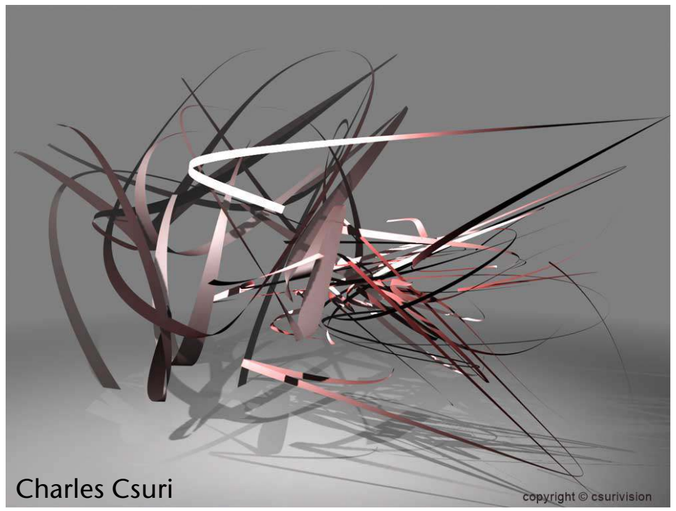
# Computer-Graphik 1

## Visibility Computations – Hidden Surfaces, Frame Buffers, and Shadows

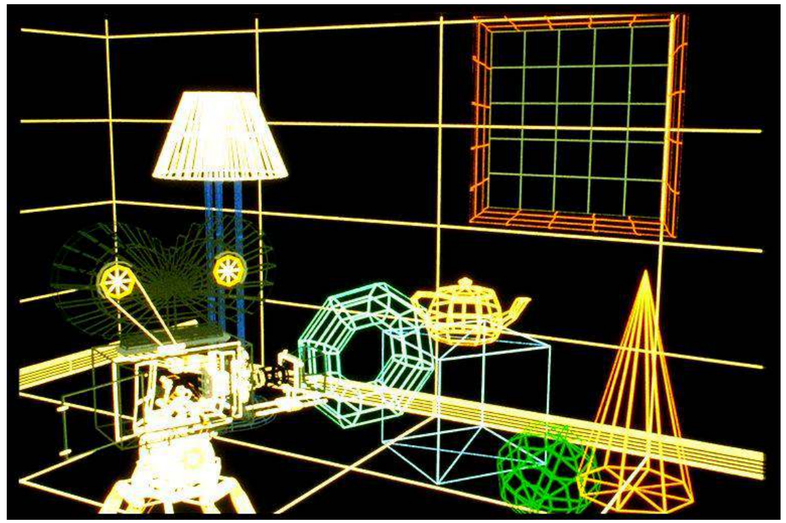


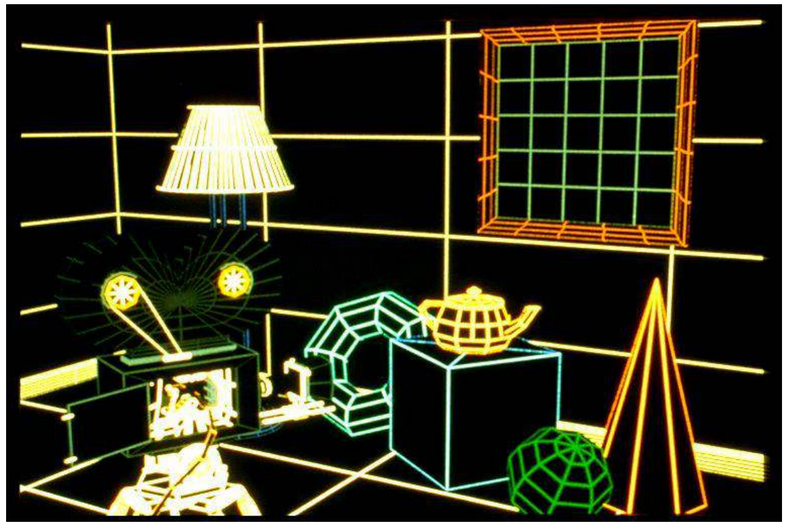

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

# Motivation

- Verdeckung (occlusion) entsteht, wenn mehrere Objekte bei der Projektion von 3D nach 2D (teilweise) die gleichen Bildschirmkoordinaten aufweisen
- Sichtbar ist das dem Auge am *nächsten* liegende Objekt
- Falls dieses Obj (halb-)durchsichtig ("transparent") ist ⟶ dahinter liegende Objekte sind auch (teilweise) sichtbar

Wireframe-Rendering *ohne* Verdeckungs-berechnung



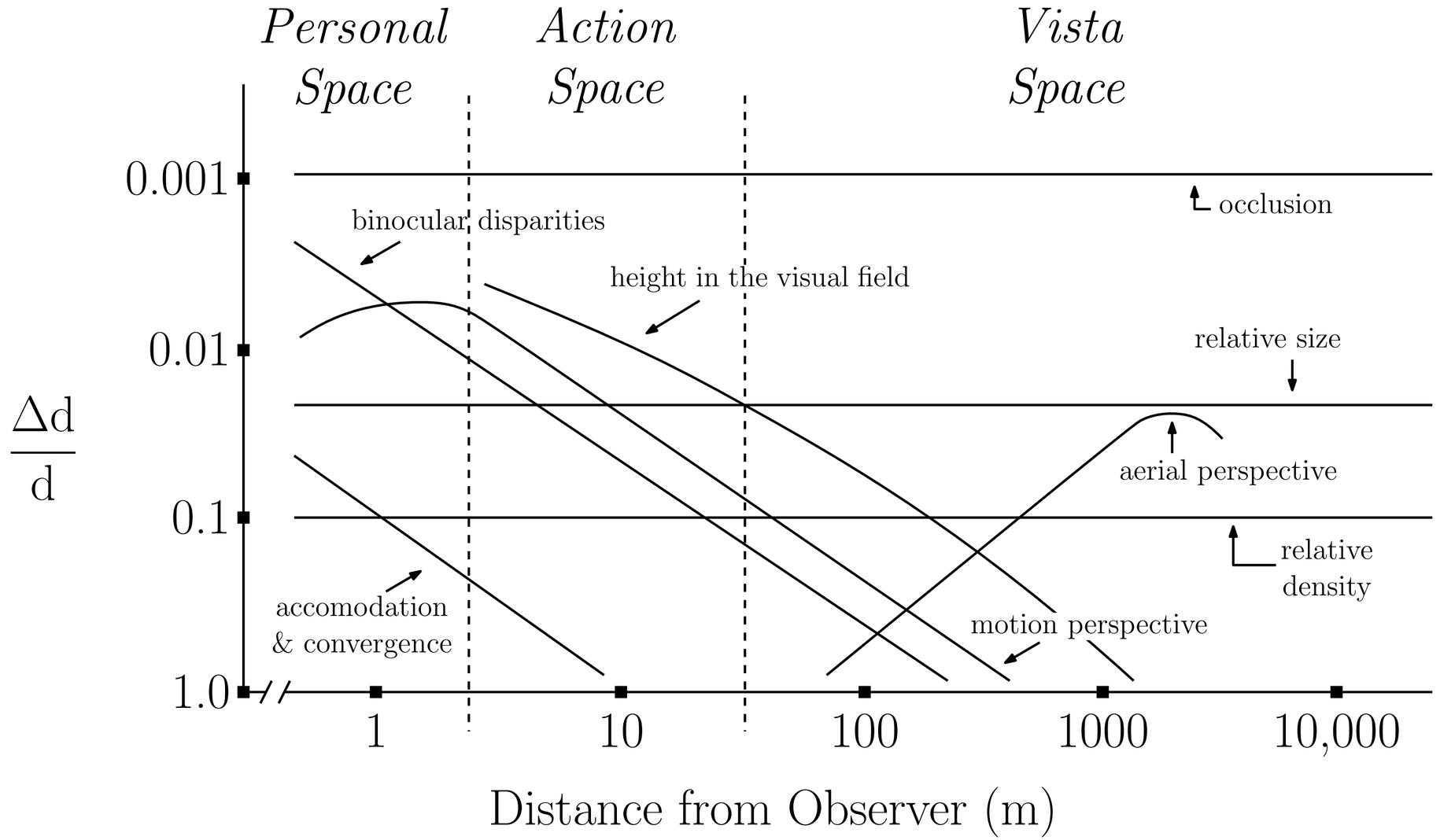


Wireframe-Rendering *mit* Verdeckungs-berechnung

Pixar "Shutterbug"

- Verdeckung ist ein extrem wichtiger (der wichtigste?) *Depth Cue*:

# Just-Noticeable Depth Thresholds (FYI)



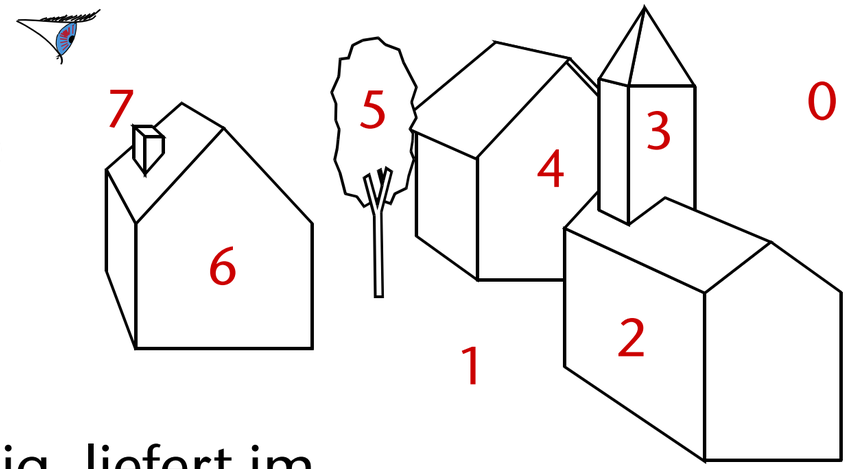
# Unterscheidung zweier wichtiger Kategorien

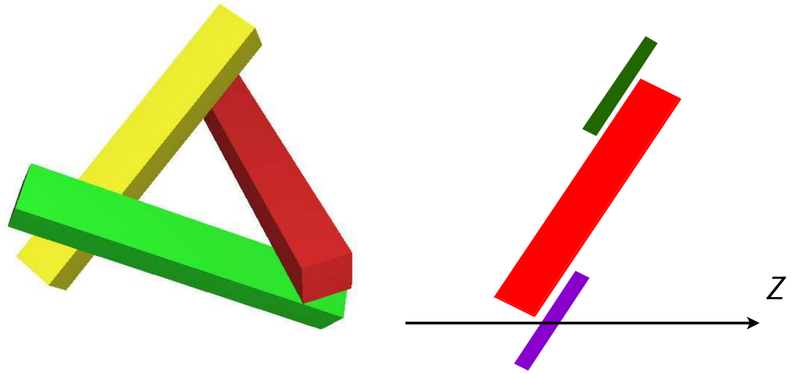
- Es gibt 2 große Problemklassen innerhalb des Bereichs "Visibility Computations"
  1. Verdeckungsberechnung: welche "Pixel" eines Polygons werden von anderen verdeckt?
     - Bezeichnungen: *Hidden Surface Elimination* (früher auch *Hidden Line Elimination*), *Visible Surface Determination*
  2. Culling: welche Polygone / Objekte *können* gar nicht sichtbar sein? (z.B., weil sie sich hinter dem Viewpoint befinden → *view frustum culling*; oder z.B., weil sie von einem anderen Objekt komplett verdeckt werden → *occlusion culling*)

     → s. "Advanced Computer Graphics"
- Achtung: die Grenzen sind fließend
  - Tendentieller Unterschied: bei HSE geht es eher darum, überhaupt ein korrektes Bild zu rendern, bei Culling geht es eher um eine Beschleunigung des Renderings großer Szenen

# Die einfachste Idee: Painter's Algorithm

- Idee: Zeichne das Bild wie ein Maler
  - Zuerst Hintergrund, dann Objekte von hinten nach vorne
- Probleme:
  - Sortierung nicht trivial, manchmal unmöglich
  - Zerlegung in Teilpolygone teuer, Viewpoint-abhängig, liefert im worst-case $O(n^2)$ viele Teile ⟶ Rendering ist nicht mehr linear in der Anzahl Polygone!

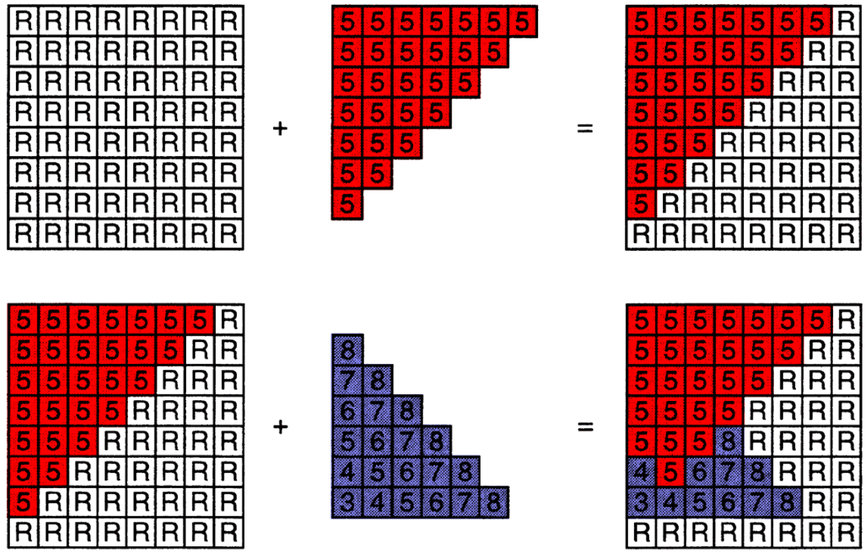


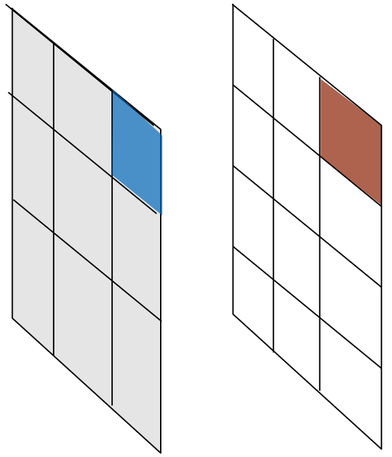




# Die Standard-Lösung heute: der Z-Buffer

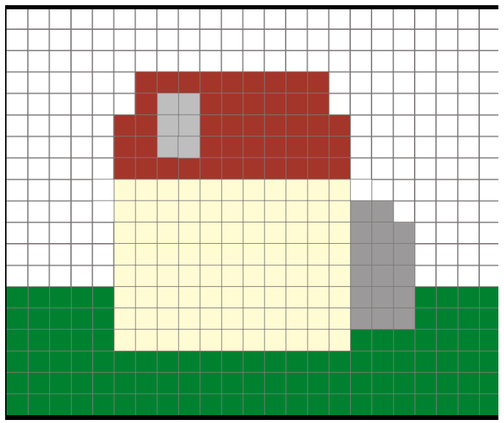

Wolfgang Strasser

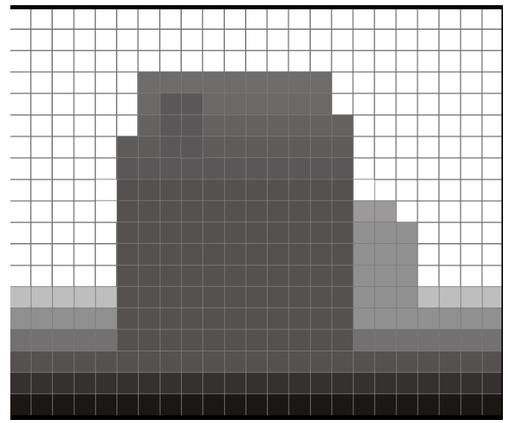
- Zusätzlicher Buffer zum Color Buffer
- Speichert pro Pixel den Abstand *z* zur Kamera
- Ein Fragment wird nur geschrieben, wenn *z* kleiner ist als der Wert im Z-Buffer
- Beispiel:




Color buffer


Z-Buffer


Color Buffer


Z-Buffer

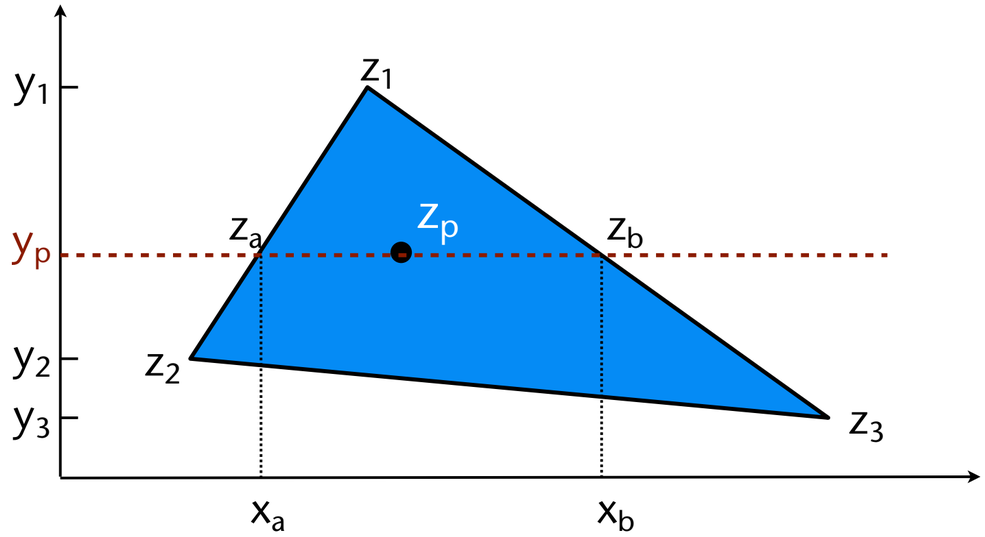
# Das Zitat der Woche

*"640 Kilobyte ought to be enough for anybody."*

Bill Gates, 1981

# Berechnung des Z-Wertes für die Scan-Conversion

1. Alternative: mehrfache lineare Interpolation




$$z_a = z_1 + \frac{y_p - y_1}{y_2 - y_1}(z_2 - z_1)$$

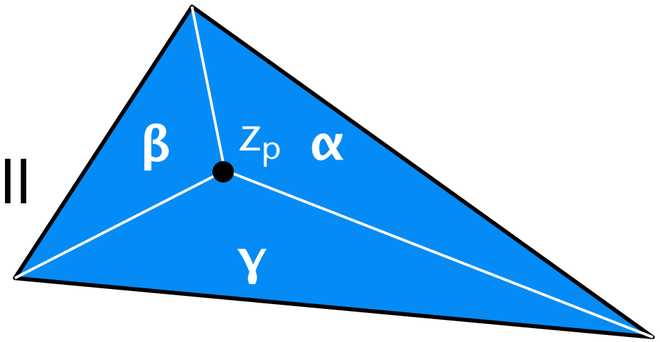
$$z_b = z_1 + \frac{y_p - y_1}{y_3 - y_1}(z_3 - z_1)$$

$$z_p = z_a + \frac{x_p - x_a}{x_b - x_a}(z_b - z_a)$$

2. Alternative:

$$z_p = \alpha z_1 + \beta z_2 + \gamma z_3$$

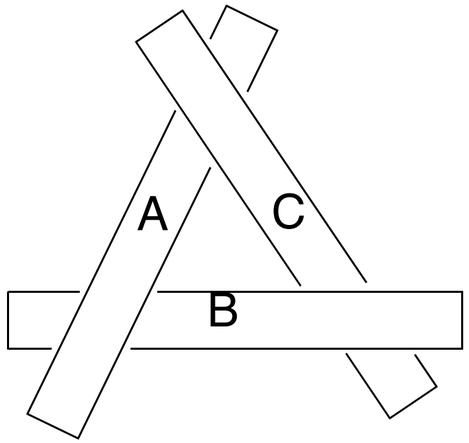
wobei $\alpha$, $\beta$, $\gamma$ = baryzentrische Koord. (können inkrementell oder mit den bekannten Formeln berechnet werden)

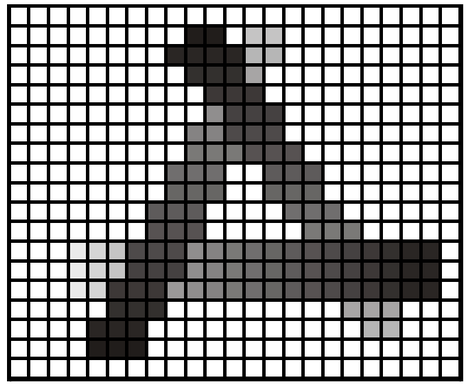


# Pseudo-Code für Rasterisierung mit Z-Buffer

```
for all pixels in window:
  framebuffer[x,y] = background color; zbuffer[x,y] = ∞;
for every triangle:
  compute projection & color at vertices
  precomputations for incremental barycentric coords
  compute bbox, then clip bbox to screen limits
  for all pixels x,y in bbox:
    calc barycentric coords (incrementally)
    compute Z of fragment at pixel (x,y)
    if α, β, γ > 0:                    // pixel is in triangle
      if Z < zBuffer[x,y]:    // pixel is visible
        compute color c of fragment
        framebuffer[x,y] = c
        zBuffer[x,y] = Z
```

Funktioniert auch
in schwierigen Fällen



# Der Z-Buffer in OpenGL (FYI)

1. Fenster mit Z-Buffer anmelden (hier am Bsp. von GLUT)

```
glutInitDisplayMode( GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH );
```

2. Einschalten:

```
glEnable( GL_DEPTH_TEST );
```

3. Wichtig: nicht nur Bildspeicher, sondern auch Z-Buffer löschen!

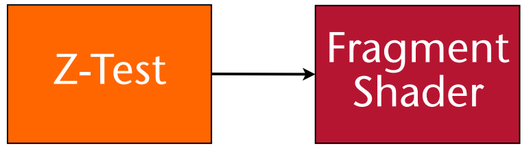
```
glClear( GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT );
```

- Achtung: unter Qt (und anderen GUI-Libs) ist (1) und (2) per Default angeschaltet
  - Mehr Info unter
    http://www.qt-project.org/doc/qt-4.8/QGLFormat.html
  - Beispiel zu QGLFormat im "OpenGL/Qt-Programmbeispiel" auf der Homepage der Vorlesung

# Der Z-Test in der Pipeline

- Wann findet der Z-Test statt?

