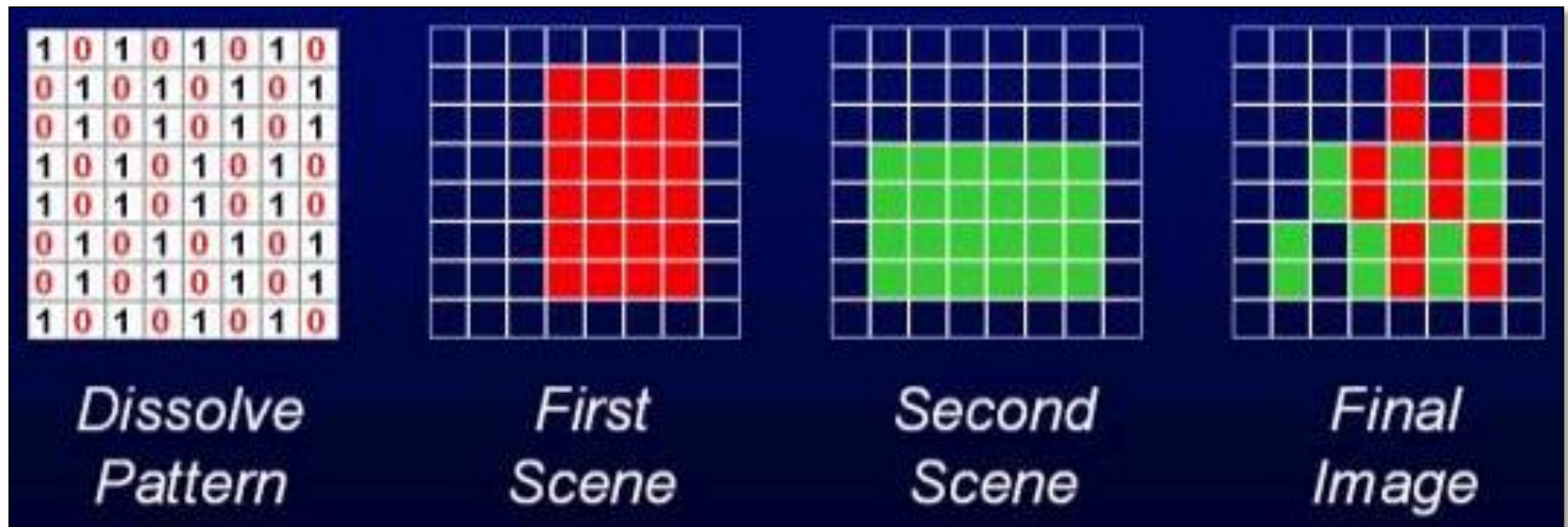


Stencil-Buffer und Stencil-Test

- ▶ mithilfe des **Stencil-Tests** kann ebenfalls entschieden werden, ob ein Fragment gezeichnet wird, z.B. Vergleich des Stencil-Werts im Buffer mit einem Referenzwert
- ▶ typische Anwendungen
 - ▶ Maskierung von Bildteilen
 - ▶ projektive Schatten und Schattenvolumen
 - ▶ Stencil-Routing



Stencil-Test und Tiefentest



Reihenfolge der Fragment-Tests (Alpha-Test nur in klassischen OpenGL)

```
if ( fragment.alpha alphafunc refalpha )  
    if ( buffer.stencil stencilfunc refstencil ) {  
        if ( fragment.depth depthfunc buffer.depth ) {  
            // es kann mehrere Color-Buffers in einem  
            // Backbuffer geben  
            foreach colorbuffer  
                // hier würde noch Blending stattfinden  
                buffer.color = fragment.color;  
            buffer.depth = fragment.depth;  
            buffer.stencil = zpass();  
        }  
    }
```

Stencil-Test und Tiefentest



Reihenfolge der Fragment-Tests (Alpha-Test nur in klassischen OpenGL)

```
if ( fragment.alpha alphafunc refalpha )  
    if ( buffer.stencil stencilfunc refstencil ) {  
        if ( fragment.depth depthfunc buffer.depth ) {  
            // es kann mehrere Color-Buffers in einem  
            // Backbuffer geben  
            foreach colorbuffer  
                // hier würde noch Blending stattfinden  
                buffer.color = fragment.color;  
            buffer.depth = fragment.depth;  
            buffer.stencil = zpass();  
        } else  
            // Stencil&Alpha bestanden, Tiefentest nicht  
            buffer.stencil = zfail();  
    } else  
        // Stencil Test nicht bestanden  
        buffer.stencil = fail();
```

Konfiguration des Stencil-Tests

▶ Stencil-Test festlegen

`glStencilFunc( stencilfunc, refstencil, mask )`

▶ vergleiche mit Referenzwert in maskierten Bits

`GL_ALWAYS`, `GL_NEVER`, `GL_LESS`, `GL_EQUAL`, ...

▶ Änderungen im Stencil-Buffer je nach dem, ob der Stencil-Test bestanden und der Tiefentest bestanden wurde oder nicht

`glStencilOp( fail, zfail, zpass )`

▶ mögliche Funktionen:

`GL_KEEP`, `GL_ZERO`, `GL_INCR`, `GL_DECR`, `GL_INVERT`, `GL_REPLACE`

▶ Beispiele

▶ zeichne Maske in den Stencil-Buffer

`glStencilFunc( GL_ALWAYS, 0x1, 0x1 );`

`glStencilOp( GL_REPLACE, GL_REPLACE, GL_REPLACE );`

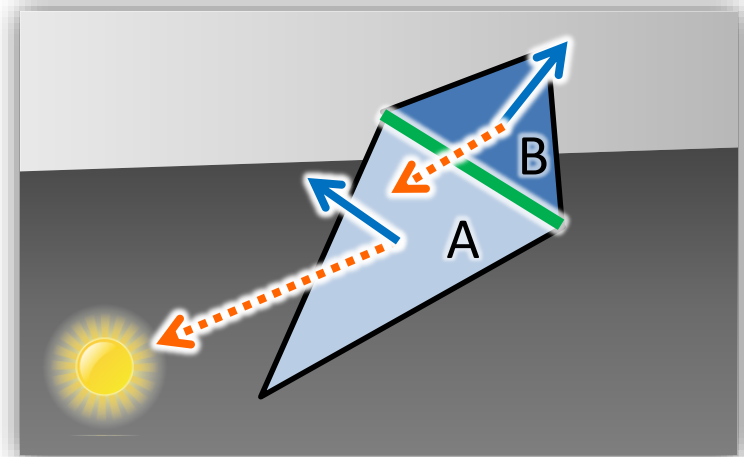
▶ zeichne Objekte nur dort, wo Stencil ungleich 1

