Position = u_mvpMatrix * a_position; // transform vertex position
}
```

---

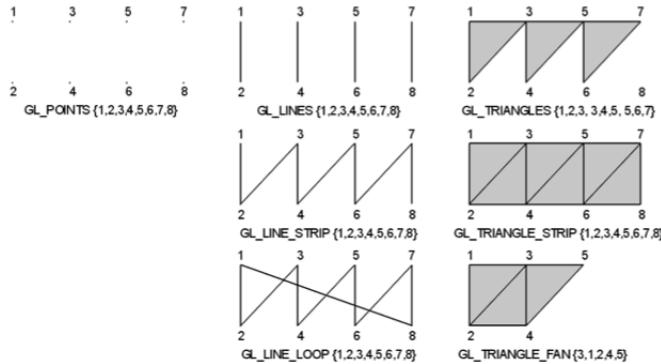
Listing 4: Minimaler Fragment Shader

```
1 varying vec4 v_col;
void main(void) {
    gl_FragColor = v_col; // Set the interpolated color to the output color
}
```

---

## 5 3D-Objekte

### 5.1 OpenGL-Primitive



### 5.2 DrawElements

Mit dem Zeichnungsbefehl **glDrawArrays** werden OpenGL-Primitive direkt mit der Vertex-reihenfolge erzeugt. Wird ein Vertex mehrfach in der Geometrie verwendet, so muss er also auch mehrfach im Array vorhanden sein.

Mit dem Zeichnungsbefehl **glDrawElements** werden OpenGL-Primitive anhand eines Index-Arrays gezeichnet. Ein mehrfach vorkommender Vertex muss so nur einmal im Array vorkommen und kann via Index mehrfach referenziert werden. Die Vertexposition verbraucht immerhin 12 Bytes, während ein Index je nach unsigned Datentyp nur 1-4 Bytes belegt.

Listing 5: Draw Elements

```
1 // Transform 2 units to the left & rebuild mvp
    _modelViewMatrix.translate(1.5f, 0.0f, 0.0f);
    mvp.setMatrix(_projectionMatrix * _modelViewMatrix);

    // Pass updated mvp and set the red color
6   glUniformMatrix4fv(_mvpLoc, 1, 0, (float*)&mvp);
   glUniform4f(_matDiffLoc, 0.0f, 1.0f, 0.0f, 1.0f);

    // Set the vertex attribute pointers to the array of structs
    GLsizei stride = sizeof(VertexPN);
11  glVertexAttribPointer(_pLoc, 3, GL_FLOAT, GL_FALSE, stride, &_v[0].p.x);
    glVertexAttribPointer(_nLoc, 3, GL_FLOAT, GL_FALSE, stride, &_v[0].n.x);

    // Draw cube with triangles by indexes
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, &i);
```

### 5.3 Vertex Buffer Object

Mit jedem Aufruf der Zeichnungsbefehle **glDrawArrays** und **glDrawElements** werden die Vertex-Arrays vom Hauptspeicher über den Datenbus auf die Grafikkarte kopiert. Dies ist unumgänglich, wenn sich die Vertexdaten sehr schnell ändern würden. Die allermeisten Objekte bleiben aber über viele Frames, wenn nicht gar für immer gleich. Auch wenn die Objekte sich bewegen, so ändern wir ja nur Transformationsmatrizen und nicht die Struktur der Dreiecksnetze. Es würde also Sinn machen, wenn wir die Vertex-Arrays im RAM der Grafikkarte speichern könnten und der Kopiervorgang nur einmal durchgeführt werden müsste. Genau diese Möglichkeit bieten uns die Vertex Buffer Objekte.