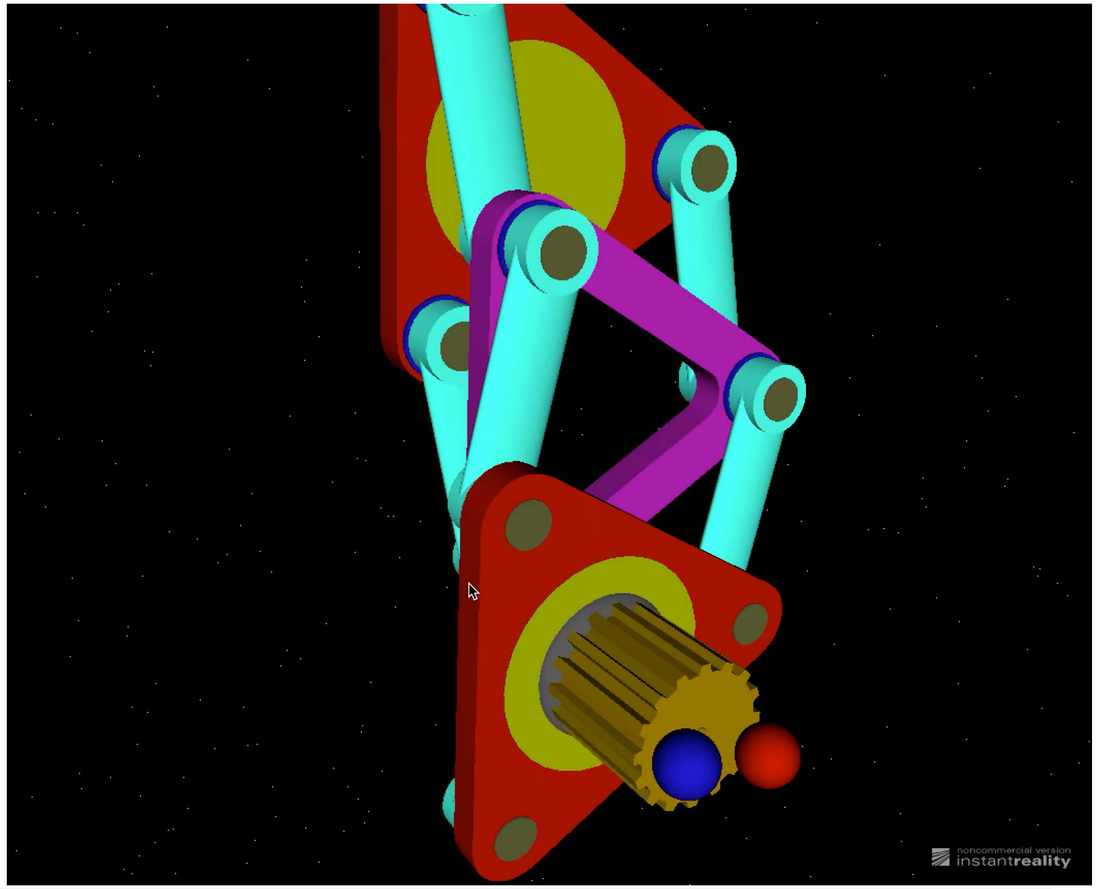

- Early-Z:

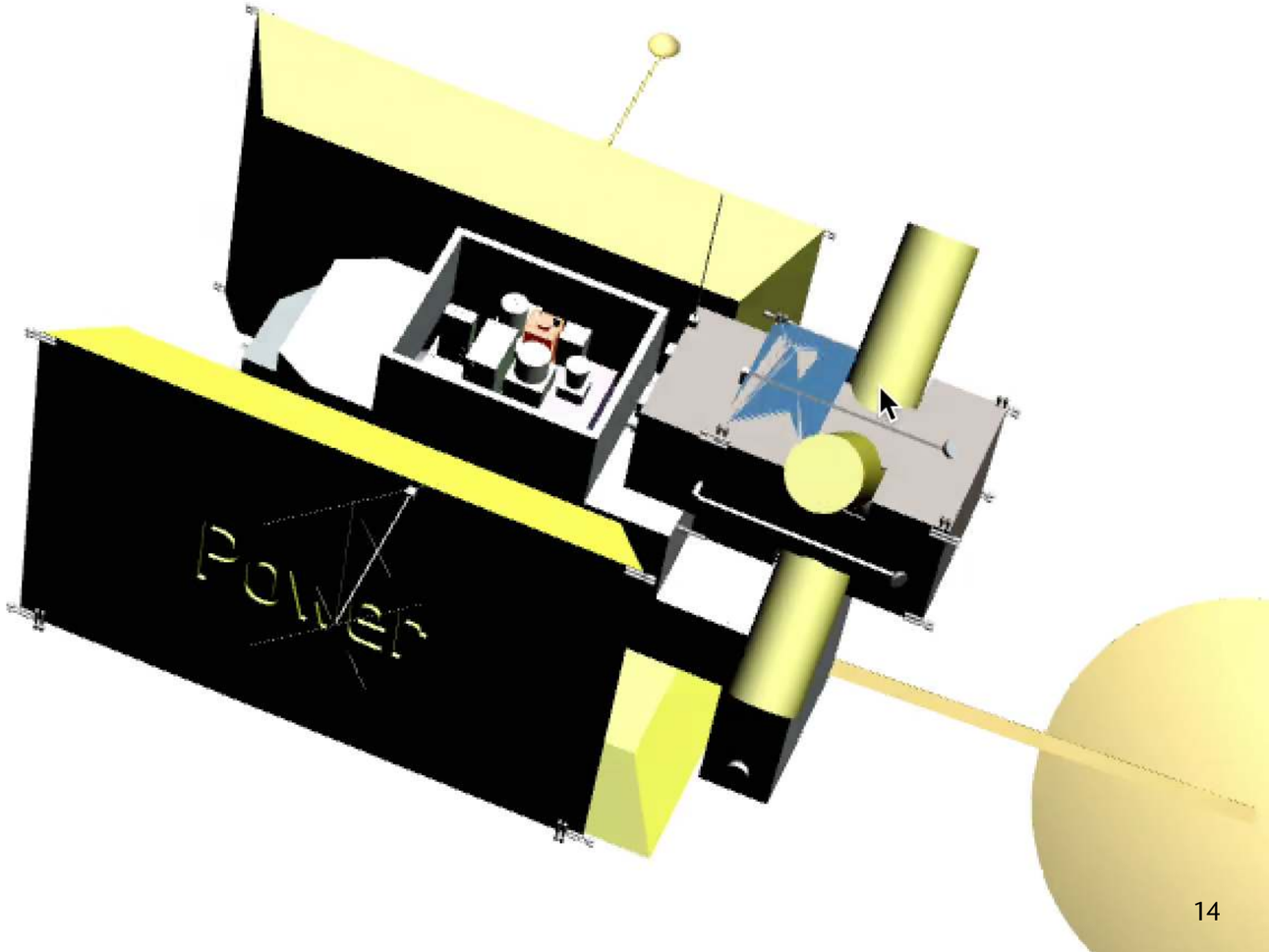



  - Spart teure Fragment-Shader-Programmausführungen
  - Reduziert Bandbreite von der GPU zum Framebuffer, und vom Texturspeicher zur GPU
  - Wird automatisch *deaktiviert*, falls das Shader-Programm den Z-Wert (**`gl_FragDepth`**) manipuliert!

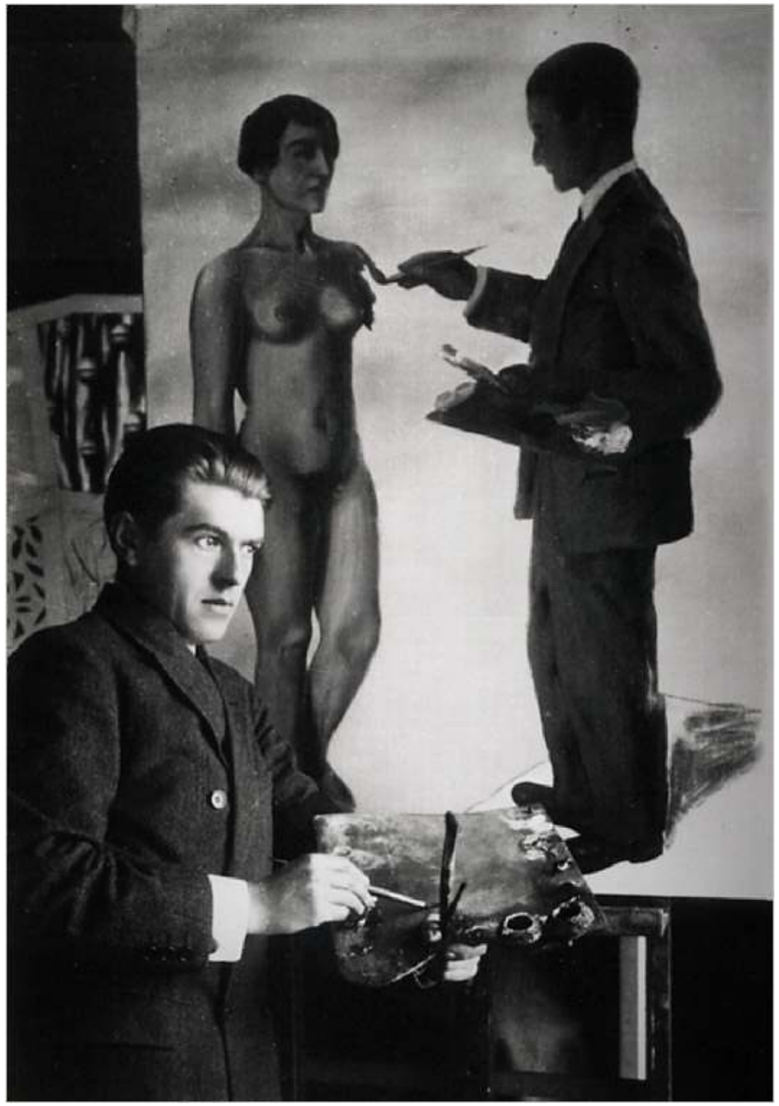
# Z-Fighting

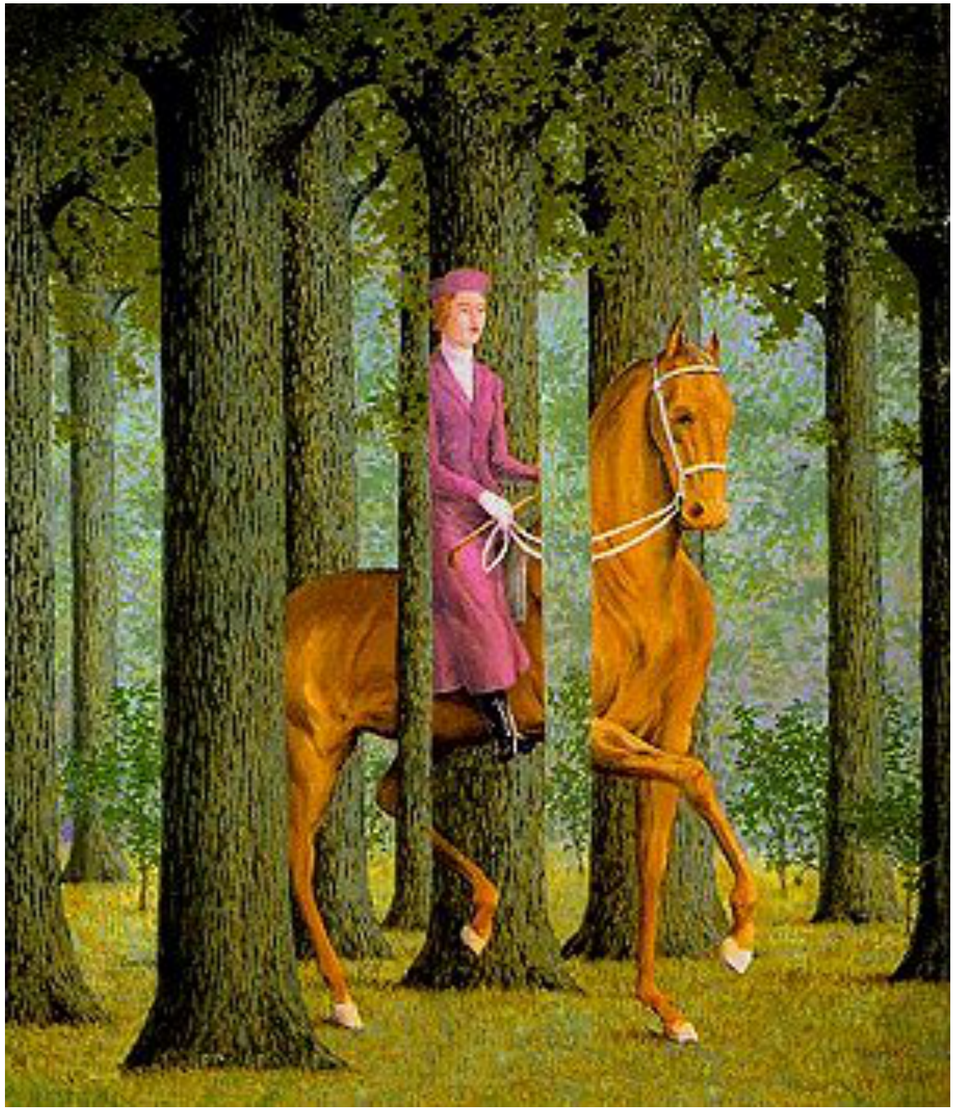
- Wegen der begrenzten Auflösung des Z-Buffers kommt es bei koplanaren (und fast koplanaren) Polygonen zum sog. Z-Fighting bzw. Flickering



# Exkurs: der Surrealist Magritte hat (auch) mit Verdeckung gespielt



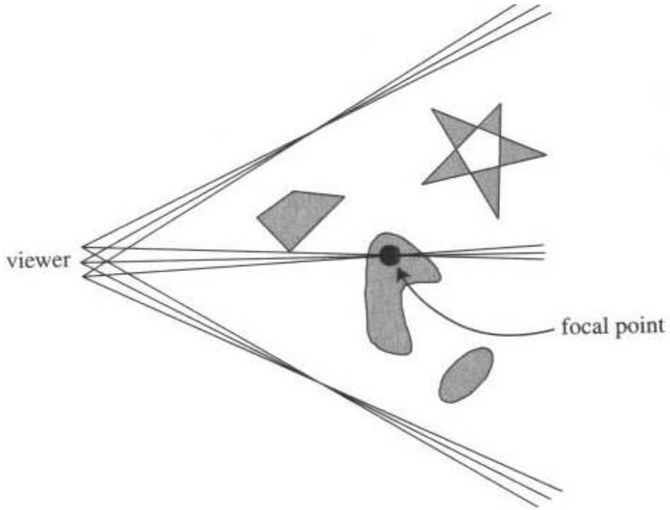


# Bewertung des Z-Buffers zur Verdeckungsrechnung

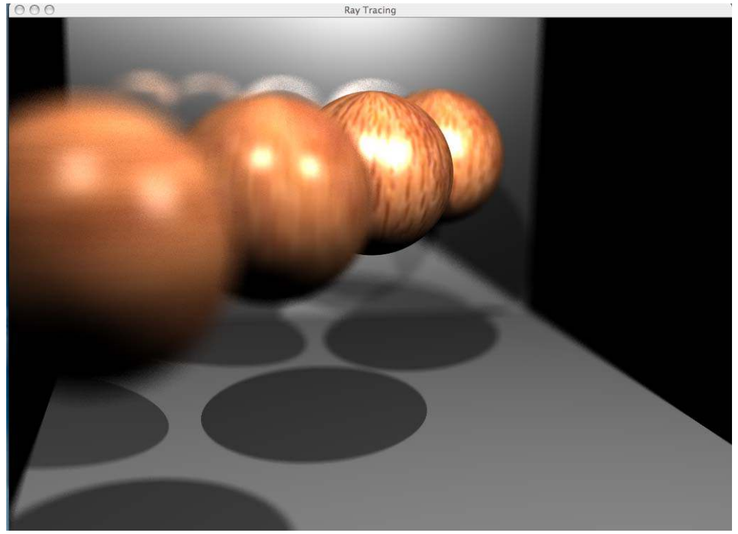
- Komplexität des Algorithmus' $\in O(n)$ , mit $n$ = Anzahl Polygone
  - Kein zusätzlicher Aufwand, z.B. durch Sortieren (z.B. $O(n \log n)$ )
- Eigentlich: $O(n+p)$, wobei $n$ = # Pgone, $p$ = # Fragmente aller Pgone (kann unter Umständen viel größer als Anzahl sichtbarer Pixel sein!)
- Läßt sich ideal in Hardware implementieren:
  - Parallelisierung ohne Kommunikations-Overhead möglich
  - Keine komplizierte "Logik" nötig (wenige **if**'s)
  - Keine komplizierten Datenstrukturen zu traversieren (z.B. verzeigerte Strukturen, z.B. Bäume)
- Nachteile:
  - Pro Pixel kann nur ein Primitiv gespeichert werden
    - Einige fortgeschrittene Effekte, z.B. Transparenz, benötigen aber alle Primitive
  - Genauigkeit des Z-Buffers ist oft stark beschränkt (*image space* vs. *object space*)
    - Auch heute noch manchmal 16-Bit Integer-Werte im Z-Buffer (z.B. in Handys, um Speicherplatz zu sparen

# Anwendung: Depth-of-Field (Tiefenunschärfe)

- Wichtiger depth cue
- Die alte Methode:
  - Rendere die Szene *n* Mal von leicht verschiedenen Viewpoints, je nach Größe der (virtuellen) Blende (*aperture*)

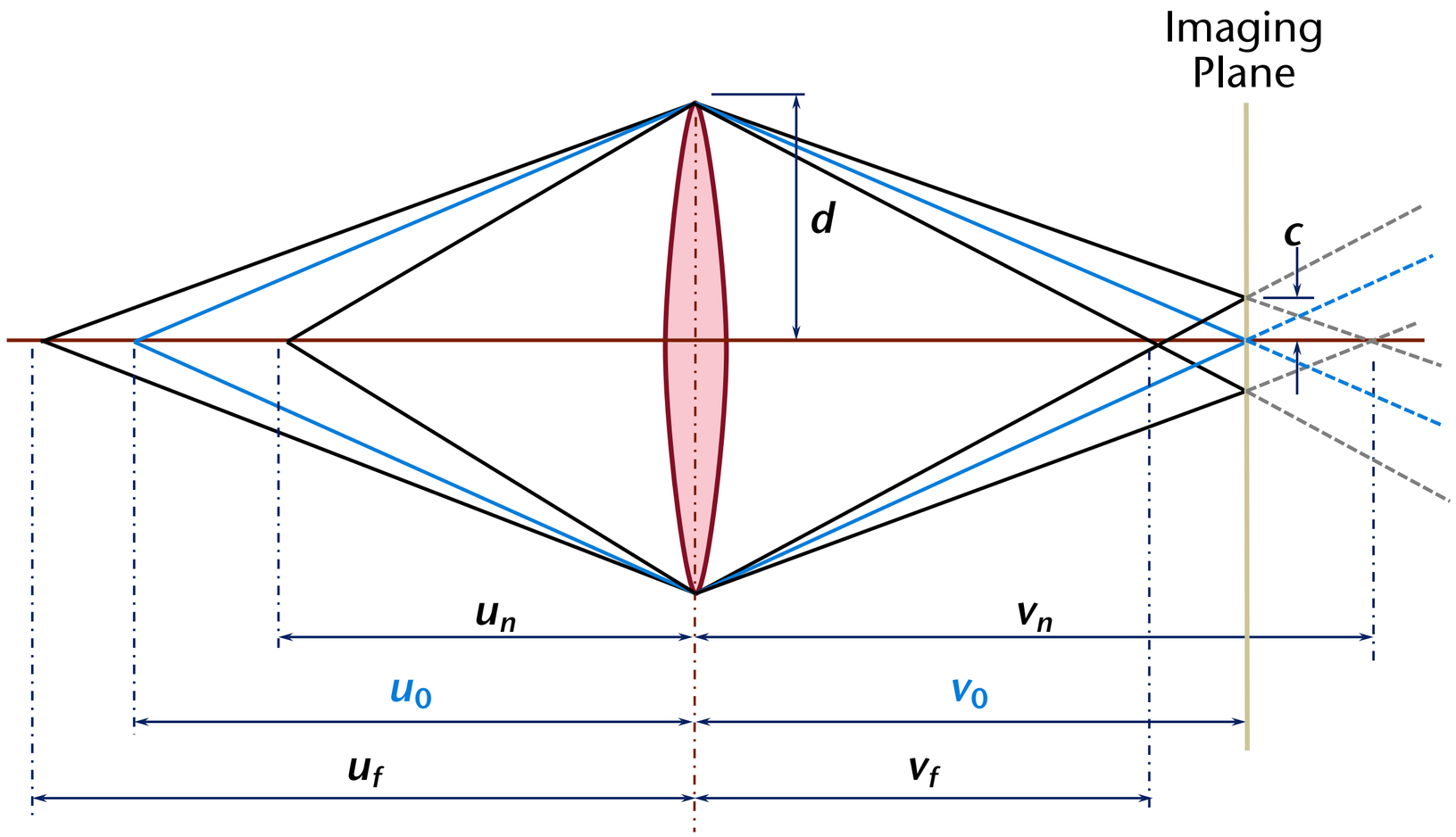



  - Akkumuliere alle Bilder im sog. Accumulation-Buffer
  - Teile am Ende die Werte im Accumulation-Buffer durch *n* ⟶ Mittelwert
- Problem: sehr teuer ⟶ 1/n * FPS



# Der *Circle of Confusion (CoC)*

- Was passiert mit Bildpunkten, die außerhalb der Fokus-Ebene sind:



# Herleitung

- Linsengleichung stellt Zusammenhang zwischen $u$ und $v$ dar:

$$\frac{1}{u} + \frac{1}{v} = \frac{1}{f} \tag{1}$$

- Wende Strahlensatz auf vorige Zeichnung an ("*using similar triangles*"):

$$\frac{v_n - v_0}{v_n} = \frac{c}{d}$$

- Ergibt also den Radius des CoC für einen beliebigen Punkt P:

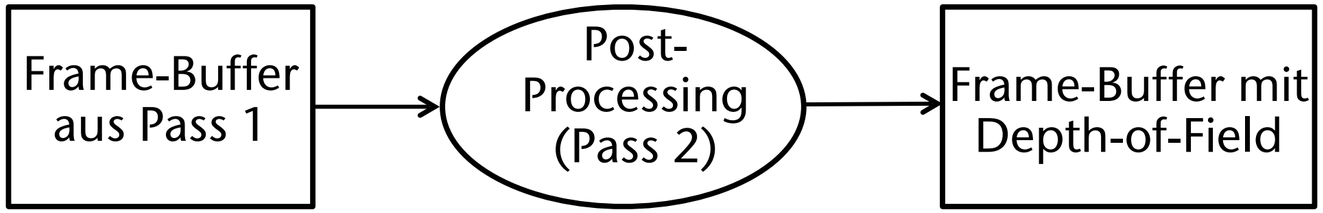
$$c = d \cdot \left| \frac{v_p - v_0}{v_p} \right| \tag{2}$$

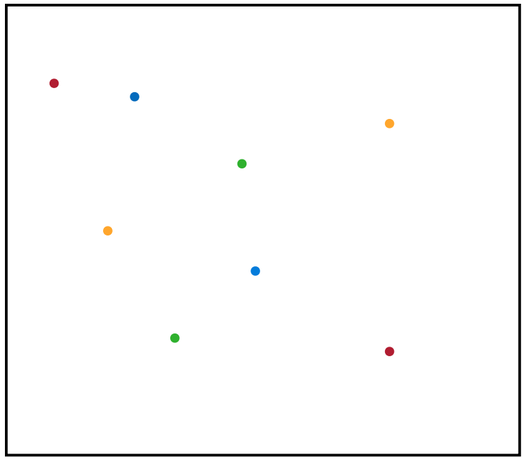
- Linsengleichung (1) einsetzen in (2) ergibt:

$$c = R(u_p) = d \cdot \left| 1 - \frac{u_0}{u_p} \cdot \frac{u_p - f}{u_0 - f} \right| \tag{3}$$

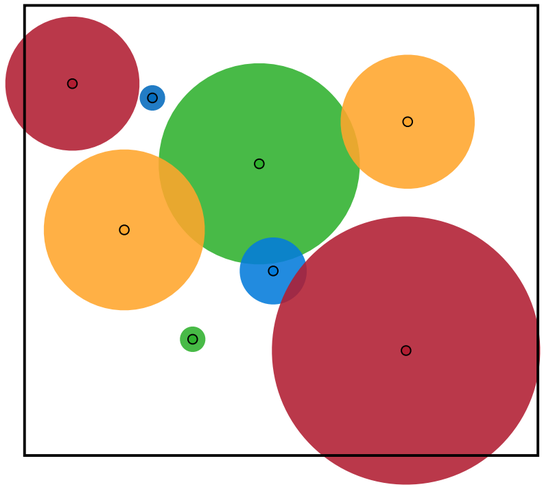
# Der Algorithmus

1. Rendere die Szene normal; speichere zu jedem Pixel dessen z-Wert
2. Post-Processing: für jedes Pixel, "verschmiere" dessen Farbwert auf die Nachbarpixel, abhängig vom Radius des CoC



1.