`glStencilFunc( GL_NOT_EQUAL, 0x1, 0x1 );`

`glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );`

Bestimmung der Objektsilhouette

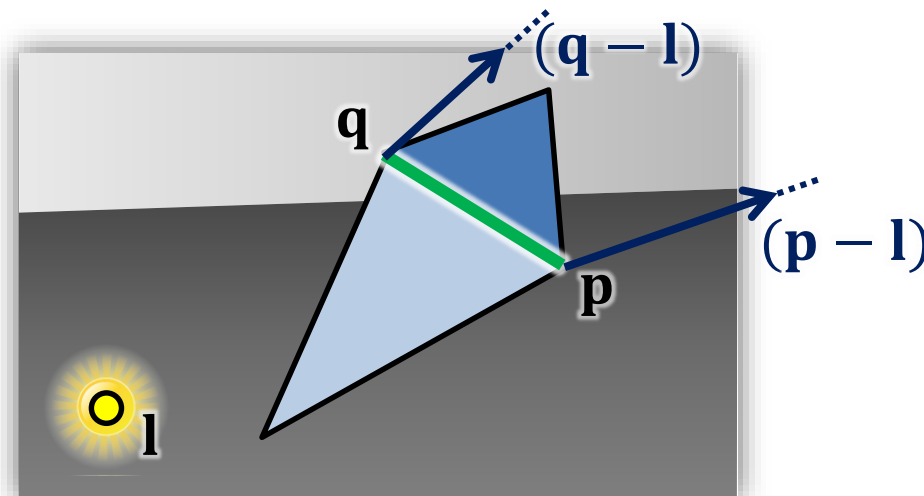
- ▶ Annahme: geschlossene Dreiecksnetze („watertight“, 2-manifold)
- ▶ zur Silhouette gehören Kanten von deren angrenzenden Dreiecken eines **zur Lichtquelle zeigt** und eines **von der Lichtquelle abgewandt** ist
- ▶ Vorgehen:
 - ▶ bestimme für jedes Dreieck, ob es zur LQ oder davon weg zeigt (Skalarprodukt zwischen der **Dreiecksnormale** und dem **Vektor zur LQ**)
 - ▶ betrachte jede Kante und ihre zwei benachbarten Dreiecke A und B:
 - ▶ zeigt A zur LQ und B zeigt weg (oder umgekehrt)
→ **Kante** ist Teil der Silhouette



Schattenvolumen eines Dreiecksnetzes

Oberfläche des Schattenvolumens

- ▶ sie die Kante $\overline{pq}$ Teil der Silhouette
 - ▶ extrudiere $\overline{pq}$ weg von der Lichtquelle ins Unendliche
 - ▶ dadurch entsteht eine **viereckige Seitenfläche des Schattenvolumen** mit den Eckpunkten $\mathbf{p}$, $\mathbf{p} + \infty \cdot (\mathbf{p} - \mathbf{l})$, $\mathbf{q} + \infty \cdot (\mathbf{q} - \mathbf{l})$ und $\mathbf{q}$
 - ▶ Punkte im Unendlichen durch homogene Koordinaten dargestellt
- ▶ und: **alle zur LQ gewandten Dreiecke sind ebenfalls Teil des Schattenvolumen**



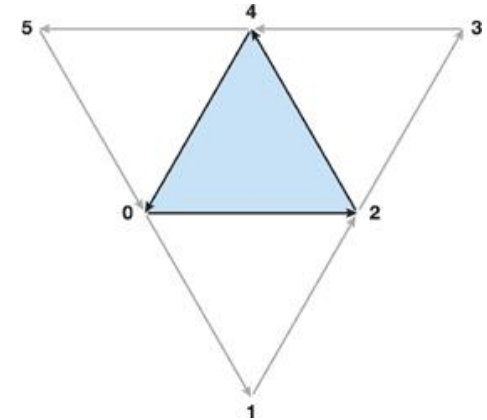
Berechnung der Silhouette auf der GPU



▶ wie berechnet man die Silhouette **effizient**?

▶ Geometry Shader:

- ▶ im GS ist es möglich auf Nachbardreiecke zuzugreifen, wenn die entsprechenden Indizes der Vertices angegeben werden
- ▶ man zeichnet Primitive vom Typ **GL_TRIANGLES_ADJACENCY_EXT** und übergibt pro Dreieck 6 Vertices statt 3
- ▶ zur Erinnerung: im GS greift man auf die Vertices zu mit **gl_PositionIn[]**
- ▶ Eingabe: Dreieck mit Nachbardreiecken
- ▶ Ausgabe: Seitenflächen des Schattenvolumen und das Dreieck selbst (wenn es zur LQ zeigt)



- ▶ detaillierte Beschreibung dieser Technik:
http://http.developer.nvidia.com/GPUGems3/gpugems3_ch11.html