Listing 6: VBO erstellen

```
void buildVBO(GLuint &vboID, void* dataPointer, GLint numElements, GLint elementSize,
    GLuint typeSize, GLuint targetTypeGL, GLuint usageTypeGL) {
```

```
// if buffer exist delete it first
5   if (vboID) glDeleteBuffers(1, &vboID);

// Generate a buffer id
glGenBuffers(1, &vboID);

// binds (activates) the buffer that is used next
10  glBindBuffer(targetTypeGL, vboID);

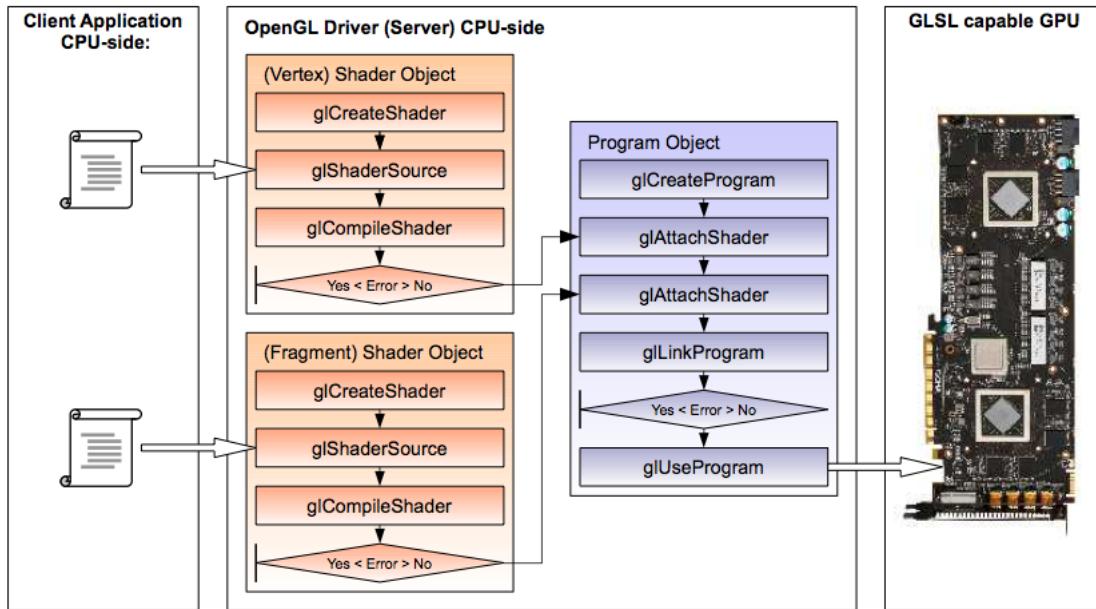
// determine the buffersize in bytes
SLint bufSize = numElements * elementSize * typeSize;

15 // copy data to the VBO on the GPU. The data could be deleted afterwards.
    glBindBufferData(targetTypeGL, bufSize, dataPointer, usageTypeGL);
    glBindBuffer(targetTypeGL, 0);
}
```

---

## 6 Shader

### 6.1 Shader Programme kompilieren, linken & aktivieren



### 6.2 einfache Datentypen

Listing 7: Datentypen

```
void          // Datentyp fuer Funktionen ohne Rueckgabewert
2 float       // Gleitkomma Variable nach IEEE Single Precision Definition
int          // Integer Variable mit 32 bit
uint         // Unsigned Integer Variable mit 32 Bit
bool         // Boolesche Variable (true/false)
vec2, vec3, vec4 // Gleitkomma Vektor mit 2, 3, oder 4 Komponenten
7 ivec2, ivec3, ivec4 // Integer Vektor mit 2, 3, oder 4 Komponenten
uvec2, uvec3, uvec4 // Unsigned Integer Vektor mit 2, 3, oder 4 Komponenten
bvec2, bvec3, bvec4 // Boolescher Vektor mit 2, 3, oder 4 Komponenten
mat2, mat3, mat4 // Matrix mit 2x2, 3x3 oder 4x4 Gleitkomma Komponenten
mat2x3, mat2x4 // Matrix mit 2 Spalten und 3 oder 4 Zeilen
12 mat3x4      // Matrix mit 3 Spalten und 4 Zeilen
sampler1D ,2D ,3D // Texturzugriffsdatentyp fuer 1D-, 2D-, oder 3D-Texturen
samplerCube    // Texturdatentyp fuer Cube Mapping Zugriff
```

### 6.3 Typen Qualifizierer

#### 6.3.1 const

- Konstanten müssen bei der Deklaration initialisiert werden.
- Kann benutzerdefiniert sein.
- Kann vordefiniert sein durch GLSL.

#### 6.3.2 attribute

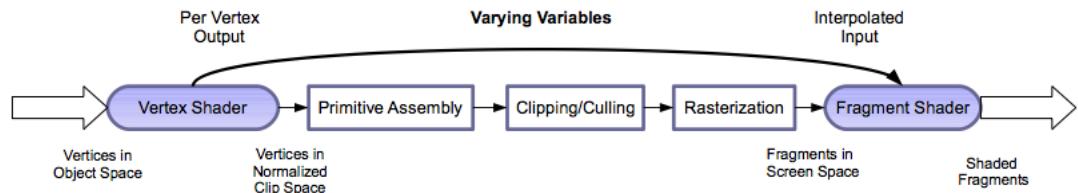
Mit attribute qualifizierte Variablen dienen als Übergabeparameter der Vertex-Attribute vom Hauptprogramm ans Vertex-Programm. Die Werte können mit jedem Vertex ändern.

### 6.3.3 uniform

Als uniform qualifizierte Variablen dienen als Übergabeparameter vom Hauptprogramm in ein Vertex- oder Fragment-Programm. Deren Werte bleiben für einen ganzen Zeichnungsbefehl konstant.