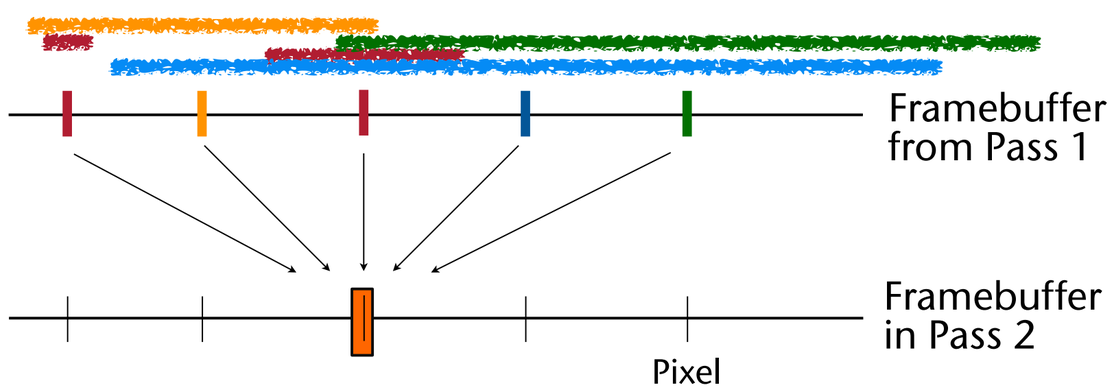


2.

- Problem: die Operation in Pass 2 ist eine sog. *Scatter-Operation* ⟶ sehr teuer bzw. unmöglich (je nach Implementierung auf der GPU)

- Lösung: mache daraus eine sog. Gather-Operation in jedem Pixel

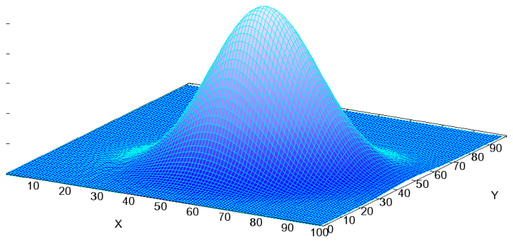
```
foreach pixel (i,j) in framebuffer 2:
  sum = (0,0,0)
  forall pixels (k,l) in m,n-neighborhood:
    r = radius_of_CoC( z_buffer_1[k,l] )
    if r > dist( (i,j), (k,l) ):
      sum += color_buffer_1[k,l]
      n += 1
  color_buffer_2[i,j] = sum / n
```

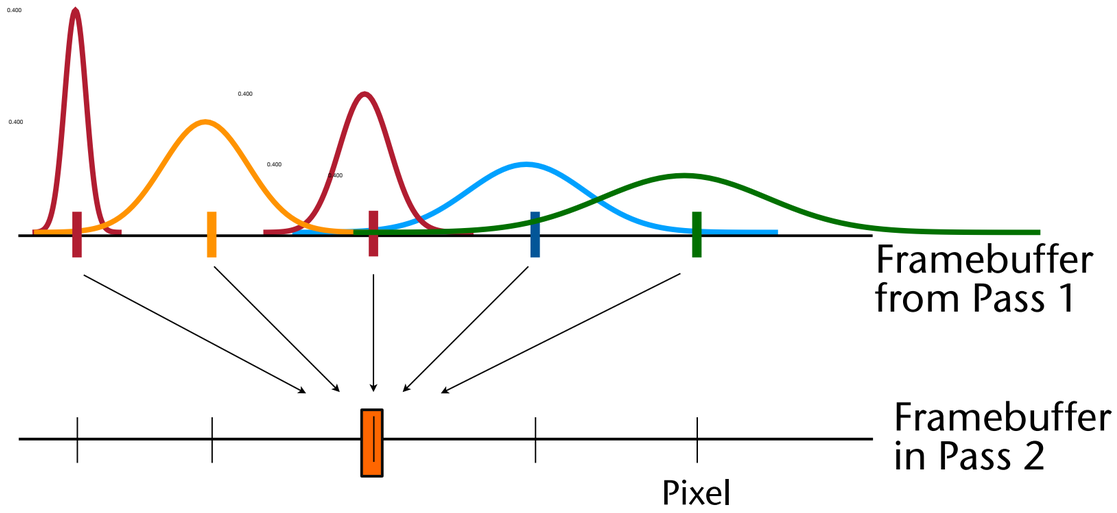



- Wichtig: Nachbarpixel, die weiter als *deren* Circle-of-Confusion entfernt sind, müssen ausgeschlossen werden!

# Etwas realistischere Variante

- Je nach Linse: CoC kann verschiedene "Footprints" haben
- Einfache Variante:
  - Approximiere Footprint durch Gauß-Funktion
  - Überlagere diese im jeweiligen "Gathering Pixel"
  - Sehr ähnlich zur Faltung (*convolution*)
  - Hier: Breite der Gauß-Funktion hängt von $z - z_0$ ab, $z$ = Tiefe des jeweiligen "Scattering Pixels"



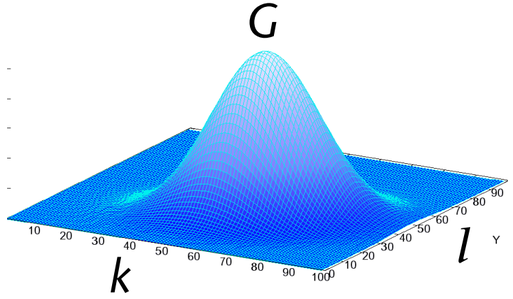



# Der verbesserte Algorithmus

- Normale Faltung: Input-Bild $I$, Output-Bild $O$, Kernel $G$

$$O(x,y) = \sum_{k=-\frac{r}{2}}^{\frac{r}{2}} \sum_{l=-\frac{r}{2}}^{\frac{r}{2}} I(x+k, y+l) \cdot G(k,l)$$

$$G(k,l) = \frac{1}{2\pi r^2} e^{-\frac{k^2+l^2}{2r^2}}$$



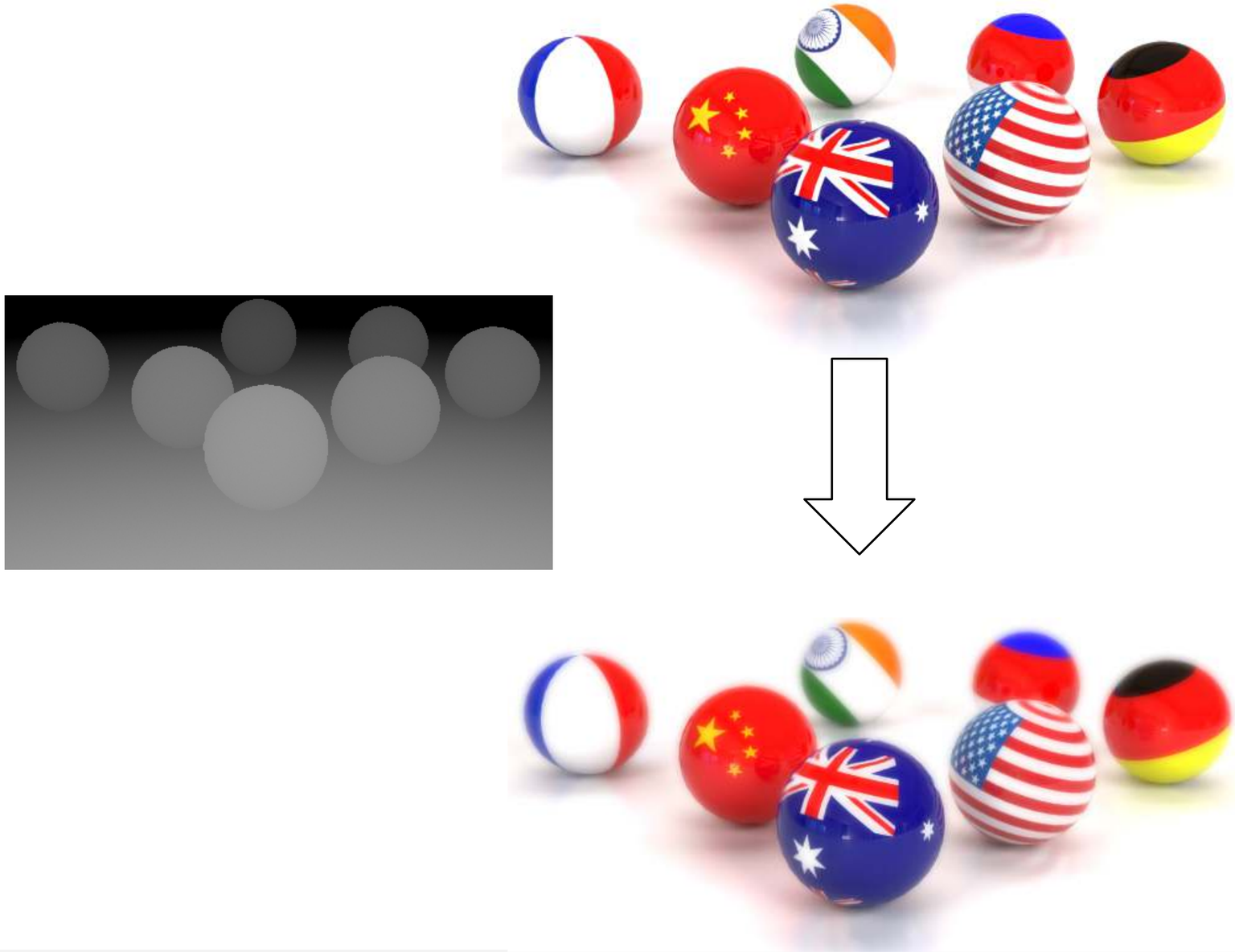

- Bei uns: die "Breite" von $G$ (Parameter $r$) muss vom Radius des CoC an der Stelle $(x+i, y+j)$ abhängen!

$$G = G(k,l,x,y) = \frac{1}{2\pi r^2} e^{-\frac{k^2+l^2}{2r^2}} \quad \text{mit} \quad r = R(\,\text{zbuffer}[x+k, y+l]\,)$$

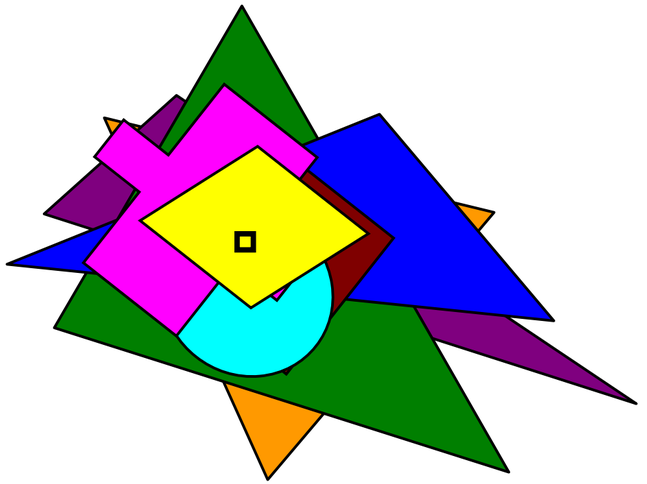
- Damit wird der Algorithmus zu:

```
foreach pixel (i,j) in framebuffer 2:
  sum_c = (0,0,0)
  sum_g = 0
  for k=-m,..,+m; l=-n,..+n:
    r = radius_of_CoC( z_buffer_1[i+k,j+l] )
    g = kernel_weight( k, l, r )
    sum_c += g * color_buffer_1[i+k,j+l]
    sum_g += g
  color_buffer_2[i,j] = sum_c / sum_g
```

# Beispiel



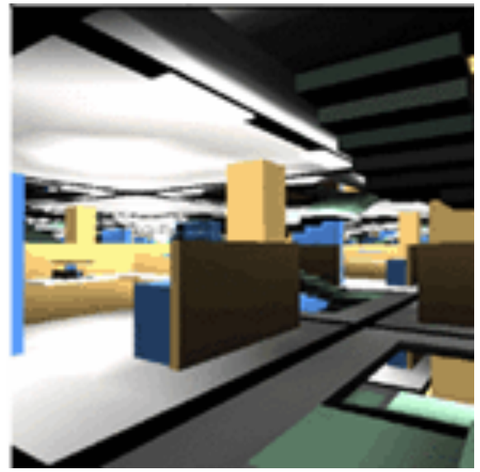
# Depth-Complexity und Overdraw

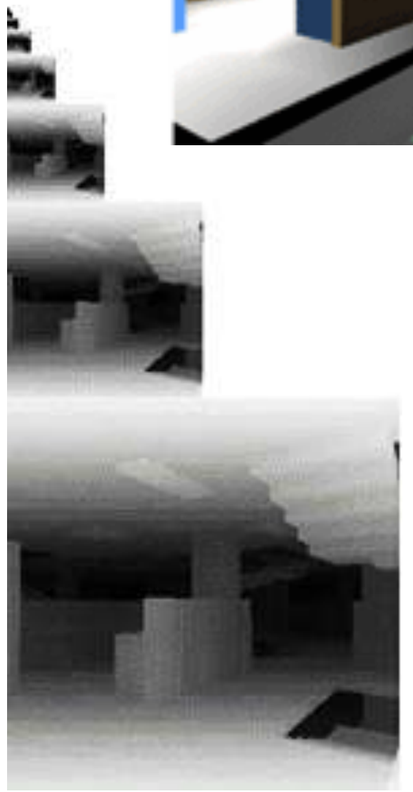


- Frage: 10 Dreiecke überdecken ein Pixel — wie groß ist die *erwartete* Anzahl Schreib-Operationen in den Color-Buffer ?
- Lösung: $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{10} \approx \ln(10) + 0.55... = 2.9289\ldots$
  - Hinweis: harmonische Reihe!
- Definition Depth-Complexity:
  - Anzahl Polygone der Szene, die "hinter" einem Pixel liegen
  - Manchmal auch diese Definition: Depth-Complexity = Anzahl Z-Tests pro Pixel
- Definition Over-Draw: Maß dafür, wie oft ein Pixel im FB überschrieben wird
- Fazit: $\text{overdraw} \approx \ln(\text{depth complexity}) + 0.5$

# Der Hierarchische Z-Buffer (HZB)

- Frage: wie kann man auch bei großer Depth-Complexity den Overdraw gering halten? (auch im worst case)
- Idee: "Z-Pyramide"
  - Einfacher Z-Buffer = höchste Auflösung
  - Weitere Levels durch Zusammenfassen von je $n \times n$ Pixel
  - Z-Wert jeweils auf max{ $n^2$ Z-Werte } setzen





| | | | |
|---|---|---|---|
| 7 | 1 | 0 | 6 |
| 0 | 3 | 1 | 2 |
| 3 | 9 | 1 | 2 |
| 9 | 1 | 2 | 2 |

farthest value

| | |
|---|---|
| 7 | 6 |
| 9 | 2 |

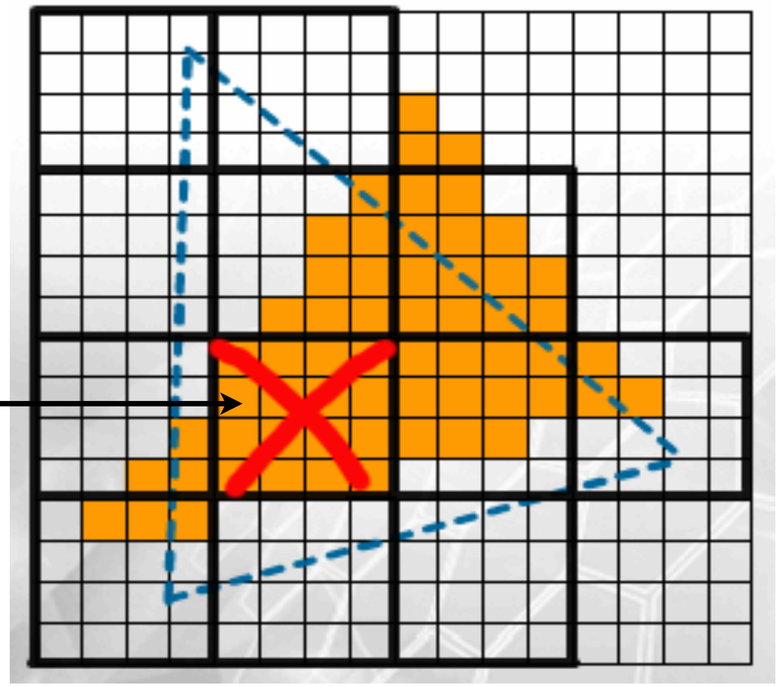
farthest value

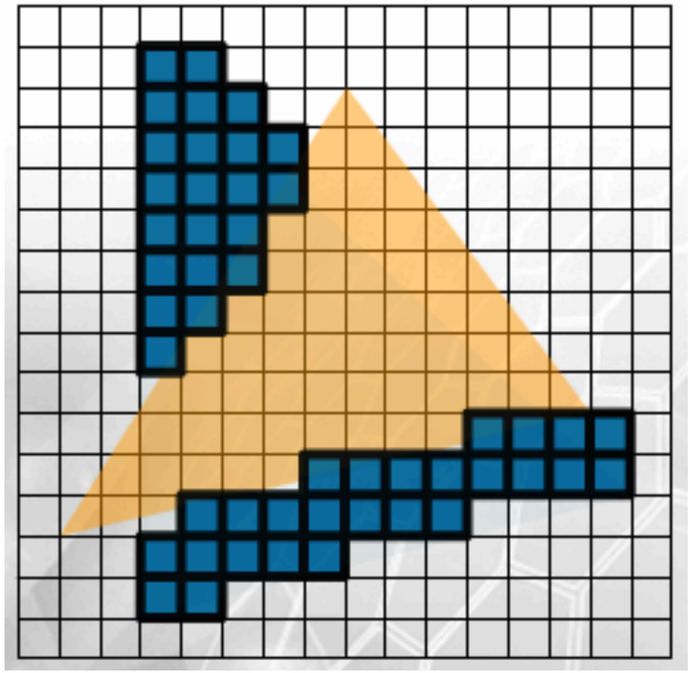
| |
|---|
| 9 |

# Beispiel mit 2 Levels

- Sei orangenes Dreieck bereits gezeichnet
- Blaues Dreieck soll (dahinter) gezeichnet werden
- Rasterisiere Dreieck erst auf grobem Level, dann ggf. innerhalb der "Kacheln"

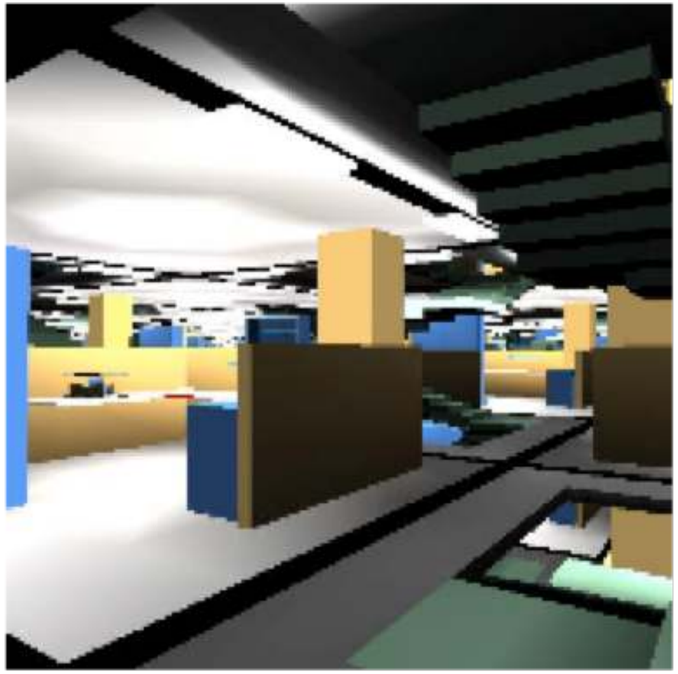
Kachel ist vollständig verdeckt, also verwerfe gesamte Kachel