Schattenvolumen: der z-fail Algorithmus

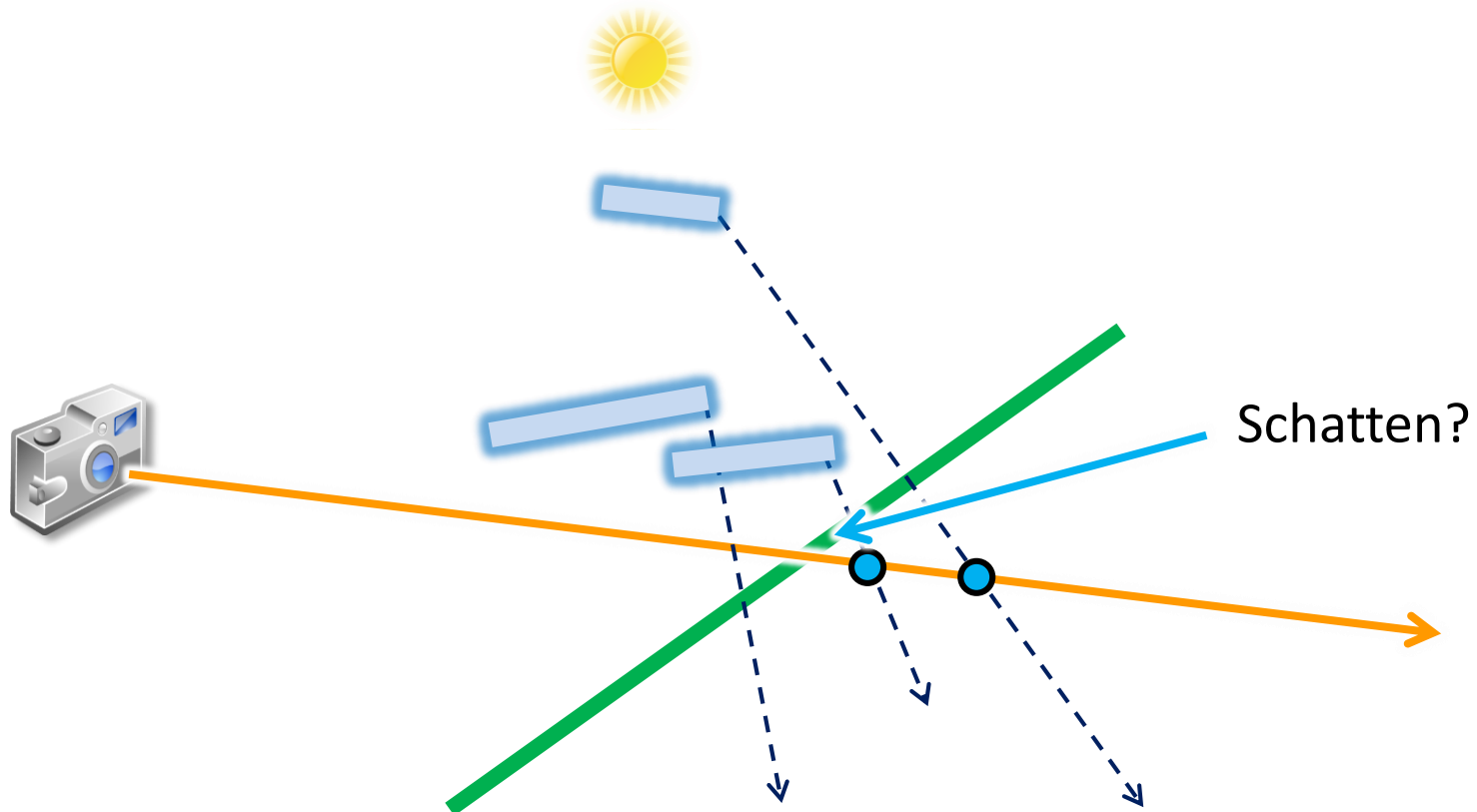


- ▶ die Probleme aufgrund des Zählens im Stencil Buffer kann man beheben mit dem sogenannten z-fail Algorithmus
- ▶ im Prinzip handelt es sich um eine andere Render- und Zählstrategie:
 - ▶ zeichne Szene (wie bisher)
 - ▶ maskiere anschließend Schreibzugriffe auf den Z-Buffer (wie bisher)
 - ▶ zeichne zuerst die Rückseiten der Schattenvolumen
 - ▶ inkrementiere Stencil Buffer, wenn der Tiefentest fehlschlägt (z-fail)
 - ▶ zeichne dann die Vorderseiten der Schattenvolumen
 - ▶ dekrementiere Stencil Buffer, wenn Tiefentest fehlschlägt
 - ▶ Resultat: Pixel mit einem Stencil-Wert > 0 befinden sich im Schatten
- ▶ Everitt et al.: „Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering“, GDC 2002

Schattenvolumen: der z-fail Algorithmus



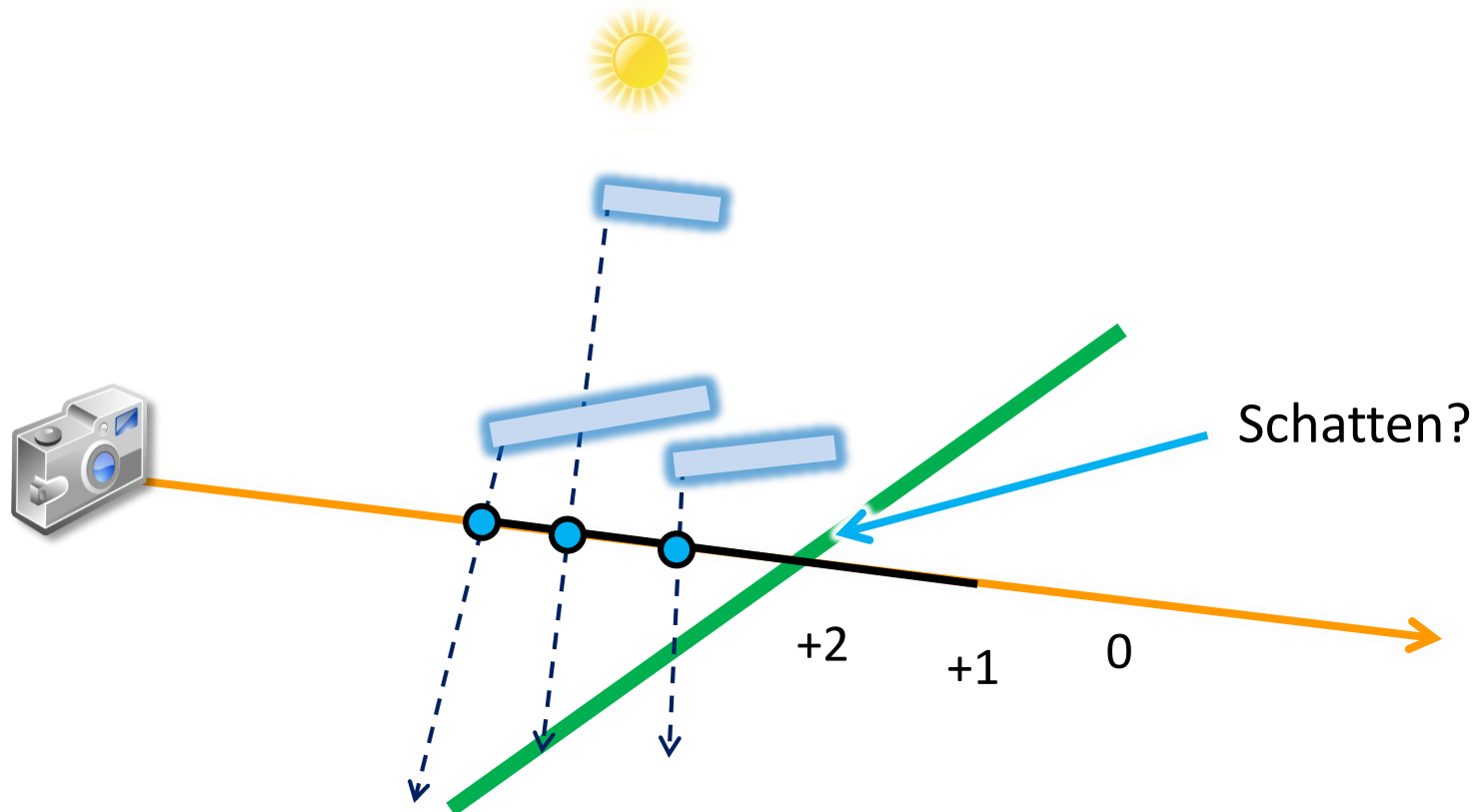
- ▶ was macht der z-fail Algorithmus?
 - ▶ anschaulich: er zählt Ein- und Austrittspunkte **von hinten nach vorne**
 - ▶ Eintrittspunkte sind an den Rückseiten, Austrittspunkte an den Vorderseiten der Schattenvolumen
 - ▶ hier: der Zähler hat den Wert 2 nach dem Zeichnen der Rückseiten