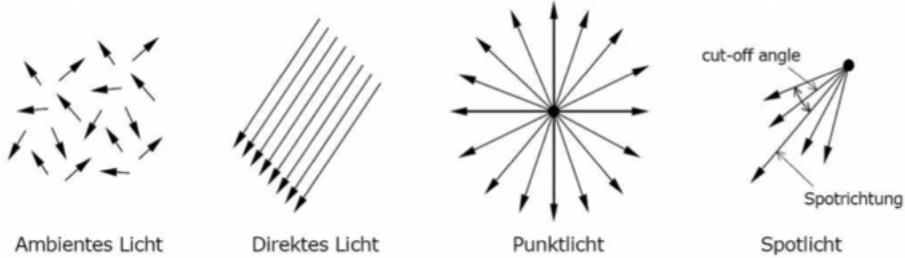
### 6.3.4 varying

Als varying qualifizierte Variablen dienen als Übergabeparameter vom Vertex- zum Fragment-Programm. Der Wert einer Varying-Variable wird im Vertex-Programm gesetzt und steht dann über das Primitiv interpoliert im Fragment-Programm zur Verfügung.



## 7 Beleuchtung

### 7.1 Lichtquellen



### 7.2 Reflexion

#### 7.2.1 Ambiente Reflexion

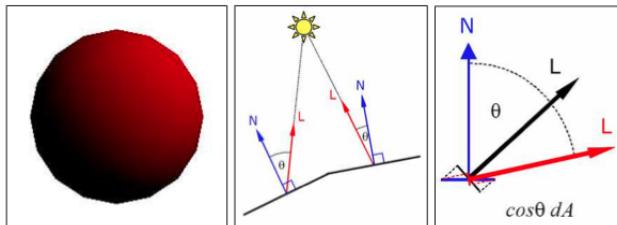
$I_a$  bezeichnet darin die konstante, ambiente Lichtintensität und  $k_a$  den ambienten Reflexionskoeffizienten (zw. 0 und 1) des reflektierenden Materials.

$$I_{ambi} = I_a * k_a \quad (28)$$

#### 7.2.2 Diffuse Reflexion

$I_d$  bezeichnet darin die konstante, diffuse Lichtintensität und  $k_d$  den diffusen Reflexionskoeffizienten (zw. 0 und 1). Wenn N und L normalisiert sind, kann der Kosinus durch das Skalarprodukt beider Vektoren ersetzt werden.

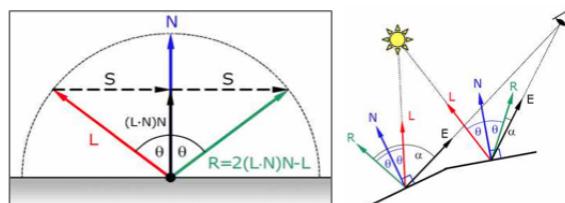
$$I_{diff} = I_d * k_d * max(\cos(\theta), 0) = I_d * k_d * max(N \bullet L, 0) \quad (29)$$



#### 7.2.3 Spiegelnde Reflexion

Die Menge des reflektierten Lichts ist vom Beobachterstandpunkt abhängig bzw. vom Winkel  $\alpha$  zwischen dem Beobachtungsvektor  $E$  (eye) und dem reflektierten Strahl  $R$ , sowie von einer Materialbeschaffenheit  $n$  bezüglich der Reflektierbarkeit (Shininess) ab.  $I_s$  bezeichnet die konstante, spiegelnde Lichtintensität und  $k_s$  den spiegelnden Reflexionskoeffizienten (zw. 0 und 1).

$$I_{spec} = I_s * k_s * max(\cos(\alpha), 0)^n = I_s * k_s * max(R \bullet E, 0)^n \quad (30)$$



#### 7.2.4 Lichtabnahme (Attenuierung)

Bevor wir alles zusammenstellen können, müssen wir noch die Energieabnahme des Lichts (Lichtabnahme) in Abhängigkeit der Distanz berücksichtigen. Darin sind  $c_1$ ,  $c_2$  und  $c_3$  Konstanten für die konstante, die lineare und die quadratische Lichtabnahme in Abhängigkeit der Distanz  $d$ . Diese Lichtabnahme wirkt nur auf die diffuse und die spiegelnde Reflexion.

$$f_{att} = \min\left(\frac{1}{c_1 + c_2 * d + c_3 * d^2}, 1\right) \quad (31)$$

### 7.3 Beleuchtungsmodell nach Phong

$$I_{Phong} = I_{amb} + \sum_{i=0}^{lights-1} f_{att} * (I_{diff} + I_{spec}) \quad (32)$$

### 7.4 Erweiterungen im alten OpenGL Beleuchtungsmodell

$$I_{OpenGL} = k_e + I_{aglobal} * k_{aglobal} + \sum_{i=0}^{lights-1} f_{att} * S_{spott} * (I_{ambi} + I_{diff} + I_{spec}) \quad (33)$$

**Emittierende Farbintensität**  $k_e$  für selbststrahlende Objekte

**Globale ambiente Hintergrundintensität**  $I_{aglobal}$  unabhängig von einer Lichtquelle

**Spotlichteffekt**  $S_{spot}$  bewirkt die Begrenzung des Spotlights und eine Lichtabnahme innerhalb des Lichtkegels von der Spotachse zum Kegelrand bewirkt. Der Wert von SspotEffect ist:

- 1, wenn das Licht kein Spotlight ist (Cut-Off Winkel=180).
