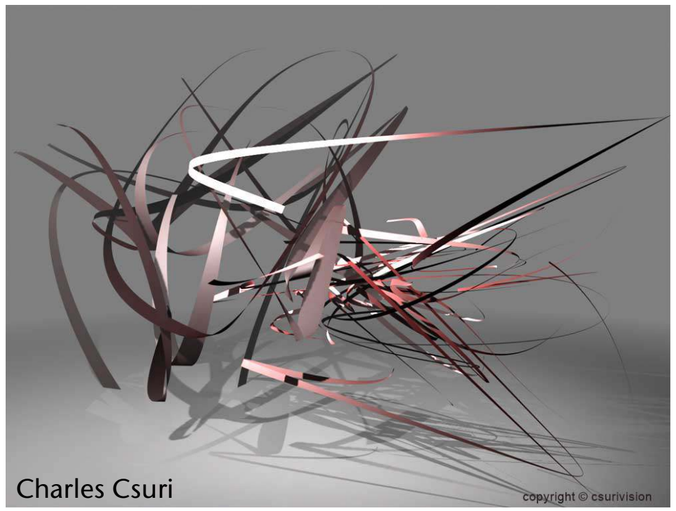
Computer-Graphik 1

Visibility Computations – Hidden Surfaces, Frame Buffers, and Shadows

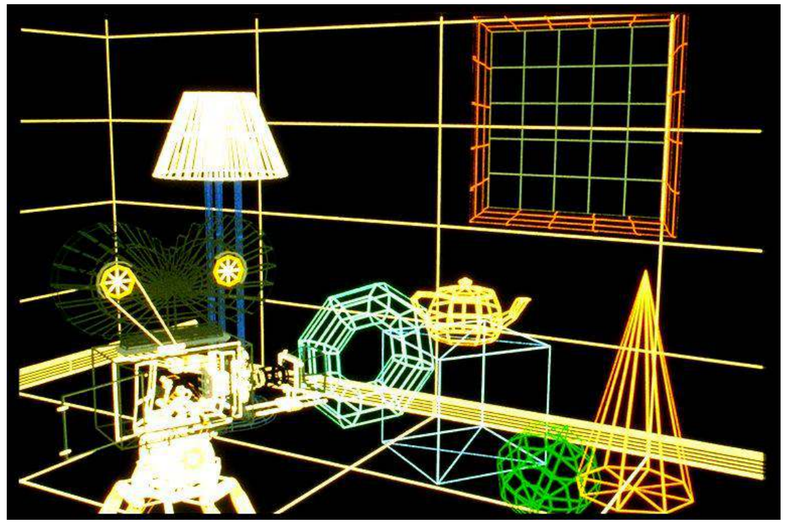


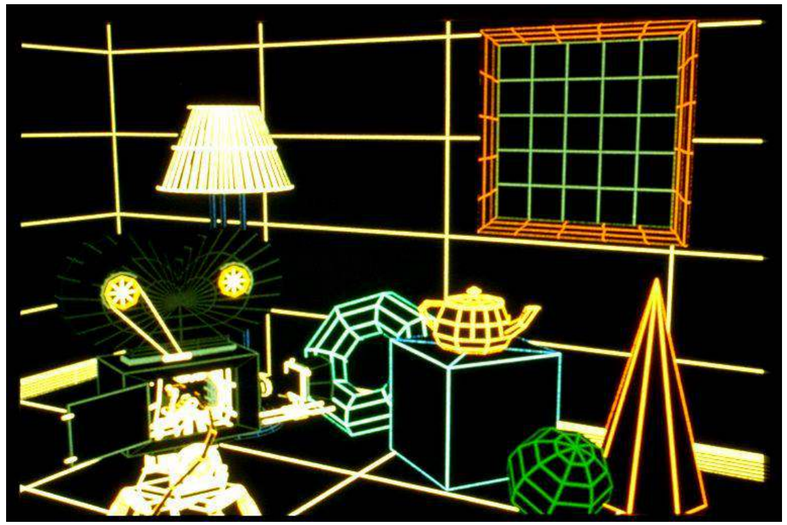
G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

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) ⟶ dahinter liegender Punkt sichtbar

Wireframe-Rendering *ohne* Verdeckungs-berechnung



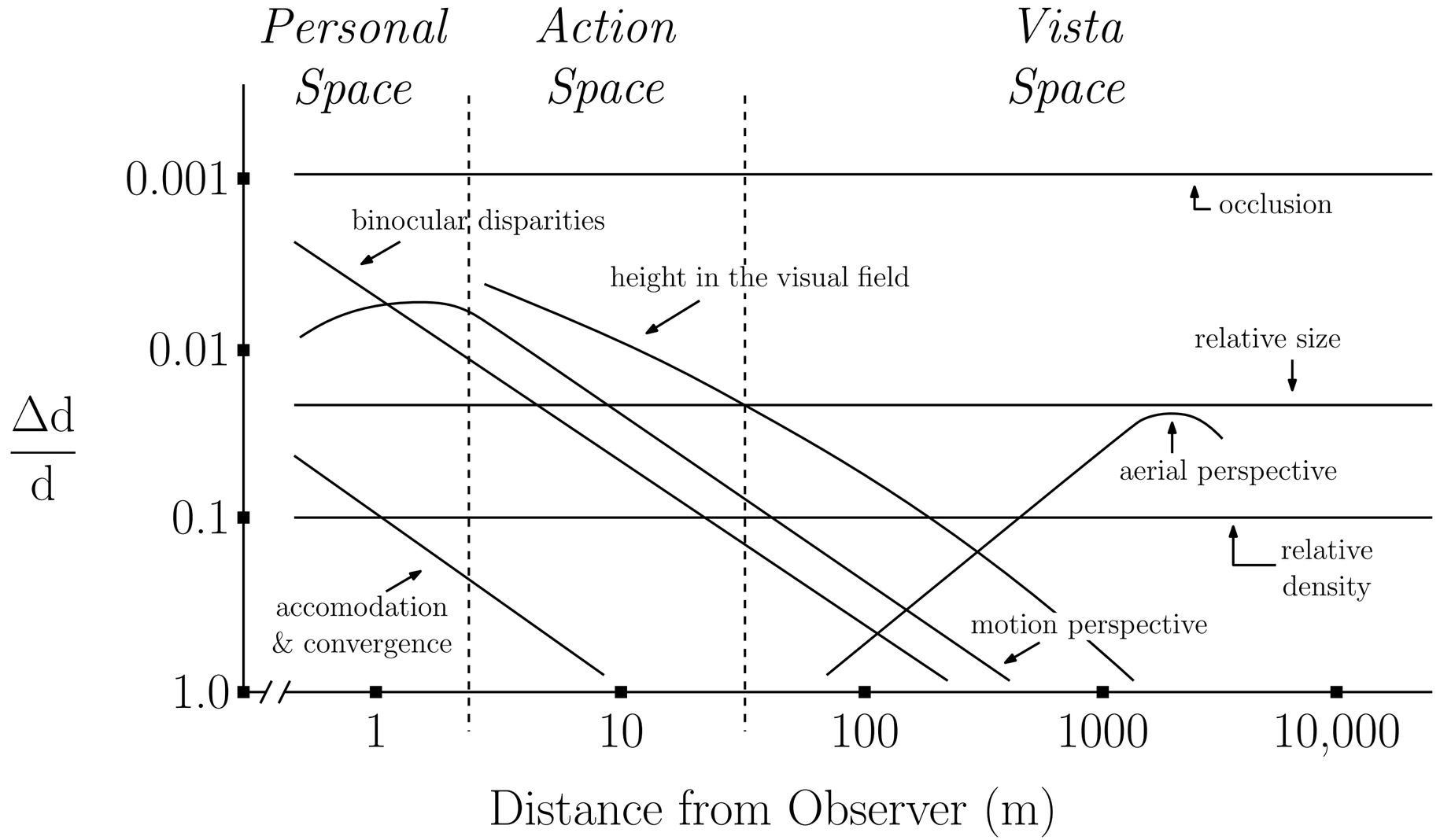


Wireframe-Rendering *mit* Verdeckungs-berechnung

Pixar "Shutterbug"

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

Just-Noticeable Depth Thresholds (FYI)

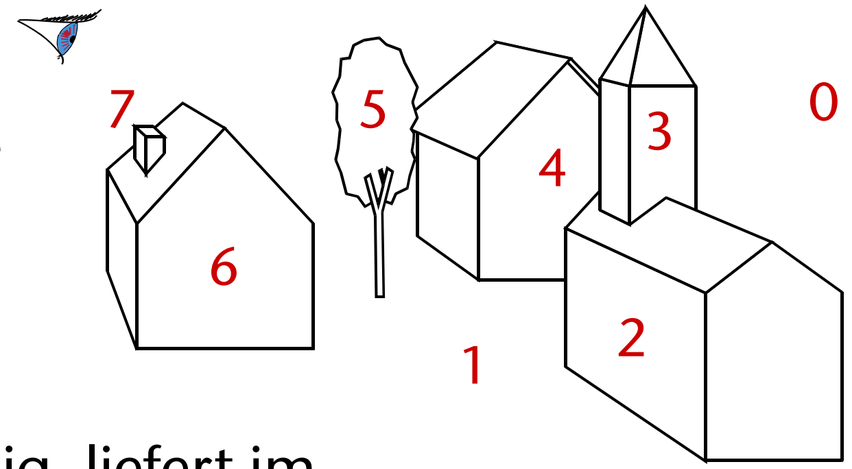


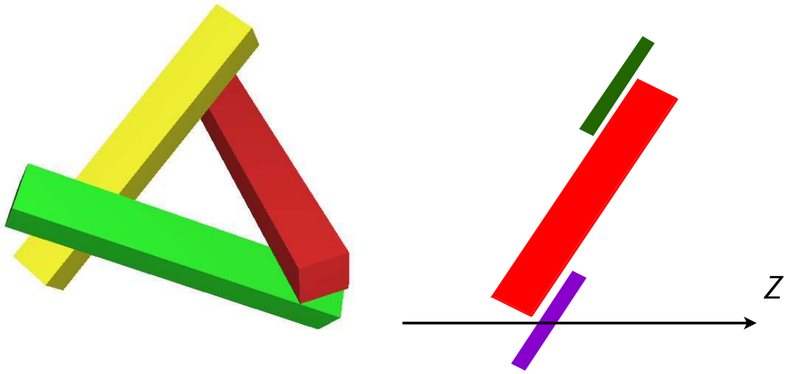
- Es gibt 2 große Problemklassen innerhalb des Bereichs "Visibility Computations"
  1. Verdeckungsberechnung: welche "Pixel" eines Polygons werden von anderen verdeckt?
     - Bezeichnungen: *Hidden Surface Elimination* (früher auch *Hidden Line Elimination*), *Visible Surface Determination*
  2. Culling: 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: Der 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!

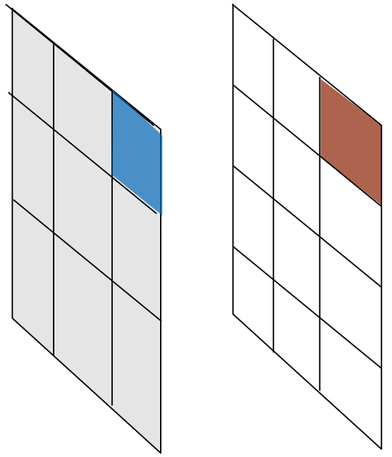


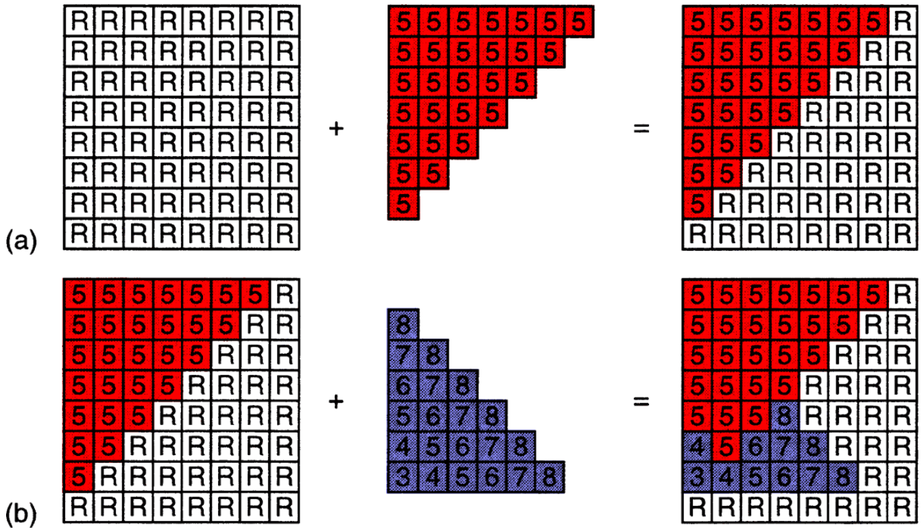


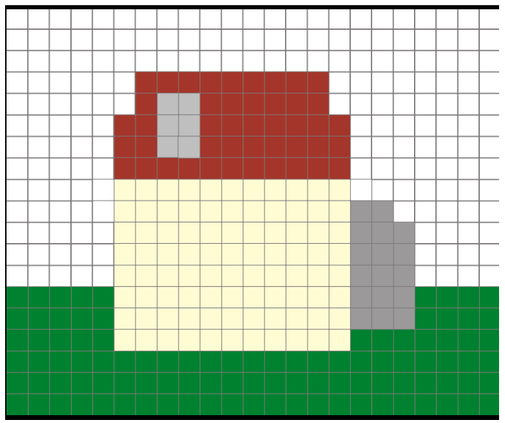
Die Standard-Lösung heute: der Z-Buffer

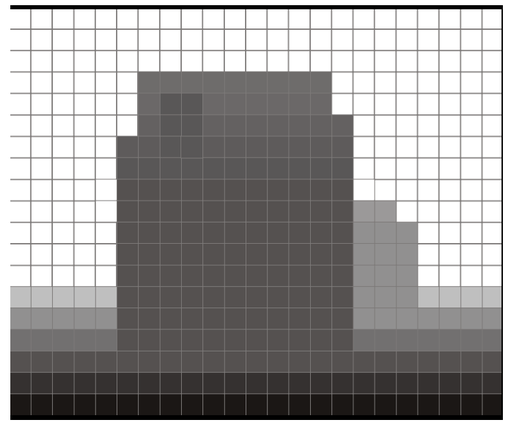

Wolfgang Strasser

- Zusätzlicher Buffer zum Color Buffer
- Speichert pro Pixel den Abstand *z* zur Kamera
- Pixel 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

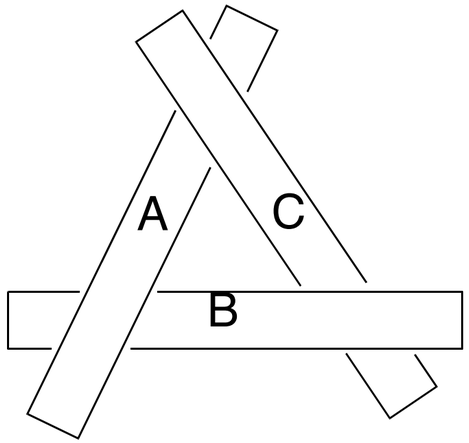
"640 Kilobyte ought to be enough for anybody."

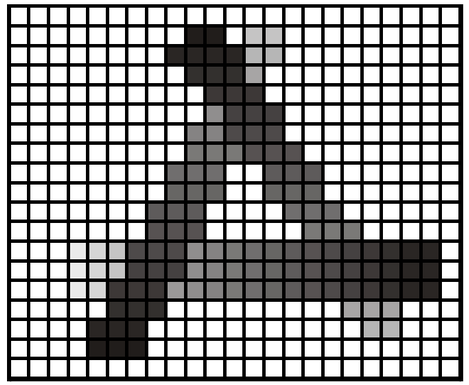
Bill Gates, 1981

Pseudo-Code für Rasterisierung mit Z-Buffer

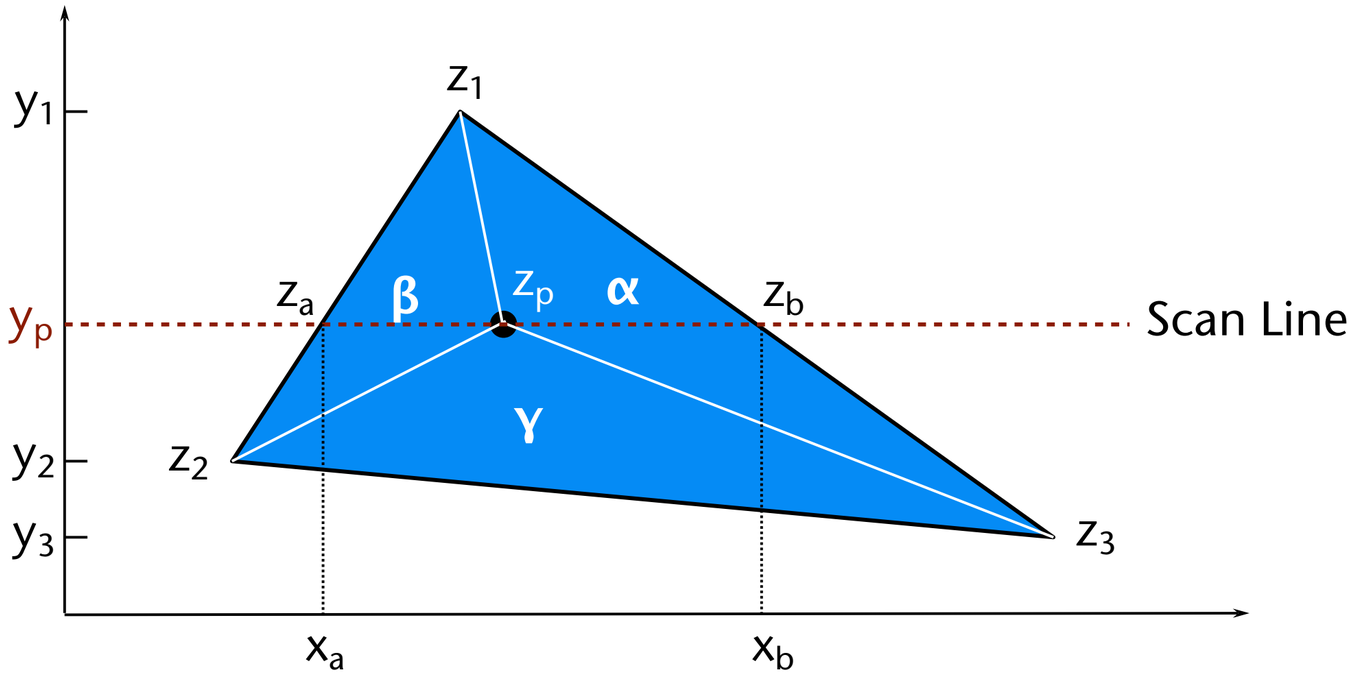
```
for all pixels in window:
  framebuffer[x,y] = BACKGROUND_COLOR; zbuffer[x,y] = ∞;
for every triangle:
  compute projection & color at vertices
  setup edge equations
  compute bbox, then clip bbox to screen limits
  for all pixels x,y in bbox:
    increment edge equations
    compute Z of current pixel / point
    compute current color c (incrementally)
    if all edge equations > 0:          // pixel is in triangle
      if current Z < zBuffer[x,y]:      // pixel is visible
        framebuffer[x,y]= c
        zBuffer[x,y] = current Z
```

Funktioniert auch
in schwierigen Fällen





Berechnung des Z-Wertes bei der Scan-Conversion



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

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

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

- Oder: $z_p = \alpha z_1 + \beta z_2 + \gamma z_3$
  wobei α, β, γ (wie gehabt) inkrementell im Algorithmus von Pineda berechnet werden

Der Z-Buffer in OpenGL

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

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

2. Einschalten:

```
glEnable( GL_DEPTH_TEST );
```

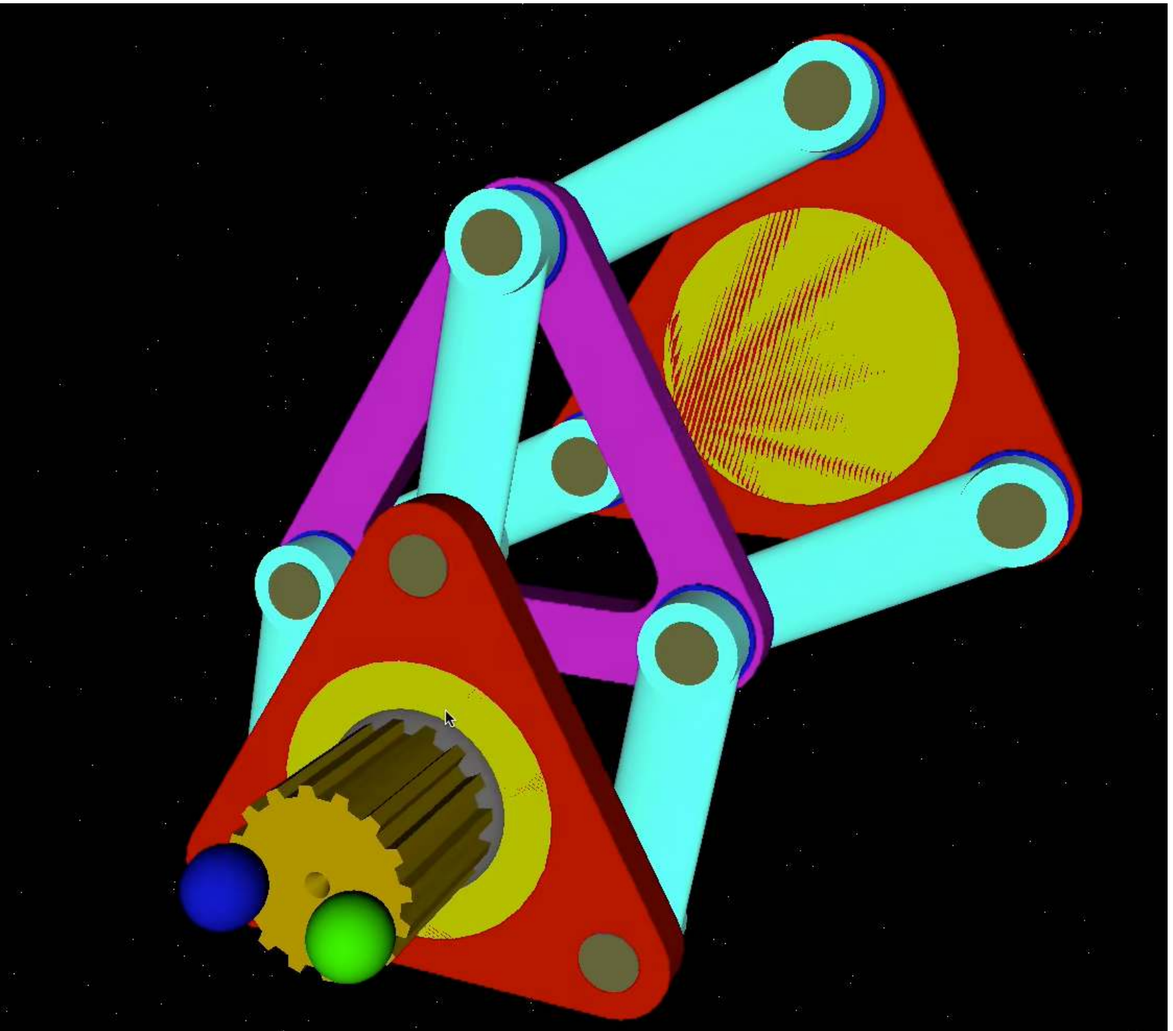
3. Wichtig: nicht nur Bildspeicher, sondern auch Z-Buffer löschen!