Schattenvolumen: der z-fail Algorithmus



- ▶ was macht der z-fail Algorithmus?
 - ▶ anschaulich: er zählt Ein- und Austrittspunkte **von hinten nach vorne**
 - ▶ nach dem Zeichnen der Vorderseiten: Zähler bleibt bei 2
 - ▶ alle Vorderseiten bestehen den Tiefentest, deshalb bleibt der Stencil Buffer unverändert



Schattenvolumen: z-pass und z-fail



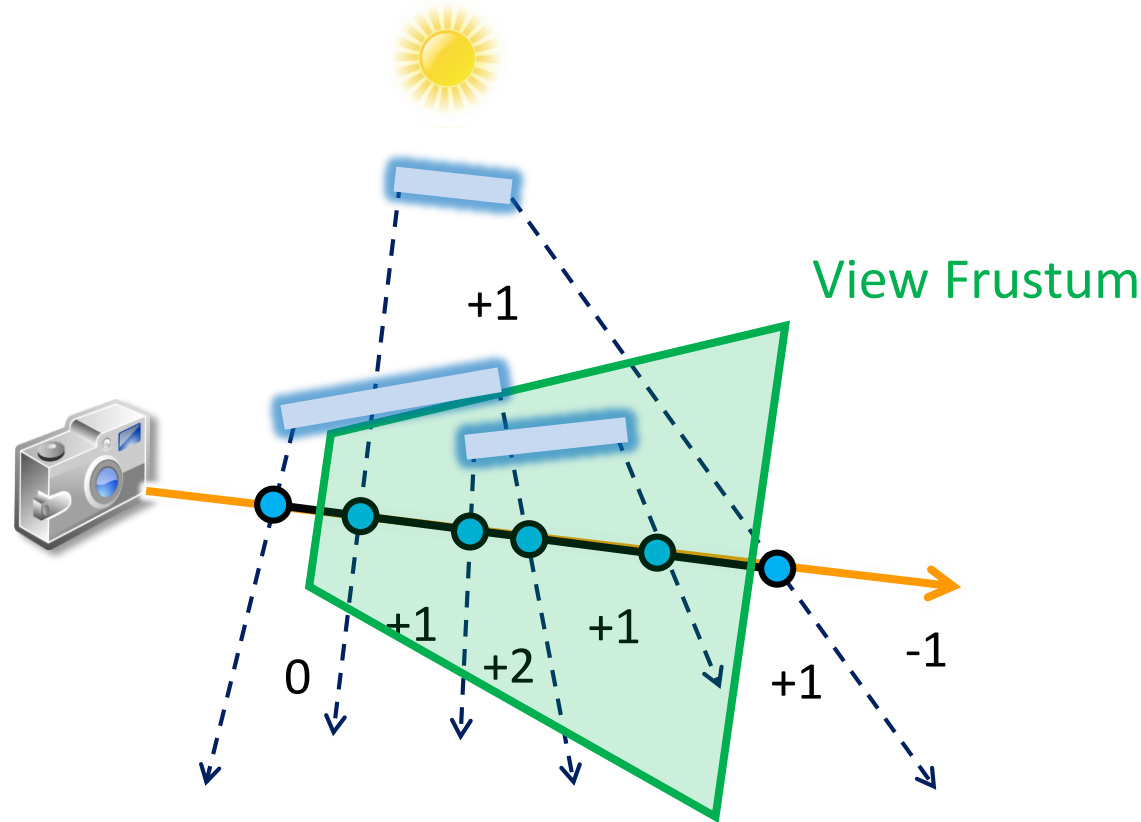
► Unterschied z-pass (Original) und z-fail

- ▶ z-pass modifiziert Stencil Buffer, wenn der Tiefentest bestanden wird
 - ▶ zählt Eintritts-/Austrittspunkte vor dem Oberflächenpunkt
 - ▶ schwierig, wenn Eintritts-/Austrittspunkte vor der Near Plane sind
- ▶ z-fail modifiziert Stencil Buffer, wenn der Tiefentest fehlschlägt
 - ▶ zählt Eintritts-/Austrittspunkte hinter dem Oberflächenpunkt
 - ▶ schwierig, wenn Eintritts-/Austrittspunkte hinter der Far Plane sind