- 0, wenn das Licht ein Spotlight ist, der Vertex aber ausserhalb des Spotkegels ist.
- $\max(L \bullet S, 0)^{spotexp}$ , wobei L der Vektor von der Vertexposition zum Licht ist und S die Spotrichtung und spotexp ein Exponent zwischen 0 und 128. Standardwert ist 0.
- Ein zunehmender Exponent bewirkt eine Abnahme des Lichtes zum Kegelrand.

### 7.5 Blinn's Halbvektor

Jim Blinn steuerte eine vereinfachte Berechnung des spekulären Anteils bei, indem er den Winkel zwischen der Normalen N und einem Halbvektor H berechnet. H ist der normalisierte Halbvektor zwischen der Blickrichtung E und der Lichtrichtung L.

$$I_{specBlinn} = I_s * k_s * \max(N \bullet H, 0)^n \quad (34)$$

$$(R \bullet E)^n \approx (N \bullet H)^{4n} \quad (35)$$

### 7.6 Schattierung

#### 7.6.1 Berechnung pro Dreieck: Flat Shading

Im alten OpenGL war es möglich die Beleuchtungsrechnung auf den letzten Vertex eines Polygons zu beschränken. Dies bewirkte eine flache Schattierung und war die einfachste und schnellste Beleuchtung.

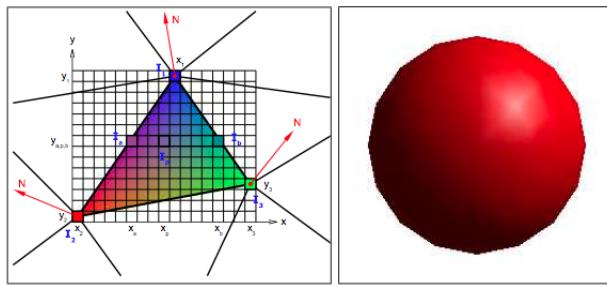
### 7.6.2 Berechnung pro Vertex: Gouraud Shading

Im alten OpenGL wurde die Blinn'sche Version des Phong Modells pro Vertex gerechnet und die Farbwerte dann über das Polygon interpoliert. Dabei werden die Farbwerte zwischen den Eckpunkten eines Polygons linear interpoliert. Der Scanline Algorithmus wird in folgende Schritte gegliedert:

1. An allen Ecken eines Dreiecks müssen die Normalen definiert sein, die aus den Flächenormalen der jeweiligen Nachbardreiecke gemittelt wurden.
2. Die Farbwerte an den Ecken gemäss Beleuchtungsmodell bestimmen.
3. Pixelzeile für Pixelzeile werden die Kantenwerte linear interpoliert.
4. Pixelintensitäten berechnen durch Interpolation zwischen den Werten.

**Nachteile:**

- Ungenaue Glanzlichter
- Keine Lichtkegel
- Keine perspektivische Verzerrungen
- Abhängigkeit von der Orientierung
- Nicht repräsentative Knotennormalen

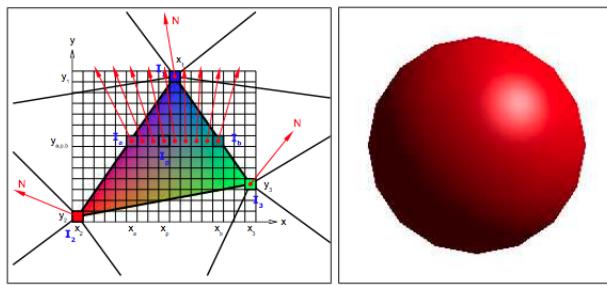


### 7.6.3 Berechnung pro Pixel: Phong Shading

Es liegt auf der Hand, dass die Phong Shading wesentlich aufwendiger ist, wenn die gesamte Rechnung pro Pixel anstatt nur pro Vertex gemacht werden muss. Zusätzlich muss ja vorgängig für jedes Pixel eine interpolierte Normale und Pixelposition berechnet werden.