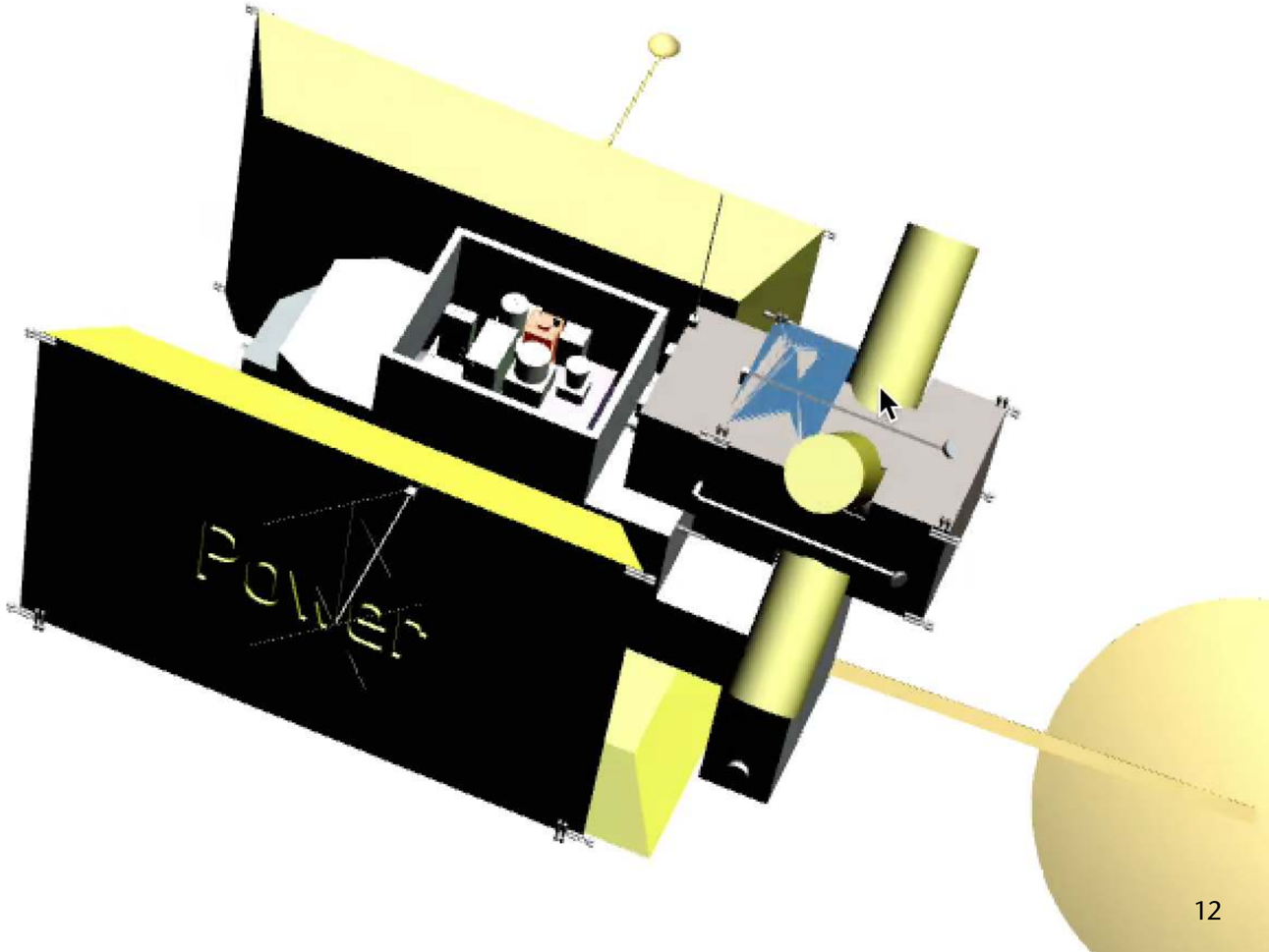
```
glClear( GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT );
```

- 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

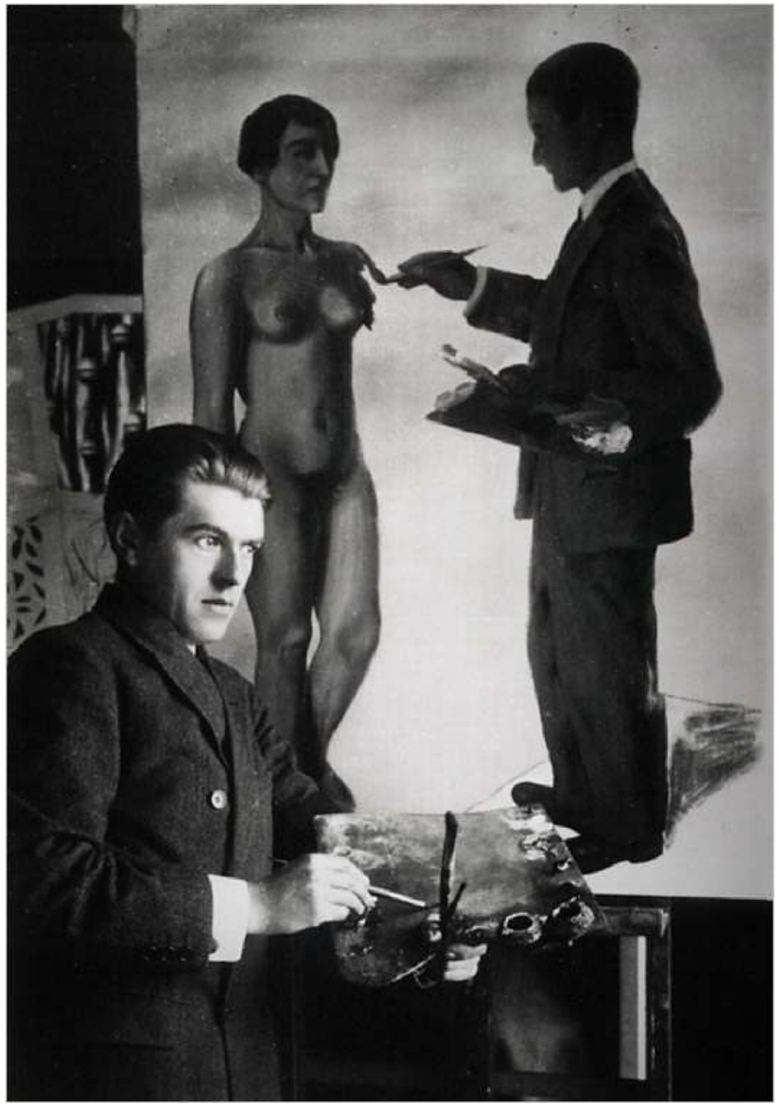
Z-Fighting

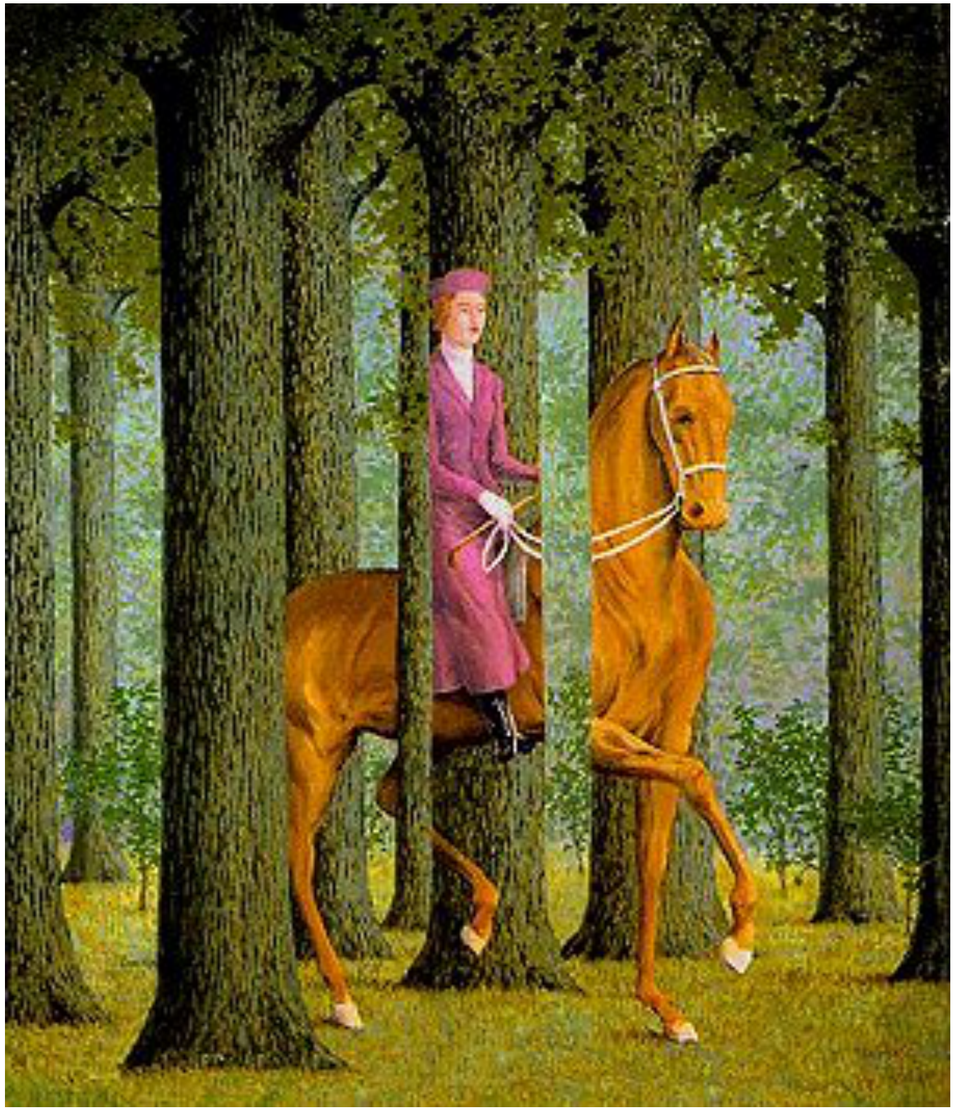
- Wegen der begrenzten Auflösung des Z-Buffers kommt es bei koplanaren (und fast koplanaren) Polygonen zum sog. Z-Fighting bzw. Flickering





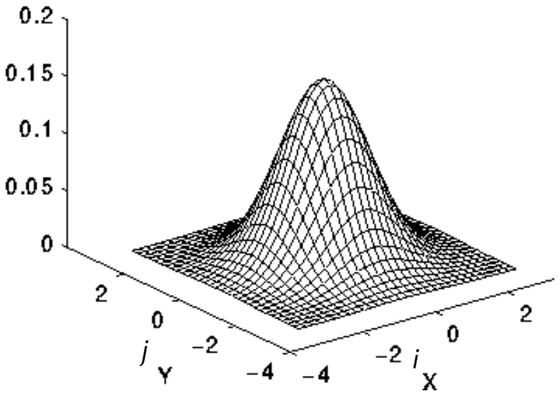
Exkurs: der Surrealist Magritte hat (auch) mit Verdeckung gespielt





Weitere Anwendung des Z-Buffers: Depth-of-Field

- Tiefenunschärfe (*depth-of-field*) = je weiter ein Objekt von der Fokusebene (*focal plane*) entfernt ist, desto "verschmierter" (*blurred*) das Bild
- Die Idee: Image Post-Processing
  - Wende auf unscharfe Pixel einen Blur-Filter mit Radius *r* an
    ⟶ Faltung (*convolution*)
  - Sei z_0 = Abstand der Fokusebene vom Viewpoint
  - Je größer $|z - z_0|$, desto größer Radius *r* des Blur-Filters
- Standard: Faltung des Bildes mit Gauß-Kernel

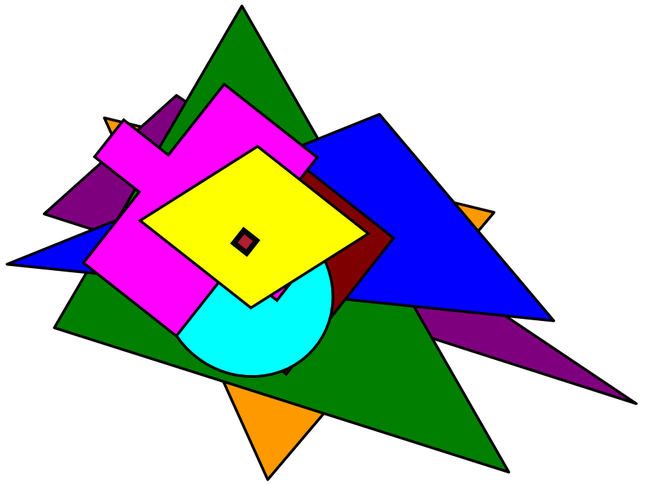


$$O(x,y) = \sum_{i=-\frac{r}{2}}^{\frac{r}{2}} \sum_{j=-\frac{r}{2}}^{\frac{r}{2}} I(x+i, y+j) \cdot G(i,j) \quad G(i,j) = \frac{1}{2\pi r^2} e^{-\frac{i^2+j^2}{2\pi r^2}} \quad \text{und} \quad r = |z(x,y) - z_0|$$

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 p = # Fragmente aller n Pgone (kann unter Umständen viel größer als Anzahl sichtbarer Pixel sein!)
- Läßt sich ideal in Hardware implementieren:
  - Parallelisierung ohne Kommunikations-Overhead
  - Keine komplizierte "Logik" (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

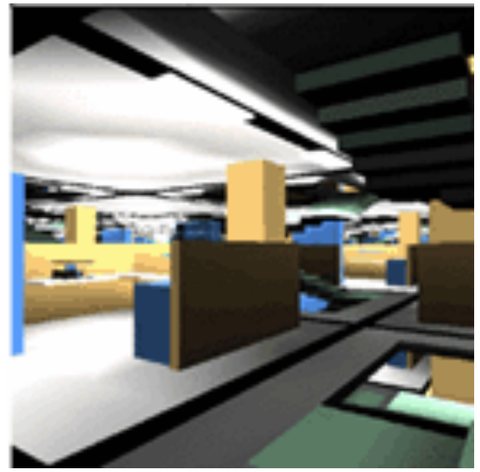
Depth-Complexity und Overdraw

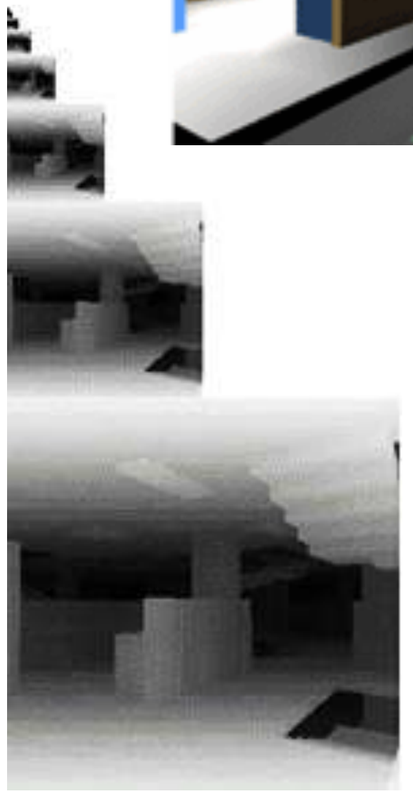


- Frage: 10 Dreiecke überdecken ein Pixel — wie groß ist die *erwartete* Anzahl Schreib-Operationen in den Color-Buffer ?
- Lösung: $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{10} \approx \ln(10) + 0.55... = 2.9289\ldots$
  - Hinweis: harmonische Reihe!
- Definition Depth-Complexity:
  - Anzahl 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 4 Pixel
  - Z-Wert jeweils auf maximalen Z-Wert setzen





7	1	0	6
0	3	1	2
3	9	1	2
9	1	2	2

farthest value

7	6
9	2

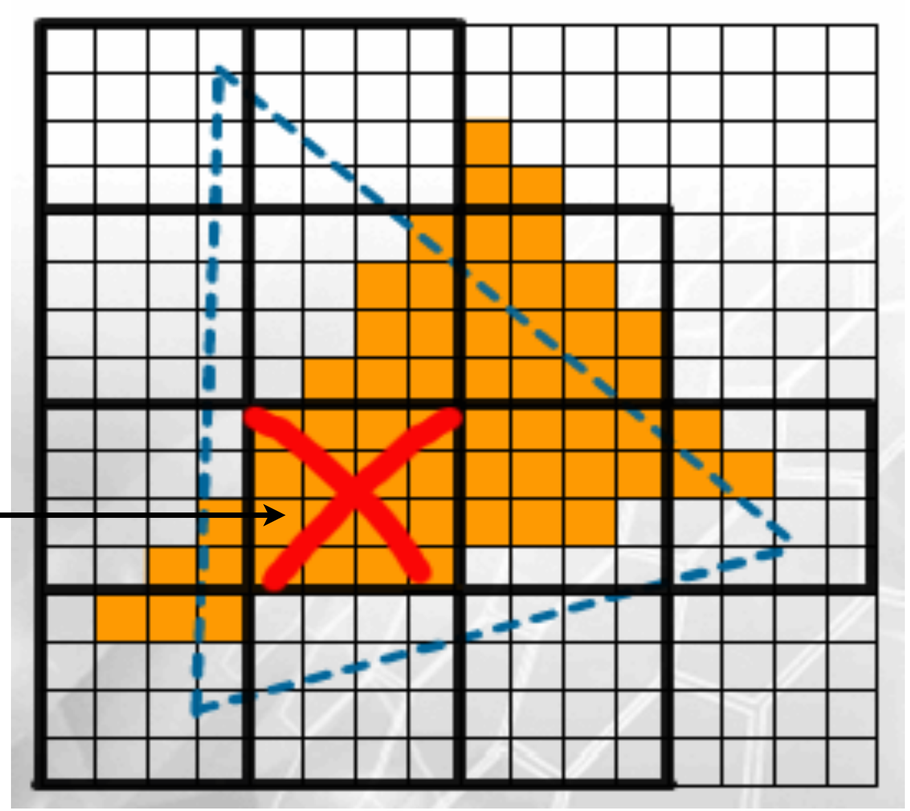
farthest value

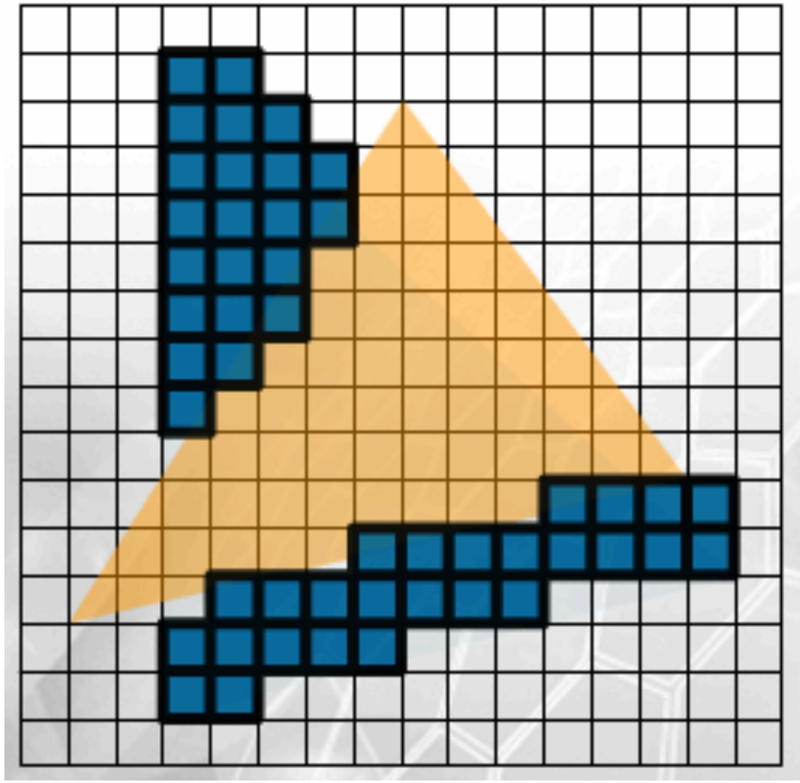
9

Beispiel mit 2 Levels

- Sei orangenes Dreieck bereits gezeichnet
- Blaues Dreieck soll (dahinter) gezeichnet werden

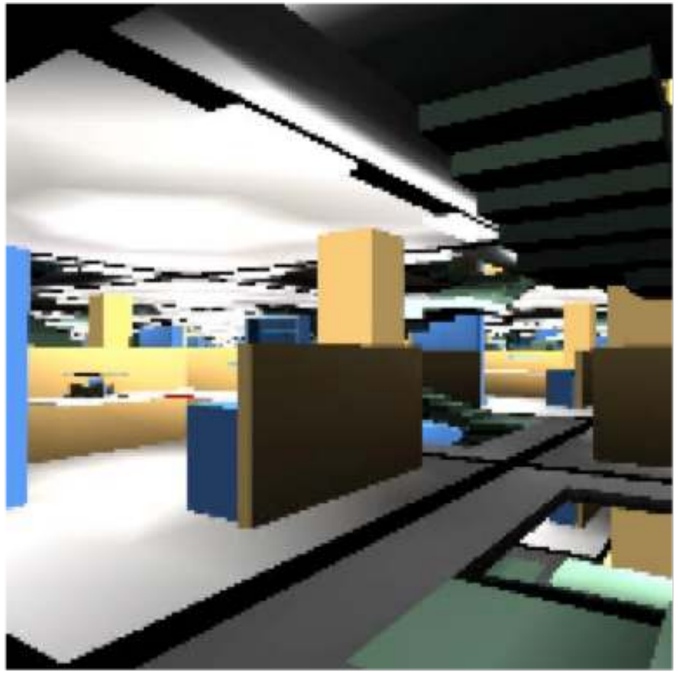
vollständig verdeckt, also verwerfe gesamten Block