Schattenvolumen: der z-fail Algorithmus

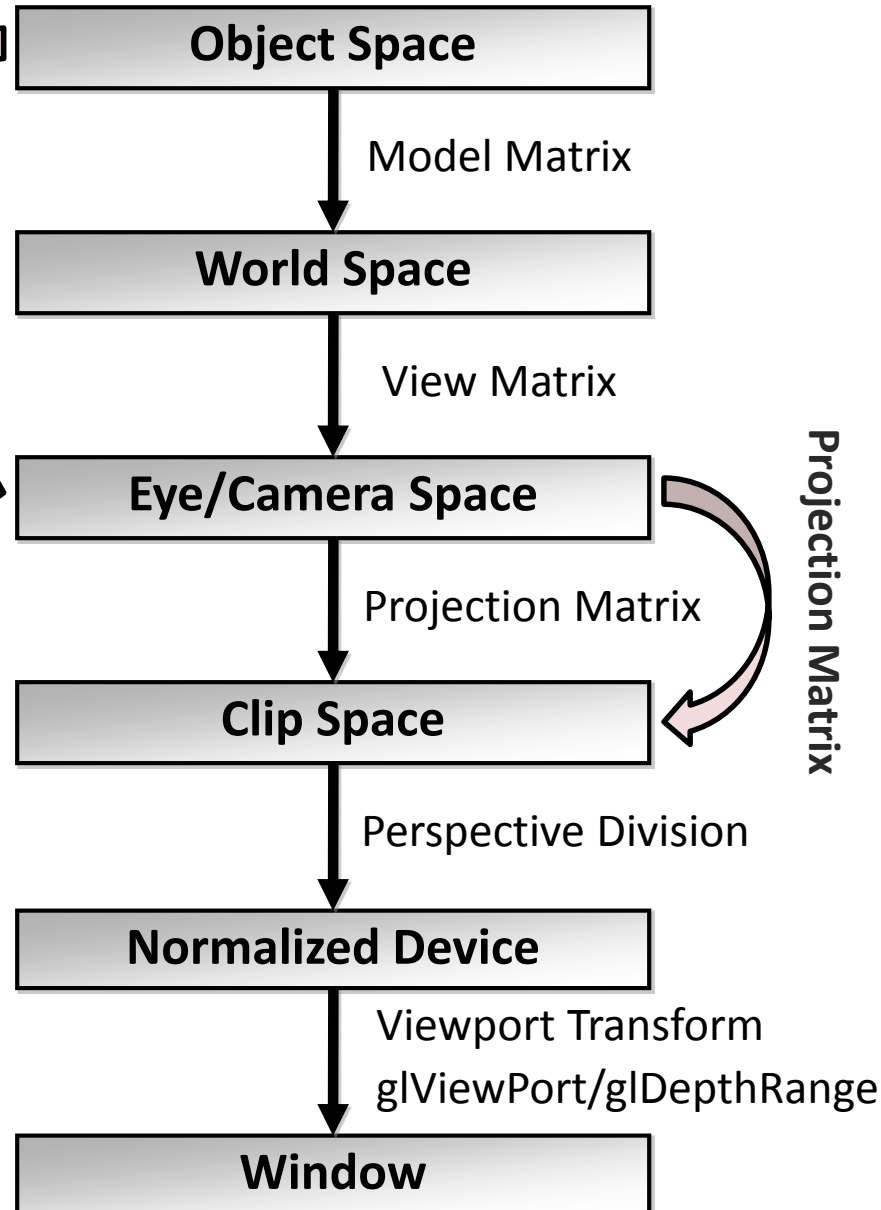


- ▶ bisher haben wir das Problem der Startwerte nicht gelöst
 - ▶ eine Initialisierung der Zähler für den z-fail Algorithmus ist nötig, wenn ein Schattenvolumen hinter der Far Plane liegt
- ▶ was haben wir damit gewonnen?

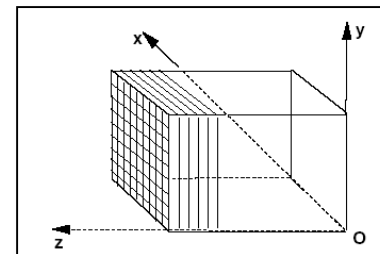
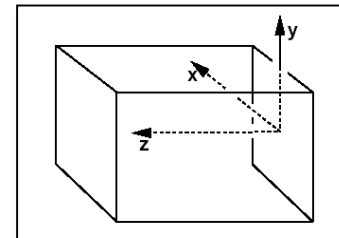
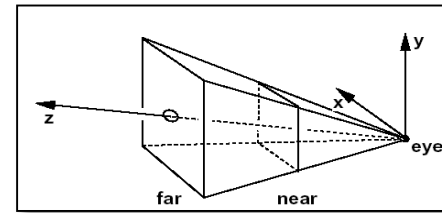
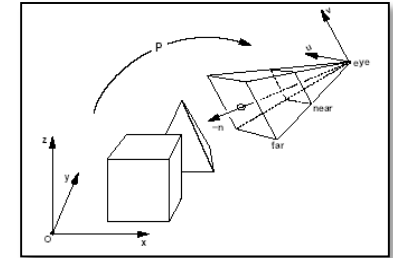
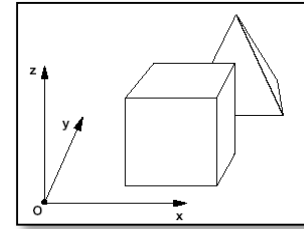
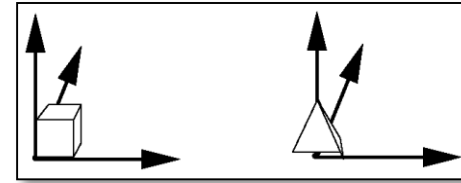


Koordinatensysteme

ModelView Matrix



Projection Matrix



`glVertex()`
`glNormal()`

Mehr Infos:
[http://www.opengl.org/
resources/faq/technical/
viewing.htm](http://www.opengl.org/resources/faq/technical/viewing.htm)

[http://www.songho.ca/
opengl/gl_transform.htm](http://www.songho.ca/opengl/gl_transform.htm)