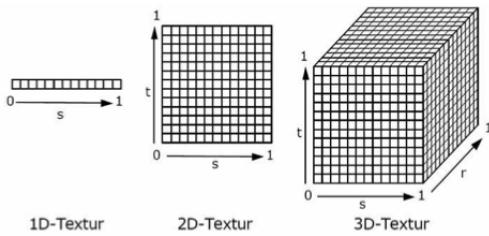
**Nachteile:**

- Polygonale Silhouetten
- Spotlichtgrenzen nicht auf Kugeloberflächen

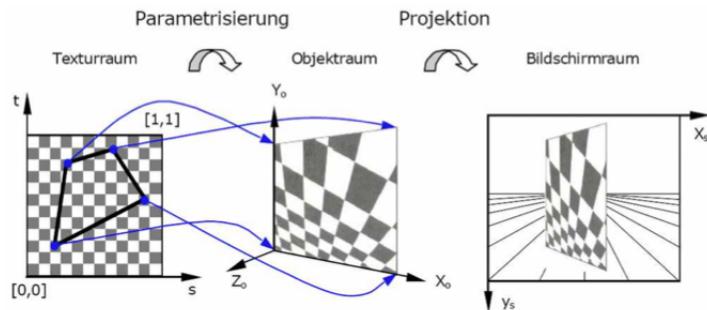


# 8 Texture Mapping

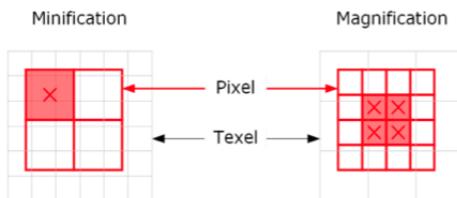
## 8.1 verschiedene Texturen



## 8.2 Textur Abbildung



## 8.3 Texturfilterung

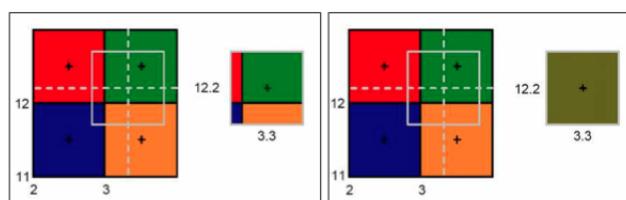


### 8.3.1 Point Sampling

Wird keine Filterung angewendet, so wird nach der Point Sampling oder Nearest Neighbour Methode gearbeitet, bei der das dem Pixelzentrum (Kreuz) am nächsten liegende Texel verwendet wird. Dies ist also im Prinzip nur ein Abschneiden der Nachkommastellen.

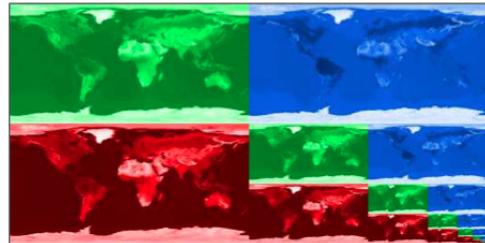
### 8.3.2 Bilineare Filterung

Bei der bilinearen Filterung werden die benachbarten Texel mit berücksichtigt. Da in X- und Y-Richtungen interpoliert wird, nennt man diesen Filter bi-linear.



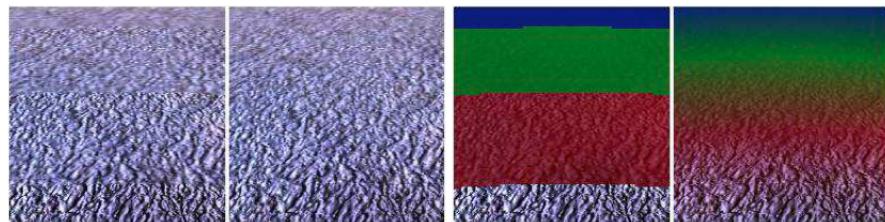
### 8.3.3 MIP-Mapping

Die Idee dahinter ist, eine Textur bereits vor der Anwendung mit mehreren Verkleinerungen anzulegen, um diese je nach Distanz zum Betrachter einzusetzen.



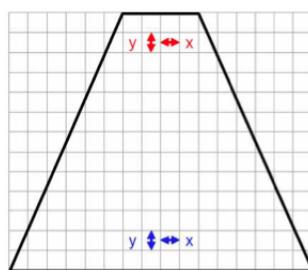
### 8.3.4 Trilineare Filterung

Der Übergang von einem MIP-Level zum nächsten fällt besonders bei grossen Flächen mit gleicher Textur auf. Die verschiedenen aufgelösten MIP-Level bilden dabei eine scharfe Kante. Beim trilinearen Filtern werden nun zuerst die entsprechenden Texel der zwei MIP-Levels bilinear gefiltert und diese dann noch einmal linear (eben tri) zwischen den beiden MIP-Levels.

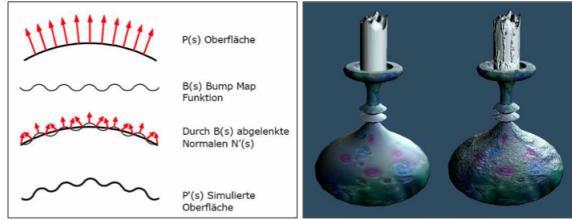


### 8.3.5 Anisotropische Filterung

Die trilineare Filterung hat nur noch einen Mangel. Die perspektivische Verzerrung wird zwar bei der Bestimmung der Texel-Position berücksichtigt, bei der bilinearen Interpolation fällt sie aber unter den Tisch. Man müsste Texturen, auf die man sehr schräg sieht, mit anderen Verfahren filtern als jene, auf die man senkrecht sieht. Da für beste Qualität nicht mehr isotrop, also gleichmäßig gefiltert werden kann, muss das ungleichmäßig (sprich anisotrop) getan werden. Beim bilinearen Filtern werden bekanntlich nur 4 Texel gemischt. Für den trilinearen Filter sind 2 bilinear gefilterte Texel.