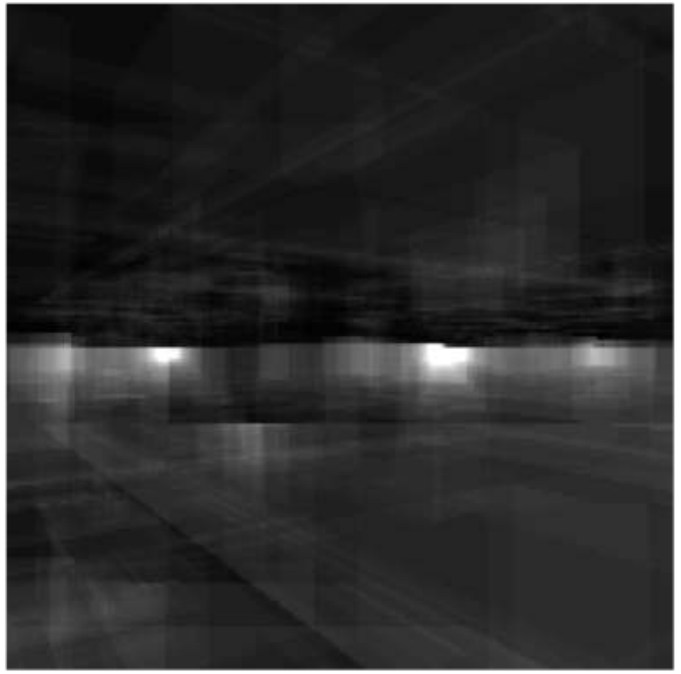
Vergleich

540 Mio Polygone



Overdraw mit
einfachem Z-Buffer

Overdraw mit HZB

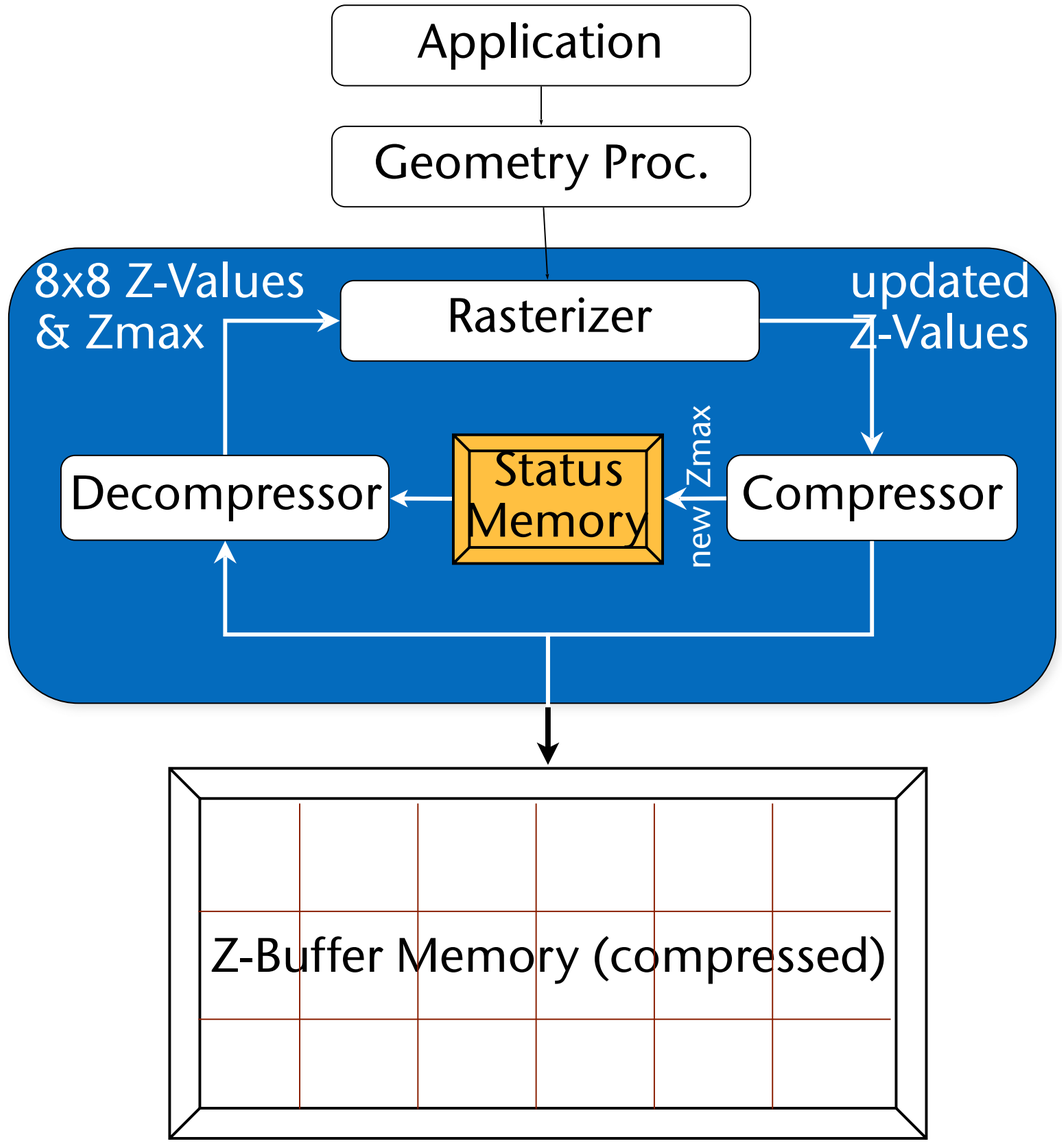


Implementierung in aktueller Graphik-Hardware

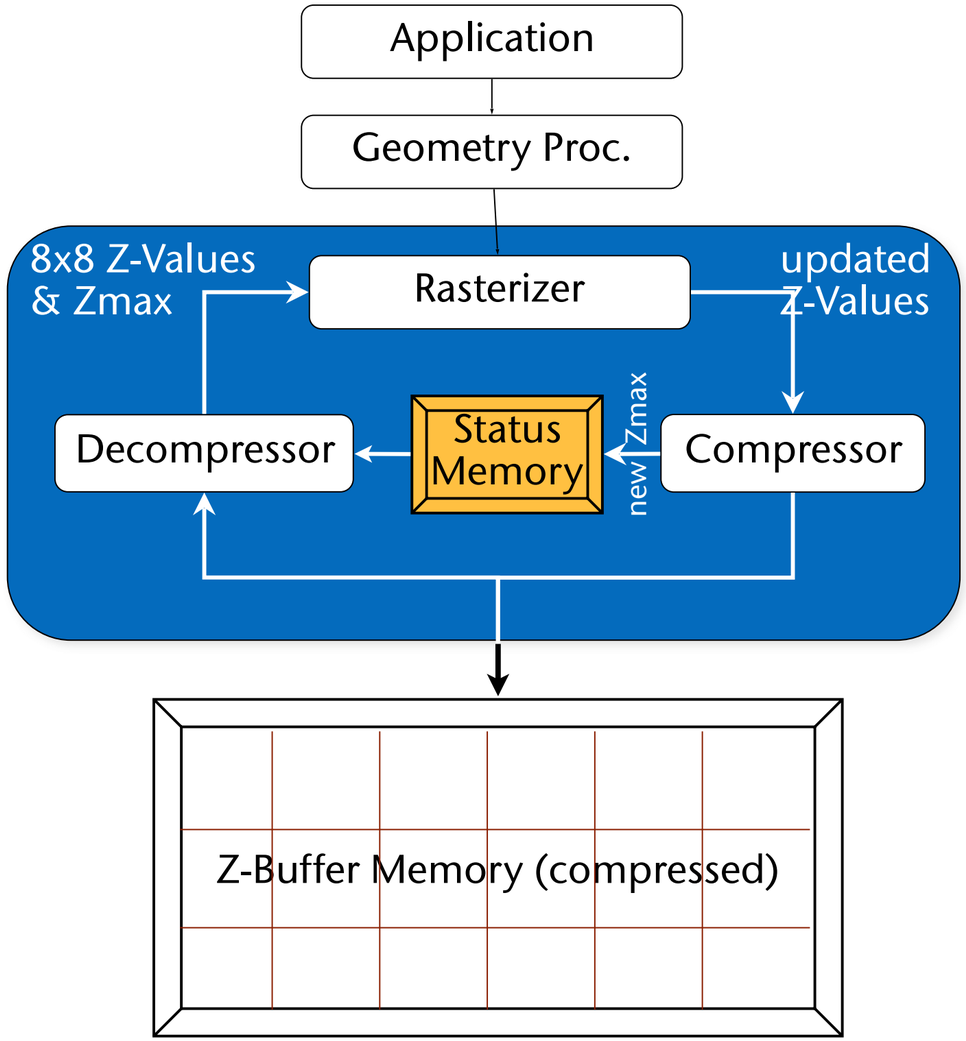
- Problem: Bandbreite zwischen Rasterizer und Speicher ist limitiert
  - Annahmen: Auflösung 1280x1024, für jedes Pixel ist depth complexity = 4
  - I/O-Aufwand pro Fragment: 1x Z-Buffer-Read + 1x Z-Buffer-Write + 1x Color-Buffer-Write + 2x Texture-Read, pro Read/Write 32 Bit

  ➢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

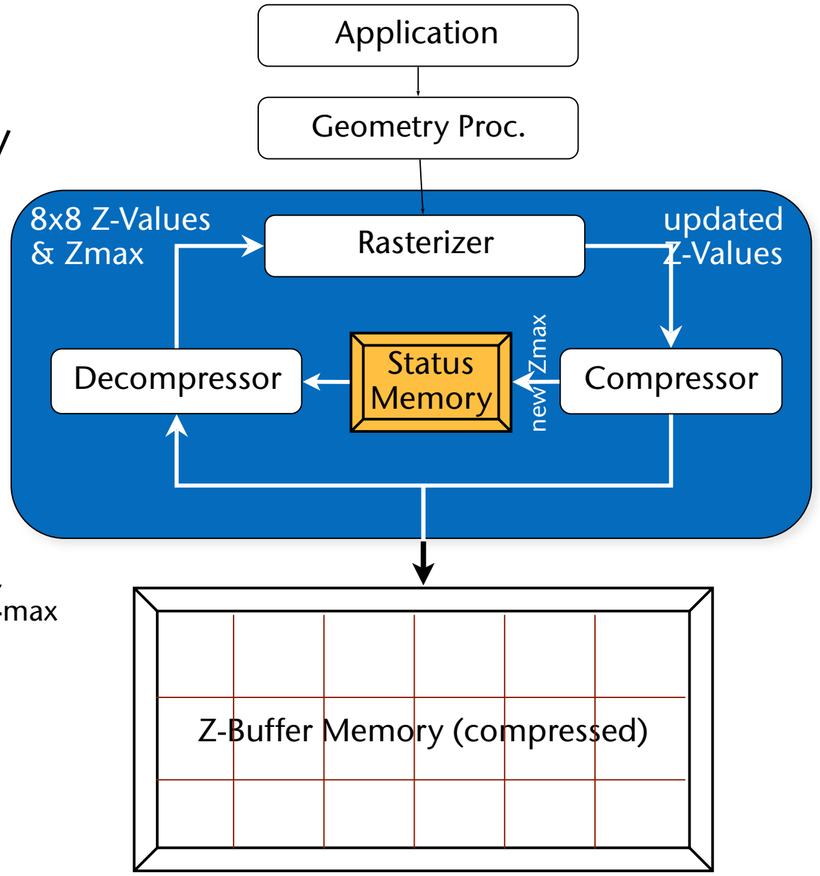
- Zentrale Idee: Status-Speicher auf dem Chip (sehr schnell) für den Zustand der Kacheln
- Pro Kachel speichere im Status-Memory:
  - Zustand ∈ {*"compressed", "uncompressed", "cleared"*}
  - Z_{max} der Kachel



- 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

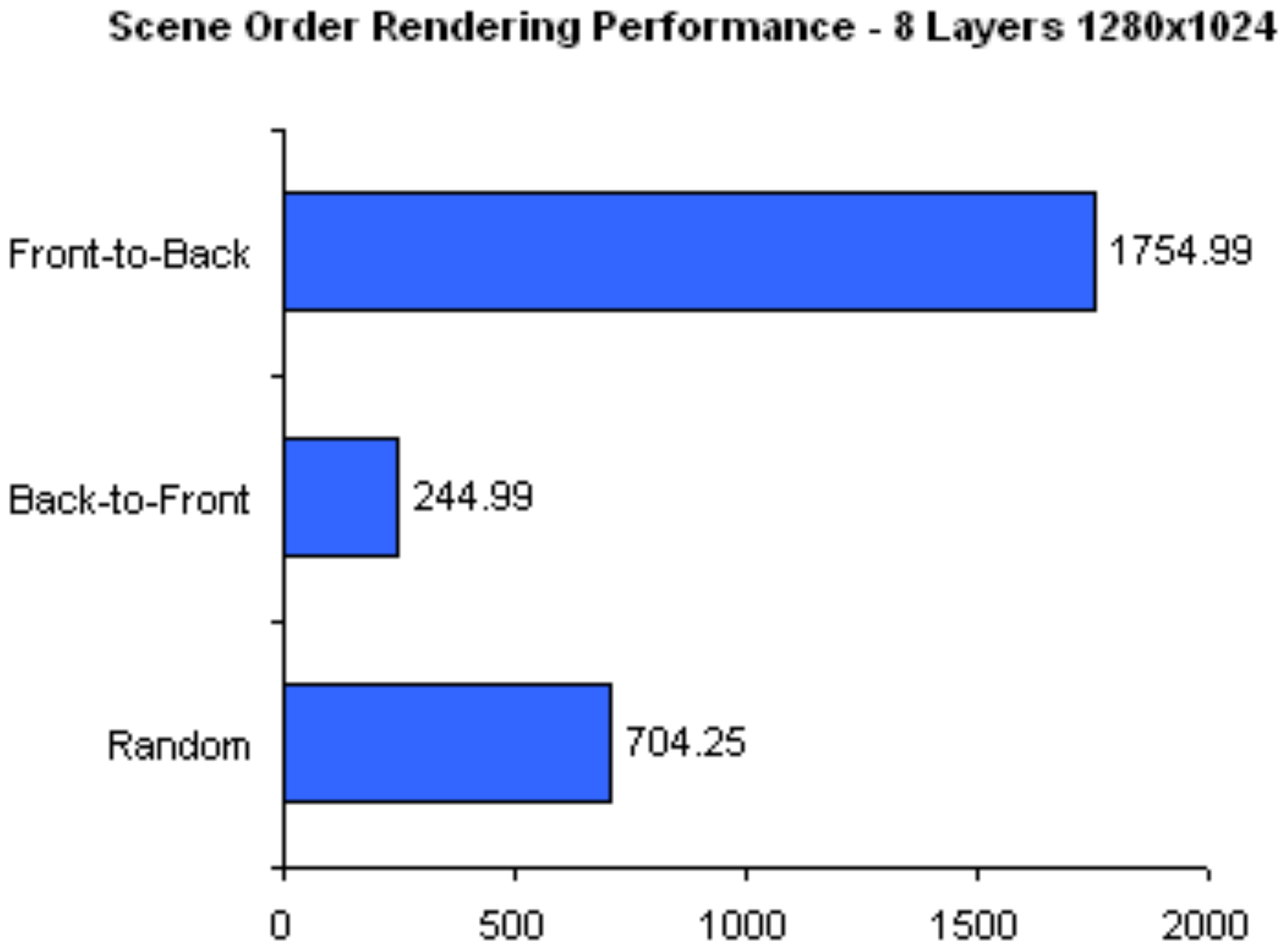


- Z-Buffer lesen:
  - Decompressor liest zuerst Z_{max} der Kachel aus dem Status Memory
  - Verschiedene Tests möglich:
    1. Teste die Z-Werte der 3 Ecken des Dreiecks gegen Z_{max}
    2. Teste die Z-Werte an den 4 Ecken der Kachel gegen Z_{max}
       - Bemerkung: diese 4 Z-Werte kann man in den Nachbarkacheln wiederverwenden
    3. Berechne alle Z-Werte der Fragmente in der Kachel und teste gegen Z_{max}
  - Fordere Z-Werte der Kachel aus dem Z-Buffer an, falls Test "fehlschlägt"
  - Falls Status der Kachel = "compressed", dekomprimiere Z-Werte vor der Weiterleitung an den Rasterizer



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

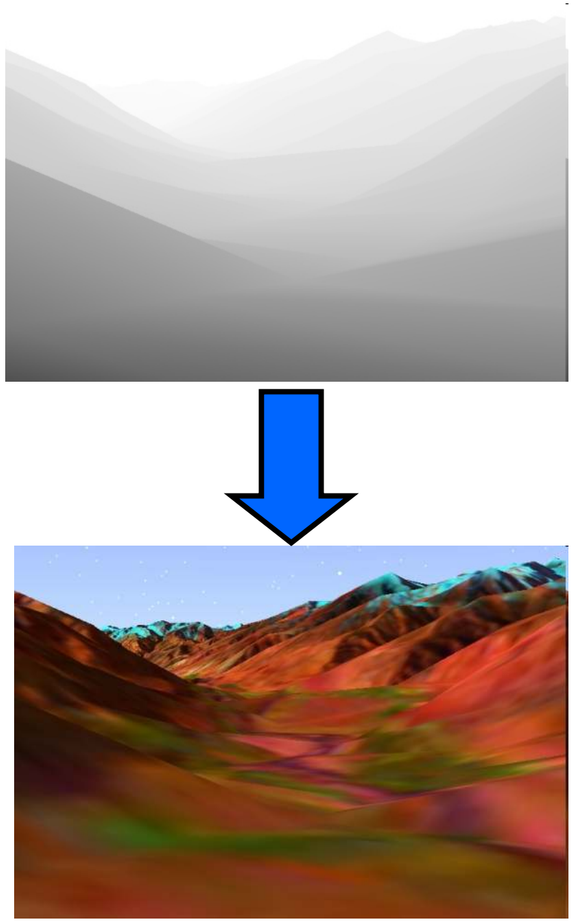
Performance-Gewinn in einer Graphikkarte



Exkurs: einfacher OpenGL-Performance-Trick "*Early-Z Pass*"

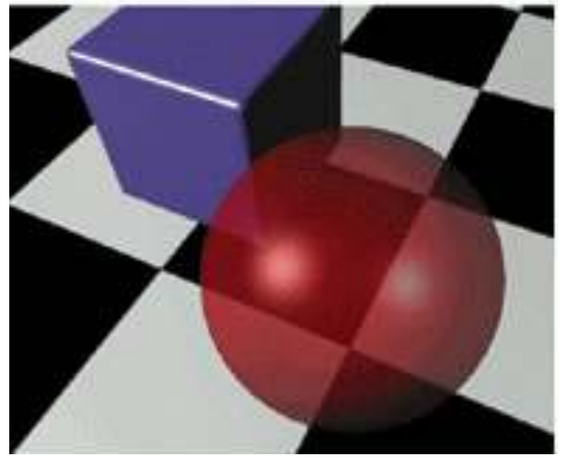
- Wegen Komprimierung des Z-Buffers: Durchsatz der GPU ist ca. *doppelt* so hoch, falls **nur** der Z-Buffer geschrieben wird (nicht Color-Buffer)
- Vorgehensweise:
  - Schalte Color-Buffer aus, nur Z-Buffer an
  - Pass 1: rendere Szene "ohne alles"
    (keine Lichtquellen, keine Farben, keine Texturen),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

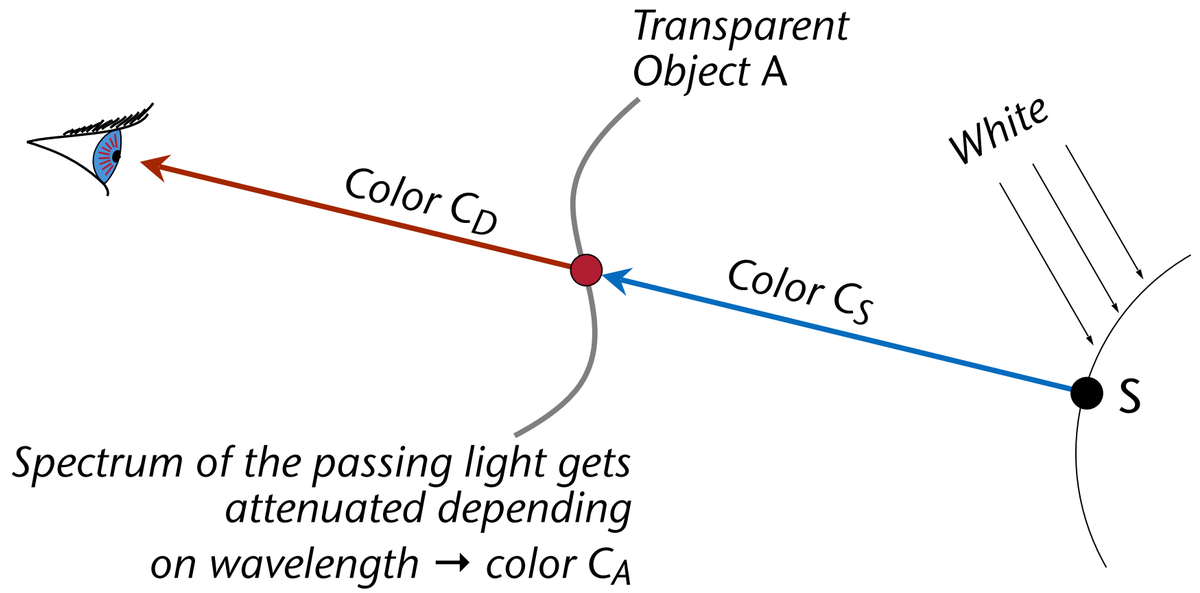
    HZB kann voll wirken ⟶ kein Overdraw



Rendering Transparent Objects

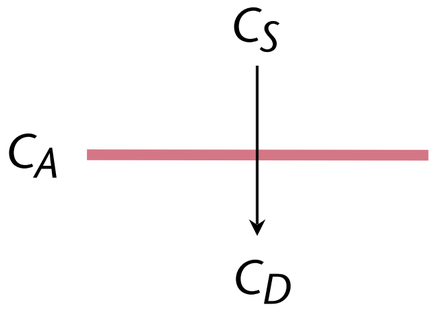
- For example: objects made of glass
- Transparency = material that lets light pass partially
- Often, some wavelengths are attenuated more than others → colored transparency
  - Extreme case: color filter in photography
- Model:





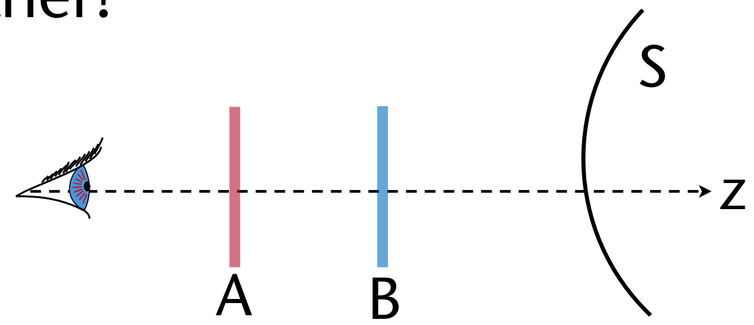
- Approximation: alpha blending
  - $\alpha \in [0, 1]$ = opacity (= opposite of transparency)
    - $\alpha = 0 \longrightarrow$ completely transparent, $\alpha = 1 \longrightarrow$ completely opaque
  - Outgoing color:

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



  - Practical implementation: α = 4th component in color vectors C = (r, g, b, α)
- 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 must be switched off!

- In Open GL:
  - Switch blending on:

```
glEnable( GL_BLEND );
```

  - Determine blending function:

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

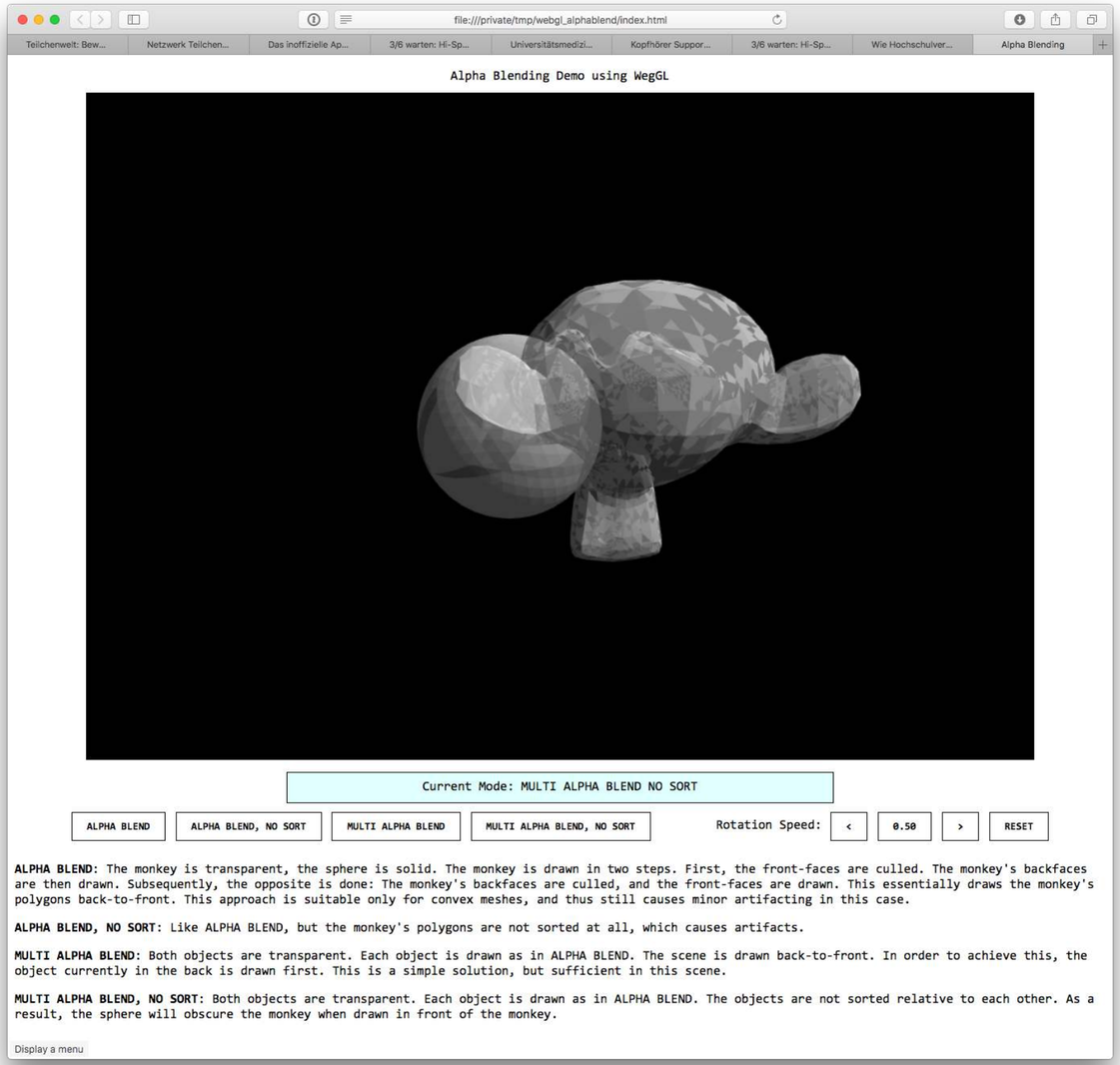
Parameter **GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA** yield

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

where C_D = color from frame buffer

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

Object-Space- vs. Image-Space-Algorithmen

- Image Space Algorithmen: arbeitet im diskreten(!) 2D-Bildraum
  - Hier: bestimme für jeden Pixel, welches Objekt sichtbar ist
  - Funktioniert auch bei dynamischen Szenen, da i.A. wenig / keine Hilfsdatenstrukturen
  - Beispiel: Z-Buffer, hierarchischer Z-Buffer
- Object Space Algorithmen: ganz allg. Algorithmen, die direkt auf den 3D-Koord. der Objekte arbeiten (mit Floating-Point)
  - Hier: bestimme vor dem Abschicken von OpenGL-Befehlen, welche Polygone andere verdecken; berechne den jeweils sichtbaren Teil eines Polygons
  - Berechnung basiert oft auf dem Aufbau komplexer Hilfsdatenstrukturen
  - Funktioniert besser bei statischen Szenen

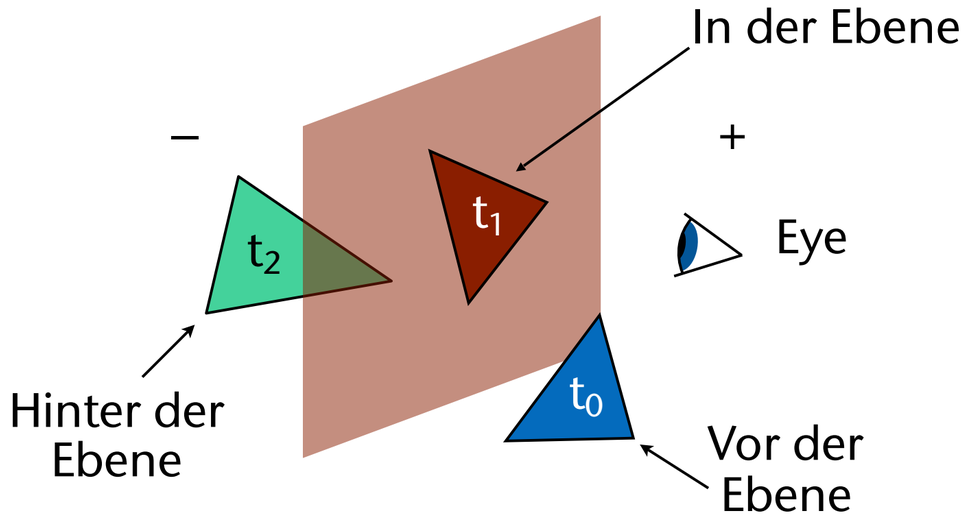
Binary Space Partition (BSP) Tree [ca. 1982]

- Ein Object-Space-Algorithmus/-Datenstruktur
  - *For depth sorting*
  - Generell für Polygon-Sortierung bzgl. eines beliebigen Punktes im Raum
- Ansatz: rekursive Unterteilung des Raumes durch Ebenen, die durch die Polygone der Szene induziert werden (Autopartition)
- Sehr effizient für statische Szenen
- Ermöglicht sehr schnell Hidden-Surface-Elimination für alle Viewpoints mittels Painter's Algorithm
- Ursprünglich fürs Rendering ohne Z-Buffer entwickelt, heute immer noch eine sehr wichtige Datenstruktur in der CG
  - Wurde sogar 1996 noch im Spiel Quake (und auch Quake II?) verwendet für Hidden-Surface-Elimination!