Schattenvolumen: der z-fail Algorithmus

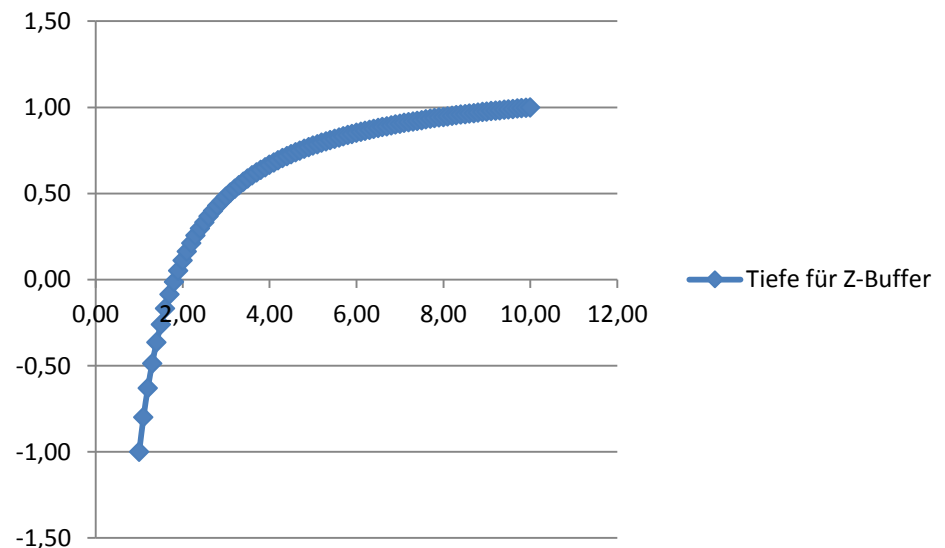


- ▶ die Lösung ist einfach: verschiebe die Far Plane ins Unendliche!
- ▶ eine Projektionsmatrix erzeugt mit `glFrustum(l, r, b, t, n, f)`

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

- ▶ Abbildung der Tiefe (hier: horiz. Achse) auf den Tiefenwert (vert. Achse)

▶ Near Plane $n = 1$,
Far Plane $f = 10$



Schattenvolumen: der z-fail Algorithmus



- ▶ was passiert in der Projektionsmatrix, wenn wir die Far Plane ins Unendliche verschieben, also $f \rightarrow \infty$?

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

▶ $\frac{f+n}{f-n} \rightarrow 1$ und $\frac{2fn}{f-n} \rightarrow 2n$

- ▶ Anm. die Vorzeichen in der Matrix sind durch die OpenGL Konvention vorgegeben

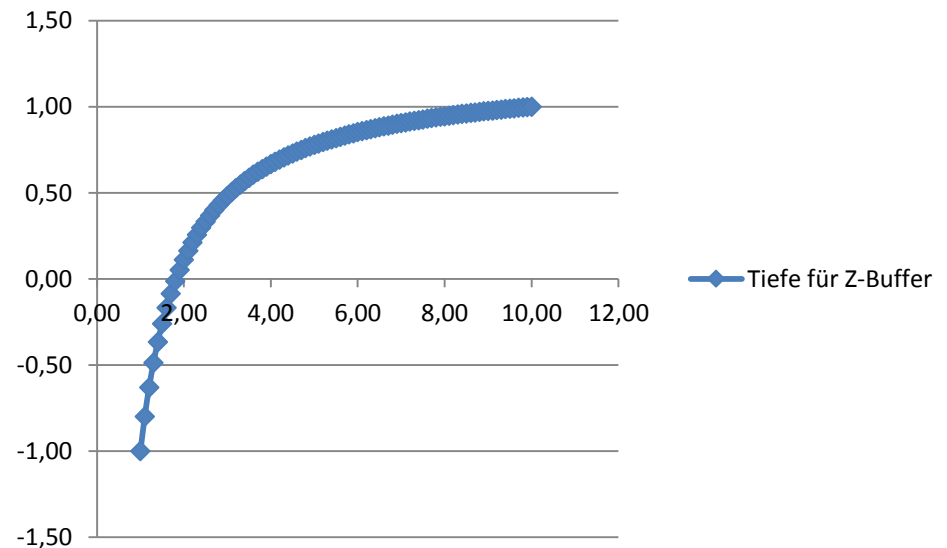
- ▶ für die Matrix ergibt sich dementsprechend:

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

Schattenvolumen: der z-fail Algorithmus



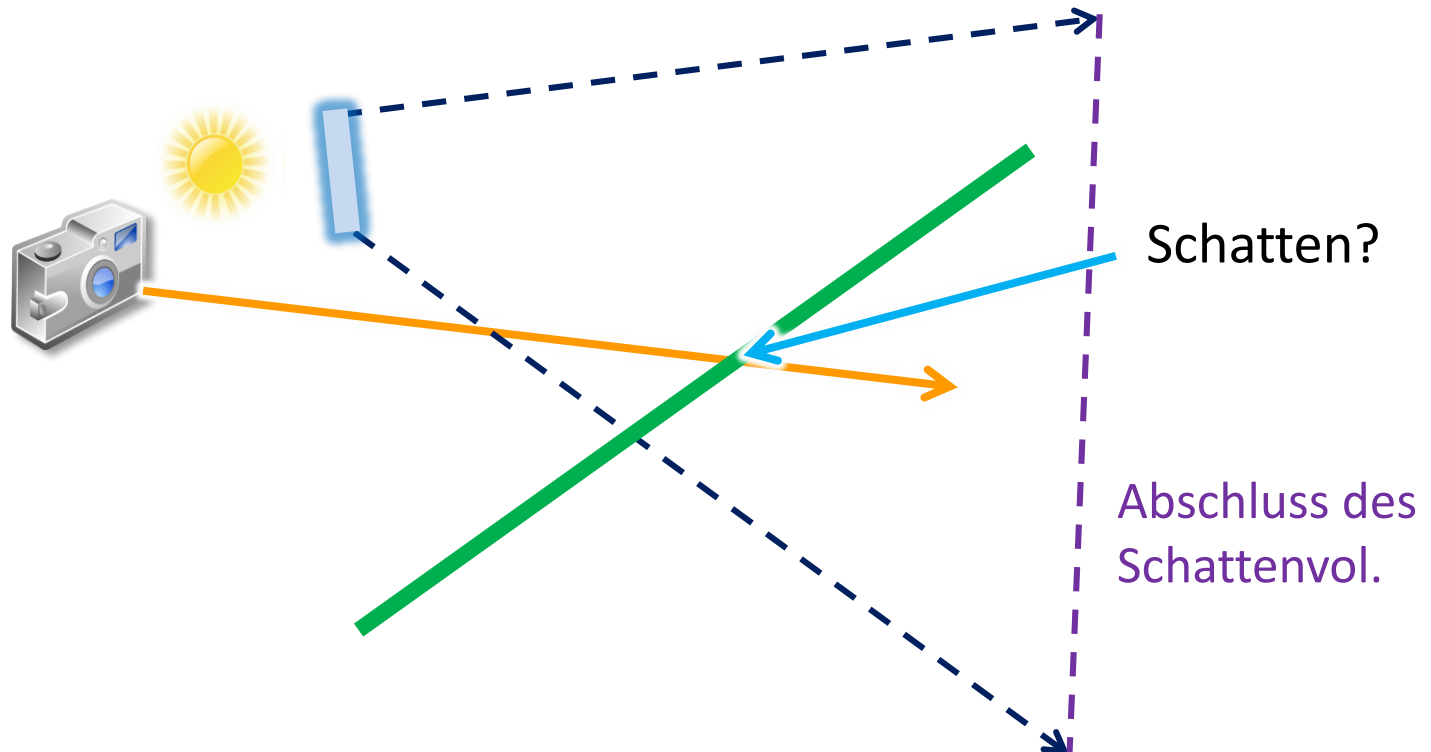
- ▶ zur Probe betrachten wir einen Punkt im Raum mit der Entfernung $|z_c|$ vor der Kamera (in OpenGL ist $z_c < 0$)
- ▶ der resultierende Tiefenwert ist $z' = \frac{-z_c - 2n}{-z_c} = 1 + \frac{2n}{z_c}$
- ▶ für $|z_c| \rightarrow \infty$: $z' \rightarrow 1$
- ▶ Far Plane repräsentiert also tatsächlich unendlich weit entfernte Punkte
- ▶ der Genauigkeitsverlust im Tiefenpuffer (für andere entfernte Flächen) ist akzeptabel, da die Abbildung der Tiefe ohnehin nicht linear ist
- ▶ wichtig für den z-fail Algorithmus ist nur: die Flächen werden tatsächlich rasterisiert