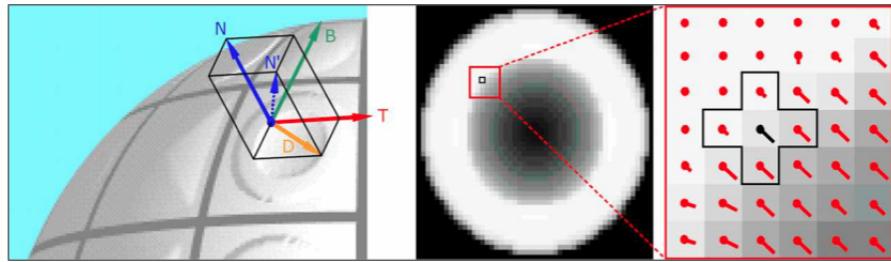


## 8.4 Bump Mapping



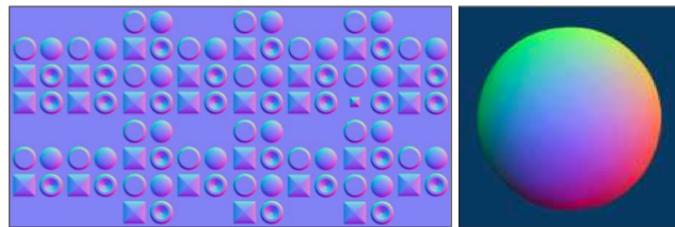
### 8.4.1 Heightfield Bump Mapping

Die Normale wird durch einen Vektor  $D$  abgelenkt, indem er zur Normalen hinzugefügt wird. Der Ablenkungsvektor  $D$  berechnet sich aus den Tangentialvektoren  $T$  und  $B$  (Binormale) sowie aus der partiellen Ableitung  $d$  (= Grauwertsteigung) bei der Texturkoordinate  $s,t$  in der Bumpmap.



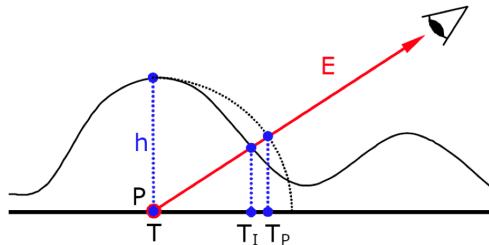
### 8.4.2 Normalmap Bump Mapping

Eine Verbesserung hinsichtlich der Performanz ist das sogenannte Normalmap Bump Mapping. Dabei wird die bereits abgelenkte Normale im Tangentenraum in einem RGB-Bild abgespeichert. Die Koordinatenkomponenten  $x$ ,  $y$  und  $z$  entsprechen den Farbkomponenten  $r$ ,  $g$  und  $b$ .



### 8.4.3 Parallax Mapping

Die Grundidee besteht darin die Texturkoordinaten (im Bild neben an nur die  $T$ -Komponente) so zu dehnen, dass wir sie den Höhen entsprechend richtig sehen. Für den Punkt  $P$  im Bild neben an würden wir ohne Parallax Mapping das Texel bei  $T$  erhalten, obwohl wir eigentlich die Farbe bei  $T_I$  (ideal) sehen sollten. Um diese Berechnung zu vereinfachen, beschränkt man den Versatz um die Höhe  $H$  entlang des Augvektors  $E$ .



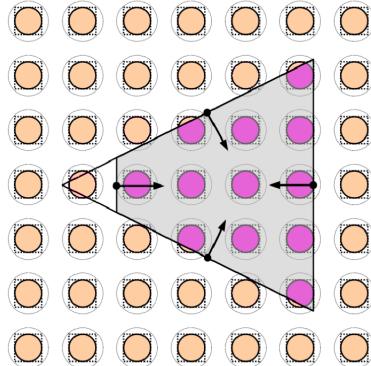
## 8.5 Texture Mapping in OpenGL

1. Texturnamen anlegen.
2. Textur binden (aktivieren).
3. Texturparameter setzen.
4. Texturdaten übergeben
5. Mipmap-Level generieren.

## 9 Frustum Culling

### 9.1 Ziel

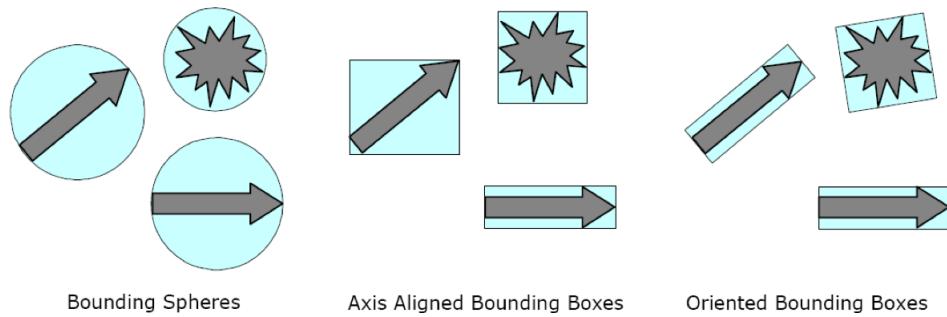
Ziel ist es, nur Objekte durch die Pipeline zu schicken, die sichtbar sind:



### 9.2 Hüllvolumen

Die Wahl des Hüllvolumens hängt von folgenden Kriterien ab:

- Passgenauigkeit des Hüllvolumens
- Kosten des Hüllvolumenschnitttests.