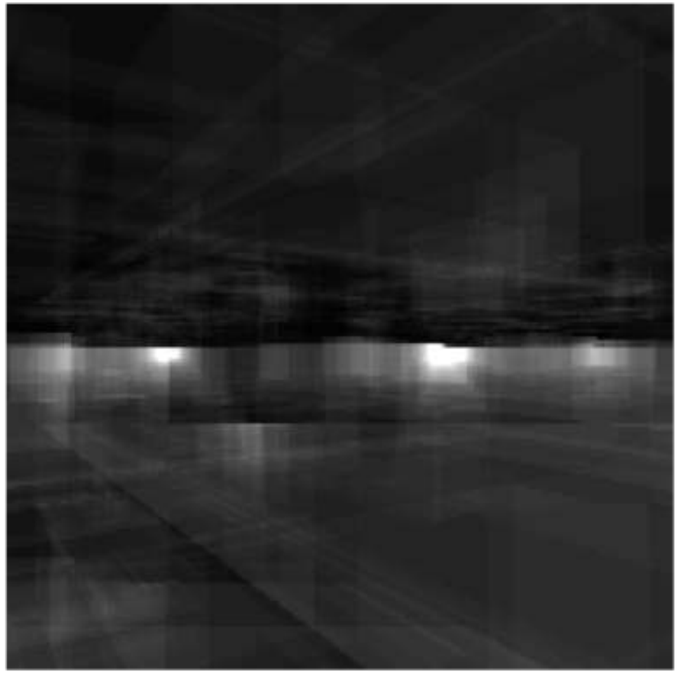
# Vergleich

540 Mio Polygone



Overdraw mit
einfachem Z-Buffer

Overdraw mit HZB



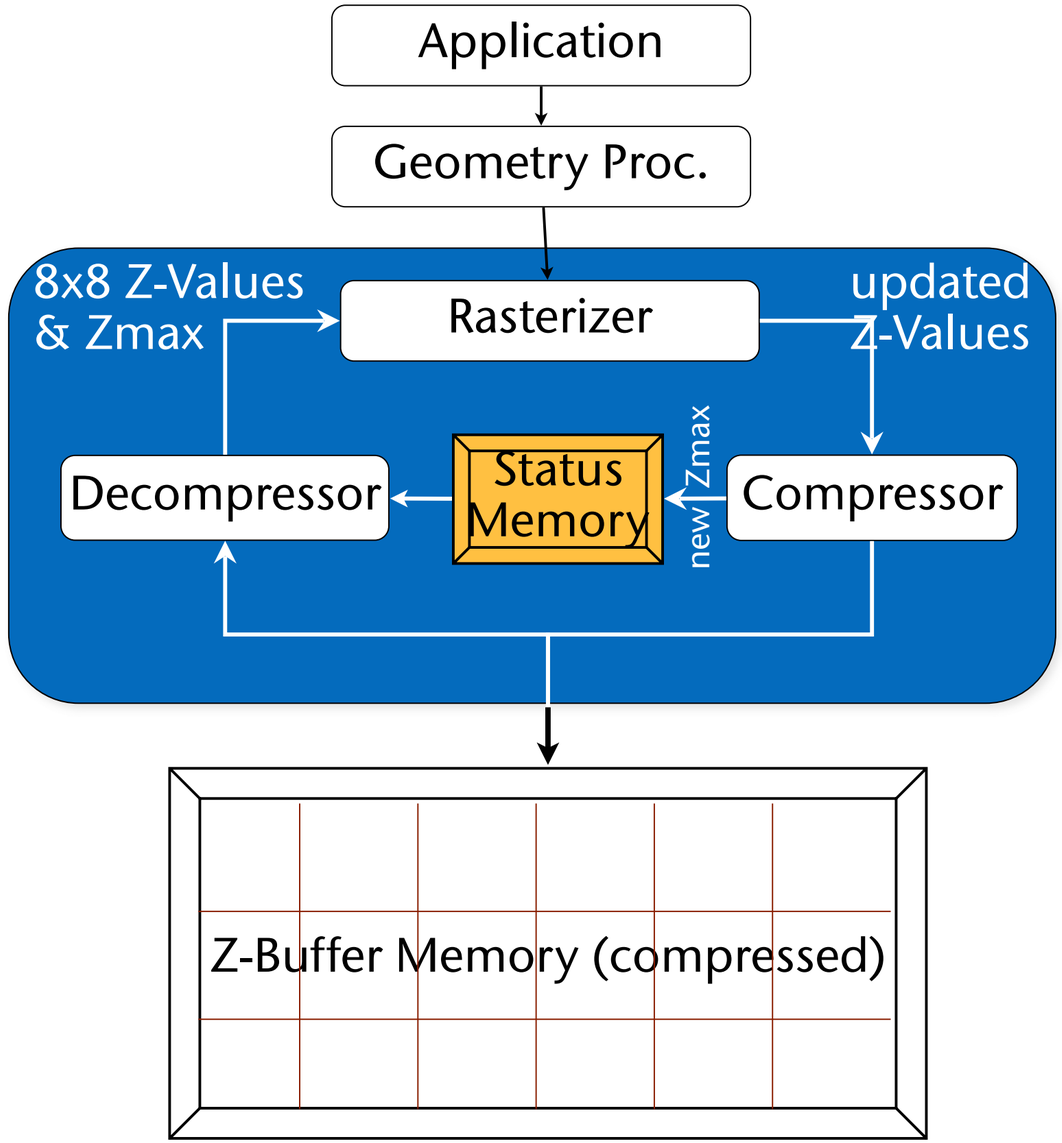
# Implementierung in aktueller Graphik-Hardware

- Problem: Bandbreite zwischen Rasterizer und Speicher ist limitiert
  - Annahmen: Auflösung 1280x1024, für jedes Pixel ist depth complexity = 4
  - I/O-Aufwand pro Fragment: 1x Z-Buffer-Read + 1x Z-Buffer-Write + 1x Color-Buffer-Write + 2x Texture-Read, pro Read/Write 32 Bit

  ➢Ergibt ca. 18 GByte/sec!
- Wie implementiert man schnell `glClear(DEPTH_BUFFER_BIT)`?
- Wie implementiert man den HZB?
- Lösungsansatz:
  - Nur 2 Levels des HZB implementieren → Z-Buffer in Kacheln aufteilen
  - Kacheln (tiles) komprimieren

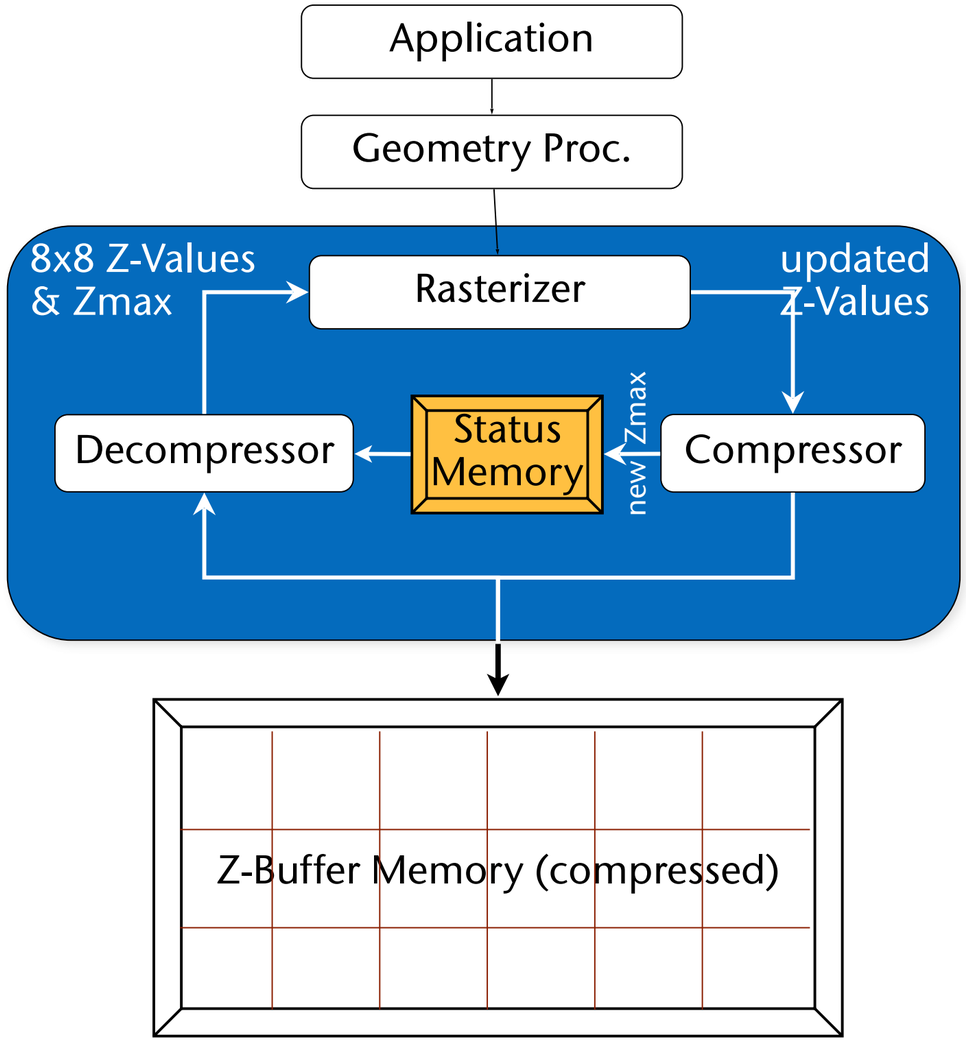
# FYI

- Zentrale Idee: Status-Speicher auf dem Chip (sehr schnell) für den Zustand der Kacheln
- Pro Kachel speichere im Status-Memory:
  - Zustand ∈ {*"compressed", "uncompressed", "cleared"*}
  - $Z_{max}$ der Kachel



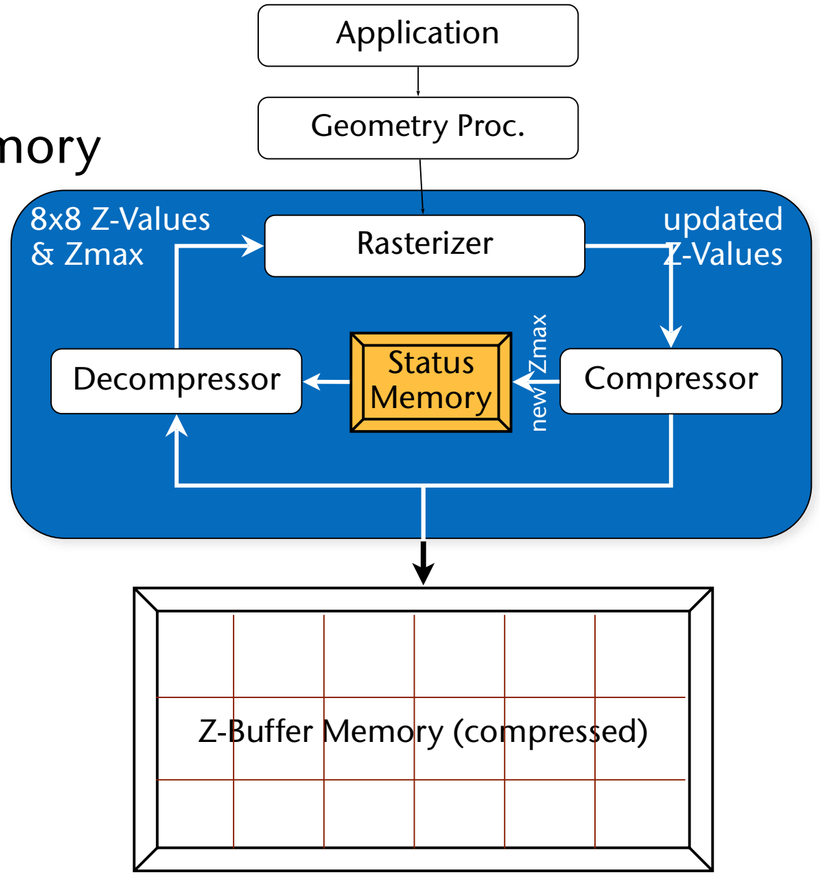
# FYI

- Z-Buffer löschen:
  - Bei **`glClear()`** wird der Status jeder Kachel auf "cleared" gesetzt
  - Beim Lesen einer Kachel: Decompressor checkt Status, sieht "cleared", schickt $Z_{far}$ an Rasterizer
  - Kein Datenfluß auf dem Bus
- Z-Buffer schreiben:
  - Compressor berechnet neues $Z_{max}$ der Kachel und schreibt es in Status Memory
  - Komprimiert aktuelle Z-Werte der Kachel
  - Falls signifikante Kompression möglich: setze Status auf "compressed", sonst "uncompressed"
  - Schreibe un-/komprimierte Kachel in Z-Buffer



# FYI

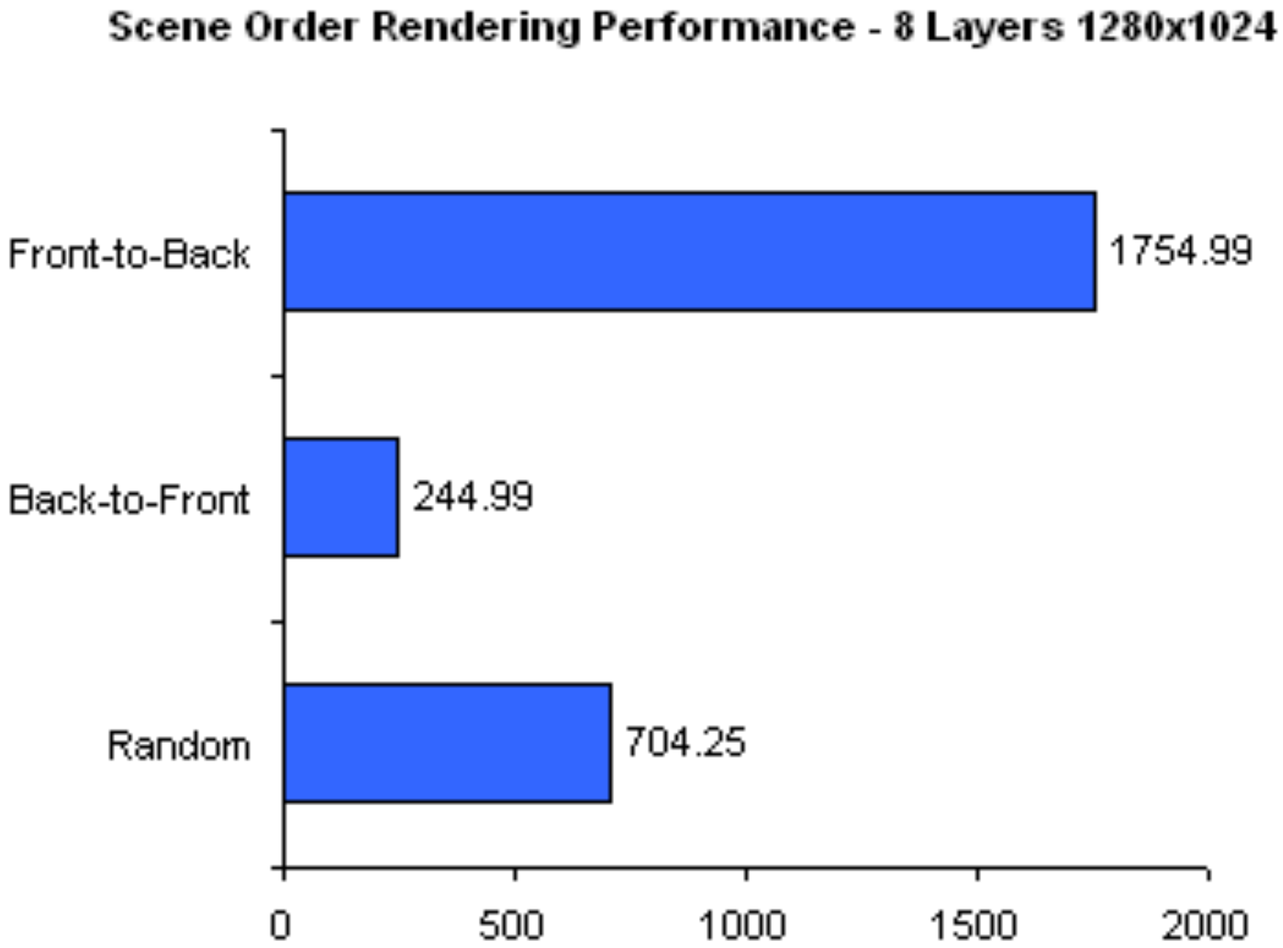
- Z-Buffer lesen:
  - Decompressor liest zuerst $Z_{max}$ der Kachel aus dem Status Memory
  - Verschiedene Tests möglich:
    1. Teste die Z-Werte der 3 Vertices des Dreiecks gegen $Z_{max}$
    2. Teste die Z-Werte des Δ's an den 4 Ecken der Kachel gegen $Z_{max}$
       - Bemerkung: diese 4 Z-Werte kann man in den Nachbarkacheln wiederverwenden
    3. Berechne alle Z-Werte der Fragmente in Kachel, teste gegen $Z_{max}$
  - Fordere Z-Werte der Kachel aus dem Z-Buffer an, falls alle Tests "fehlschlagen"; ggf. dekomprimieren




- Kompression (z.B. DPCM) erreicht bis zu Faktor 4:1
- Nennt sich *"HyperZ"* oder "*Lightspeed Memory Architecture*" bei den Graphikkartenherstellern

# Performance-Gewinn in einer Graphikkarte

FYI

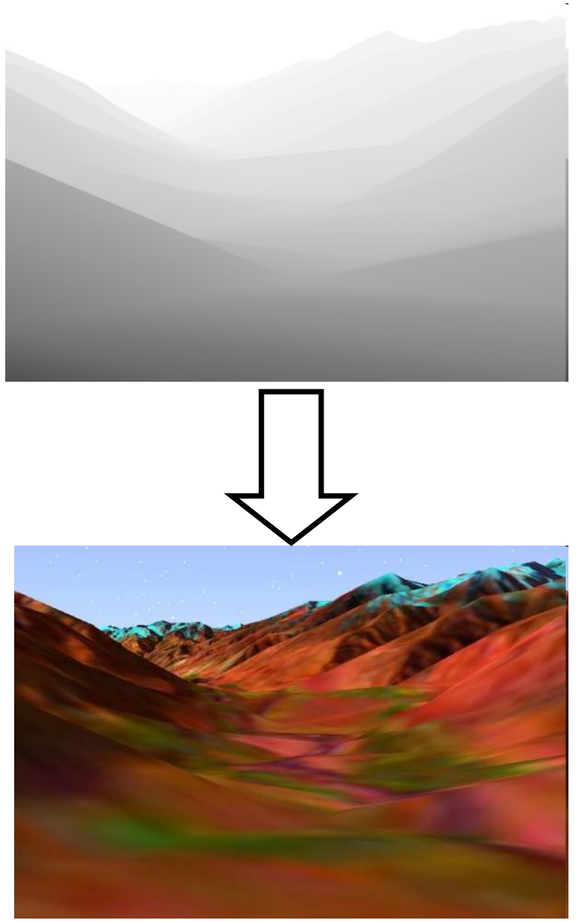


# Einfacher Rendering-Performance-Trick "*Early-Z Pass*"

- Wegen Komprimierung des Z-Buffers: Durchsatz der GPU ist ca. *doppelt* so hoch, falls **nur** der Z-Buffer geschrieben wird (nicht Color-Buffer)
- Vorgehensweise:
  - Schalte Color-Buffer aus, nur Z-Buffer an
  - Pass 1: rendere Szene "ohne alles"
    (keine Lichtquellen, keine Farben, keine Texturen),
    schreibe nur Z-Buffer = *"lay down depth"*
  - Pass 2: rendere Szene "mit allem",

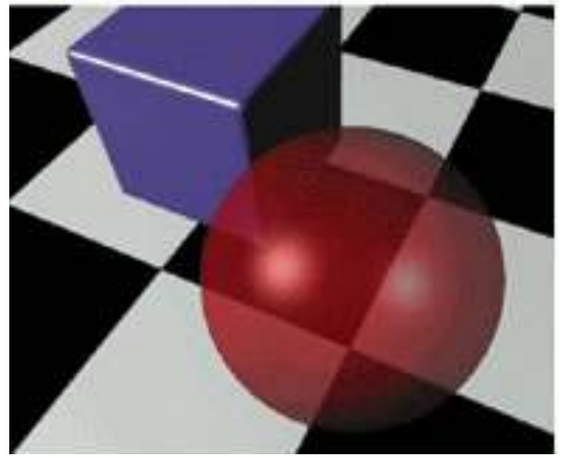
    lösche *nicht* den Z-Buffer ⟶ HZB kann voll wirken

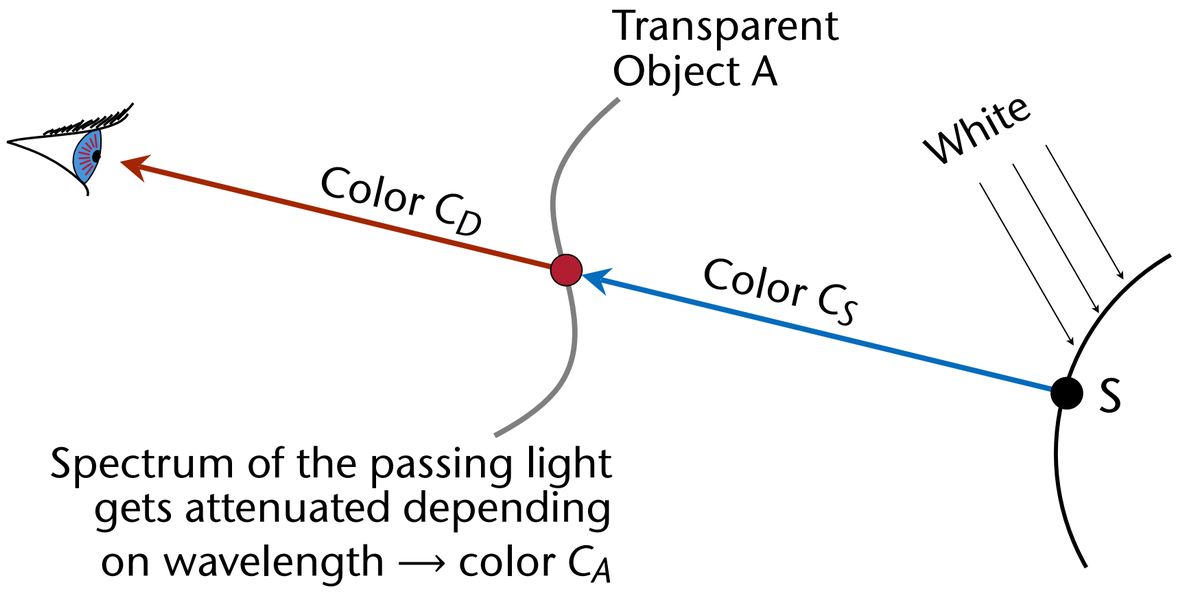
    ⟶ *kein* Overdraw



# Rendering Transparent Objects

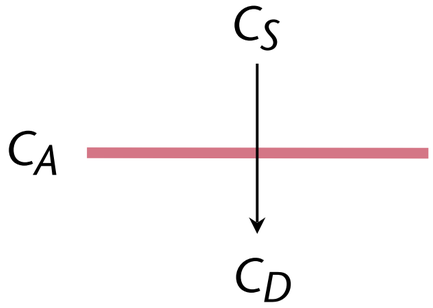
- For example: objects made of glass
- Transparency = material that lets light pass partially
- Often, some wavelengths are attenuated more than others ⟶ colored transparency
  - Extreme case: color filter in photography
- Model:





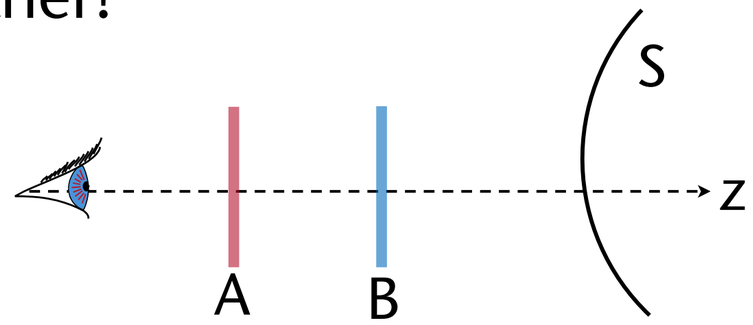
- Approximation: alpha blending
  - $\alpha \in [0, 1]$= opacity (= opposite of transparency)
    - $\alpha = 0 \longrightarrow$ completely transparent, $\alpha = 1 \longrightarrow$ completely opaque
  - Outgoing color:

$$C_D = \alpha C_A + (1 - \alpha) C_S$$




  - Practical implementation: $\alpha$ = 4th component in color vectors C = (r, g, b, $\alpha$)
- During rendering, the graphics card performs these fragment operations:
  1. Read color from frame buffer → $C_S$
  2. Compute $C_D$ by above equation
  3. Write $C_D$ into framebuffer
- For later: "transparent color" $C_A$ of the object = transmission spectrum (similar to reflectance spectrum of opaque objects)

- Problem: several transparent objects behind each other!
  - Assume Z-buffer is off, i.e., no Z-test is done




1. First *A* then *B* results in:

$$C'_D = \alpha_A C_A + (1 - \alpha_A)\, C_S$$
$$C_D = \alpha_B C_B + (1 - \alpha_B)\, C'_D$$
$$= \alpha_B C_B + (1 - \alpha_B)\, \alpha_A C_A + (1 - \alpha_B)\,(1 - \alpha_A)\, C_S$$

2. First *B* then *A* results in:

$$C'_D = \alpha_B C_B + (1 - \alpha_B)\, C_S$$
$$C_D = (1 - \alpha_A)\, \alpha_B C_B + \alpha_A C_A + (1 - \alpha_B)\,(1 - \alpha_A)\, C_S$$

- Conclusion: you must render transparent polygons/particles from back to front! and the Z-buffer can be switched off

# Benutzung in OpenGL

- Switch blending on:

```
glEnable( GL_BLEND );
```

- Determine blending function:

```
glBlendFrame( Glenum s, Glenum d );
```

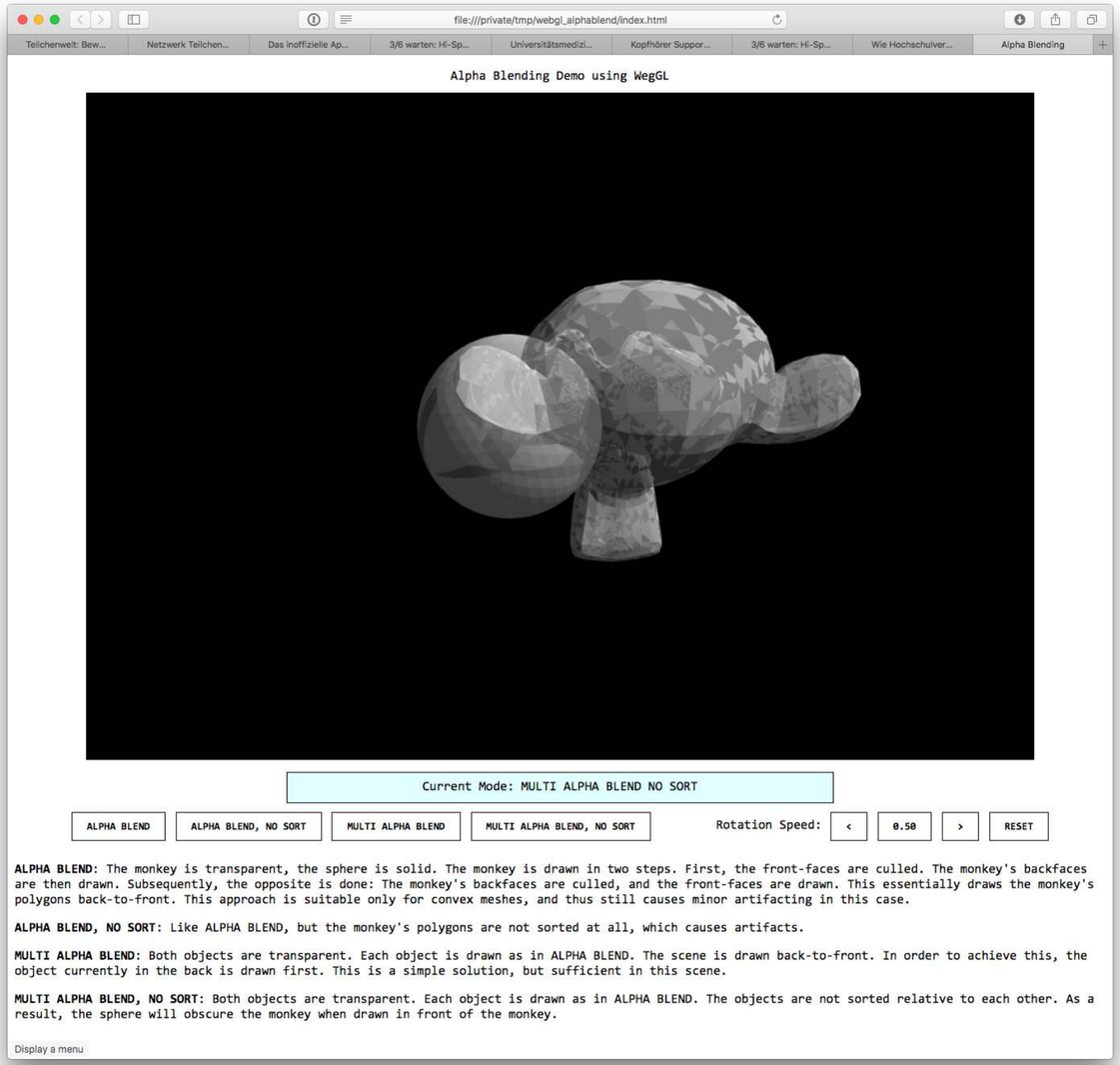
Parameters **GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA** yield

$$C'_D = \alpha C_A + (1 - \alpha)\, C_D$$

where $C_D$ = color from frame buffer

- There are many more variants, e.g., you can just accumulate colors using (**GL_ONE, GL_ONE**)

# Demo


file:///private/tmp/webgl_alphablend/index.html
Alpha Blending
Alpha Blending Demo using WegGL
Current Mode: MULTI ALPHA BLEND NO SORT
ALPHA BLEND
ALPHA BLEND, NO SORT
MULTI ALPHA BLEND
MULTI ALPHA BLEND, NO SORT
Rotation Speed:
<
0.50
>
RESET
ALPHA BLEND: The monkey is transparent, the sphere is solid. The monkey is drawn in two steps. First, the front-faces are culled. The monkey's backfaces are then drawn. Subsequently, the opposite is done: The monkey's backfaces are culled, and the front-faces are drawn. This essentially draws the monkey's polygons back-to-front. This approach is suitable only for convex meshes, and thus still causes minor artifacting in this case.
ALPHA BLEND, NO SORT: Like ALPHA BLEND, but the monkey's polygons are not sorted at all, which causes artifacts.
MULTI ALPHA BLEND: Both objects are transparent. Each object is drawn as in ALPHA BLEND. The scene is drawn back-to-front. In order to achieve this, the object currently in the back is drawn first. This is a simple solution, but sufficient in this scene.
MULTI ALPHA BLEND, NO SORT: Both objects are transparent. Each object is drawn as in ALPHA BLEND. The objects are not sorted relative to each other. As a result, the sphere will obscure the monkey when drawn in front of the monkey.
Display a menu

# Ein "Stencil" im echten Leben

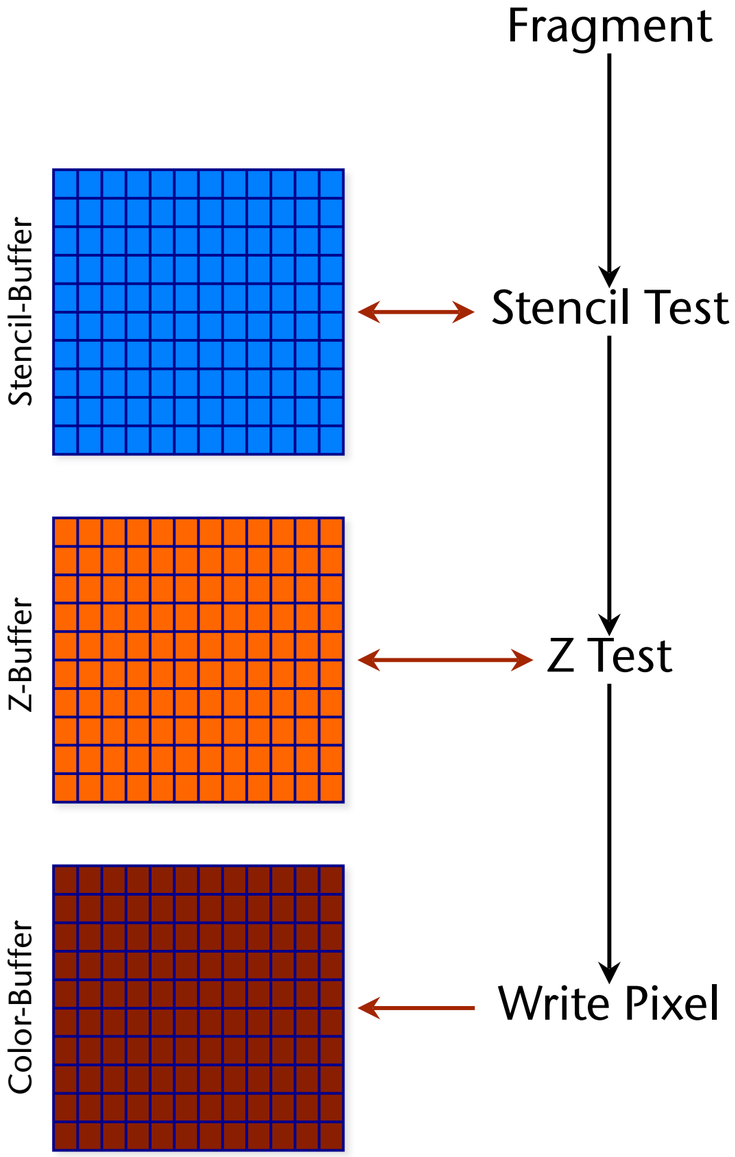
# Der Stencil Buffer

- Der Stencil-Buffer ist eine Art "Vergleichs-Buffer"
  - Ähnlich zu Z-Buffer (*test & pass/kill*) , aber mit anderen Features
- Die zwei Operationen bei eingeschaltetem Stencil-Buffer:

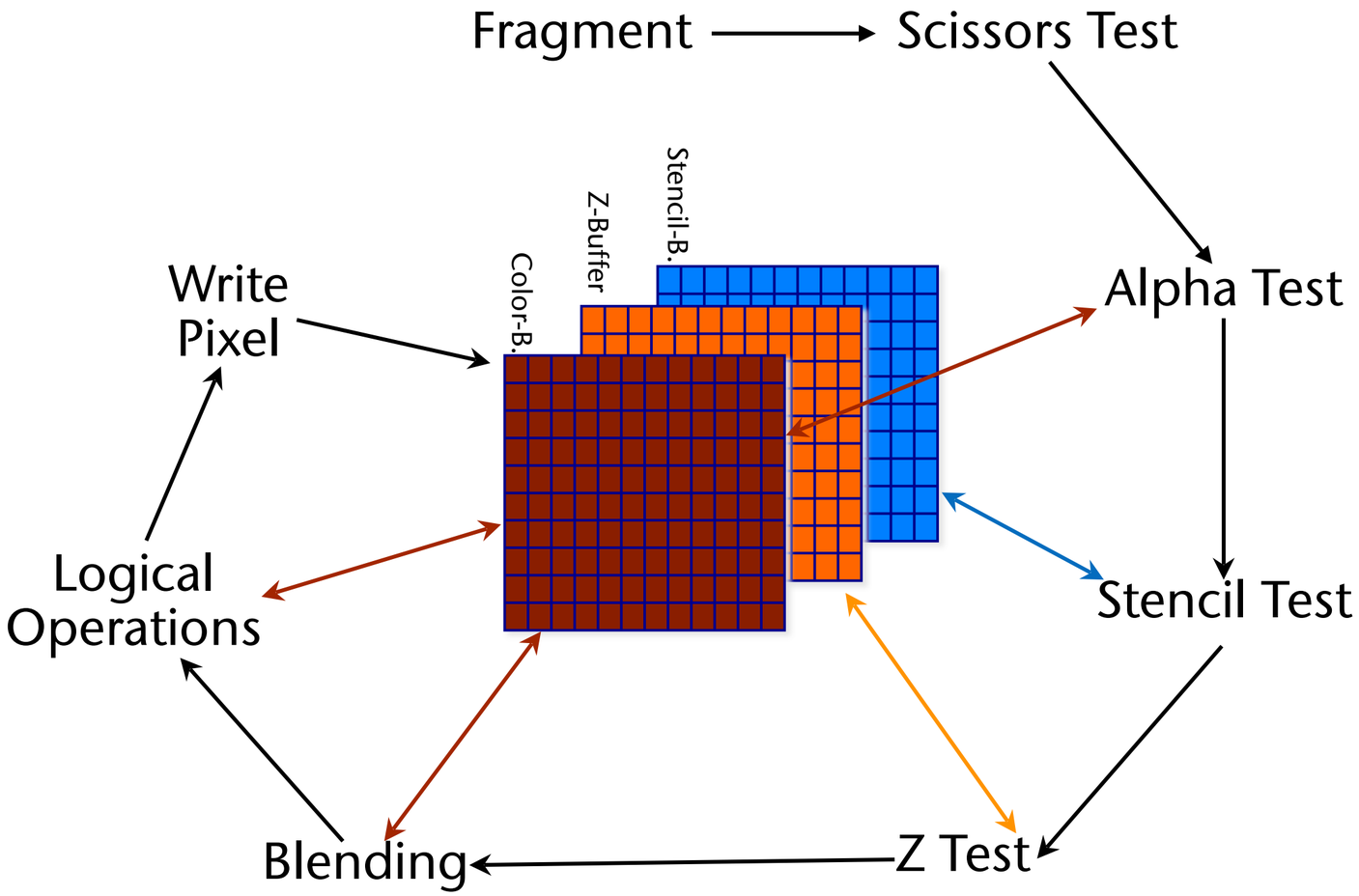
1. **`glStencilFunc(GLenum func, GLint ref, GLuint mask)`**: der *Stencil-Test,* legt fest, ob in den *Color-Buffer* geschrieben wird
   - Form des Tests: **`s`** ***`func`*** **`ref`** .
   - *s* = aktueller Wert im Stencil-Buffer an der Pixelstelle, *ref* = ein Referenzwert, *mask* = Bit-Maske
   - Mögliche Operationen für ***`func`*** : GL_LESS, GL_GREATER, GL_EQUAL, etc.
2. **`glStencilOp(GLenum sfail, GLenum zfail, GLenum zpass)`**: die *Stencil-Operation,* legt für jeden der 3 Fälle fest, welche Operation auf den Wert im *Stencil-Buffer* ausgeführt wird
   - Mögliche Operationen: GL_ZERO = Stencil löschen, GL_INCR = gespeicherten Stencil-Wert erhöhen, GL_DECR = gespeicherten Stencil-Wert erniedrigen, u.a. ...

# Die Reihenfolge der Tests

```
doStencilAndZTest( int x, int y, int z, color f ):
  // stencil test
  check stencilbuf[x,y] against reference value
  if stencil test failed:
    perform sfail operation on stencilbuf[x,y]
  else:
    // z-test
    if z > zbuffer[x,y]:
      perform zfail operation on stencilbuf[x,y]
    else:
      colorbuf[x,y] = f
      perform zpass operation on stencilbuf[x,y]
```

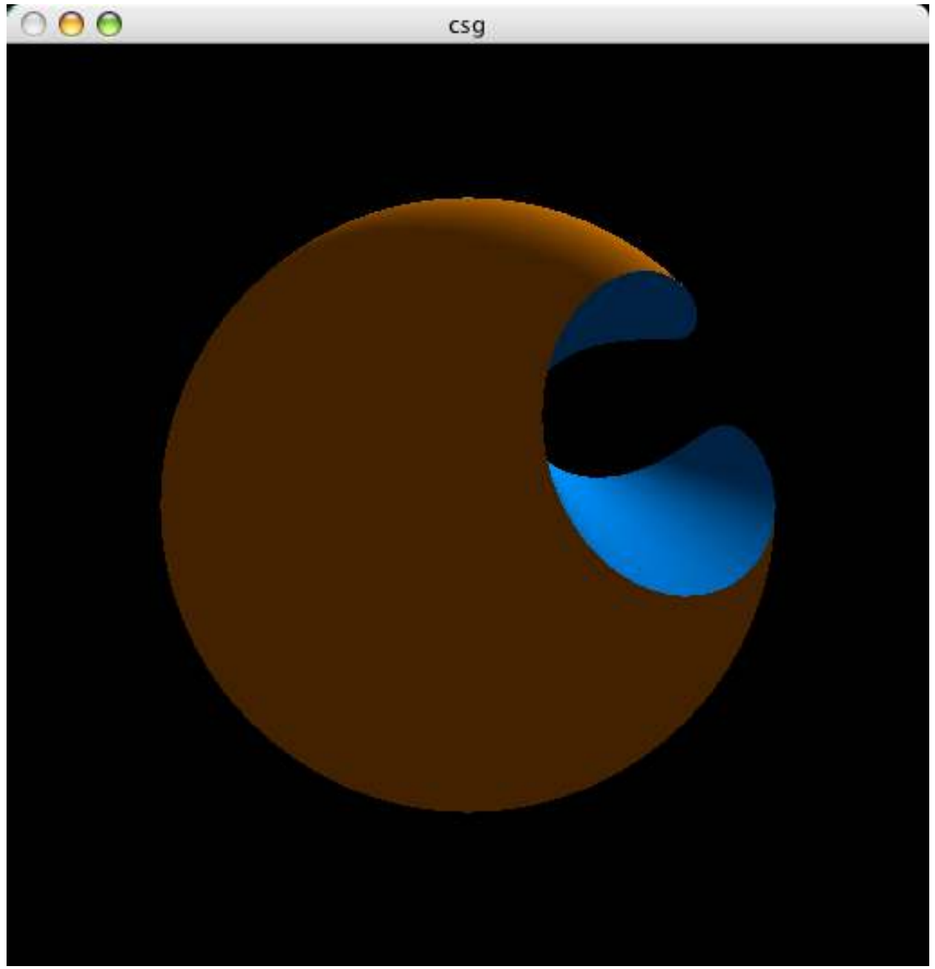


# Die volle Abfolge von Tests und Operationen in der GPU

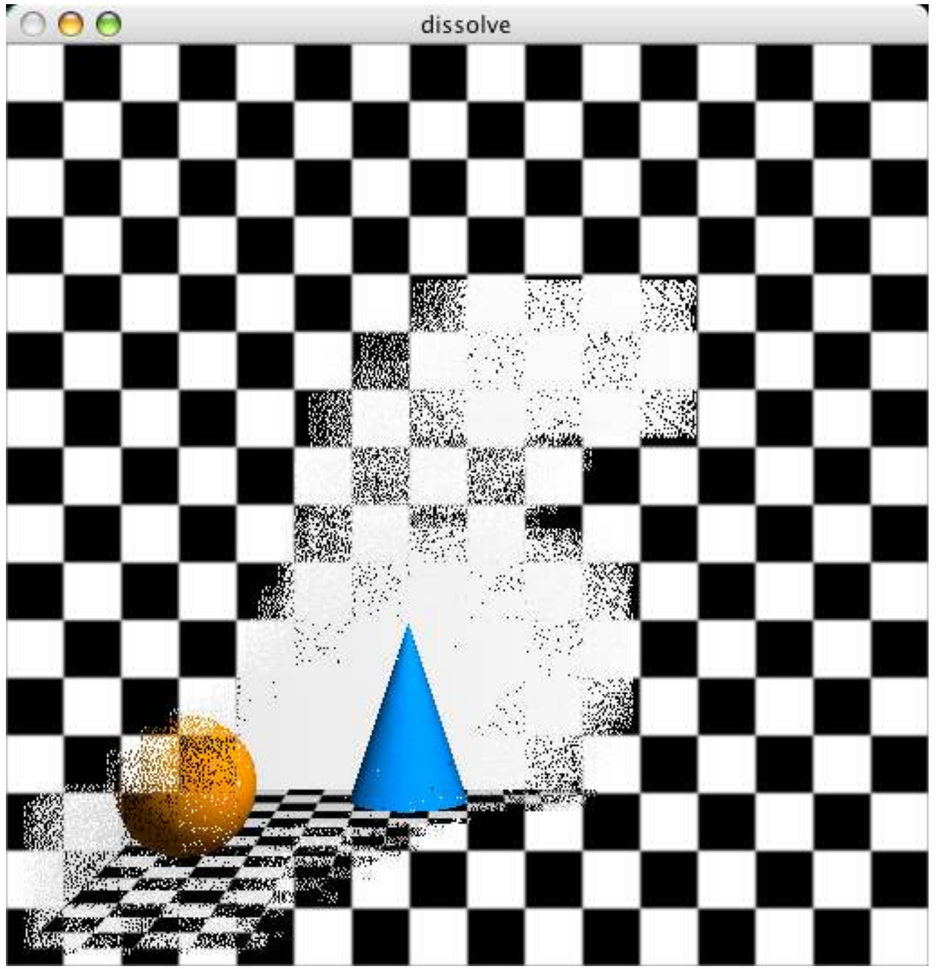


# Beispiele für komplexere Operationen/Effekte mittels Stencil-Buffer

CSG-Operationen (Schnitt, Differenz, ...)