Schattenvolumen: der z-fail Algorithmus



- ▶ einen „pathologischen Fall“ gibt es noch zu beachten:
beim z-fail Algorithmus müssen die Schattenvolumen auch „hinten“ geschlossen sein
- ▶ hinter dem betrachteten Punkt auf der grünen Fläche würde sonst nie ein Fragment rasterisiert werden, das den Tiefentest nicht besteht
- ▶ deshalb hat das Schattenvolumen eine **Rückseite**



```
varying vec3 normal;
```

```
void main() {  
    gl_Position = gl_Vertex;  
    normal = gl_Normal;  
}
```

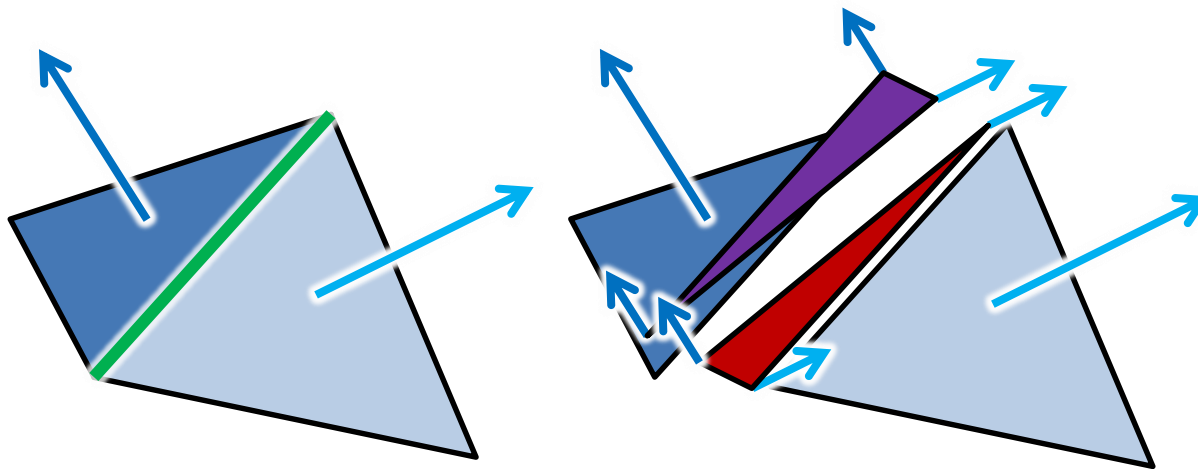
```
varying in vec3 normal[];
```

```
void main() {  
    for ( int i = 0; i < gl_VerticesIn; i++ ) {  
        gl_Position  = gl_ModelViewProjectionMatrix * gl_PositionIn[ i ];  
        ...  
        gl_FrontColor = vec4( kd );  
        EmitVertex();  
    }  
    EndPrimitive();  
}
```

▶ wie berechnet man die Silhouette **effizient**?

▶ **Vertex Shader:**

- ▶ der Trick erfordert einen Vorbereitungsschritt: erzeuge zwei „degenerierte“ Dreiecke (Flächeninhalt 0) **an jeder Kante**
- ▶ verwende die Normale der jeweiligen Dreiecke für die Vertizes dieser Dreiecke
- ▶ extrudiere im Vertex Shader alle Vertizes, deren Normalen von der Lichtquelle weg zeigen
- ▶ Nachteil: deutlich mehr Dreiecke – wie viele denn eigentlich?



- ▶ beschreiben den Zusammenhang zwischen Anzahl der **Vertizes** V , **Kanten/Edges** E und **Facetten** F eines konvexen Polyeders

- ▶ einfache (konvexe) Polyeder

$$V - E + F = 2$$

- ▶ Polygonnetz mit Rand

$$V - E + F = 1$$

- ▶ die Zahl $\chi = V - E + F$ heißt Euler-Charakteristik eines Polyeders

- ▶ gleiches χ kennzeichnet topologisch äquivalente Polyeder

- ▶ wenn Sie der Beweis interessiert:

- ▶ „Nineteen Proofs of Euler's Formula”