- Erstellungskosten des Hüllvolumens.



# 10 Alpha Blending

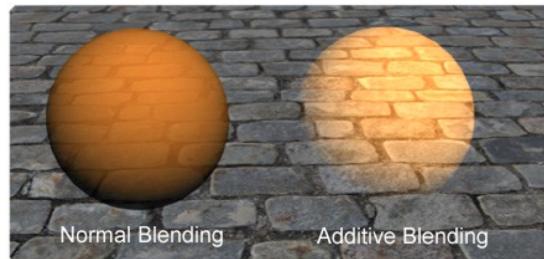
## 10.1 Einleitung

- Beim Alpha Blending wird eine neue Farbe mit der bestehenden vermischt.
- Der Alpha Wert ist der Dekungsgrad.
- Der Alpha Wert der Farbe ist der 4. Parameter bei den Farbangaben RGBA.

## 10.2 Blendfunktion

- Die Blendfunktion bestimmt, wie die bestehende Farbe ( $R_D G_D B_D A_D$ ) mit der Neuen ( $R_S G_S B_S A_S$ ) kombiniert wird.
- Der resultierende Farbe entsteht durch komponentenweise Addition, wobei man jede Komponente über die Source- bzw. Destination-Faktoren beeinflussen kann:  
$$RGBA = ((R_S S_R + R_D D_R), (G_S S_G + G_D D_G), (B_S S_B + B_D D_B), (A_S S_A + A_D D_A))$$

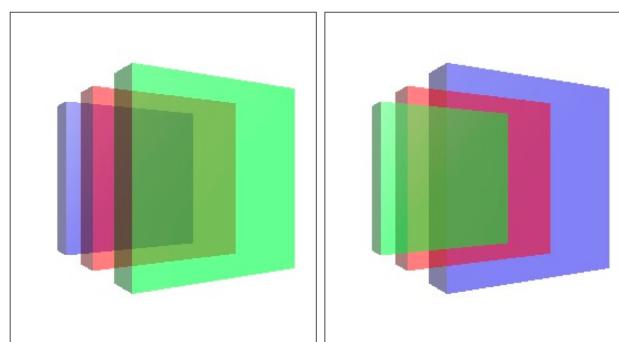
## 10.3 Additive Blending



## 10.4 Zeichnungsreihenfolge

Zeichnungsreihenfolge ist entscheidend:

1. Alle opaques (=nicht-transparente) Objekte
2. Alle transparenten Objekte



# 11 Deferred Shading

Methode um in dreidimensionalen Szenen die Geometrieverarbeitung von der Lichtberechnung zu trennen. So sind mehrere hundert dynamische Lichter in geometrisch komplexen Szenen möglich

## 11.1 Prinzip

In klassischen Rendermethoden wird anhand von Tiefe (depth), Ausrichtung (normals) und Farbe (albedo) eines Eckpunktes und Farbe, Intensität und Einfallswinkel einer Lichtquelle der finale Farbwert für den jeweiligen Eckpunkt ermittelt. Für jede Lichtquelle muss so jeder Eckpunkt zur Berechnung herangezogen werden.

Beim Deferred Shading werden nun Tiefenwert, Ausrichtung und Farbe eines jeden Pixels in jeweils eine Textur in Bildschirmgröße gespeichert. Dies wird durch sogenannte Multiple Render Targets ermöglicht, wobei in jedem Rendervorgang in verschiedene Framebufferobjekte (die Texturen) gleichzeitig geschrieben werden kann. Statt dass nun jeder Eckpunkt mit den Lichtquellen verrechnet werden muss, muss nur noch jedes Pixel (in dem alle benötigten Werte – depth, normals und albedo – vorhanden sind) bei der Berechnung berücksichtigt werden. Die Berechnung selbst erfolgt durch klassische Beleuchtungsmodelle, wie zum Beispiel nach Phong. Dabei kann zusätzlich noch Glanzlicht mit einbezogen werden. Technisch geschieht das im Pixel- bzw. Fragment-Shader am Ende der Grafikpipeline.

## 11.2 Vorteile

- Independent of scene complexity & dynamics.
- Fully in hardware.
- Well suited for deferred renderers (as normal map is typically available).

## 11.3 Nachteile

- Noise removal requires extra blur stage.
- Limited range of sample sphere makes approach relatively local and view dependent.

## 11.4 OpenGL

### 11.4.1 Framebuffer objects

- FBOs encapsulate a framebuffer that can be used for off- screen rendering.
- Each FBO has a given dimension, and a number of attachments (n color buffers, depth buffer, and stencil buffer).
- Attached buffers are either textures or renderbuffers.