Grundlegende Idee

- Annahme (vorerst): keine zwei Polygone schneiden sich
- F_p sei die implizite Gleichung der Ebene, in der das Polygon p liegt (supporting plane)
- Ein Hidden-Surface-Algo für folgende Szene:

```
if F_t1(eye) > 0 :
    draw t2
    draw t1
    draw t0
else:
    draw t0
    draw t1
    draw t2
```



  - F_{t1} = Ebenengleichung für supporting plane von t_1
- Funktioniert für beliebigen Viewpoint!

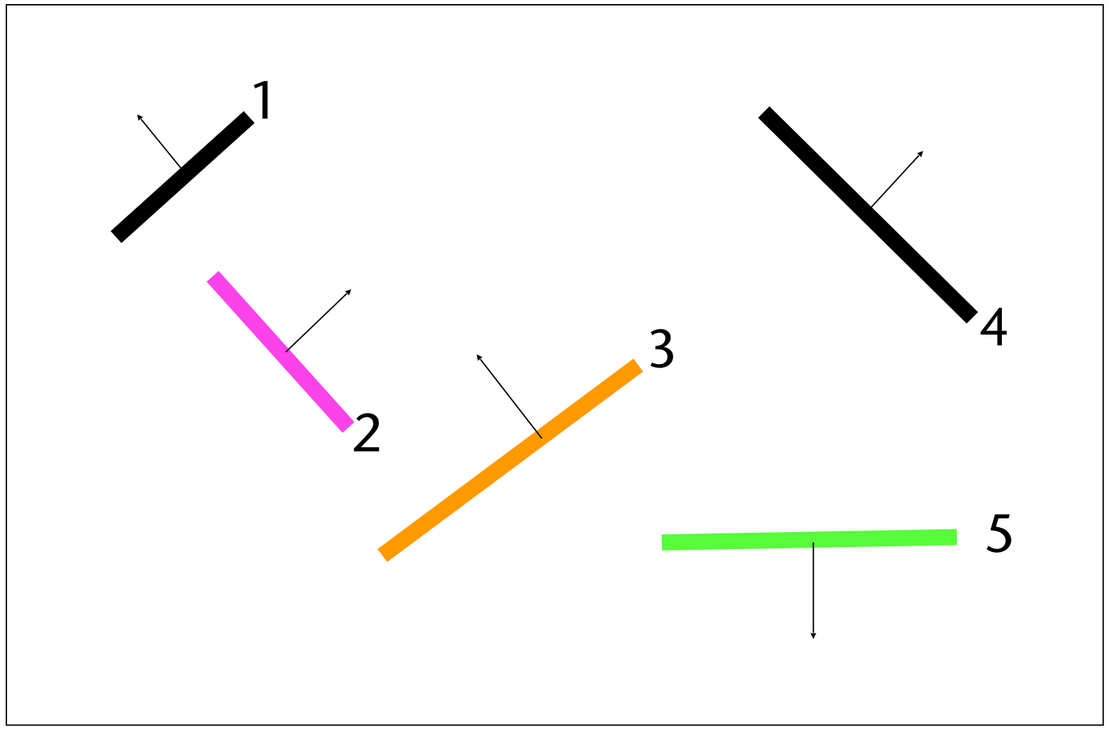
Rekursiver Aufbau und (informelle) Definition eines BSP Tree

1. Wähle ein (zufälliges) Polygon; erzeuge damit folgenden Wurzelknoten:
   - Speichere dieses Polygon in der Wurzel
   - Das Polygon definiert gleichzeitig eine sog. Splitting-Ebene
   - Speichere diese auch in der Wurzel (Normale **n** und Offset *d*)
   - Speichere außerdem alle weitere Polygone, die exakt **in** dieser Ebene liegen
2. Partitioniere die Menge der restlichen Polygone in zwei Teilmengen, je nachdem auf welcher Seite sie liegen
   - Schneidet ein Polygon die Ebene, dann unterteile dieses in zwei Polygone, jeweils ein Teil auf einer Seite (*splitting*)
3. Baue rekursiv je einen BSP für alle Polygone auf der negativen bzw. positiven Seite und hänge diese als Kinder an die Wurzel
4. Stop, wenn ein Unterbaum nur noch ein Polygon enthält

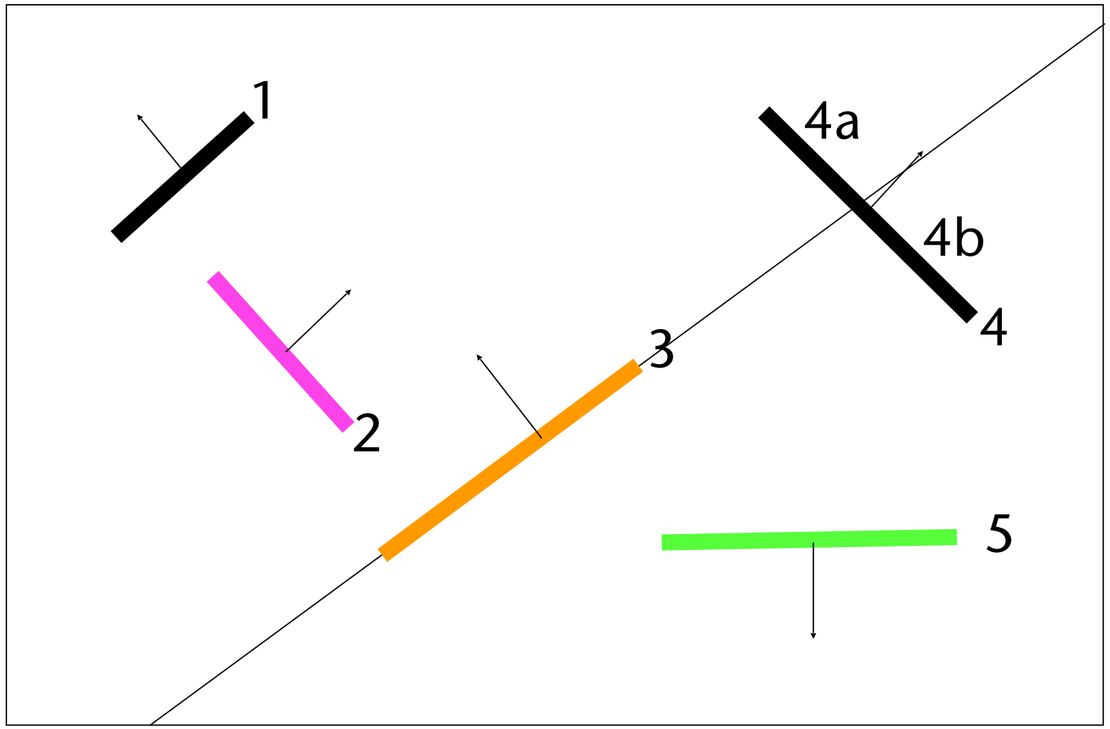
- NB: diese Art BSP heißt Auto-Partition

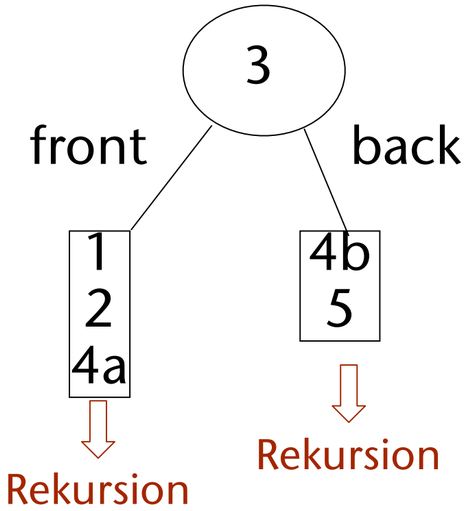
Beispiel

- Ausgangsszene:

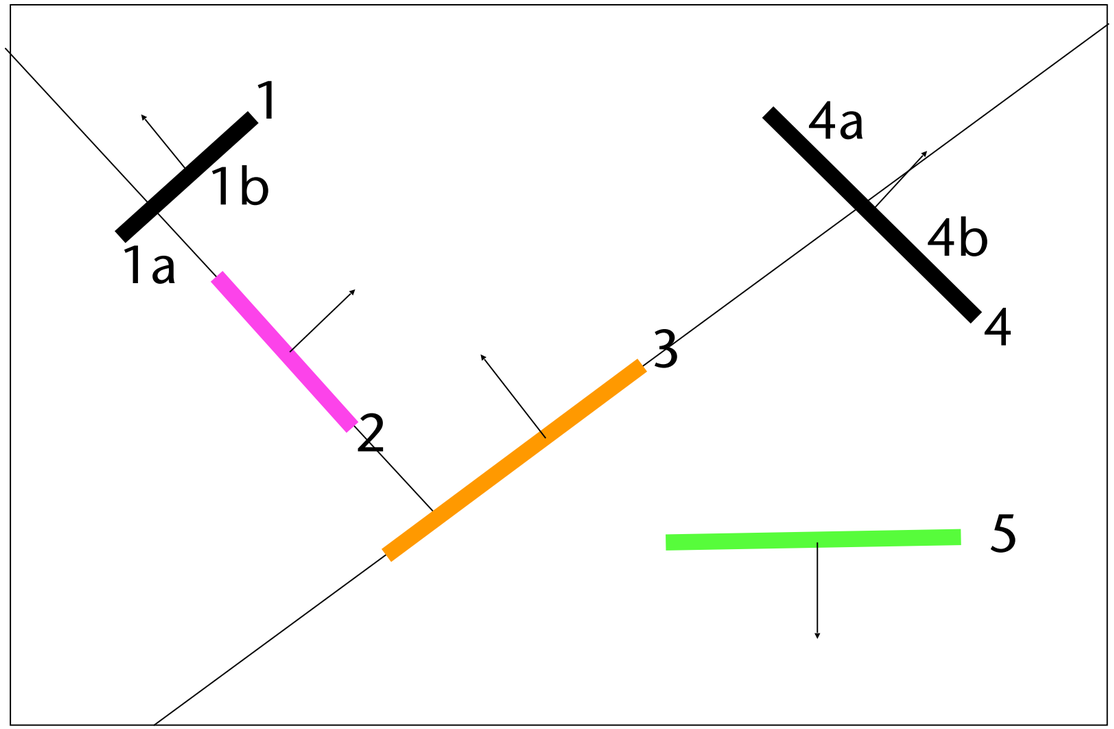


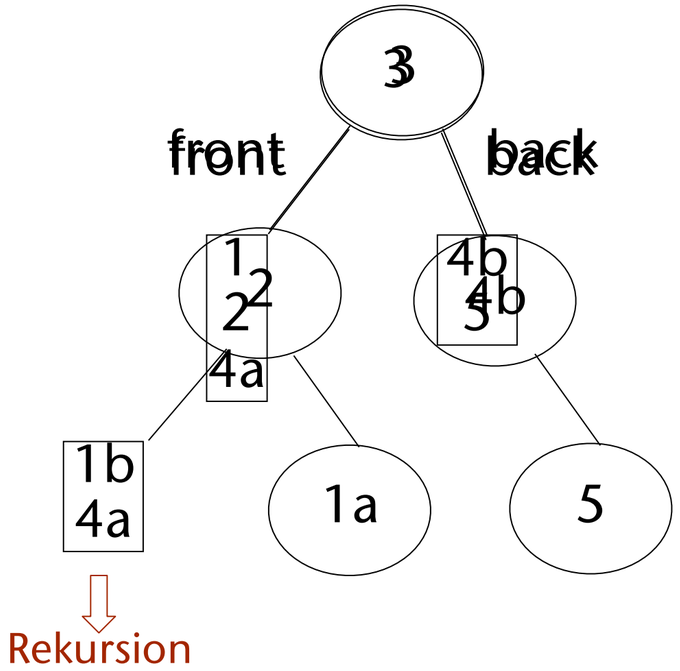
- Wähle z.B. Polygon 3 als Wurzelelement



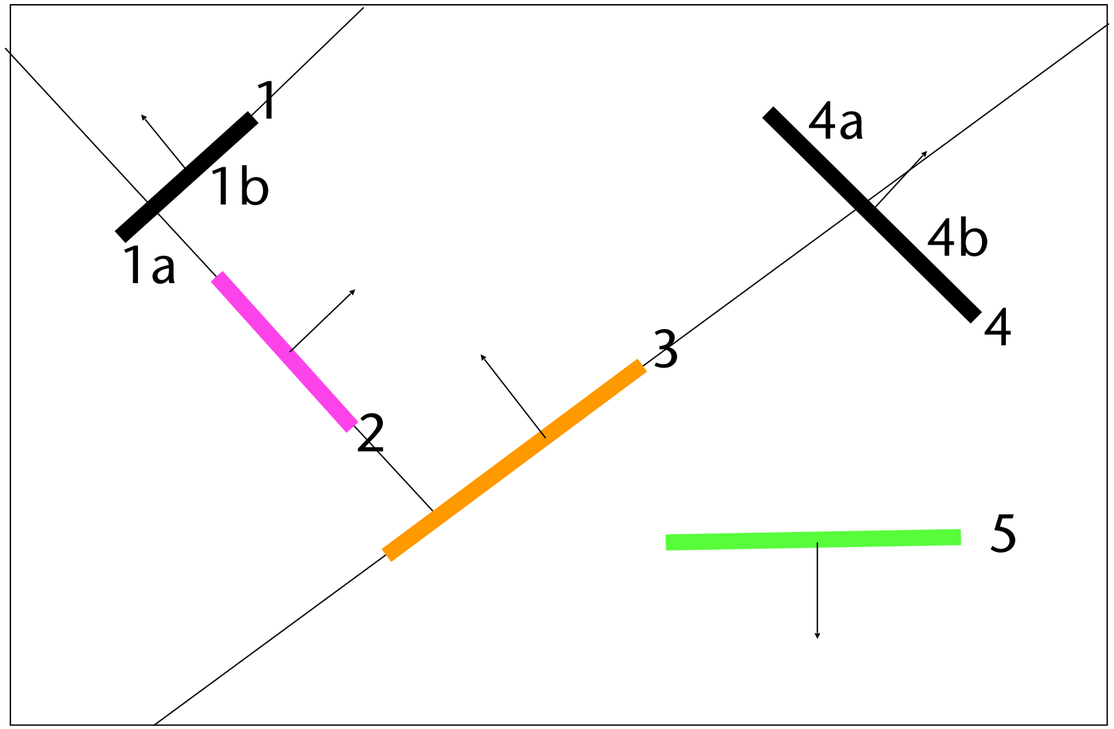


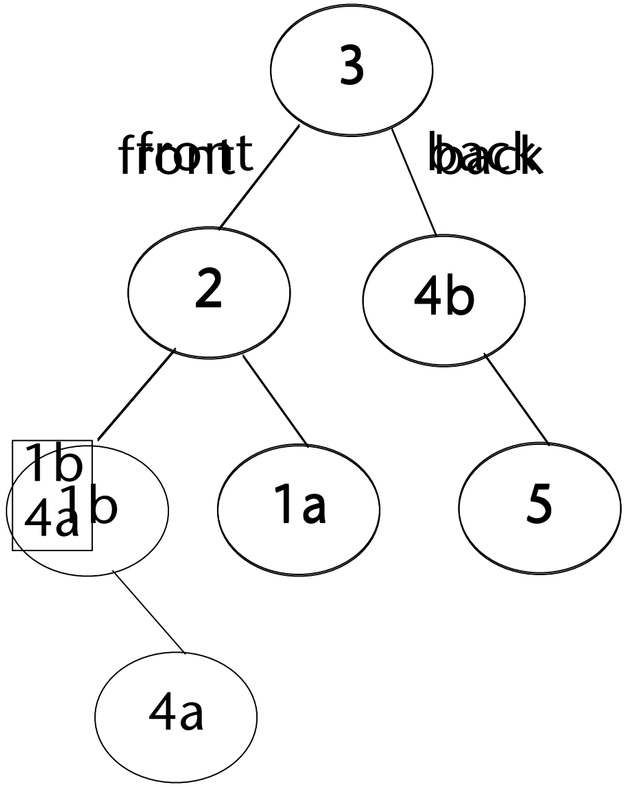
- Wähle Polygon 2 und 4b als nächste Knoten





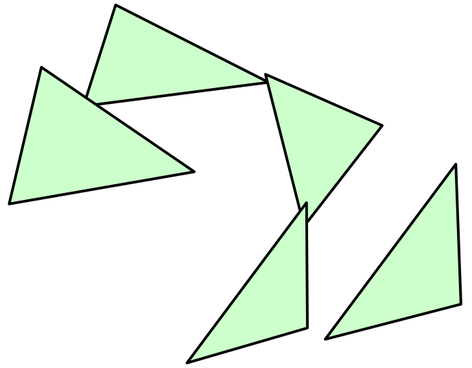
- Nun wähle Polygon 1b als Knoten



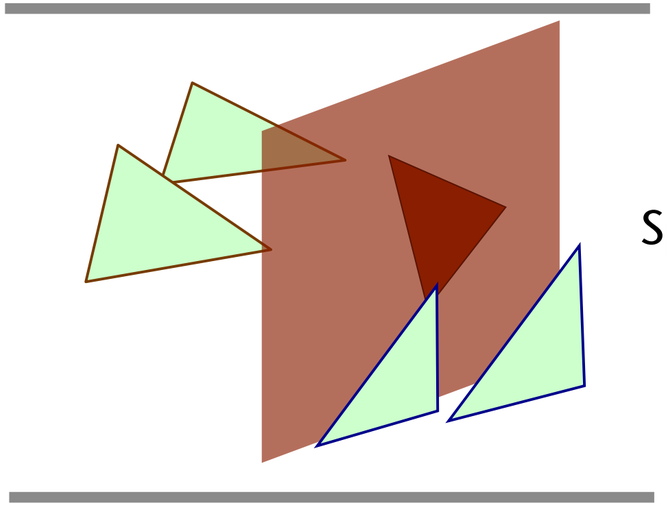


Der Pseudo-Code zur Konstruktion

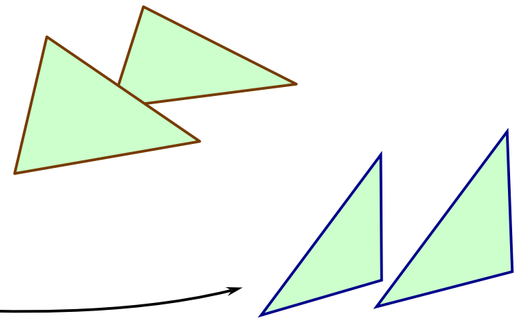
```
buildBSP( polygons ):
  node = new Node
  node.t = polygons[0]     // or pick random
  front = new PolygonSet
  back  = new PolygonSet
  if polygons.size > 1:
    for p in polygons:
      if p lies exactly in node.plane: ...
      if p intersects node.plane :
        split p
        do following test with each part
      if p on back-side of node.plane:
        back.push( p )
      else
        front.push( p )
  node.front = buildBSP( front )
  node.back = buildBSP( back )
  return node
```



Polygon-Menge



Splitting



Rekursions-schritt

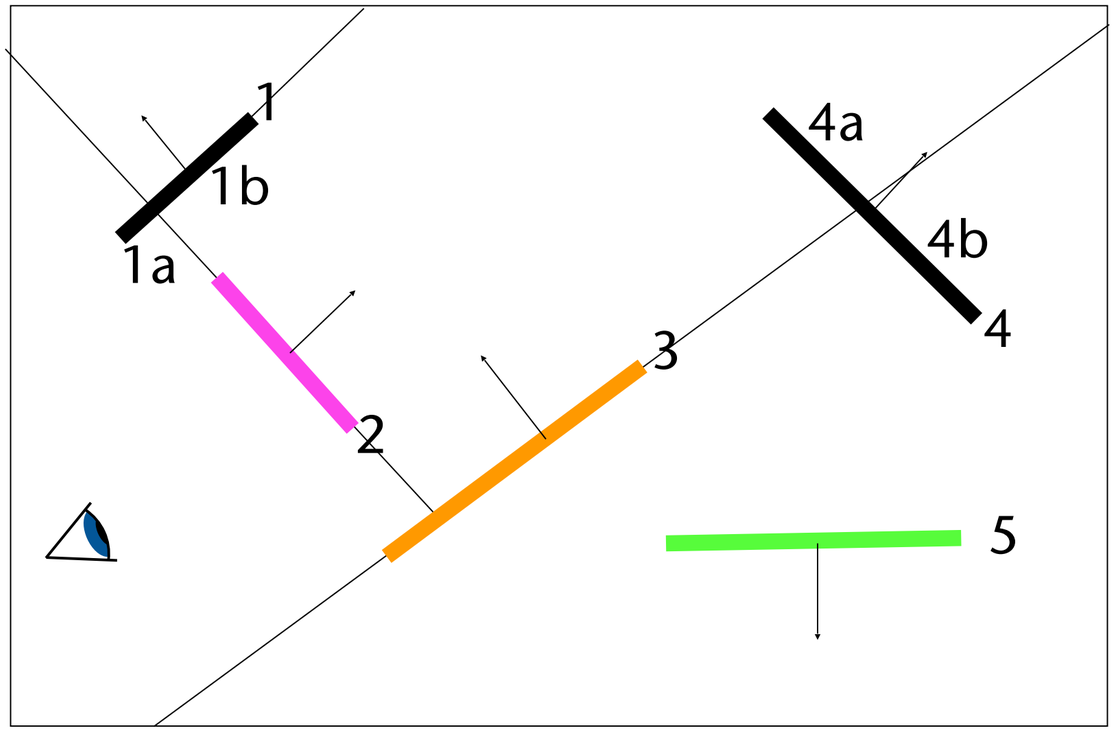
Der Rendering-Algorithmus mit BSPs

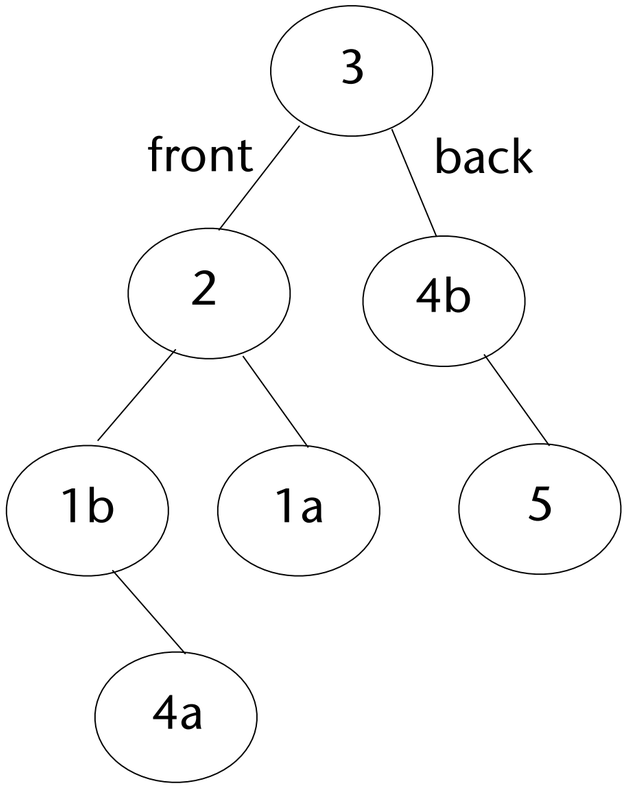
- Rendering (back to front):
  - Beginne bei der Wurzel
  1. Zeichne Polygone rekursiv auf der Gegenseite vom Viewpoint aus
  2. Zeichne Polygon(e) im Knoten
  3. Zeichne rekursiv Polygone auf der selben Seite wie Viewpoint

```
draw(node, eye):
  if node is child:
    rasterize( node.triangle )
  if node.plane(eye) < 0 :
    draw( node.pos, eye )
    rasterize( node.triangle )
    draw( node.neg, eye )
  else
    draw( node.neg, eye )
    rasterize( node.triangle )
    draw( node.pos, eye )
```