<http://www.ics.uci.edu/~eppstein/junkyard/euler/>

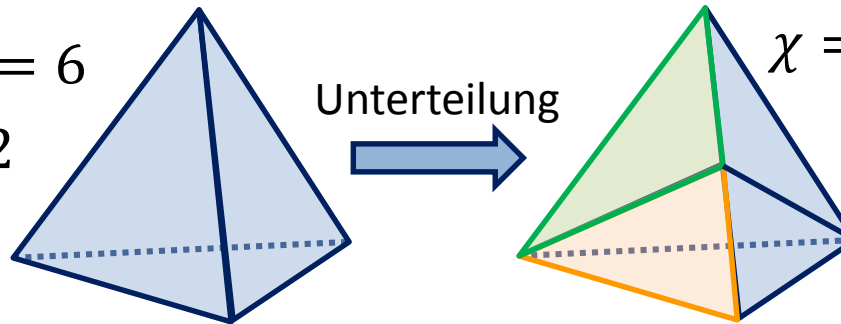
- ▶ beschreiben den Zusammenhang zwischen Anzahl der **Vertizes** V , **Kanten/Edges** E und **Facetten** F eines konvexen Polyeders
- ▶ einfache (konvexe) Polyeder

$$V - E + F = 2$$

- ▶ Beispiel: Tetraeder

- ▶ $V = F = 4$ und $E = 6$

- ▶ $\chi = 4 - 6 + 4 = 2$



$$V = 5, F = 6, E = 9$$
$$\chi = 5 - 9 + 6 = 2$$

- ▶ Beispiel: geschlossenes Dreiecksnetz

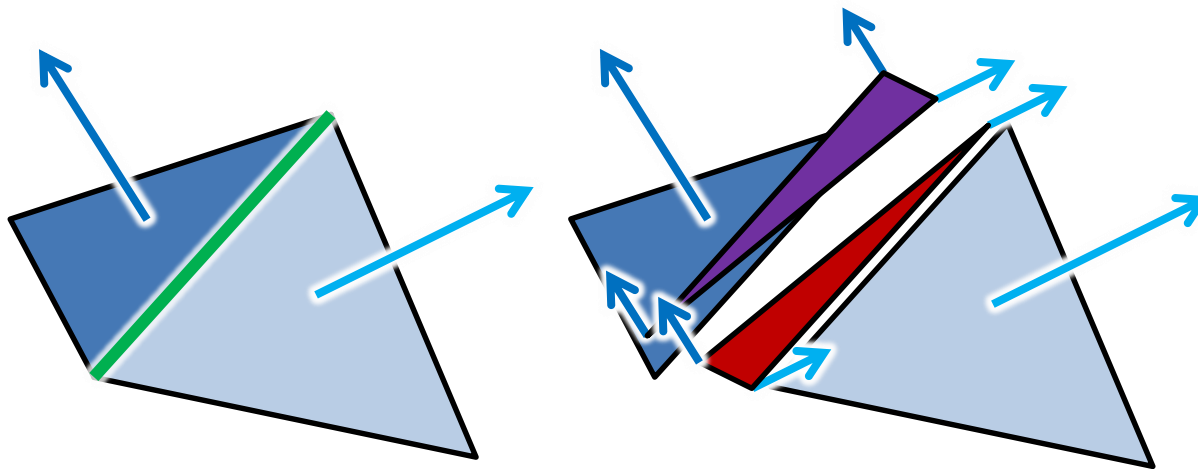
- ▶ jedes Dreieck besitzt 3 Kanten

- ▶ jede Kante besitzt 2 Nachbardreiecke

- ▶ insgesamt gibt es $\frac{2}{3}$ so viele Dreiecke wie Kanten, also $F = \frac{2}{3}E$ bzw. 1.5 mal so viele Kanten wie Dreiecke

Berechnung der Silhouette auf der GPU

- ▶ wie berechnet man die Silhouette **effizient**?
- ▶ Vertex Shader:
 - ▶ erzeuge zwei „degenerierte“ Dreiecke (Flächeninhalt 0) an jeder Kante
 - ▶ Nachteil: deutlich mehr Dreiecke – wie viele denn eigentlich?
 - ▶ insgesamt gibt 1.5 mal so viele Kanten wie Dreiecke, also $E = \frac{3}{2}F$
 - ▶ für jede Kante erzeugen wir 2 neue Dreiecke, also hat das Dreiecksnetz insgesamt $F + 2 \cdot \frac{3}{2}F = 4F$ Dreiecke!

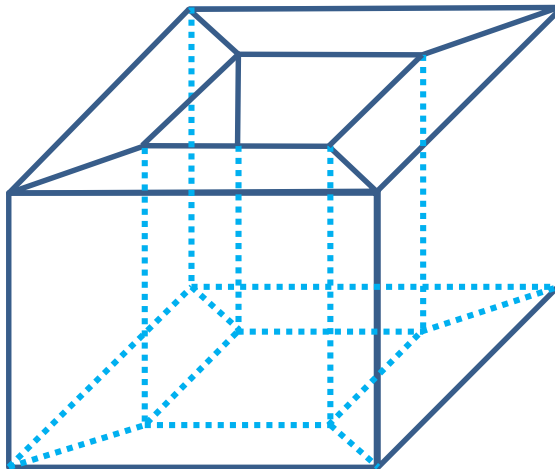


Erweiterte Euler-Formel

- ▶ ...beschreibt den Zusammenhang zwischen Anzahl der **Vertizes** V , **Kanten/Edges** E und **Facetten** F eines Polyeders
- ▶ Körper mit Löchern und Bohrungen (z.B. Torus)

$$V - E + F - H = 2(C - G)$$

- ▶ H Anzahl der Löcher in Facetten (holes)
 - ▶ C Anzahl der miteinander verbundenen Komponenten (connected components)
 - ▶ G Anzahl der Bohrungen durch den Körper (Genus)
- ▶ Beispiel:



$$\begin{array}{ccccccccc} V & - & E & + & F & - & H & = & 2(C - G) \\ 16 & & 32 & & 16 & & 0 & = & 1 & 1 \end{array}$$

Schattenvolumen Tricks

- ▶ ein Trick um ein Schattenvolumen in einem Vertex Shader zu erzeugen:
zeigt die Normale eines Vertex von der LQ weg, dann bewege ihn von der LQ weg ins Unendliche
- ▶ wird ein Dreiecksnetz (ohne Vorverarbeitungsschritt) mit diesem Shader gezeichnet erhalten wir direkt ein approximatives Schattenvolumen
 - ▶ nur für geschlossene, ausreichend tessellierte Dreiecksnetze