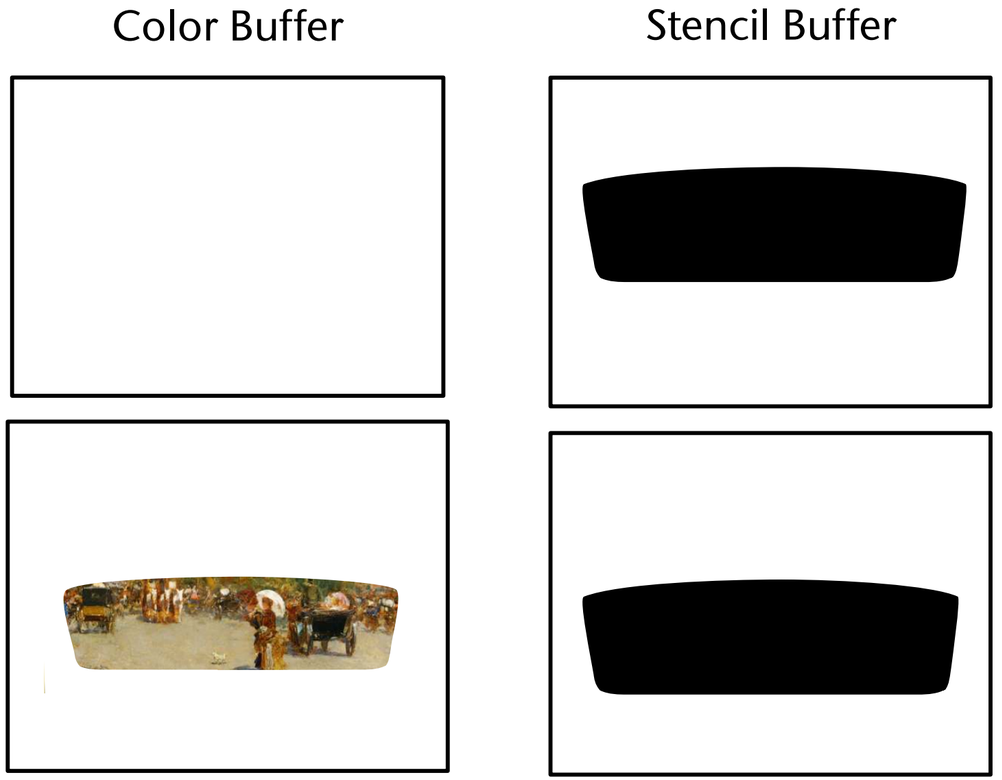



"Dissolve"



# Typisches, einfaches Anwendungsbeispiel

- Szene durch ein Objekt maskieren:
  1. Alle Buffer inkl. Stencil-Buffer löschen
  2. Maske rendern, dabei Stencil-Buffer überall dort auf 1 setzen, aber Color-Buffer unverändert lassen(!)
  3. Szene zeichnen, aber nur dort, wo Stencil-Wert = 1

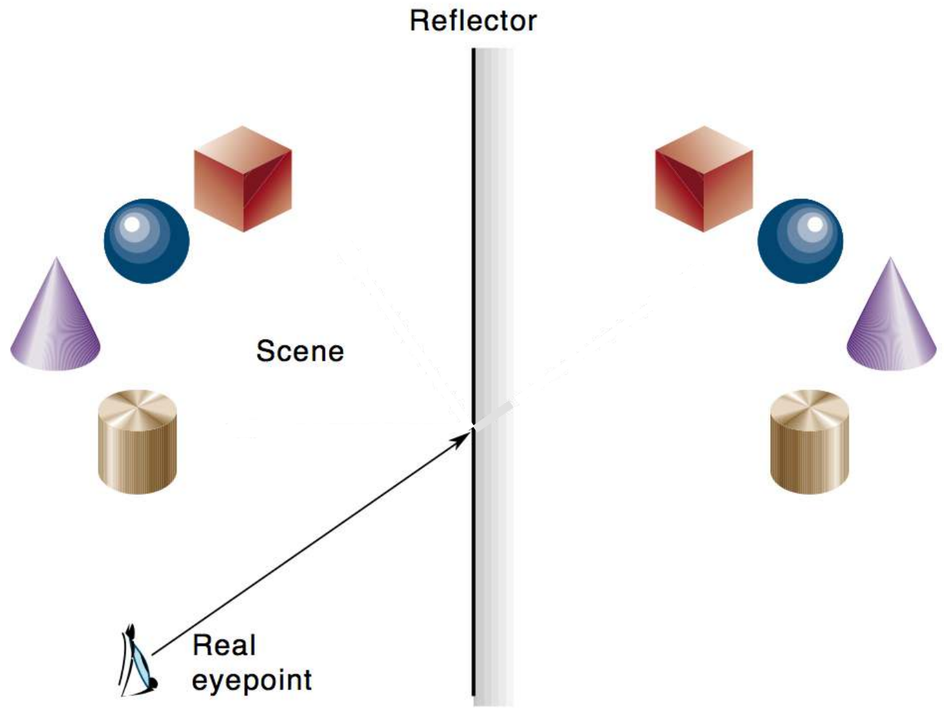


# Anwendung des Stencil-Buffer zum Rendering planarer Spiegel

# Rendering *Planar Reflections* Using the Stencil Buffer

- Grundlegende Idee:
  - Erzeuge für jedes Objekt ein "virtuelles" gespiegeltes Objekt
  - Betrachte Spezialfall, daß die Spiegelebene die Ebene z=0 ist (xy-Ebene)
  - Setze Viewpoint

  1. Rendere alle Polygone mit z' = -z
  2. Rendere die Szene normal




- Dies ist ein Beispiel für einen multi-pass Rendering-Algo
- Achtung: rendere in Pass 1 nur Polygone, wenn sie nach der Spiegelung wirklich *hinter* der Spiegelebene liegen! (Clipping-Plane in Spiegelebene)

- Problem:
  - Normale Spiegel (Wandspiegel, Autospiegel) haben nur eine begrenzte Ausdehnung
  - Der simple Algorithmus zeigt gespiegelte Objekte, wo gar kein Spiegel ist!
- Lösung: der Stencil-Buffer
  - Erzeuge im Stencil-Buffer eine Maske mit genau der Form des Spiegels

# Der 2-Pass Algorithmus im Detail

- Clear color & z buffer; set up viewpoint, etc.

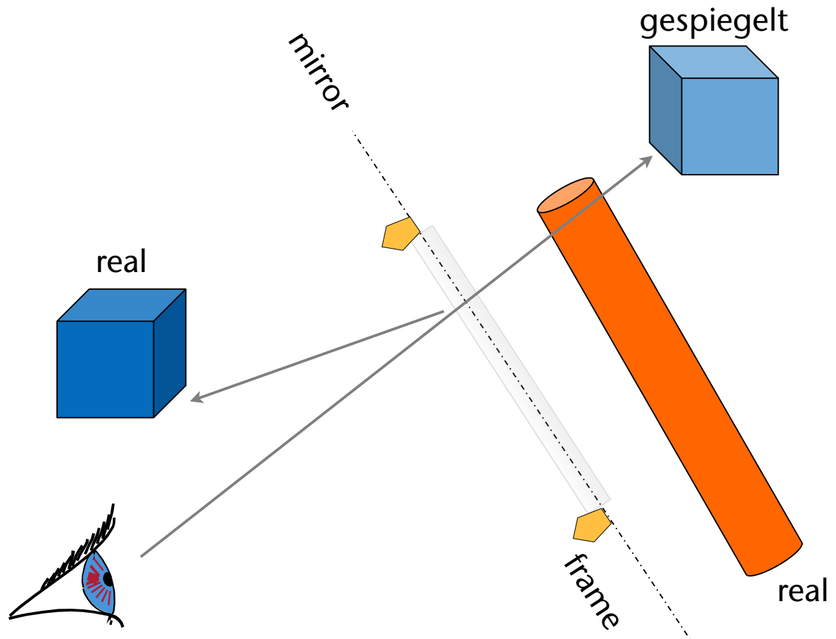
1. Pass: render objs that could potentially be seen in mirror
   - Set up clipping plane such that *only* objects *behind* mirror are rendered
   - Compute transformation for reflection and apply to all polygons
   - Render scene (without geometry of mirror itself)

2. Pass: draw mirror (not its frame), and mask out everything outside mirror

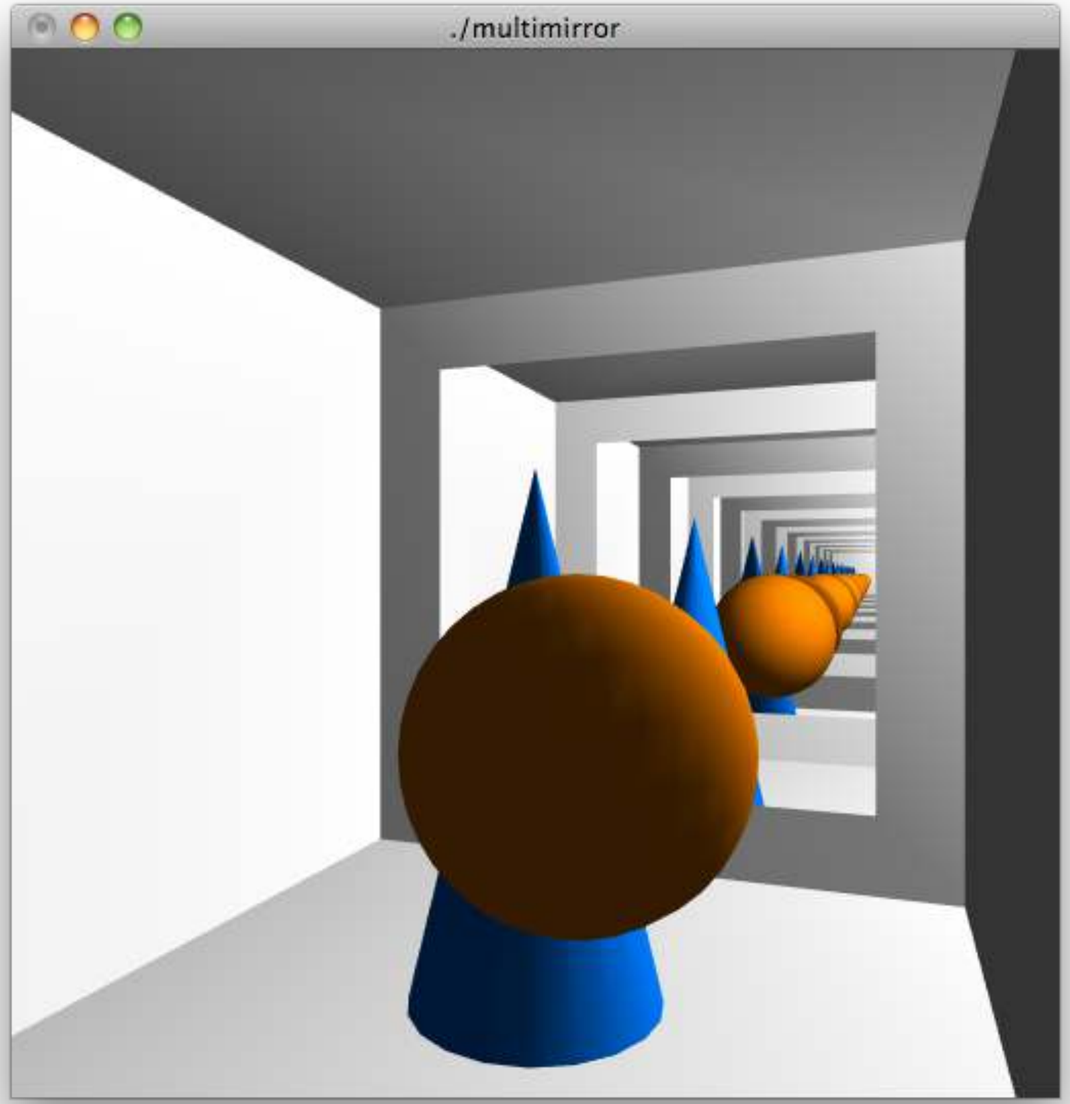
- Clear stencil and z buffer, *but leave color buffer intact*
  - `glClear( GL_STENCIL_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )`
- Configure stencil buffer such that 1 will be stored at each pixel touched by a polygon (of mirror)
  - ```
    glStencilOp( GL_REPLACE, GL_REPLACE, GL_REPLACE );
    glStencilFunc( GL_ALWAYS, 1, 1 );
    glEnable( GL_STENCIL_TEST );
    ```
- Draw the geometry of the mirror
  - This sets stencil bits & fills z buffer with depth values of mirror geometry
  - Enable blending for color buffer to see mirror surface
- Clear color buffer outside mirror geometry:
  - Configure the stencil test to pass *outside* the mirror polygon:
    ```
    glStencilFunc(GL_NOTEQUAL, 1, 1);
    glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
    ```
  - Clear color buffer, so that pixels outside the mirror return to the background color:
    ```
    glClear(GL_COLOR_BUFFER_BIT)
    ```


    ```

- Pass 3:
  - Disable stencil test
  - Disable clipping plane
  - Render scene as usual
    - Including frame of mirror



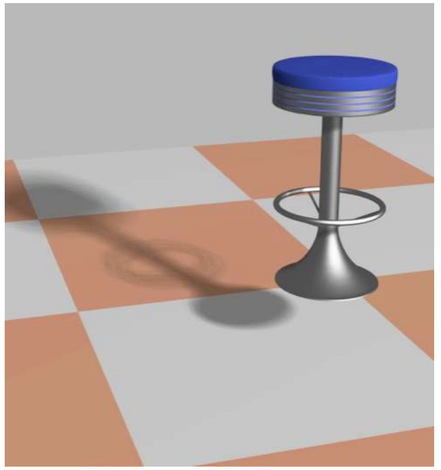
# Demo

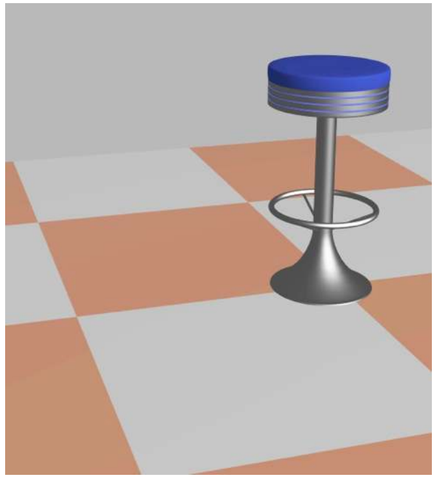


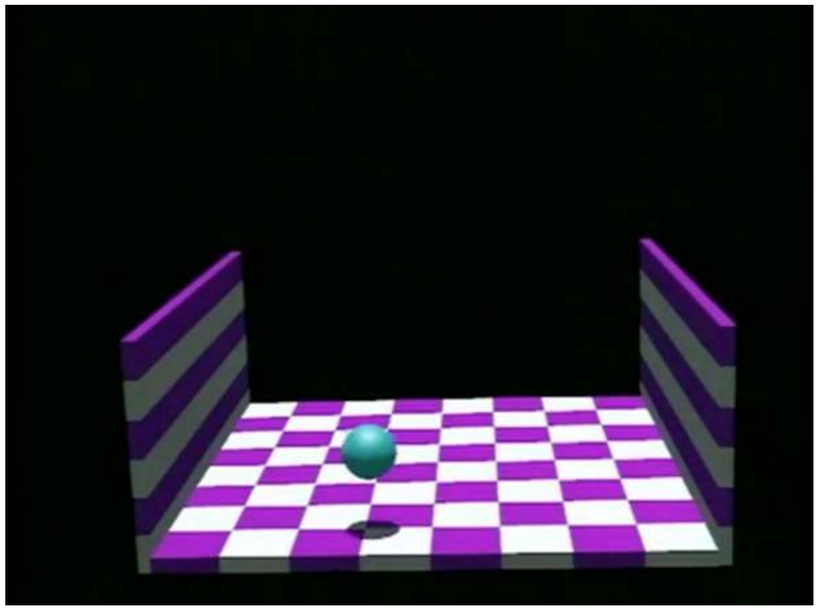

Iterative Anwendung des Verfahrens auf mehrere hintereinander liegende Spiegel ⟶ Hack, der nur für diesen speziellen Fall die Illusion von "Spiegel im Spiegel" schafft!

# Schatten

- Warum ist Schatten so wichtig?
  - Bessere "Verankerung" der Objekte in der Szene:
    - Mehr Information über die relative Lage der Objekte im Raum
    - Ein wichtiger Depth Cue (Tiefeninformation)
  - Hervorhebung der Beleuchtungsrichtung
  - Erhöhung des Realismus einer Szene

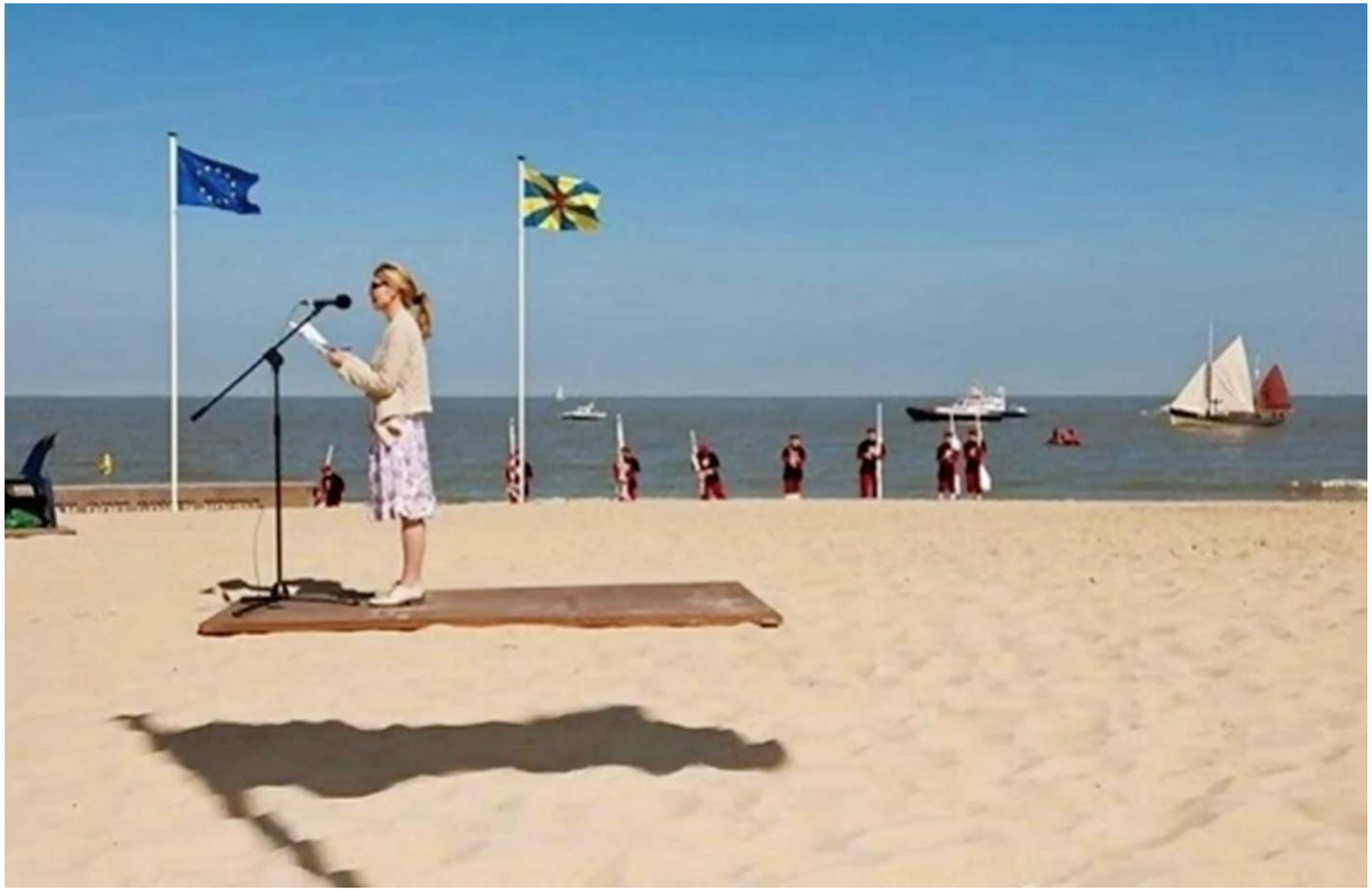






Die Trajektorie des Balls im Bildraum ist in beiden Fällen genau dieselbe!

# Eine optische Täuschung mit Schatten

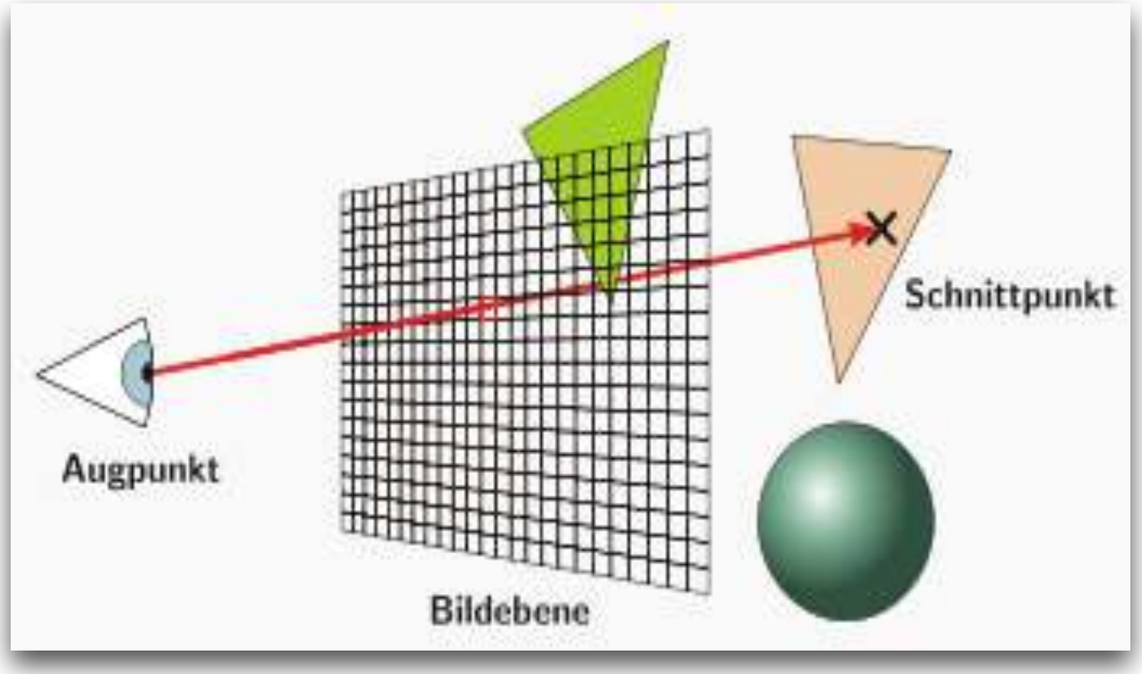


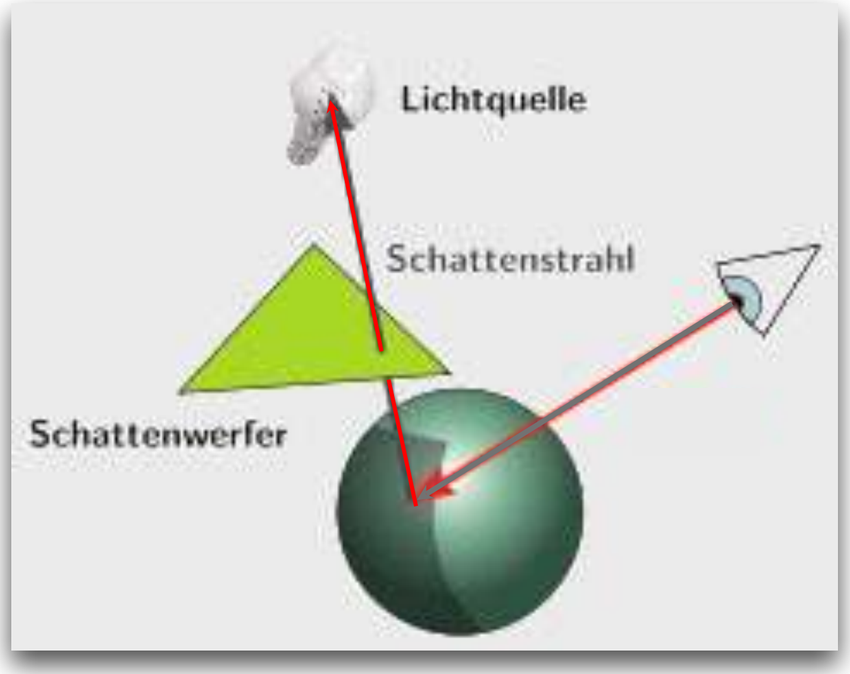
# Wrong Shadows Reveal Poorly Done Foto Faking


Pro 7, Galileo, Fake Fotos, 30. 5. 2013

# Rendering Shadows using Shadow Volumes

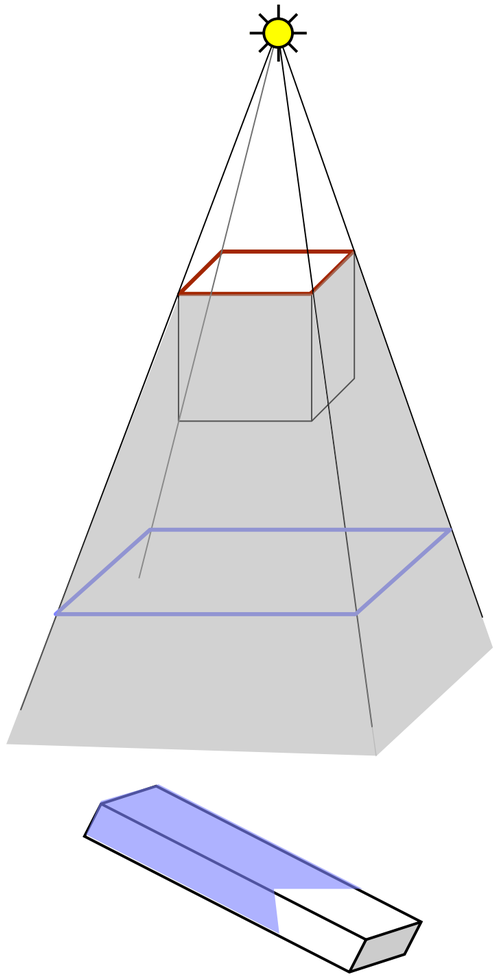
- Zusammenhang zwischen Visibility und Shadows:
  - Visibilitätsberechnung = welche Objekte sind vom Betrachter aus sichtbar
  - Schattenberechnung = welche Objekte sind von der Lichtquelle aus sichtbar



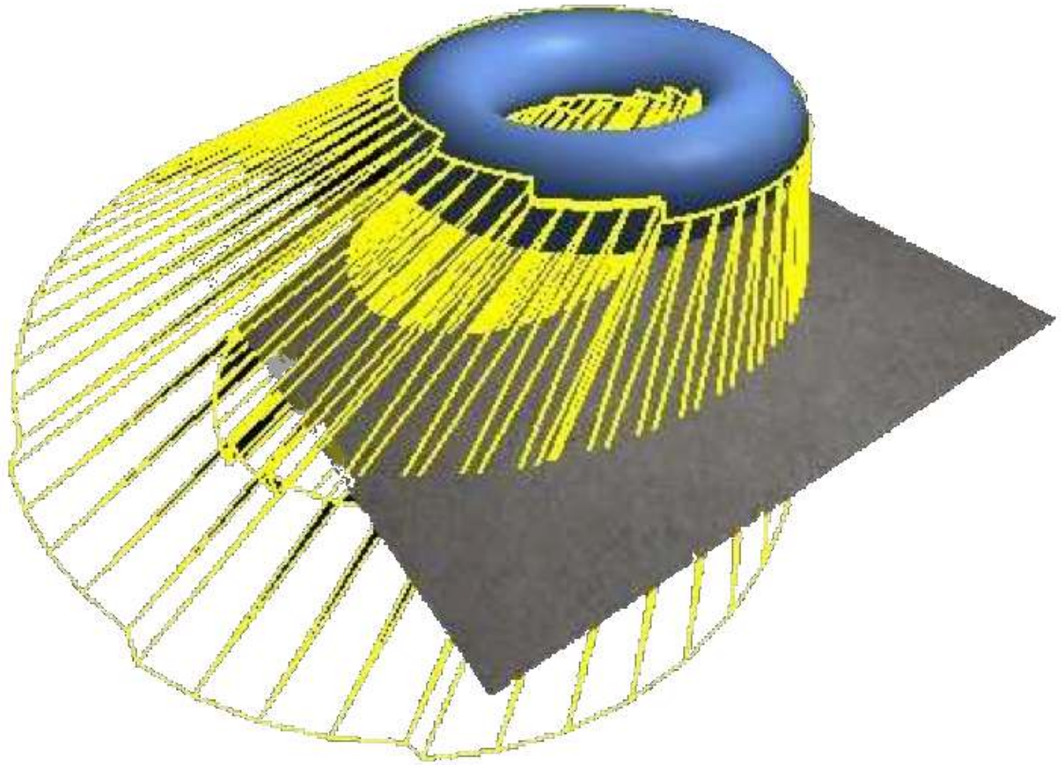



# Das Schattenvolumen

- Ansatz im Folgenden: modelliere die (Teil-)Volumina des Universums, die *kein* Licht von einer gegebenen Lichtquelle erhalten
- Das Schattenvolumen (*shadow volume*):
  - Ein Kegelstumpf, mit der Lichtquelle als Spitze
  - Erzeugt durch einen "*shadow caster*"
  - Jede Silhouettenkante (*silhouette edge*) des Casters , von der Lichtquelle aus gesehen(!), erzeugt genau ein Quad im Shadow Volume
  - Das Shadow Volume ist (im Prinzip) unendlich
- Liegt ein Objekt (teilweise) im Inneren des Schattenvolumens, so heißt dieses "shadow receiver"

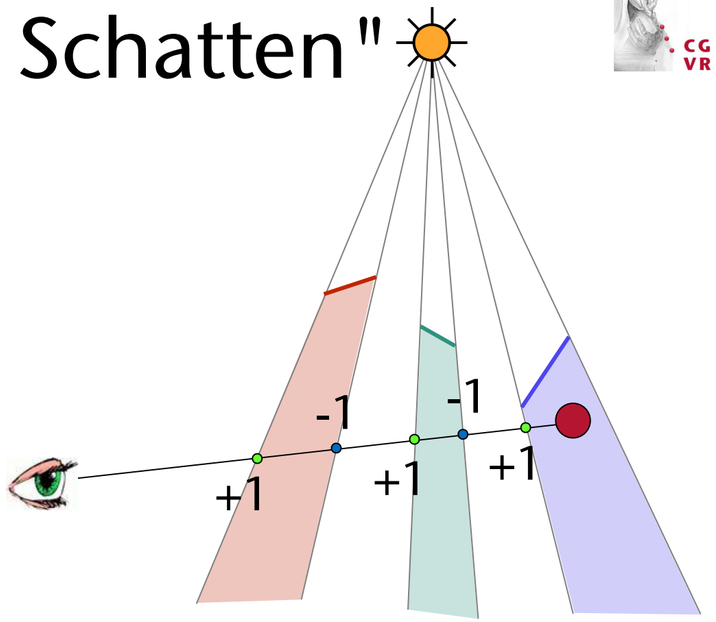


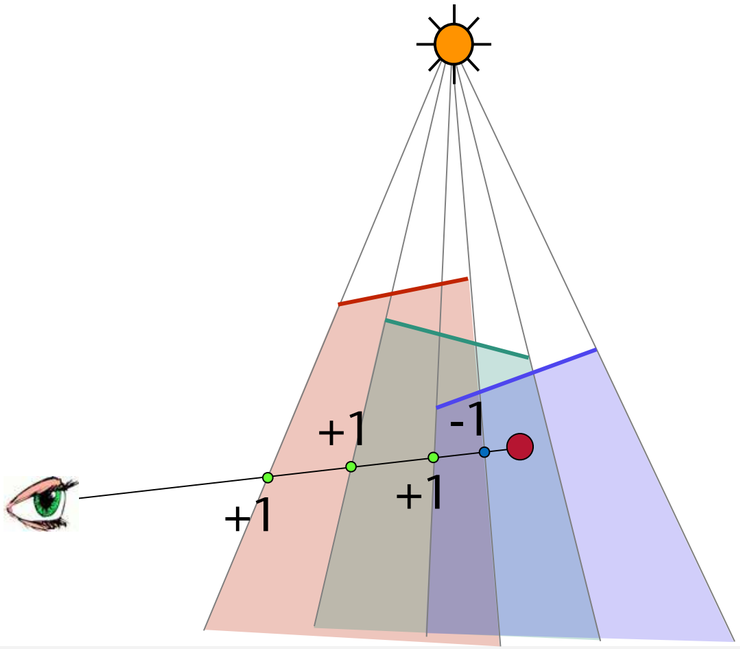
- Beispiel für ein komplexeres Shadow Volume
- Beachte: die Silhouettenkanten liegen nicht notwendigerweise alle in einer Ebene!



# Ein geometrisches Prädikat für "Pkt liegt im Schatten"

- Die prinzipielle Idee (ähnlich zu Inside-/Outside-Test bei der Rasterisierung von allgemeinen Polygonen):
  - Zähle Schnitte zwischen Sehstrahl und Schattenvolumen
  - Initialisierung mit 0, +1 bei Eintritt in Schattenvolumen, -1 bei Verlassen
  - Zähler zeigt an, in "wievielen Schatten" sich ein Punkt zugleich befindet
- Spezialfall: Beobachter ist selbst im Schatten!
- Bezeichnung: front- / back-facing polygons



# Der Algorithmus im Detail (Variante "Z-Pass-Algo")

- Hier oBdA: nur 1 Lichtquelle
  - Sonst: pro Lichtquelle 2 weitere Passes
- Pre-processing: berechne alle Shadow Volumes
  - In echter Implementierung: während des Renderns der Szene generieren

1. Pass: rendere Szene mit normaler Beleuchtung durch die Lichtquelle