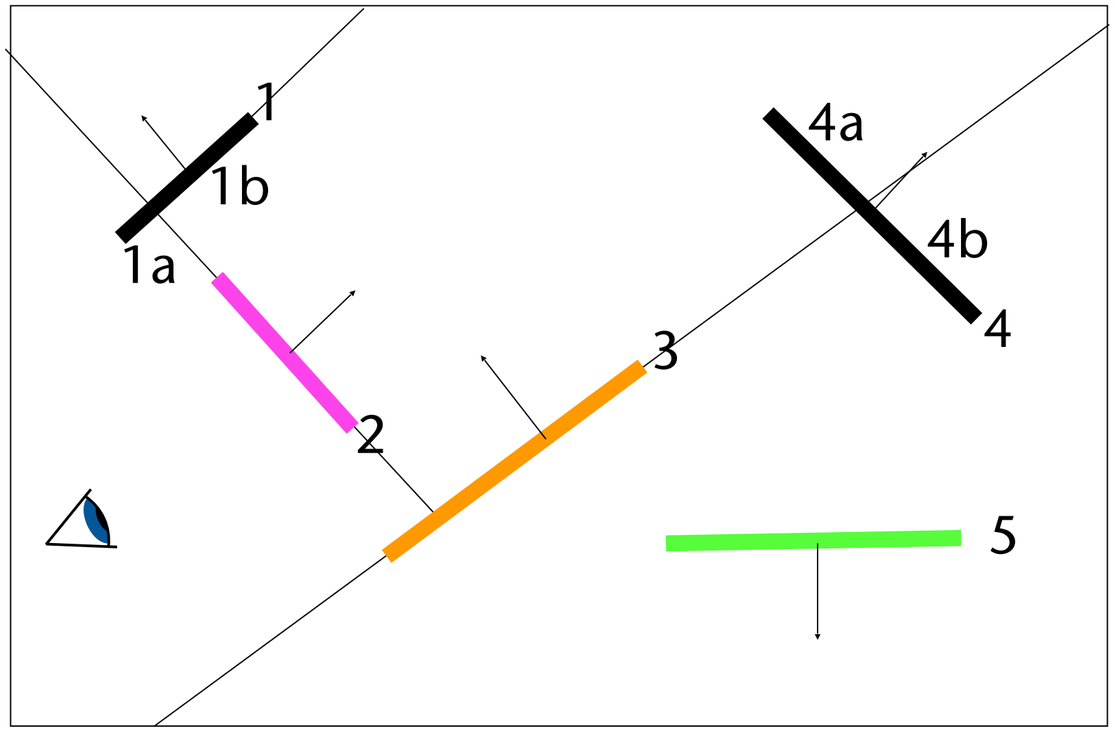
Beispiel fürs Rendering mittels BSP (Traversierung)

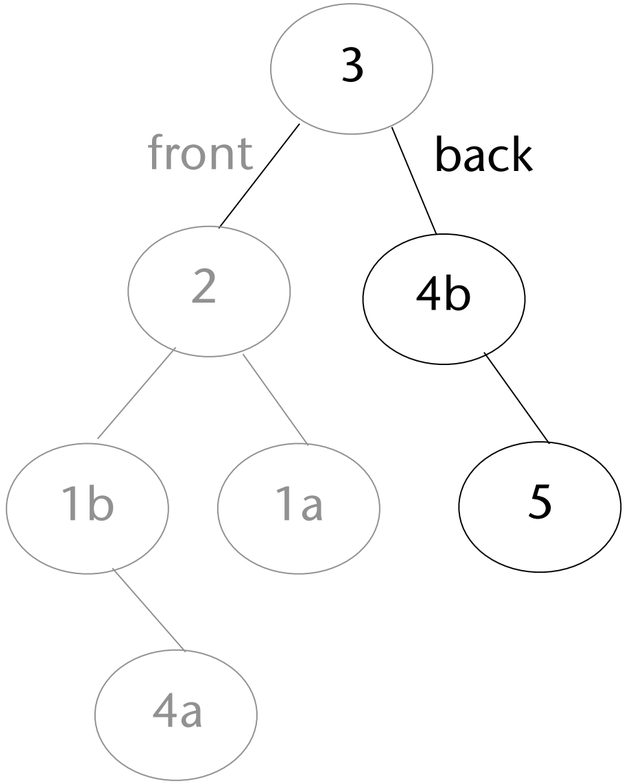
- Angenommen, der Viewpoint befindet sich wie hier dargestellt



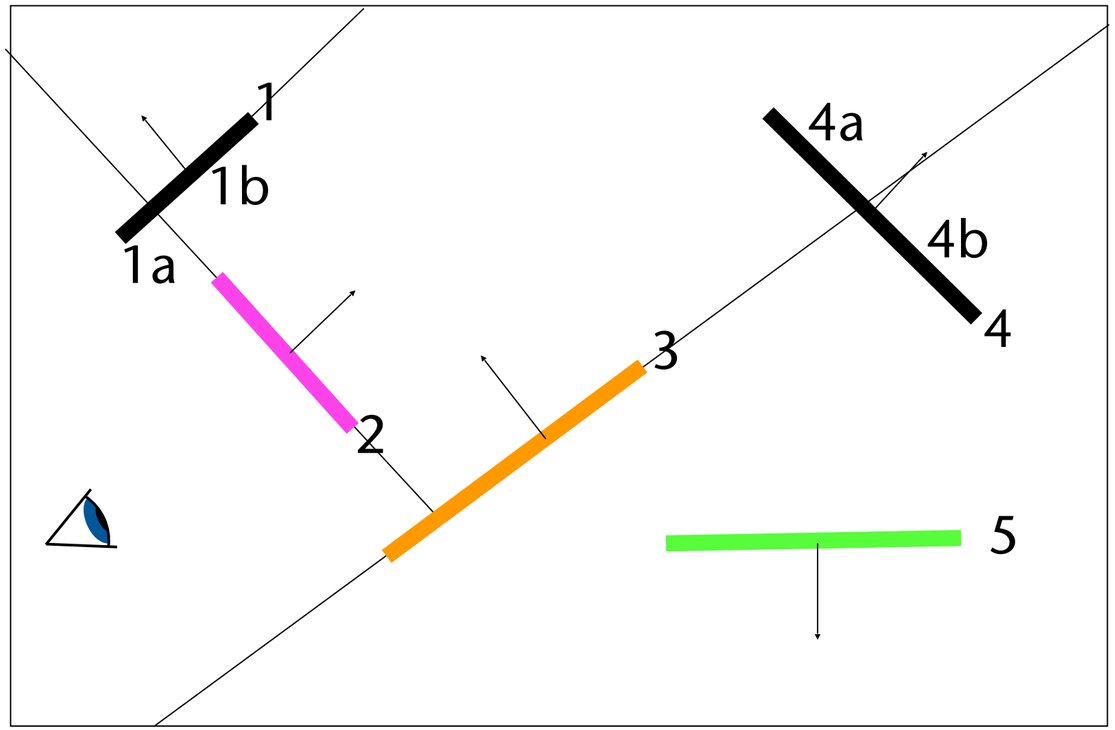


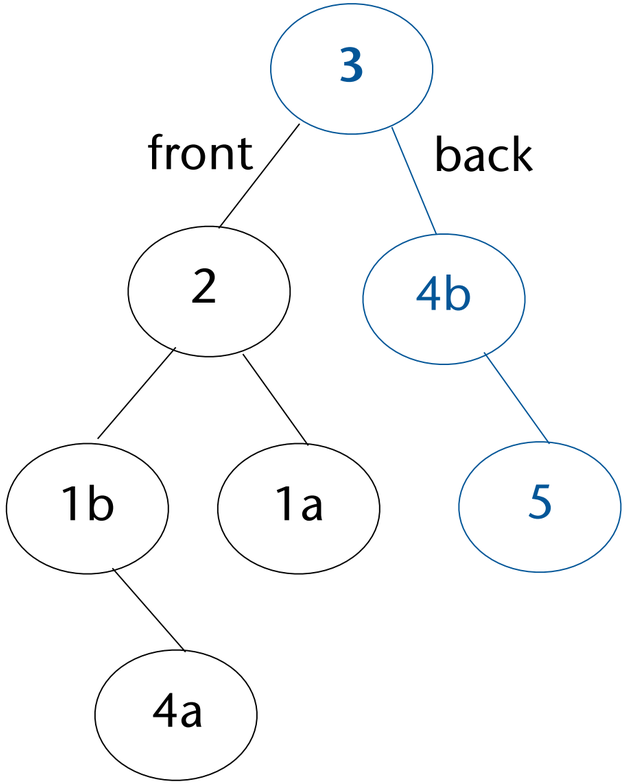
1. Viewpoint liegt auf der Vorderseite von 3 ⟶ zuerst die Polygone auf der Rückseite von 3 rendern
2. Rendere zuerst die Polygone auf der Vorderseite von 4b (nichts), dann 4b, dann auf der Rückseite von 4b (Polygon 5)



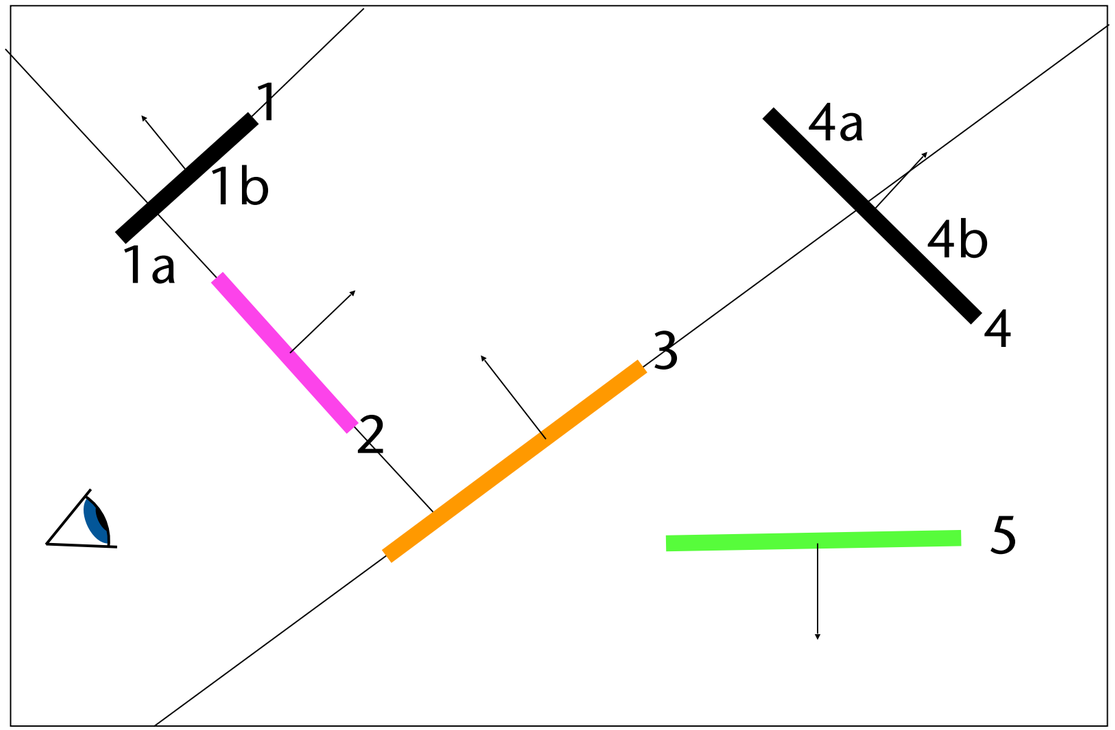


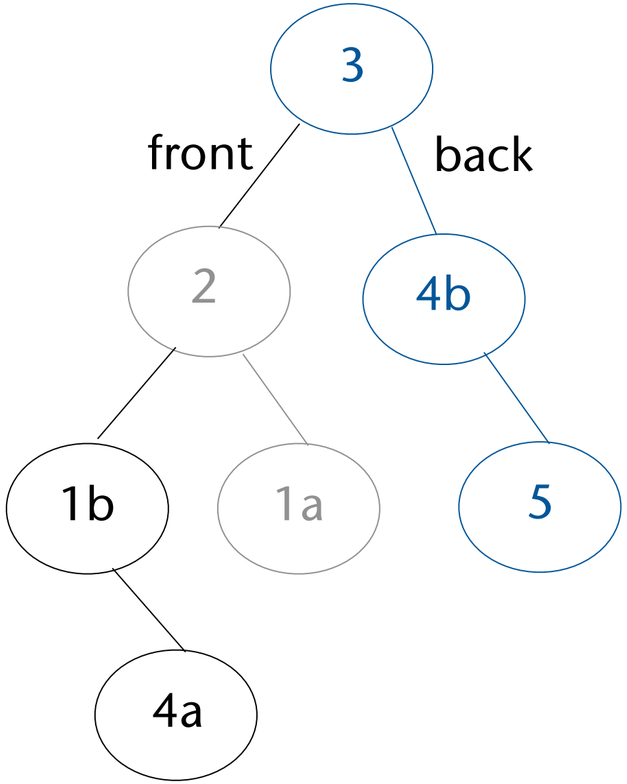
3. Nun (nachdem die Traversierung aus der Rekursion in den rechten Teilbaum zurück kommt) rendern wir Polygon 3 und dann die Vorderseite von 3 (= linker Teilbaum von 3)



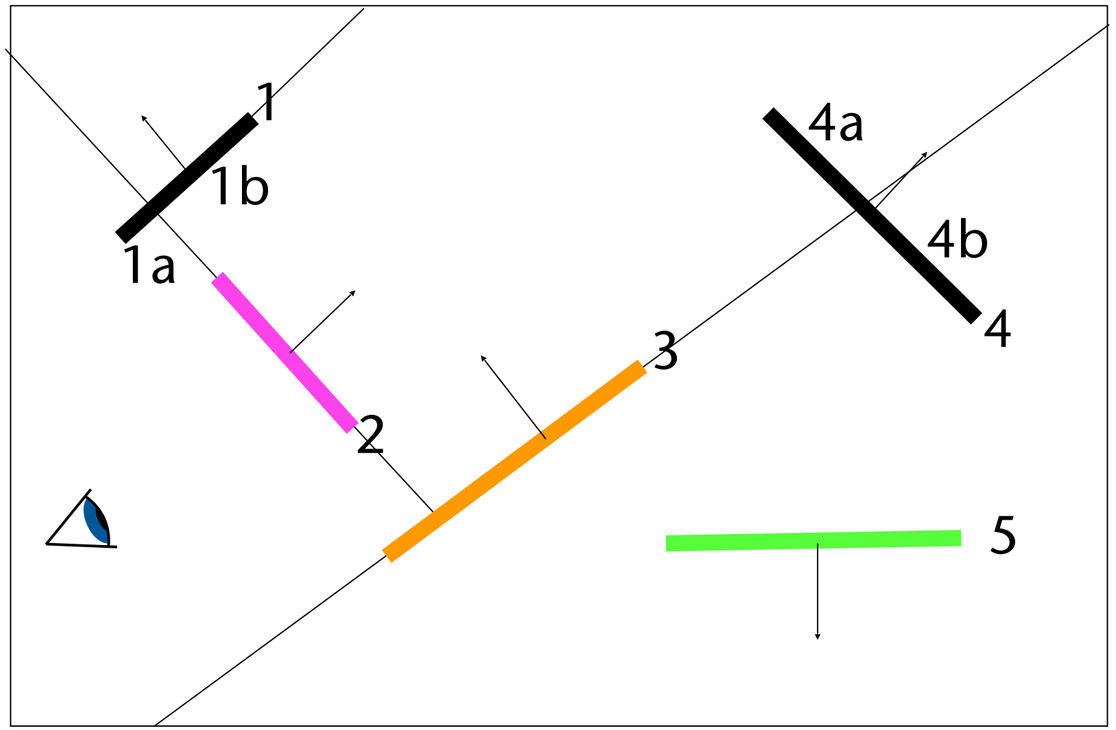


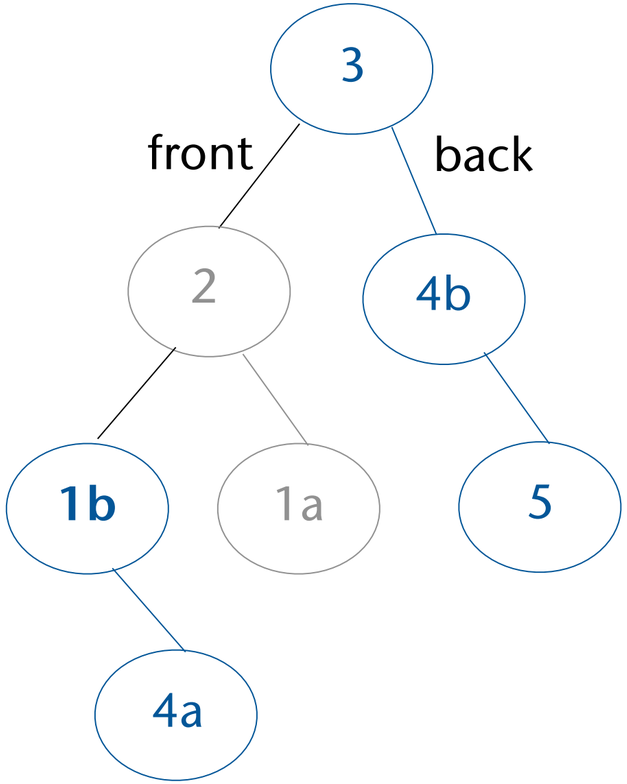
4. Der Viewpoint liegt auf der Rückseite von 2, somit werden erst die Polygone auf der Vorderseite von 2 gerendert





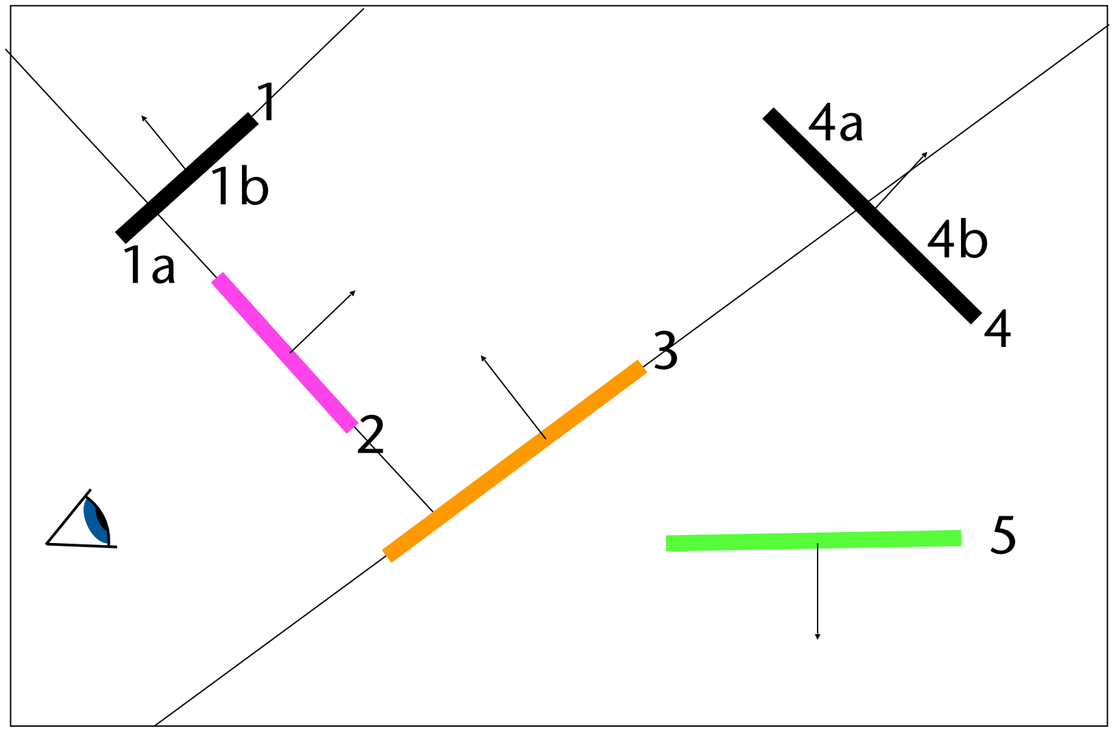
5. Der Viewpoint liegt auf der Rückseite von 1b, somit werden erst die Polygone auf der Vorderseite von 1b gezeichnet (nichts)
6. Rendere dann 1b, danach die Polygone auf der Rückseite von 1b, also 4a

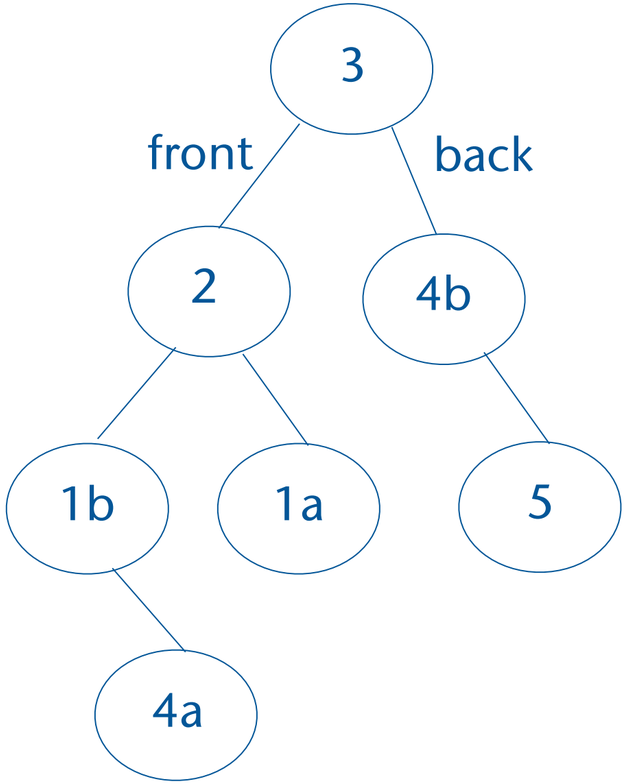




7. Danach rendere 2, danach die Polygone auf der Rückseite von 2, also 1a

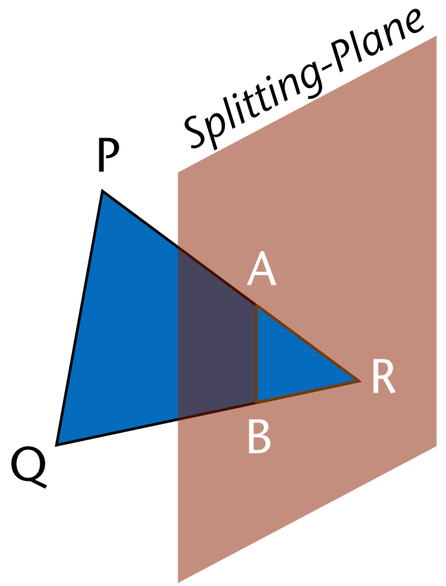
- Ergibt Gesamtreihenfolge: 4b, 5, 3, 1b, 4a, 2, 1a