- FBOs can be enabled for writing and reading.

### 11.4.2 FBO relevant API

- glGenFramebuffers, glDeleteFramebuffers
- glBindFramebuffer - Bind for reading or writing
- glClearBuffer
- glFramebufferTexture2D - Attach texture to FBO
- glCheckFramebufferStatus - Important: Check if FBO is correctly set up

## 12 Shadow Rendering

### 12.1 Einleitung

- Schatten für Echtzeitanwendungen werden normalerweise mit Deferred Rendering implementiert
- In einem ersten Schritt werden Informationen gesammelt die dann für die Beleuchtungsberechnung in einem zweiten Schritt verwendet werden

### 12.2 Shadow Volumes

- Jedes Polygon erzeugt ein Schattenvolumen (Pyramidenstumpf)
- Für jeden Sichtstrahl wird ein Zähler inkrementiert beim Eintreten in das Schattenvolumen und dekrementiert beim Austreten.
- Zonen mit einem Zähler == 0 sind beleuchtet, != 0 sind im Schatten
- Wie kann man das erreichen ohne Ray-Tracing?

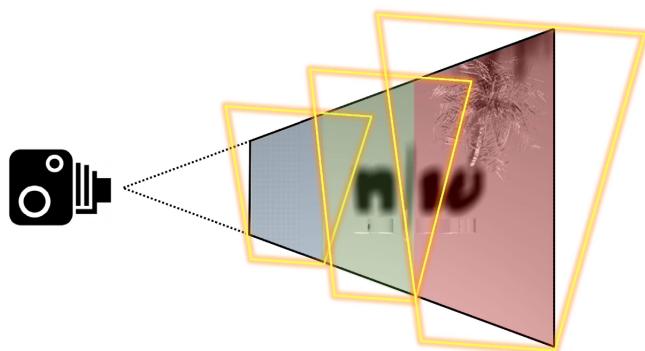
### 12.3 Shadow Maps

- Rendern der Szene um Farb- und Tiefenwerte zu erhalten
- Rendern der Shadow Volumes
- Kombinieren der Resultate

### 12.4 Variance Shadow Maps

- Multisampling vermeiden, Filter der GPU verwenden
- Wie bei PCF interessiert uns der Zusammenhang der Shadow Map Werte einer Region und des Tiefenwerte des Fragments das wir zeichnen
- Statt Shadow Map Anteile (PCF) betrachtet man die Verteilung der Tiefenwerte
- Die Tschebyscheff-Ungleichung liefert uns die Wahrscheinlichkeit bei bekanntem Mittelwert und Varianz ob ein Wert auftritt oder nicht

### 12.5 Cascade Shadow Maps



## 12.6 Vergleich

	<b>Vorteile</b>	<b>Nachteile</b>
<b>Shadow Volumes</b>	<ul style="list-style-type: none"> <li>• Präzise, geometrische Schatten</li> <li>• Unabhängig von der Abmessung der Szene</li> </ul>	<ul style="list-style-type: none"> <li>• Braucht Szenen Geometrie (funktioniert nicht Alphamasken)</li> <li>• Nur begrenzt brauchbar für Echtzeitanwendungen</li> </ul>
<b>Shadow Maps</b>	<ul style="list-style-type: none"> <li>• Einfaches Prinzip</li> <li>• Keine Geometrie Daten nötig (funktioniert z.B. auch mit Alphamasken)</li> <li>• Schnell (Szene muss nur einmal zusätzlich pro Lichtquelle gezeichnet werden)</li> </ul>	<ul style="list-style-type: none"> <li>• Ungeeignet für grosse Szenen (Auflösung der Shadow Map)</li> <li>• Unschöne (scharfe) Ränder</li> </ul>
<b>Variance Shadow Maps</b>	<ul style="list-style-type: none"> <li>• Ähnlich gute Filterung wie PCF</li> <li>• GPU verwenden für Statistik</li> <li>• Effizient</li> </ul>	<ul style="list-style-type: none"> <li>• Mehrkanal Float32 Texturen (wegen <math>Depth^2</math>) → sind z.B. Extensions in OpenGL ES</li> </ul>
<b>Cascade Shadow Maps</b>	<ul style="list-style-type: none"> <li>• Kamera-abhängige Auflösung</li> <li>• Sehr gute Resultate auch bei grossen Szenen</li> <li>• Kann einfach mit Filterung kombiniert werden</li> </ul>	<ul style="list-style-type: none"> <li>• Idealerweise implementiert mit 3D Texturturen</li> <li>• Szene muss pro Shadow-Map neu gezeichnet werden</li> </ul>
<b>Photonmapping, Raytracing, Radiosity, ...</b>	<ul style="list-style-type: none"> <li>• Beste bekannte Verfahren für Global Illumination</li> <li>• Viele Lichteffekte (nicht nur Schatten) werden miteinbezogen</li> </ul>	<ul style="list-style-type: none"> <li>• Sehr Rechenaufwendig</li> <li>• Noch nicht brauchbar für Echtzeitanwendungen</li> </ul>