```
glClearStencil(0);                  // init stencil to 0
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);                // enable light source that casts shadow
glEnable(GL_DEPTH_TEST);            // standard depth testing ..
glDepthFunc(GL_LEQUAL);             //    .. with <=
glDepthMask(1);                     // update depth buffer
glDisable(GL_STENCIL_TEST);         // no stencil testing in this pass
glColorMask(1,1,1,1);               // update color buffer
renderScene();
```

(Old OpenGL Syntax)

2. Pass: render Shadow Volumes; zähle im Stencil-Buffer die Anzahl Eintritte und Austritte für das Pixel, das an der jeweiligen Stelle im Framebuffer sichtbar ist

```
glDepthMask(0);                                  // don't modify depth buffer!
glColorMask(0,0,0,0);                            // .. nor color buffer
glDisable(GL_LIGHTING);                          // no need to compute lighting
glEnable(GL_DEPTH_TEST);                         // only pgons of shadow vol. truly
glDepthFunc(GL_LESS);                            //  in front of visible pixel count
glEnable(GL_STENCIL_TEST);                       // use stencil testing
glStencilMask(~0u);                              // use all bits of stencil buffer
glEnable(GL_CULL_FACE);                          // we need one pass for back/front
glCullFace(GL_BACK);                             // for all front-facing pgons ..
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);          // .. passing the depth test
renderShadowVolumePolygons( light0 );            // .. increase stencil value
glCullFace(GL_FRONT);                            // for all back-facing pgons ..
glStencilOp(GL_KEEP, GL_KEEP, GL_DECR);          // .. passing the depth test
renderShadowVolumePolygons( light0 );            // .. decrease stencil value
```

3. Pass: rendere die Szene ohne Lichtquelle (= Schatten); schreibe Pixel nur dann in den Color-Buffer, wenn sie im Schatten der Lichtquelle sind ("Licht ausknipsen")