Splitting von Dreiecken

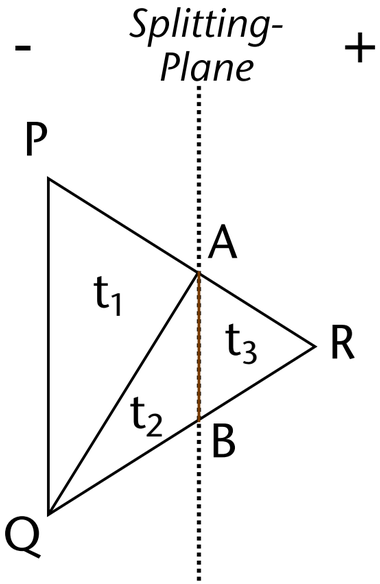
- Dreieck schneidet die Ebene ⟶ unterteilen
  - I.A. entstehen 3 neue Teil-Dreiecke
- Achtung: Reihenfolge der Eckpunkte muß beibehalten werden, damit sich Normale nicht ändert!
- O.B.d.A. liegt R allein auf einer Seite der Ebene und es gilt $F_{plane}(R) > 0$:
  - Füge t_1, t_2 in den negativen Unterbaum ein
  - Füge t_3 in den positiven Unterbaum ein



$t_1 = (P, Q, A)$

$t_2 = (Q, B, A)$

$t_3 = (A, B, R)$



- Wie bestimmt man A und B ?
  - A: Schnittpunkt der Gerade zwischen P und R und der Ebene F_{plane}
  - Verwende Parameterform der Geradengleichung

$$X(t) = P + t(R - P)$$

  - Setze X in die Ebenengleichung ein:

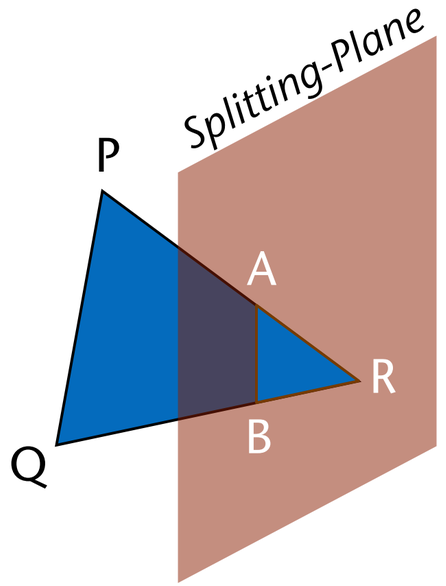
$$f_{\text{plane}}(X) = (\mathbf{n} \cdot X) - d$$

$$= \mathbf{n} \cdot (P + t(R - P)) - d \stackrel{!}{=} 0$$

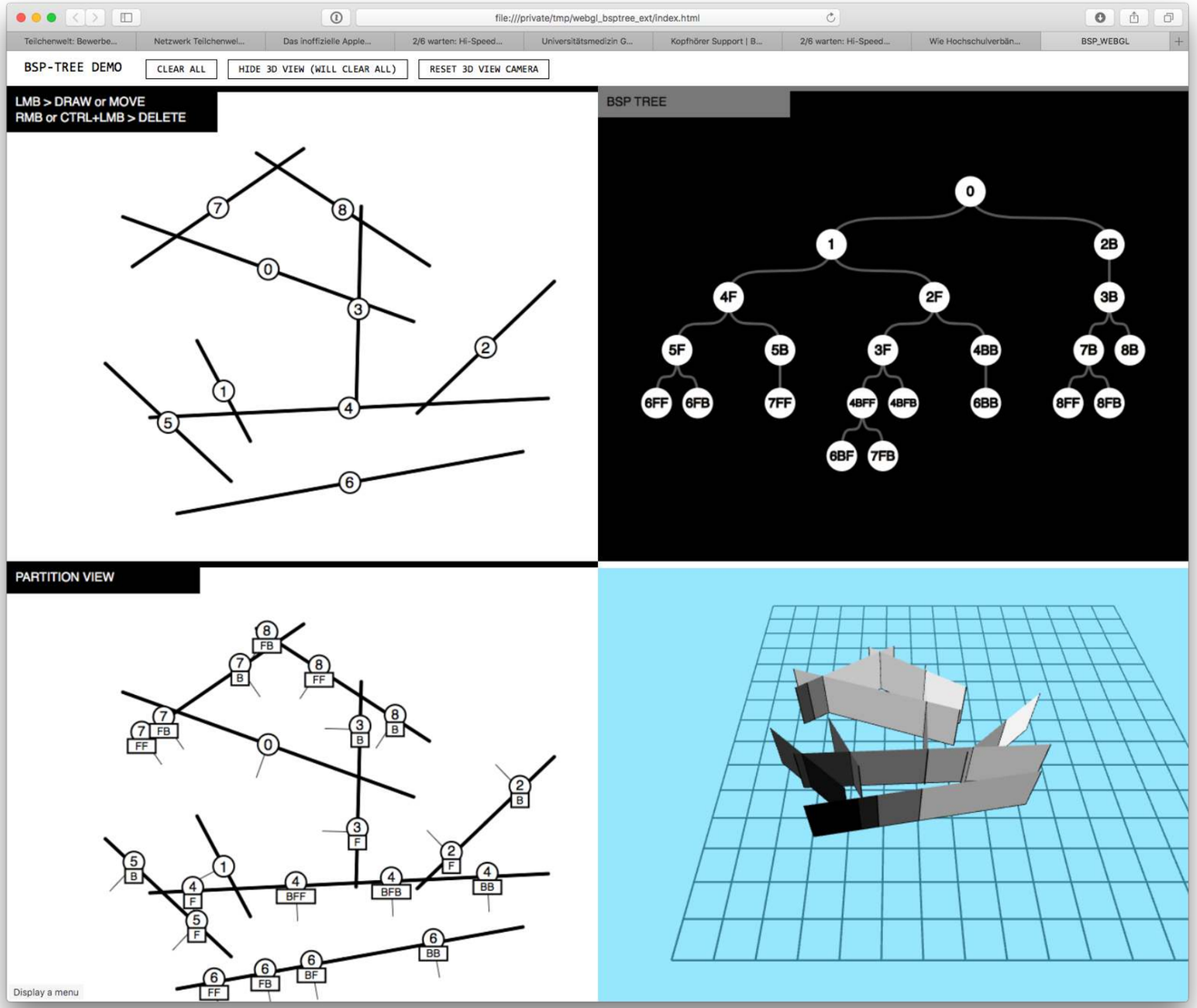
  - Berechne t und setze es in X(t) ein, um A zu berechnen:

$$t = \frac{d - (\mathbf{n} \cdot P)}{\mathbf{n} \cdot (R - P)}$$

  - Wiederhole für B



Demo


file:///private/tmp/webgl_bsptree_ext/index.html
BSP-TREE DEMO
CLEAR ALL
HIDE 3D VIEW (WILL CLEAR ALL)
RESET 3D VIEW CAMERA
LMB > DRAW or MOVE
RMB or CTRL+LMB > DELETE
BSP TREE
PARTITION VIEW
Display a menu

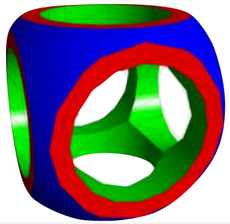
Abstrakte Sichtweise auf einen BSP

- BSP-Konstruktion: liefert zu einer gegebenen Menge von Polygonen eine Datenstruktur, so dass zu jedem beliebigen Query-Punkt eine Sortierung der Polygonen (bzw. Polygonteile) ausgegeben werden kann
- BSP-Traversierung: liefert zu einem bestimmten Query-Punkt eine sortierte Folge der Polygone (bzw. Polygonteile) ausgegeben wird
  - Das funktioniert von fern zu nah, aber auch umgekehrt
  - Die Blickrichtung ist dabei völlig irrelevant

Zusammenfassung

Vorteile

- Sehr effiziente Datenstruktur, um Polygone bzgl. eines Punktes zu sortieren
- Unabhängig vom Viewpoint
- Auch für andere Aufgaben geeignet
  - Z.B. Konstruktion von Schnitt u.a. Boole'sche Operationen auf Objekten (CSG) ⟶ VL "Comp. Geometry"



Nachteile

- Evtl. viele kleine Polygonteile (wg Splitting)
- Bei Rendering: starkes Over-drawing (viele Pixel werden "umsonst" geschrieben)
  - Wg. Back-to-front–Sortierung
- Schwierig, den Baum ausgeglichen zu halten
- Ungeeignet für dynamische Szenen

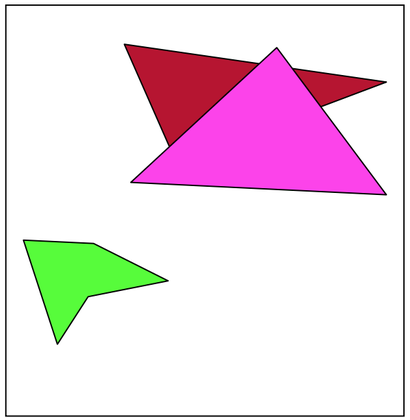
Weitere Anwendung des BSP Tree: Camera "Collision Test"

- Aufgabe: gegeben sind ...
  - eine Menge Polygone (Szene des Spiels)
  - zwei Punkte P_1, P_2 (= alte/neue Kamera-Position)
  - Teste ob P_1P_2 eines der Polygone (z.B. eine Wand) durchdringt
- Lösung:
  - Baue BSP über der Menge der Polygone (Preprocessing)
  - Algorithmus zur Laufzeit:

```
checkColl( p1, p2, node ):
  d1 = node.distToPlane( p1 )
  d2 = node.distToPlane( p2 )
  if d1<0 && d2<0:
     return checkColl( p1, p2, node.left )
  if d1>0 && d2>0:
     return checkColl( p1, p2, node.right )
  // line p1p2 intersects plane of the node,
  // and (possibly) a polygon in left or right
  return
    intersect( p1, p2, node ) ||
    checkColl( p1, p2, node.left ) ||
    checkColl( p1, p2, node.right )
```

Warnock's Algorithmus

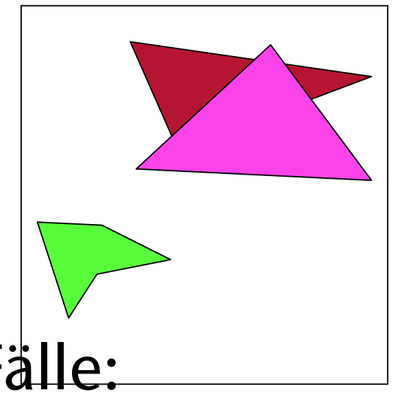
- Ein Image-Space-Verfahren, das auf einer rekursiven Unterteilung des *image space (Bildschirm)* beruht, bis die einzelnen Gebiete "homogen" sind



- Heute nicht mehr relevant (im Moment)
- Zeigt aber sehr schön folgendes algorithmisches Prinzip:
  - Kann man eine geometrische Entscheidung nicht für den ganzen Bereich fällen, so teile diesen erst einmal auf (hier: Bildraum wird aufgeteilt)
  - Ist im Prinzip eine Variante von Divide-and-Conquer
  - Ergibt einen sog. Quadtree

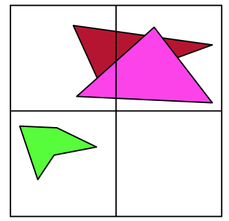
Idee

- Unterteile den Bereich in 4 gleiche Gebiete
- Treffe für jedes Teilgebiet die Entscheidung, welches Polygon (vorne) gezeichnet werden soll
- Zwischen einem Polygon und einem Gebiet gibt es folgende 4 Fälle:

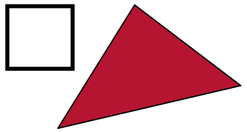


- Beobachtungen:
  - Nicht vom Gebiet geschnittene Polygone beeinflussen das Gebiet nicht
  - Schneidet ein Polygon das Gebiet, so beeinflusst der außerhalb liegende Teil das Gebiet nicht

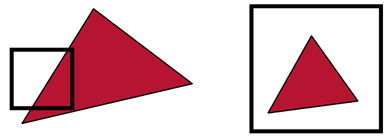
Der Algorithmus



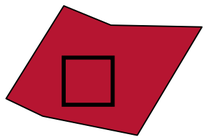
- Unterteile rekursiv den Bildschirm und die Menge der Polygone
- Bei jeder Rekursion wird das Teilgebiet untersucht:



1. Kein Polygon innerhalb des Gebietes ⟶ fülle mit der Hintergrundfarbe
2. Nur 1 Polygon liegt ganz oder teilweise innerhalb des Gebiets ⟶ fülle Gebiet mit der Hintergrundfarbe und zeichne anschließend den Teil des Polygons, der innerhalb liegt



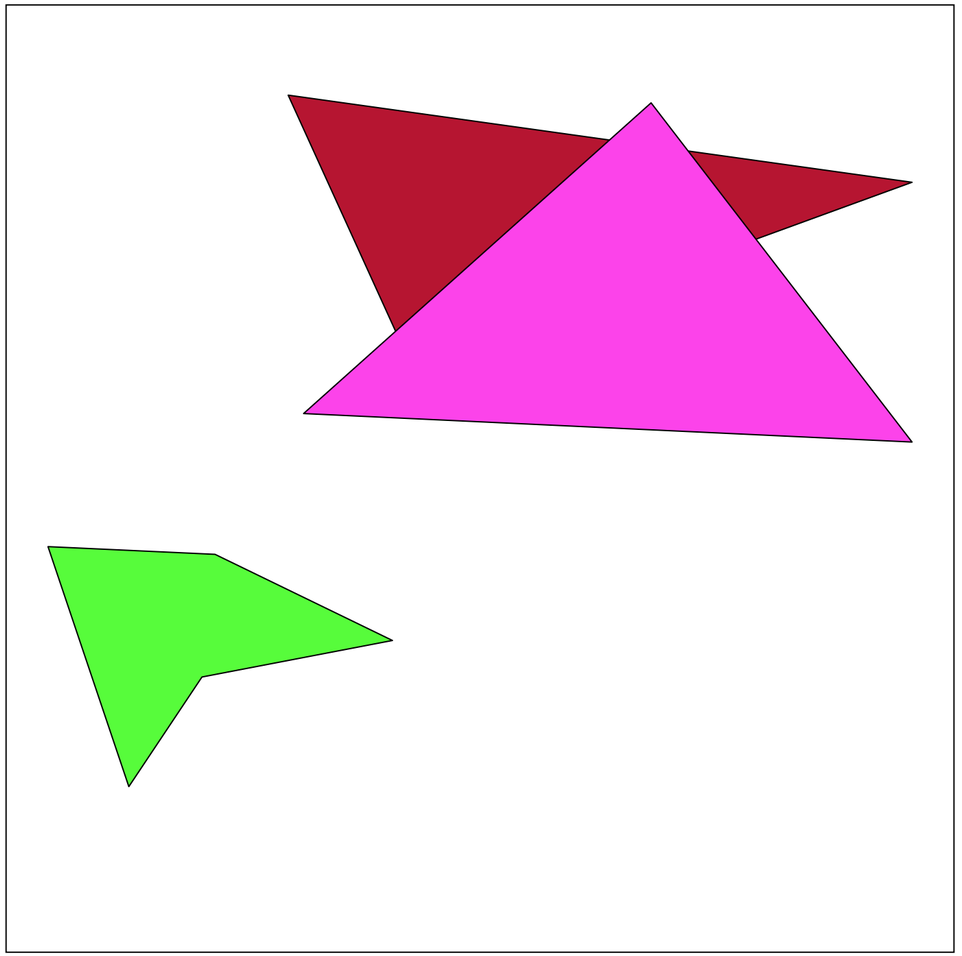
3. Wird das Gebiet von genau 1 Polygon umschlossen (kein Schnitt mit einem anderen Polygon) ⟶ färbe Gebiet komplett mit der Farbe des Polygons



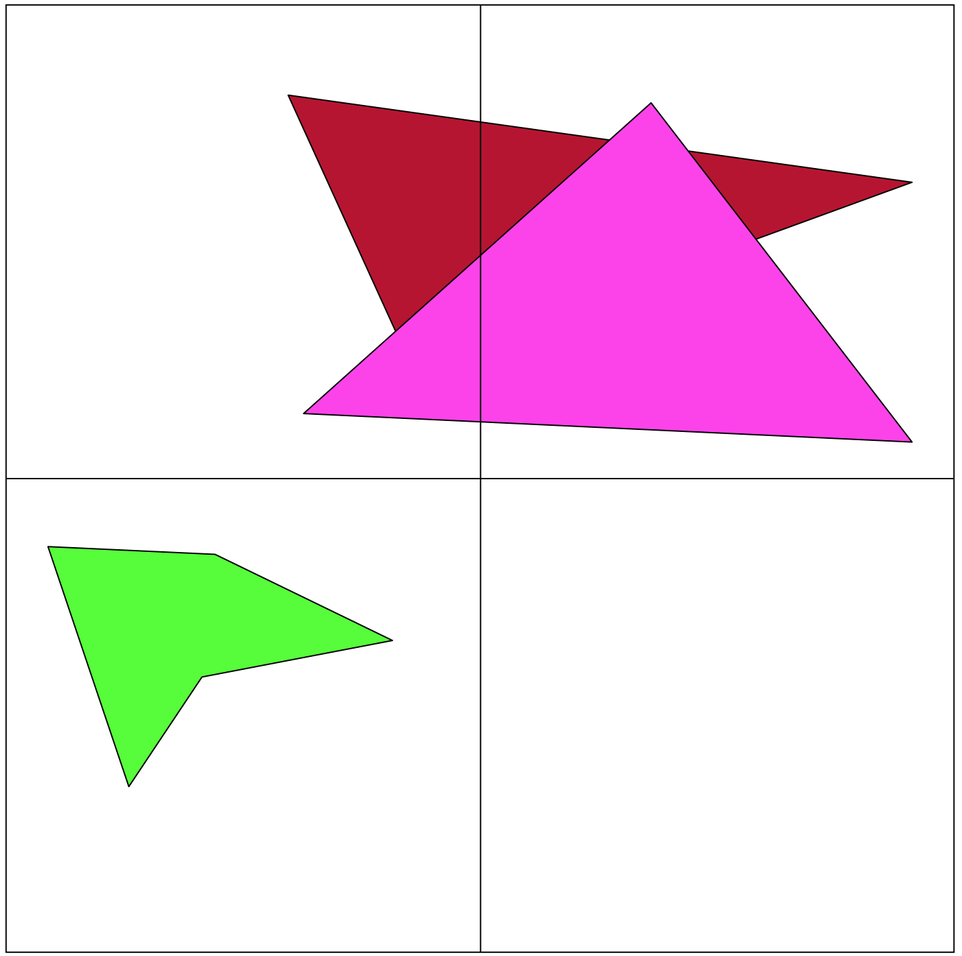
4. Umschließt, schneidet oder enthält das Gebiet mehr als 1 Polygon, aber ein Polygon liegt vor allen anderen ⟶ fülle den entsprechenden Teil des Gebiets mit der Farbe dieses Polygons

- Anderenfalls: unterteile das Gebiet und fahre mit Rekursion fort
  - Abbruchkriterium: Gebiet = 1 Pixel