```
glEnable(GL_LIGHTING);                          // switch off light source 0
glDisable(GL_LIGHT0);                           //    but keep all others
glEnable(GL_DEPTH_TEST);                        // use z-test as usual
glDepthFunc(GL_EQUAL);                          // must match from 1st step
glDepthMask(0);                                 // no need to update z buffer
glEnable(GL_STENCIL_TEST);                      // only render pixels that are
glStencilFunc(GL_GEQUAL, 1, ~0u);               //   inside the shadow
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);         // no need to update stencil
glColorMask(1,1,1,1);                           // do modify the color buffer
renderScene();
```

- Dieser Algorithmus heißt "z-pass algorithm", weil in Pass 2 nur diejenigen Fragmente der Shadow-Volumes den Stencil-Buffer verändern, die den Z-Test passieren (= vor der eigtl Geometrie sind)

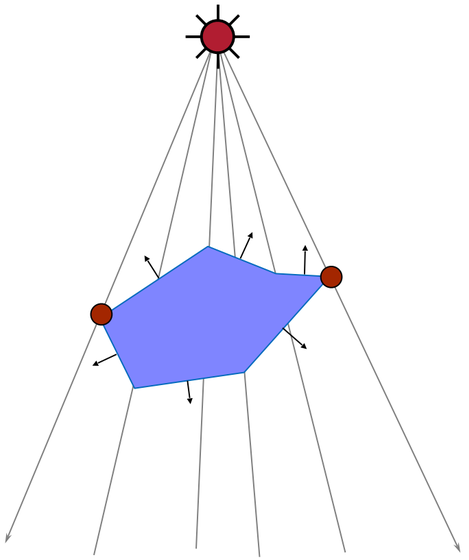
# Varianten des Algorithmus'

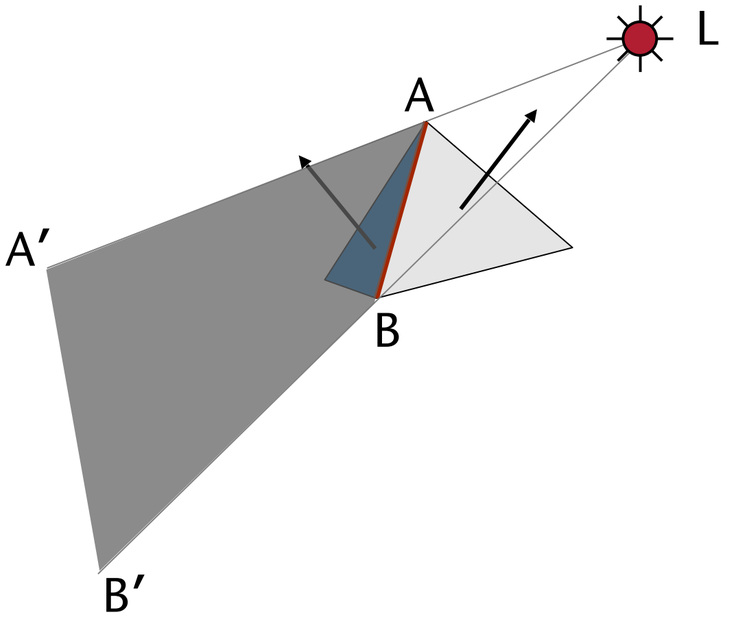
- Es gibt eine GL-Extension, so daß man eine Stencil-Operation für front-facing, und *gleichzeitig* eine andere Stencil-Operation für back-facing Polygone angeben kann
  - Ist aber nicht auf allen Graphikkarten / Plattformen verfügbar
- Es gibt Probleme, falls die Schattenvolumengeometrie durch Clipping abgeschnitten wird
  - Eine Variante des Algos (der "*z-fail algo*") kommt damit zurecht
- Für mehrere Lichtquellen:
  - Rendere in Pass 1 die Szene ohne alle Lichtquellen (nur *ambient light*)
  - Für jede Lichtquelle:
    - Führe Pass 2 und Pass 3 durch
    - In Pass 3: akkumuliere Pixel-Farbwerte auf den bestehenden Wert im Color Buffer (also nicht ersetzen; geht mit passender sog. Blending-Funktion)

# Bemerkungen zu Details

- Berechnung der Silhouettenkanten der *shadow caster*:
  - Eine Kante (hat genau 2 inzidente Polygone) ist Silhouettenkante ⇔

    ein Polygon zeigt zur Lichtquelle und ein Polygon zeigt weg von der Lichtquelle (Skalarprodukt)
- Berechnung der Seitenflächen eines Shadow Volumes (eines *shadow caster's*):
  - Verlängere die Eckpunkte der Silhouettenkante
  - Kann man mit speziellen Shadern während des Renders der Szene erledigen (CG2)



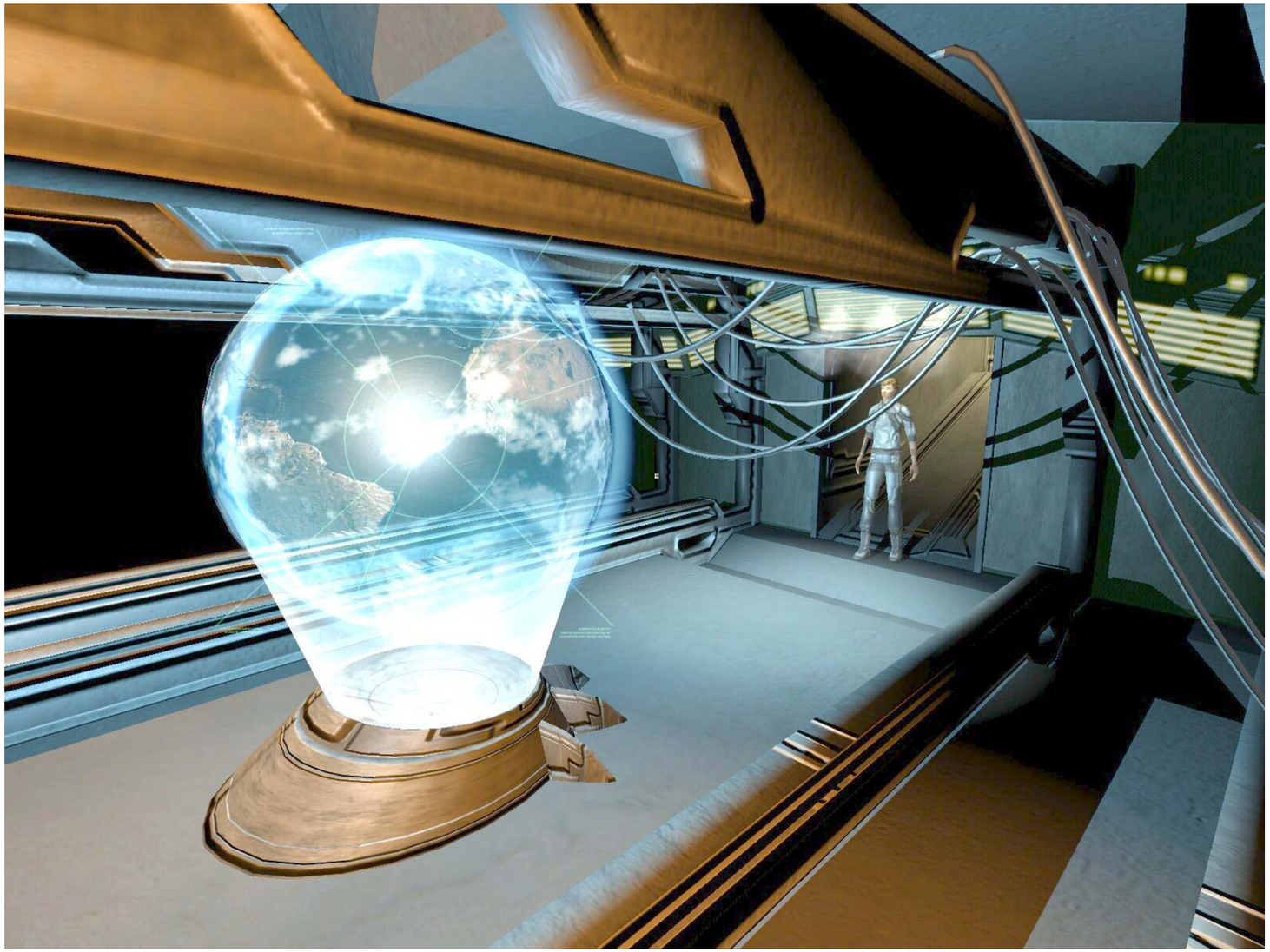


# Beispiele

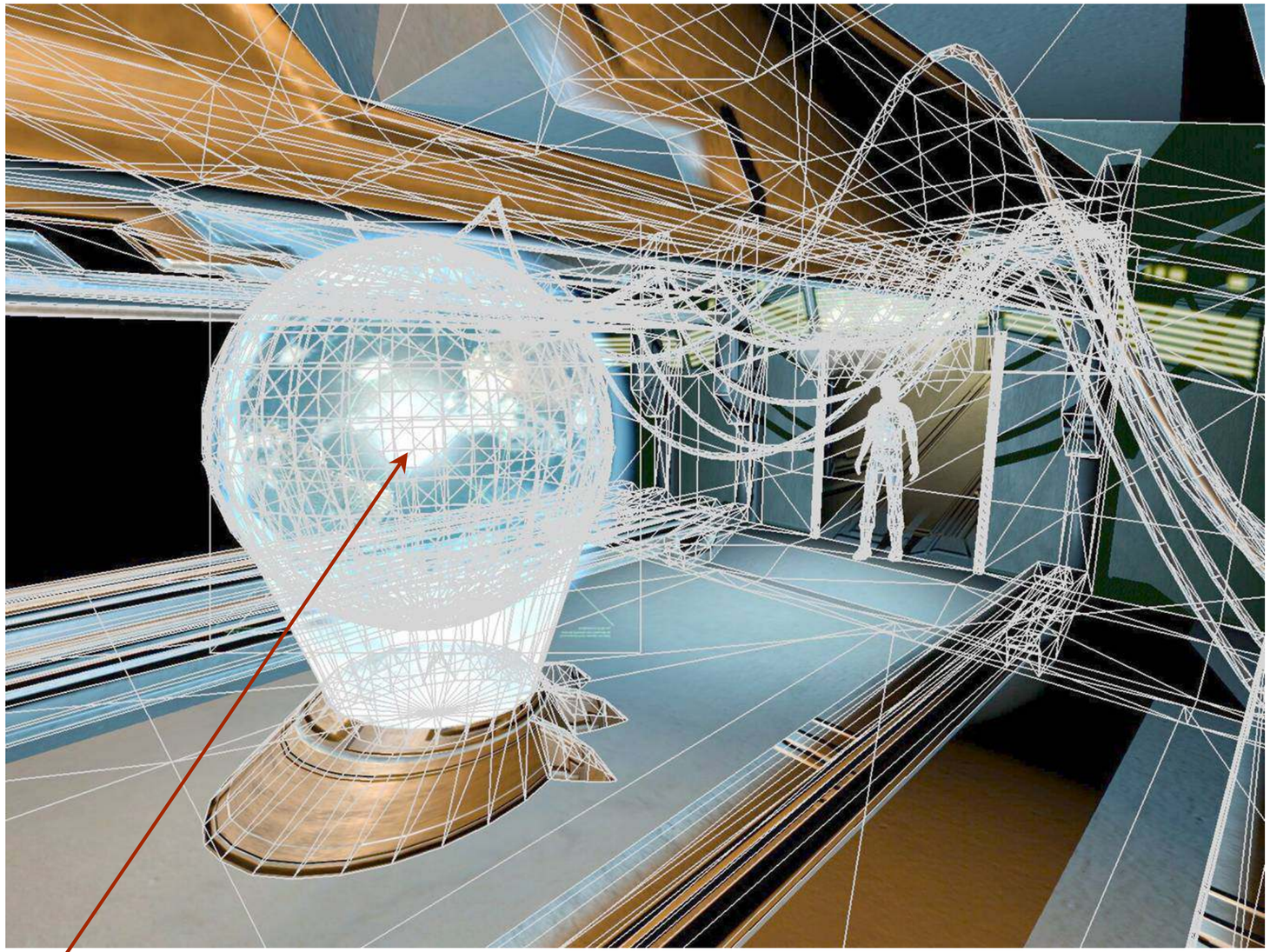
Shadowed scene


Stencil buffer contents

green = stencil value of 0 , red = stencil value of 1 , darker reds = stencil value > 1

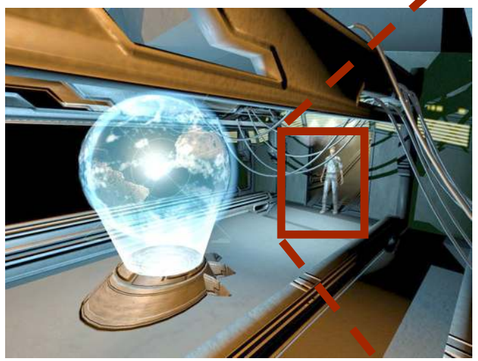


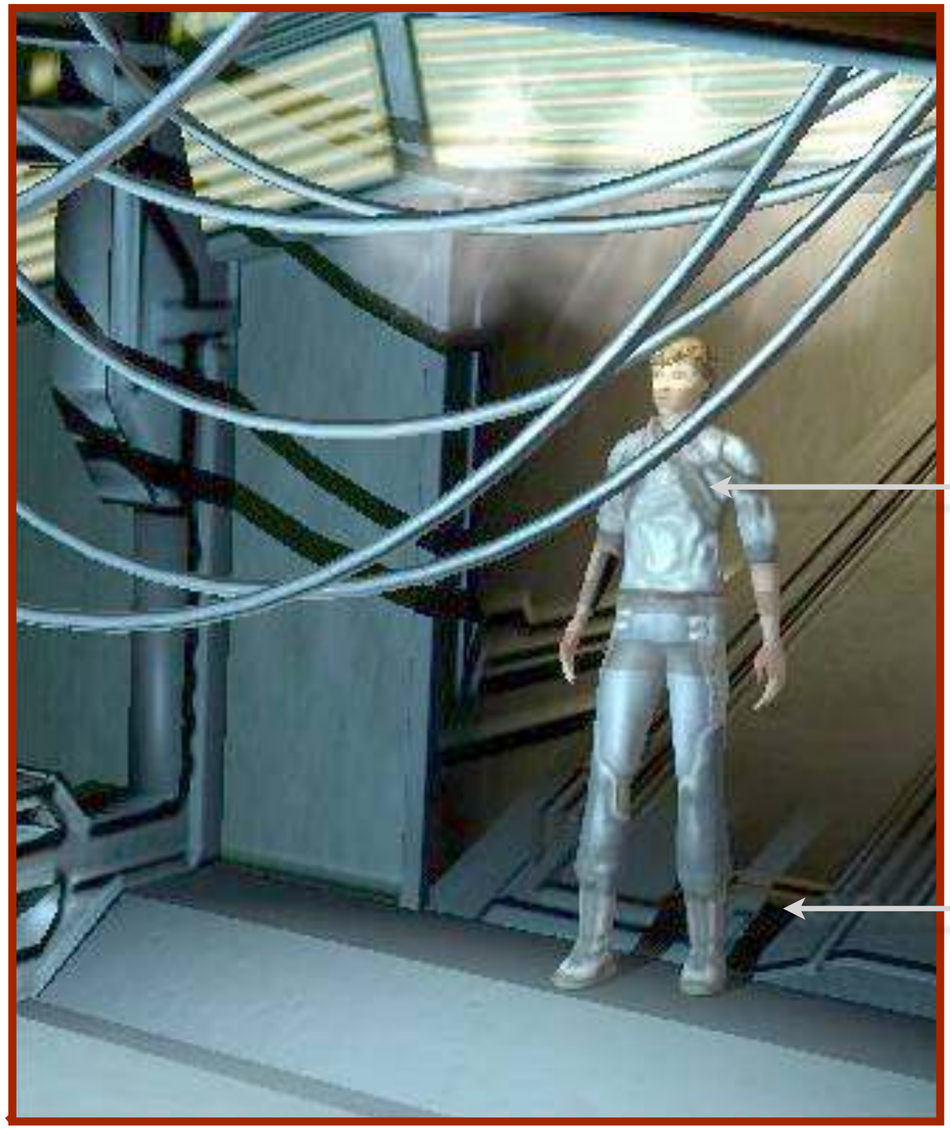
*Abducted* game images courtesy Joe Riedel at Contraband Entertainment



Primary light source location

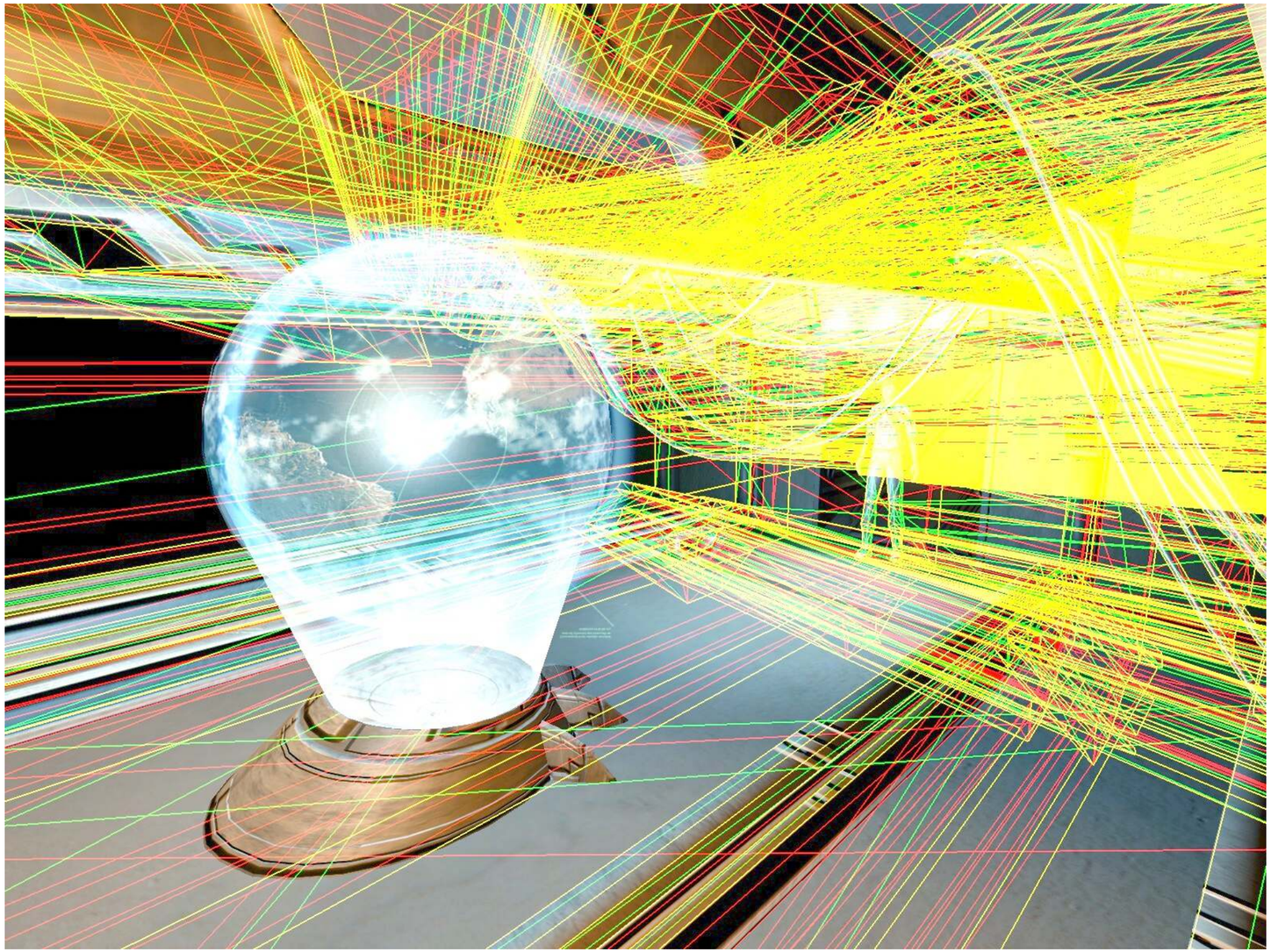
Wireframe shows geometric complexity of visible geometry



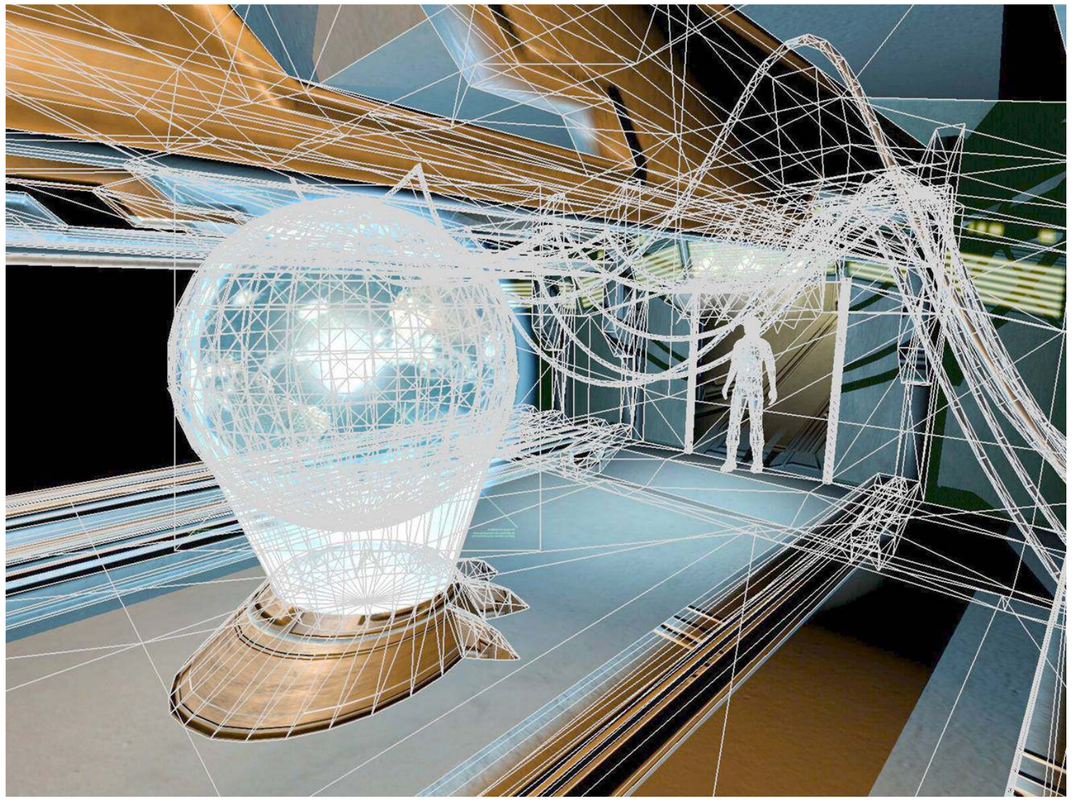


Notice cable shadows on player model

Notice player's own shadow on floor



Wireframe shows geometric *complexity of shadow volume* geometry



Visible geometry

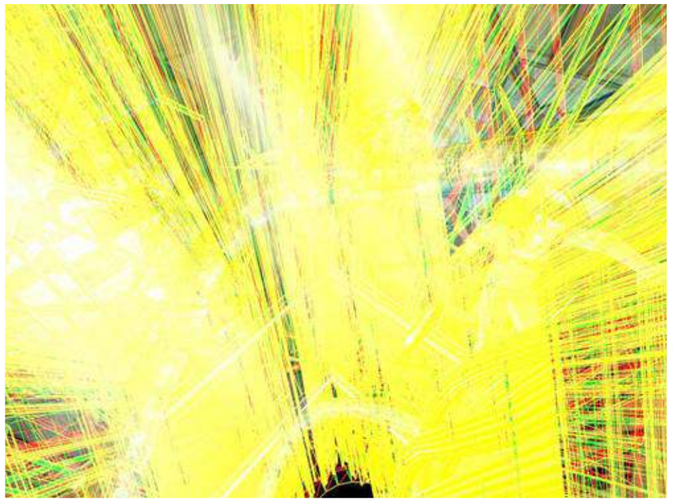
<

<

Shadow volume geometry

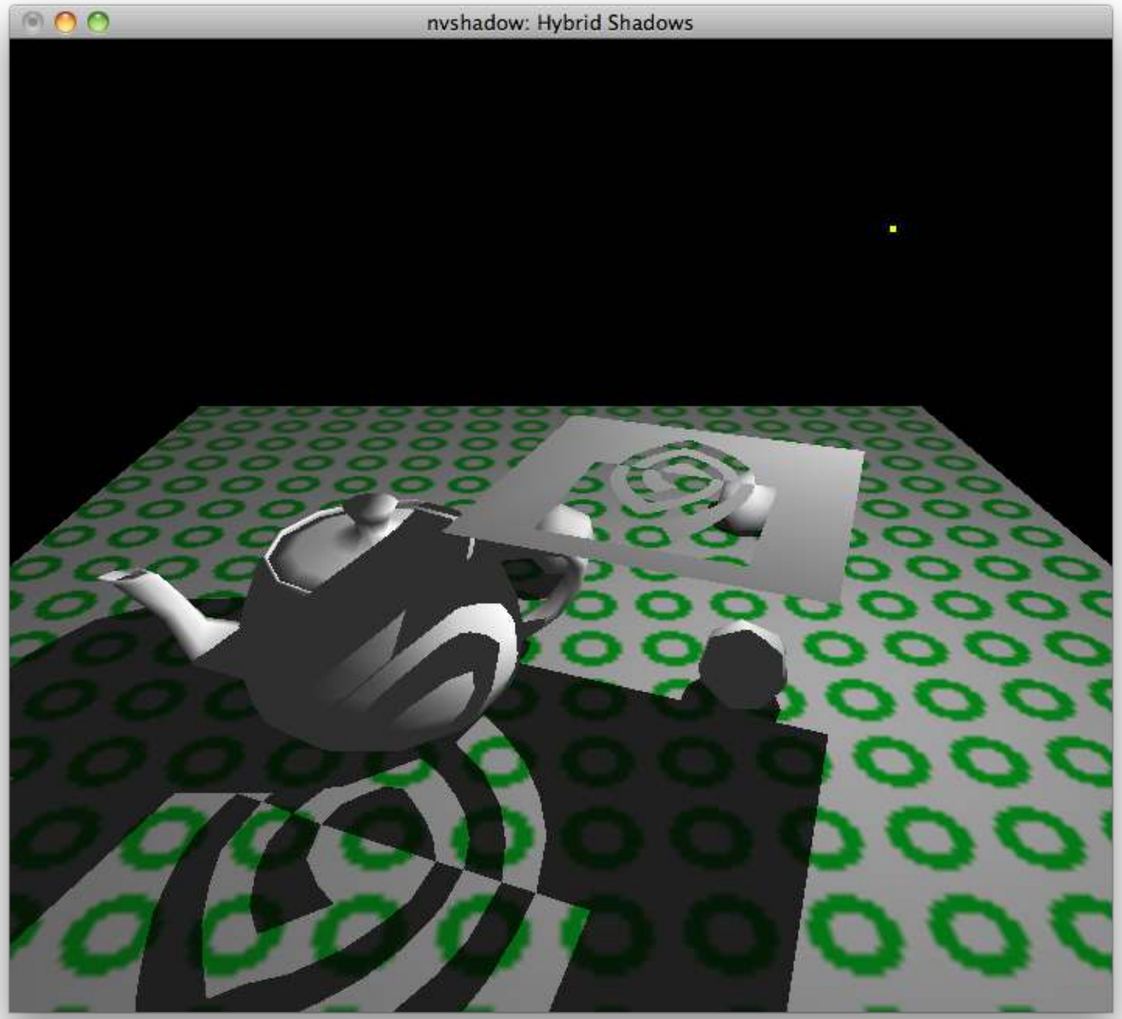
Typically, shadow volumes generate considerably more pixel updates than visible geometry

Visible geometry



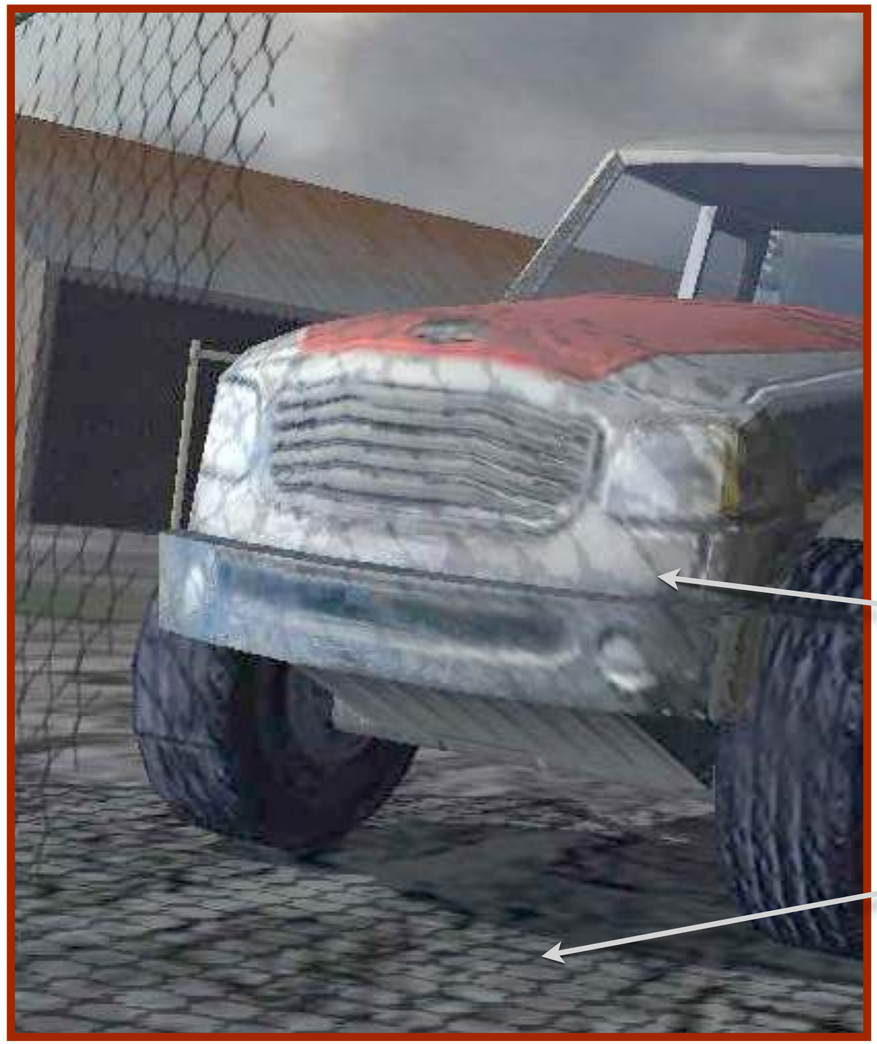
Shadow volume geometry

# Demos




http://www.opengl.org/resources/features/StencilTalk/

# Situations When Shadow Volumes Are Too Expensive



Chain-link fence is shadow volume nightmare!

Chain-link fence's shadow appears on truck & ground with shadow maps

Fuel game image courtesy Nathan d'Obrenan at Firetoad Software

# Sichtbare Schattenvolumen in der Realität