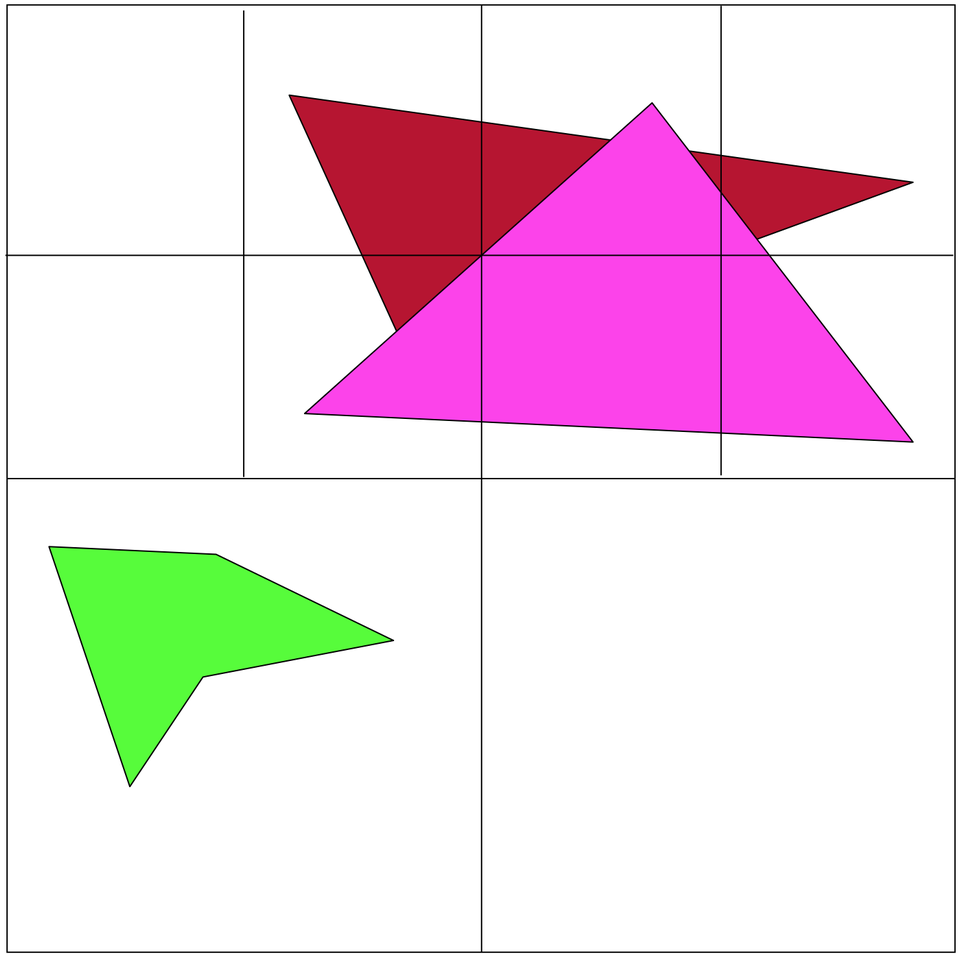
Beispiel



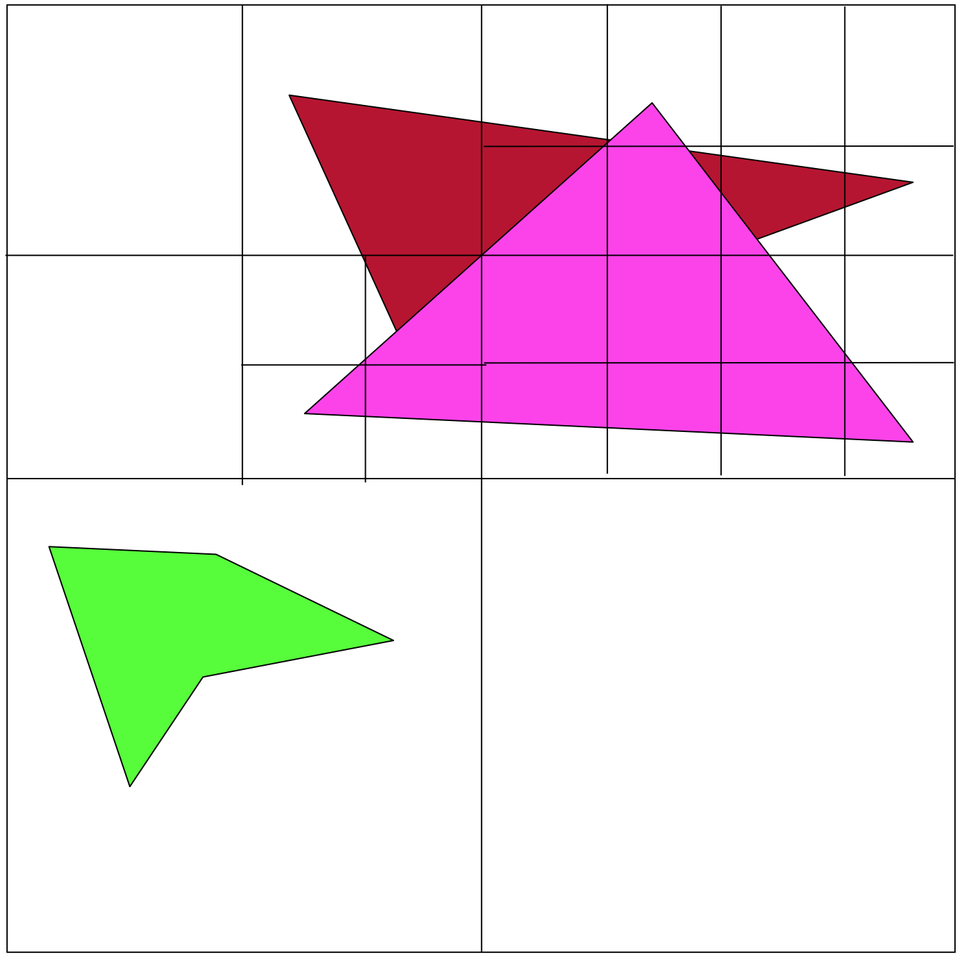
Ausgangsszene



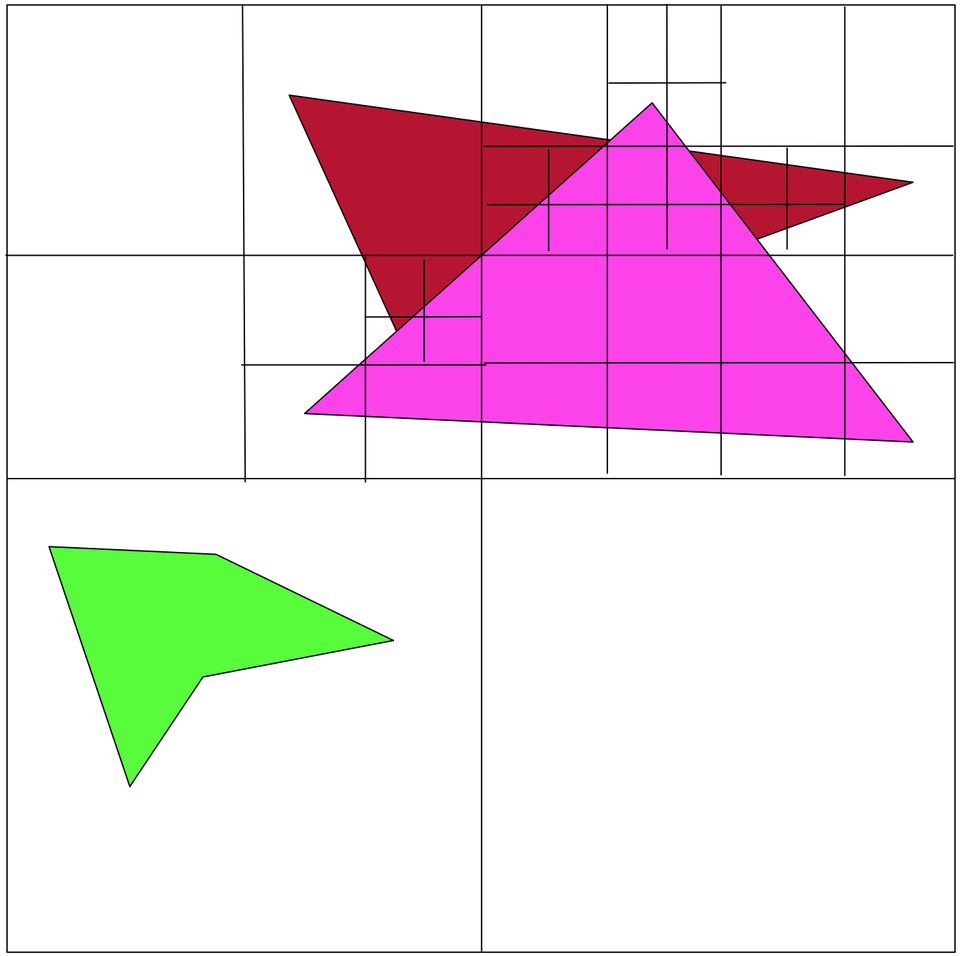
Erste Unterteilung



Zweite Unterteilung



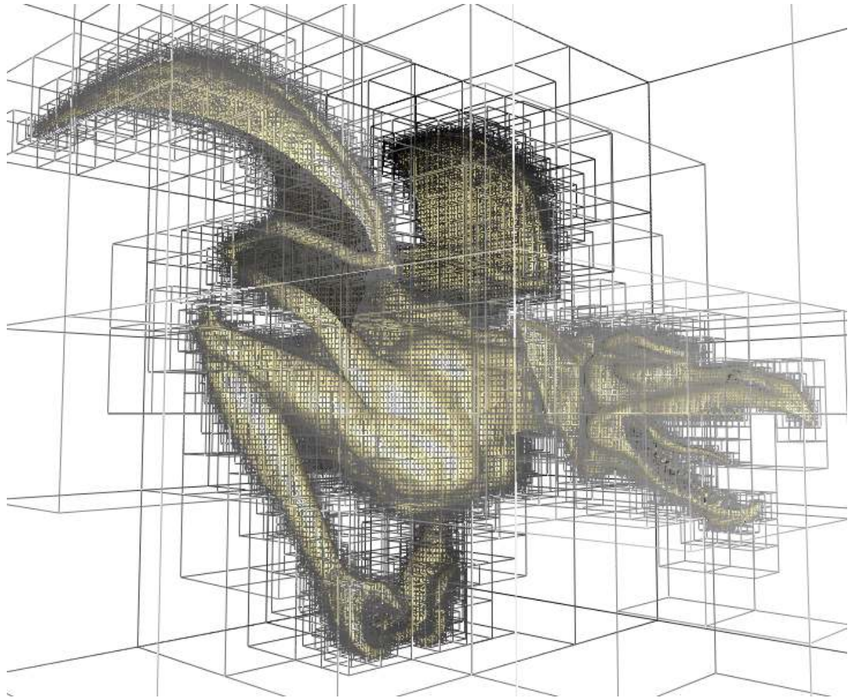
Dritte Unterteilung



Vierte Unterteilung

Beispiel-Anwendung: Octree Textures

- Octree = 3D-Analgon zu Quadtrees
- Anwendung: Texturierung eines Meshes
- Problem: Generierung von Textur-Koordinaten
- Konstruktion des Octree: rekursive Unterteilung bis
  1. Blatt nur noch 1 Polygon, 1 Kante, oder 1 Vertex enthält; oder,
  2. Blatt leer ist; oder,
  3. maximale Tiefe erreicht ist
- Nur Blätter vom Typ 1 speichern eine Farbe



- Texturierung zur Laufzeit (naïve Version):
  - Rasterisiere Polygone wie bisher
  - Interpoliere nicht Farben, sondern die (x,y,z)-Koordinaten der Vertices im *Object*-Space (erledigt der Rasterizer für uns)
  - Im Fragment-Shader: Traversiere Octree bis zum dem Blatt, das (x,y,z) des aktuellen Fragments enthält; verwende die in diesem Blatt gespeicherte Farbe
  - Zum Anti-Aliasing: interpoliere zwischen Nachbar-Zellen des Octrees
- Vorteil: keine Texture-Koordinaten nötig!
  - Die 3D-Koordinaten des Punktes (Fragments) im Object-Space reichen
  - Weiterer Vorteil: keine Verzerrungen
- Nachteile: ...

- Ausgelassene Details: MIP-Mapping, Interpolation der Farben in der Octree-Textur, sehr dünne Objekte, ...
- Nachteile:
  - Laufzeit zur Texturierung eines Pixels ist sehr verschieden
  - Speicherung ist ineffizient (?)

Master-Arbeit!

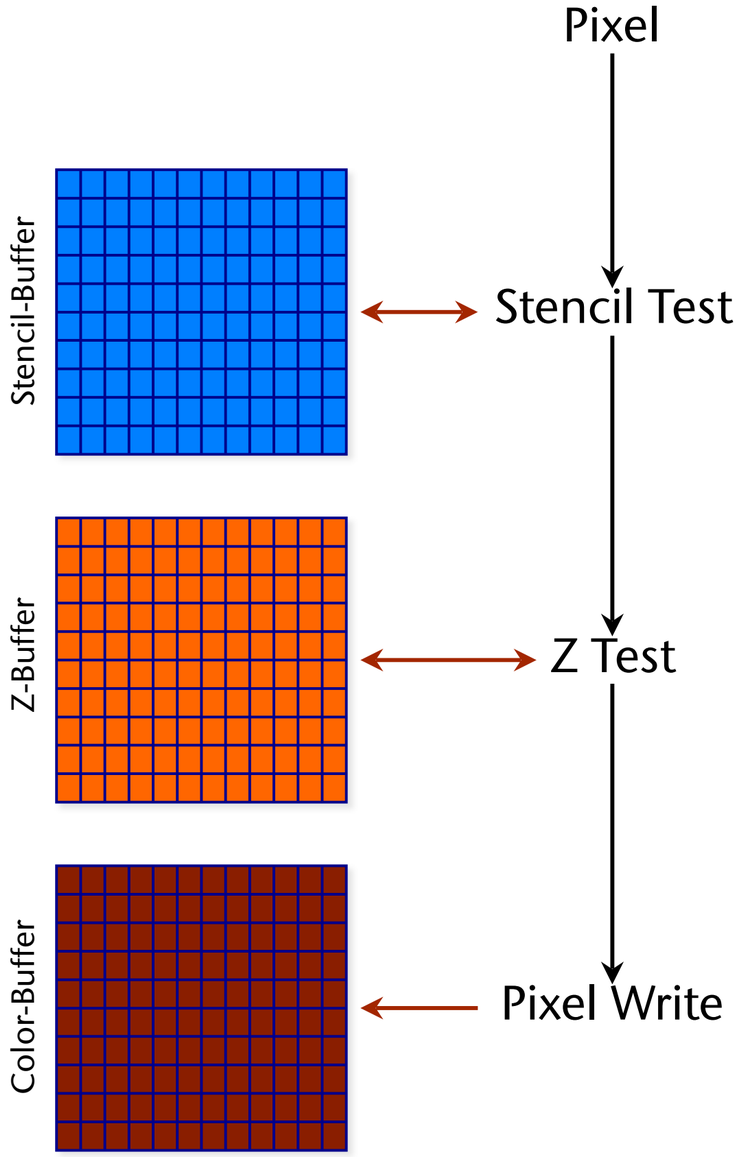
Der Stencil Buffer

- 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:

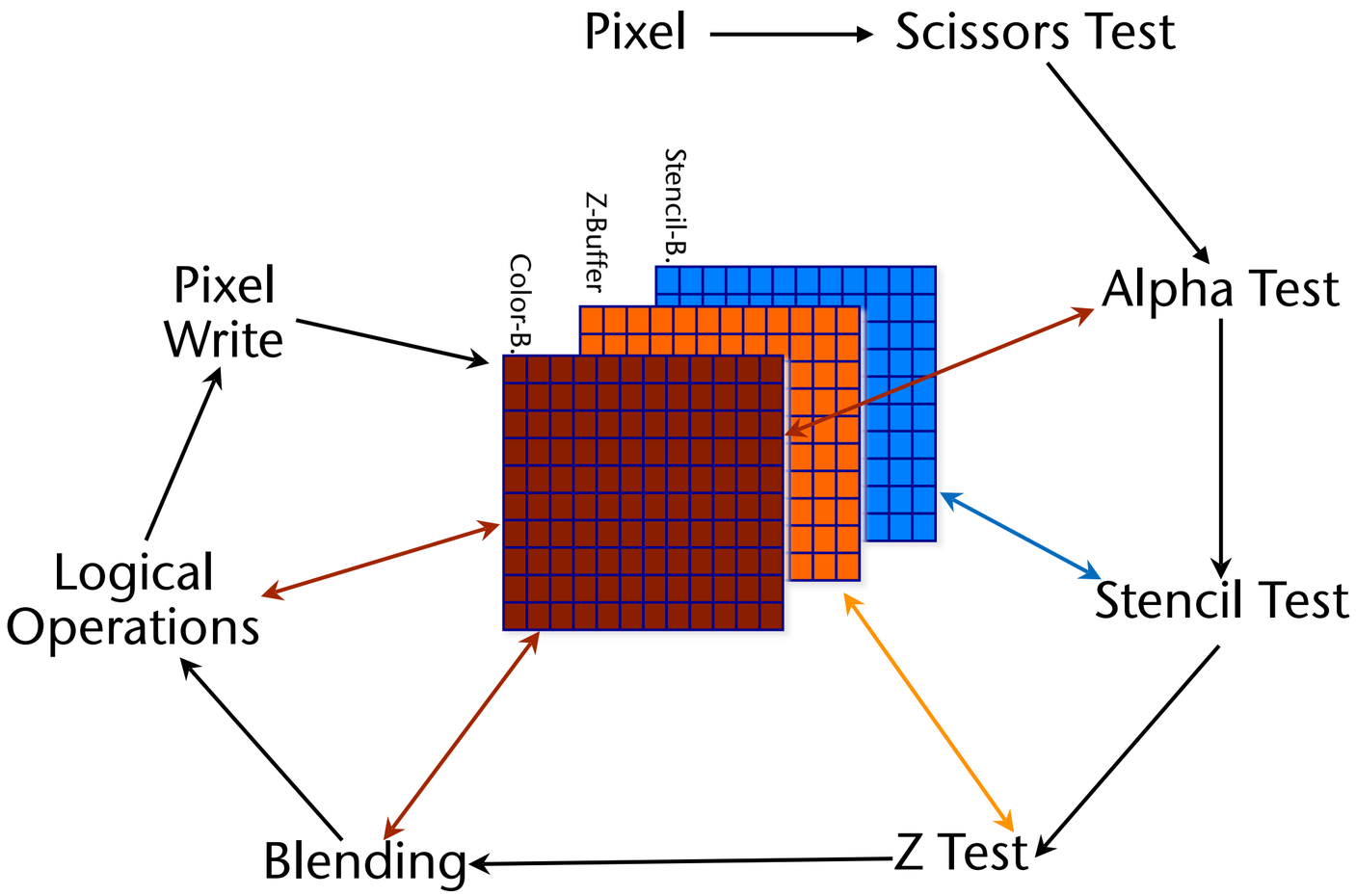
1. **`glStencilFunc(GLenum func, GLint ref, GLuint mask)`**: der *Stencil-Test,* legt fest, ob in den *Color-Buffer* geschrieben wird
   - Form des Tests: **`s func ref`** .
   - Dabei ist *s* = aktueller Wert im Stencil-Buffer an der Pixelstelle, *ref* = ein Referenzwert, mask = Bit-Maske
   - Mögliche 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]
```

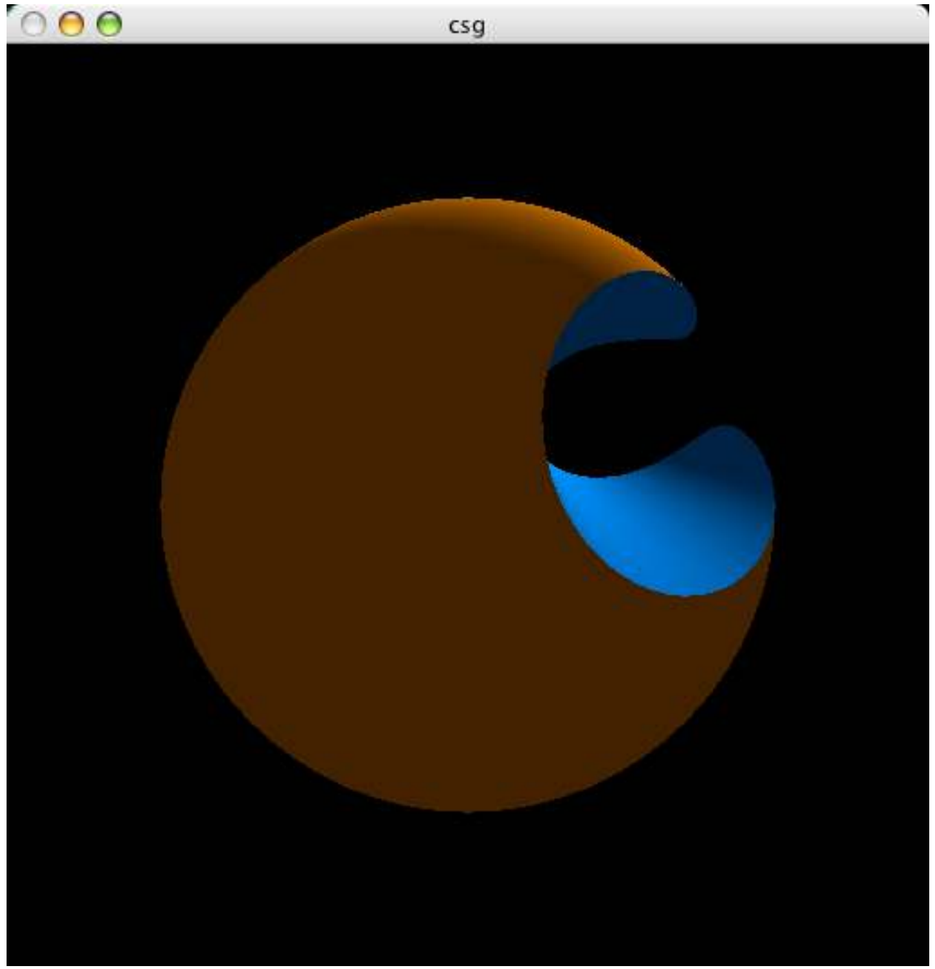


Die volle Abfolge von Tests und Operationen in der GPU

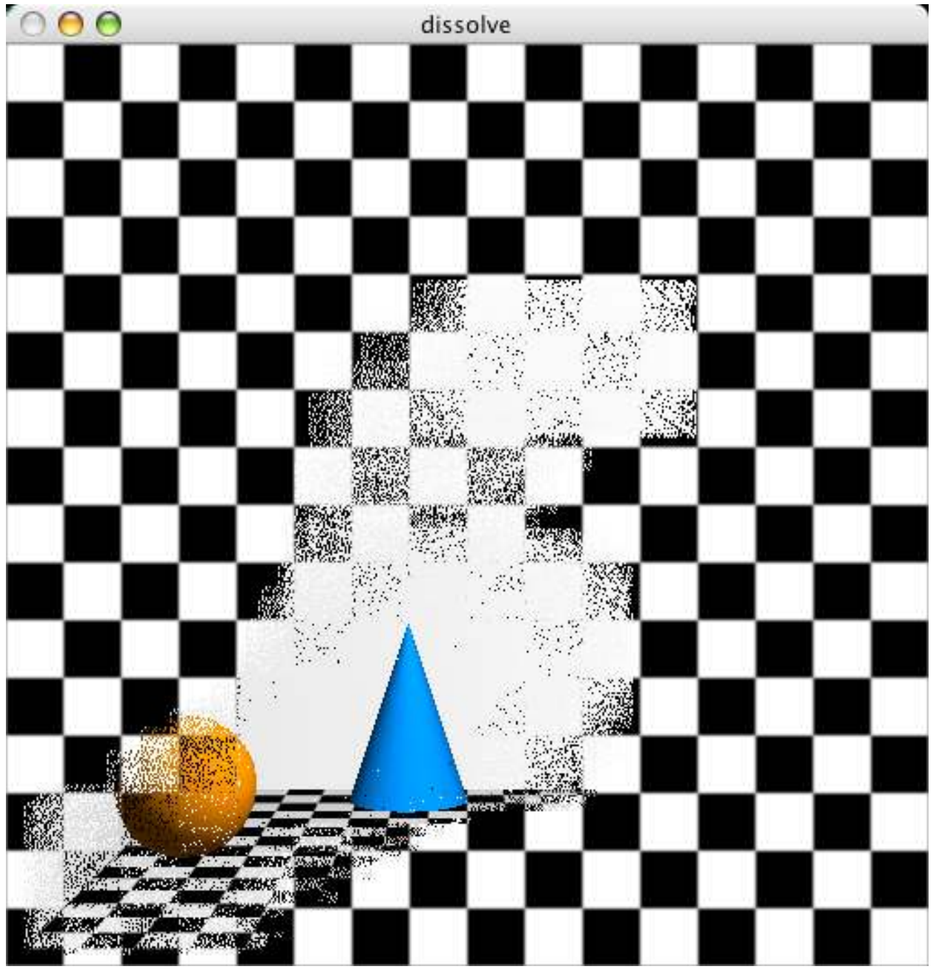


Beispiele für komplexere Operationen/Effekte mittels Stencil-Buffer

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



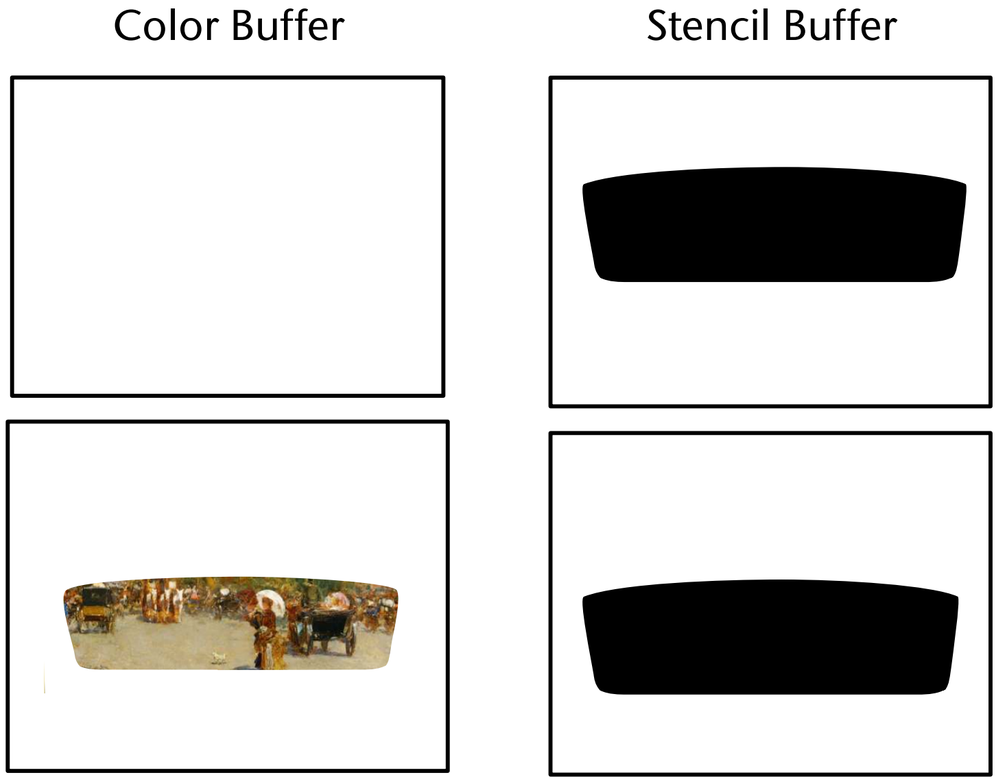
"Dissolve"



Ein "Stencil" im echten Leben

Typisches, einfaches Anwendungsbeispiel

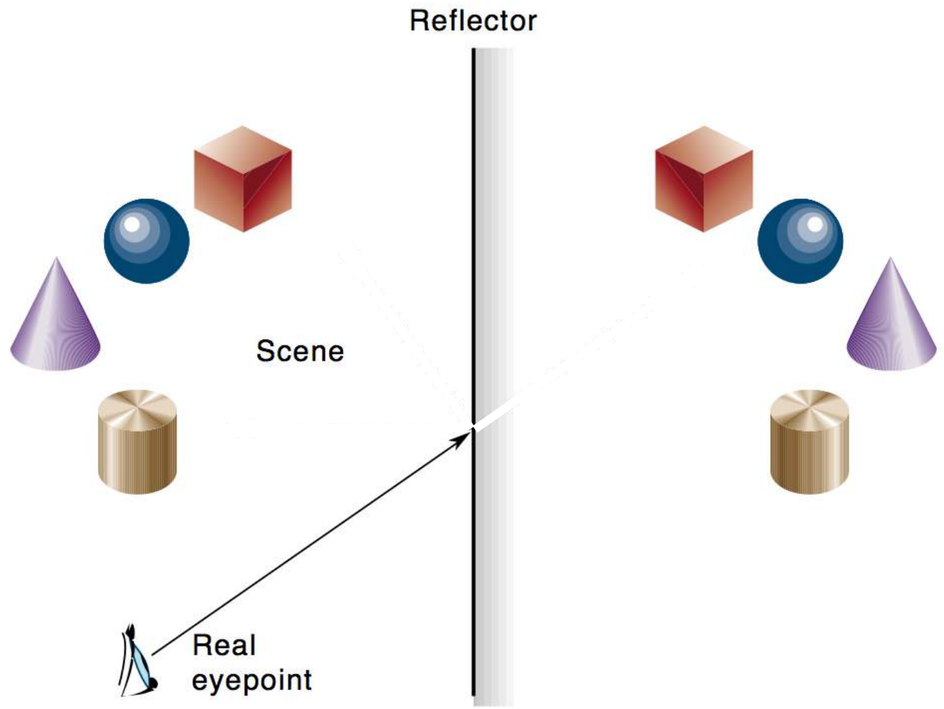
- Szene durch ein Objekt maskieren:
  1. Alle Buffer inkl. Stencil-Buffer löschen
  2. Maske rendern, dabei Stencil-Buffer überall dort auf 1 setzen, aber Color-Buffer unverändert lassen(!)
  3. Szene zeichnen, aber nur dort, wo Stencil-Wert = 1



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
  - Rendere alle Polygone mit z' = -z
  - Rendere die Szene normal



- Dies ist ein Beispiel für einen multi-pass Rendering-Algo
- Achtung: rendere in Schritt 2 nur Polygone hinter der Spiegelebene (mittels Clipping-Plane in Spiegelebene ⟶ später)

- 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.
- Pass 1:
  - Set up (user-defined) clipping plane such that objs *in front* of mirror are *not* rendered
  - Compute reflection transformation and apply to all polygons
  - Render scene (without geometry of mirror itself)
- Pass 2: 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 );
      ```
      ```

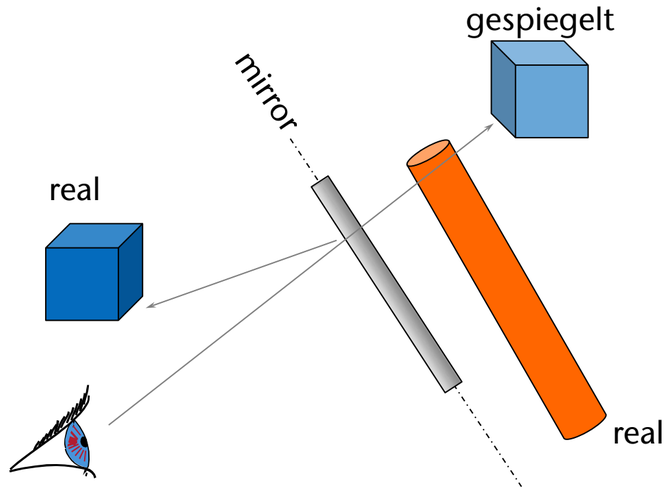
- Draw the geometry of the mirror
  - This sets stencil bits & fills z buffer with depth values of mirror geometry
  - Enable blending for color buffer to see mirror surface
- Clear color buffer outside mirror geometry:
  - Configure the stencil test to pass *outside* the mirror polygon:

```
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:

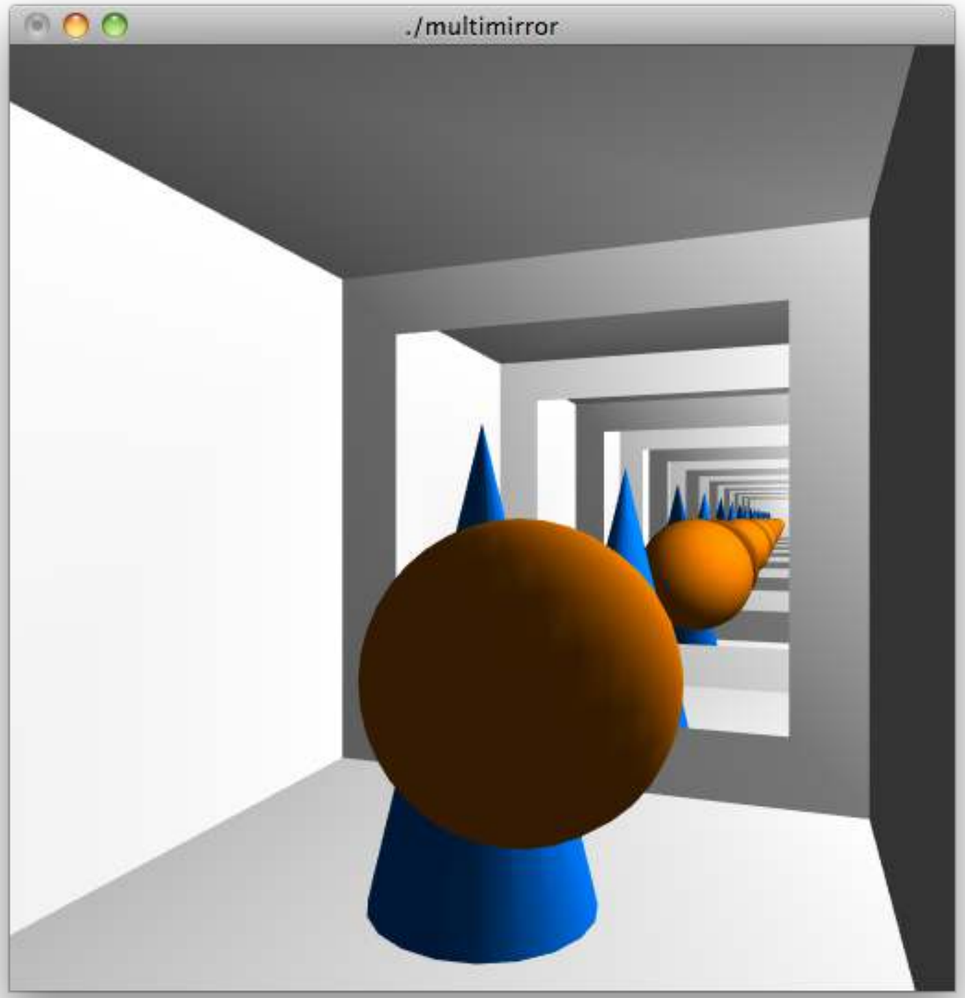
```
glClear(GL_COLOR_BUFFER_BIT)
```

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



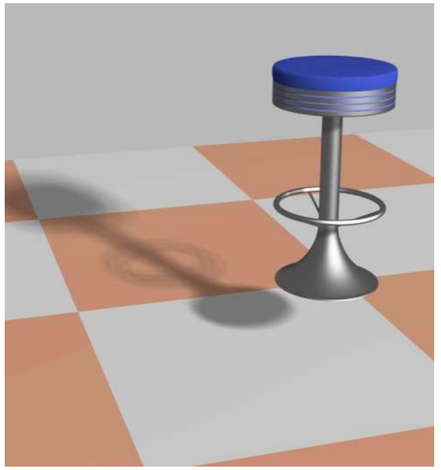
Demo

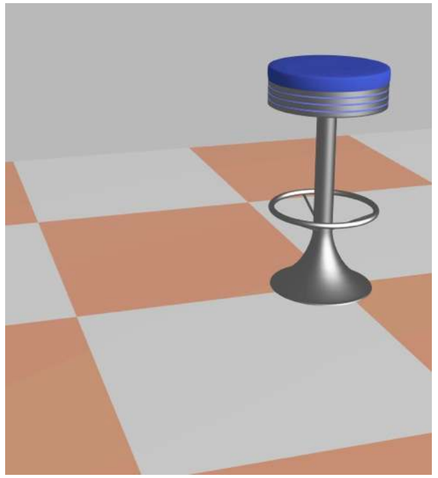
Iterative Anwendung des Verfahrens

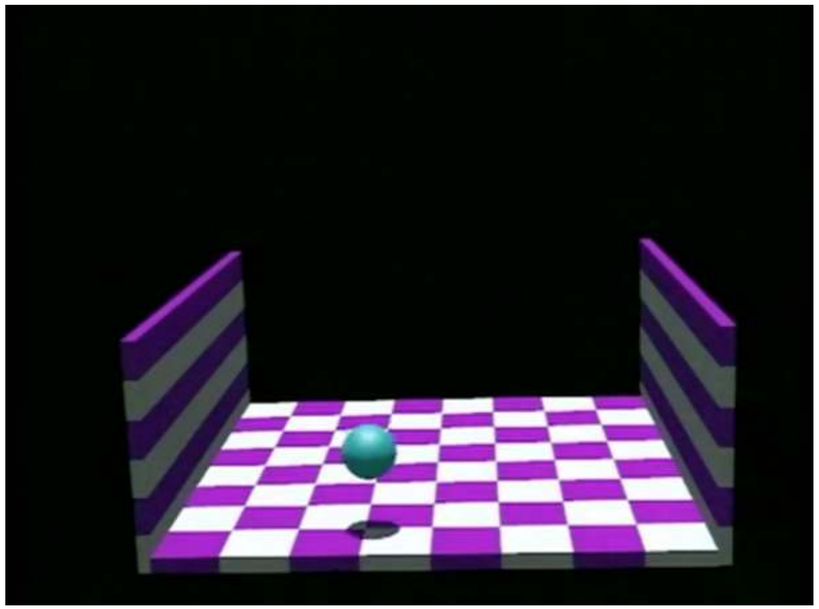


Schatten

- 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

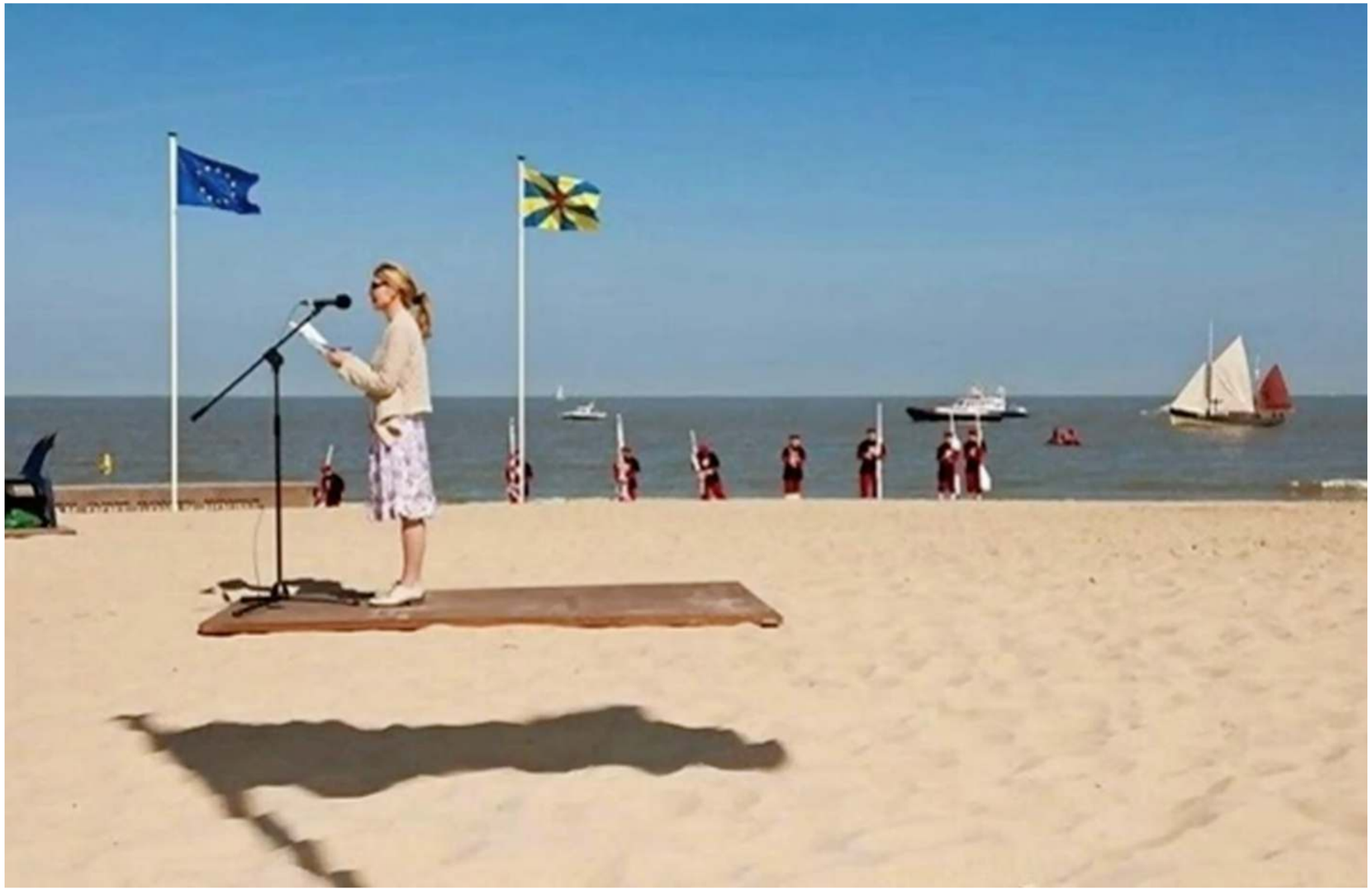






Die Trajektorie des Balls im Bildraum
ist in beiden Fällen genau dieselbe!

Eine optische Täuschung mit Schatten

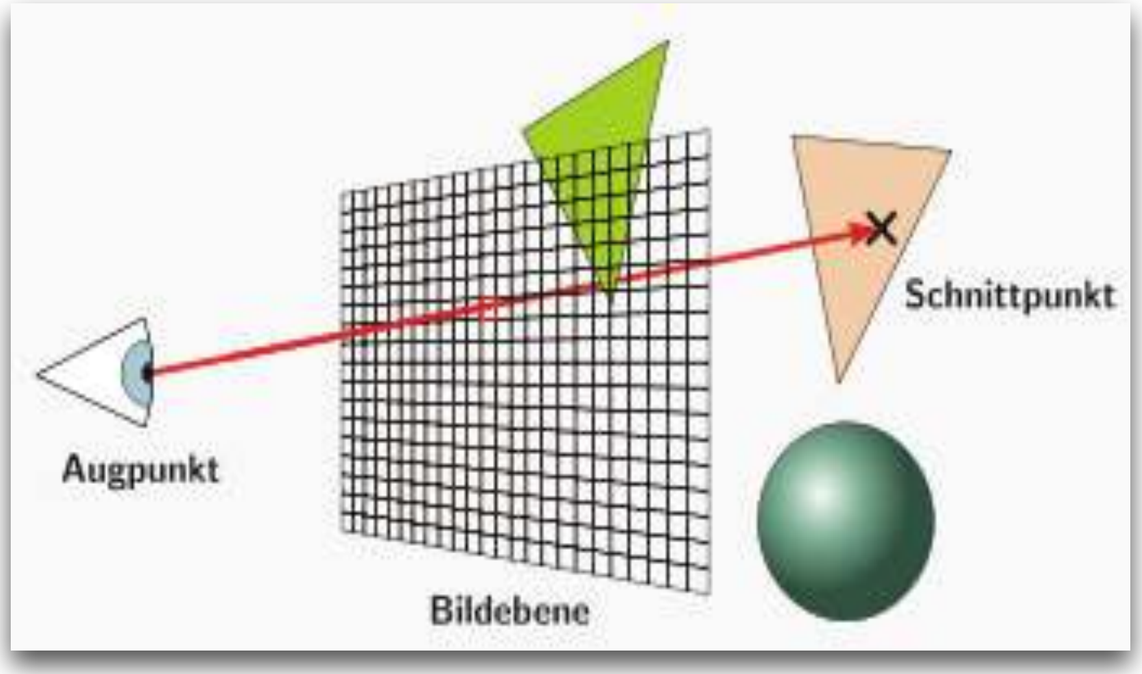


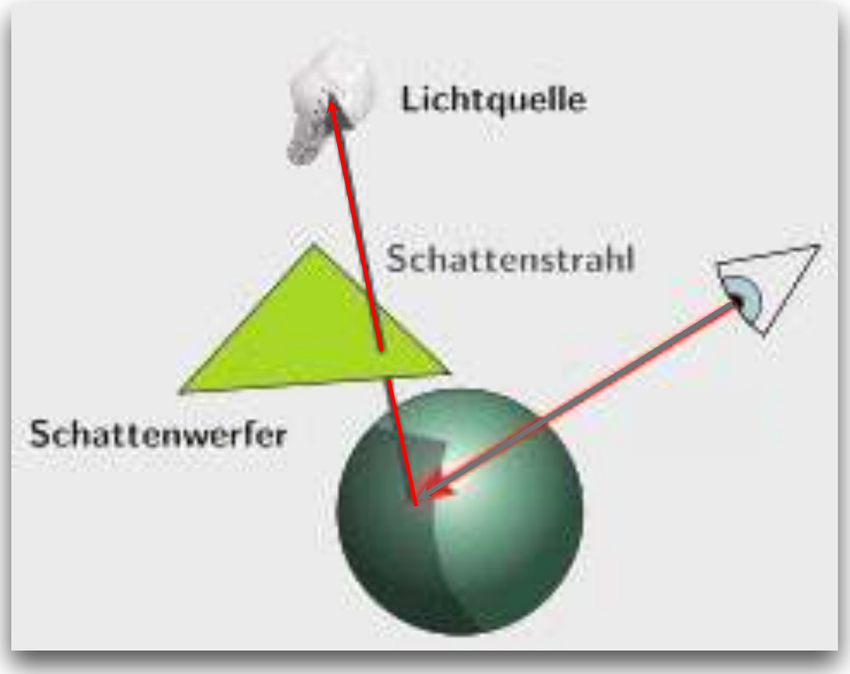
Wrong Shadows Reveal Poorly Done Foto Faking

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

Rendering Shadows using Shadow Volumes

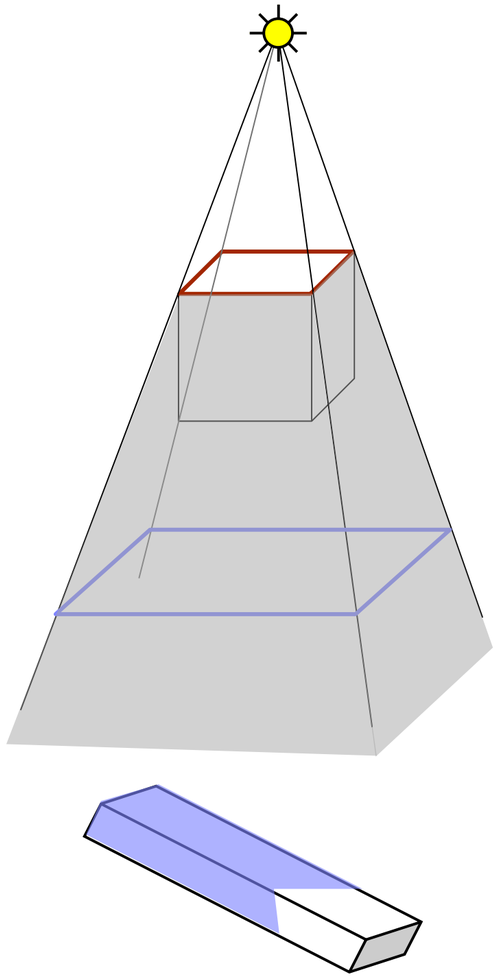
- Zusammenhang zwischen Visibility und Shadows:
  - Visibilitätsberechnung = welche Objekte sind vom Betrachter aus sichtbar
  - Schattenberechnung = welche Objekte sind von der Lichtquelle aus sichtbar



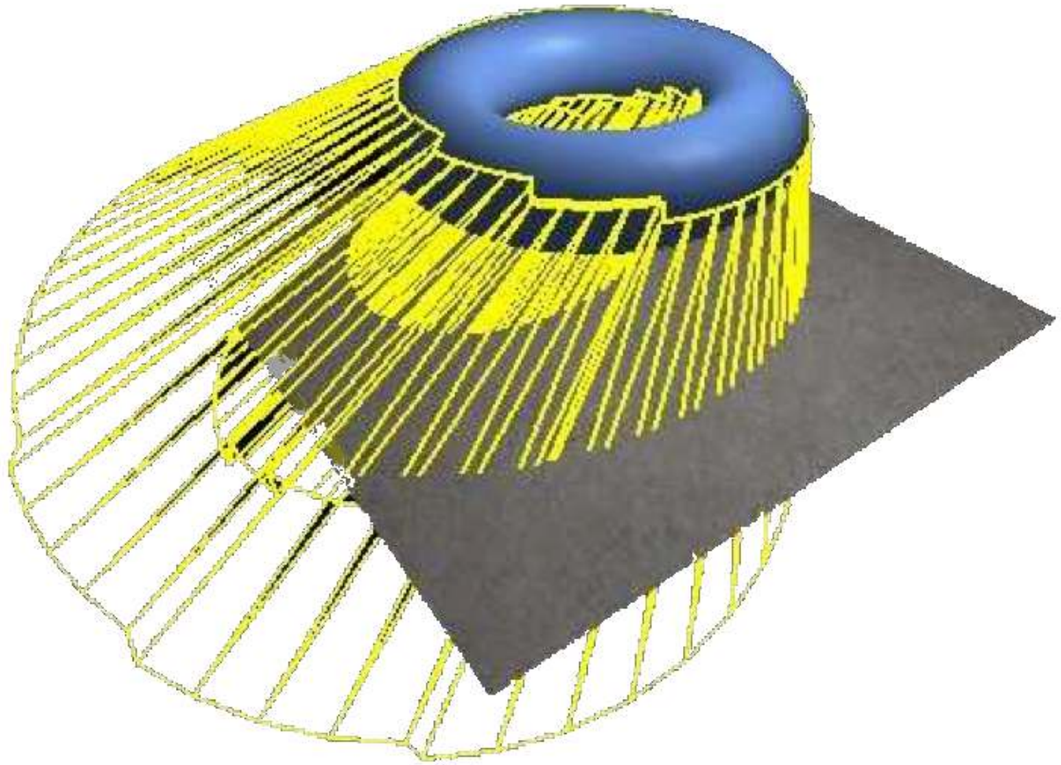


Das Schattenvolumen

- 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"

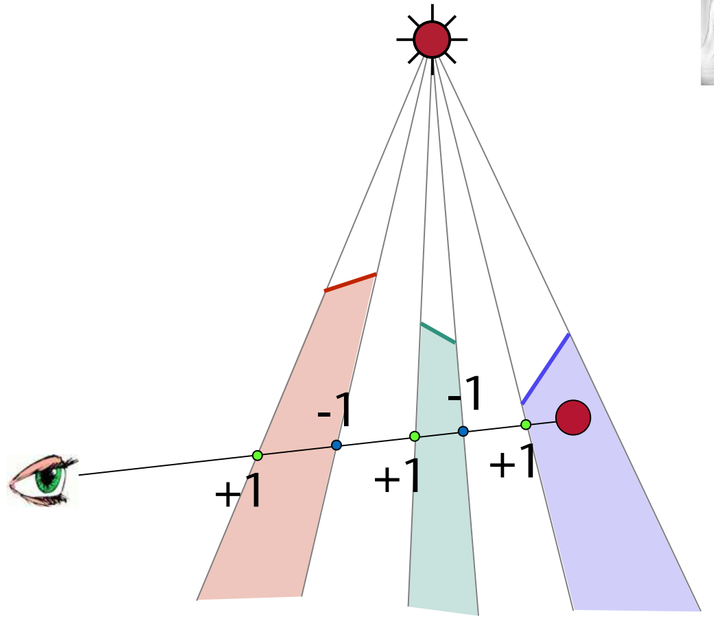


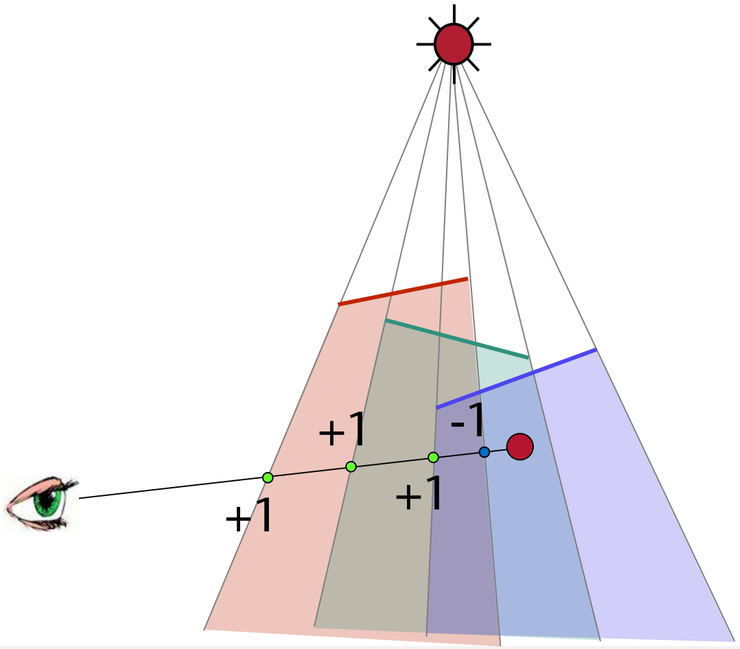
- Beispiel für ein komplexeres Shadow Volume
- Beachte: die Silhouettenkanten liegen nicht notwendigerweise alle in einer Ebene!



Ein geometrisches Prädikat für "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-Wert verändern, die den Z-Test passieren (= vor der eigtl Geometrie sind)

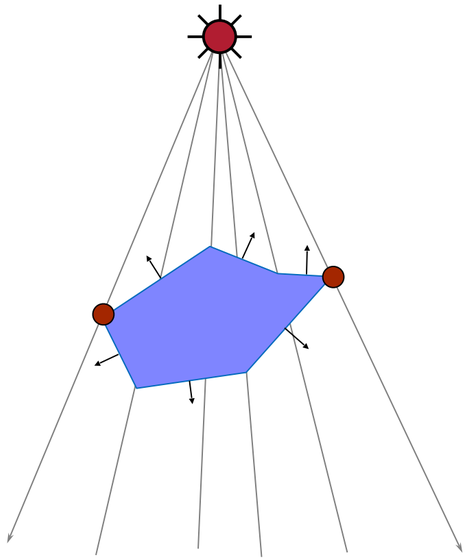
Varianten des Algorithmus'

- 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)

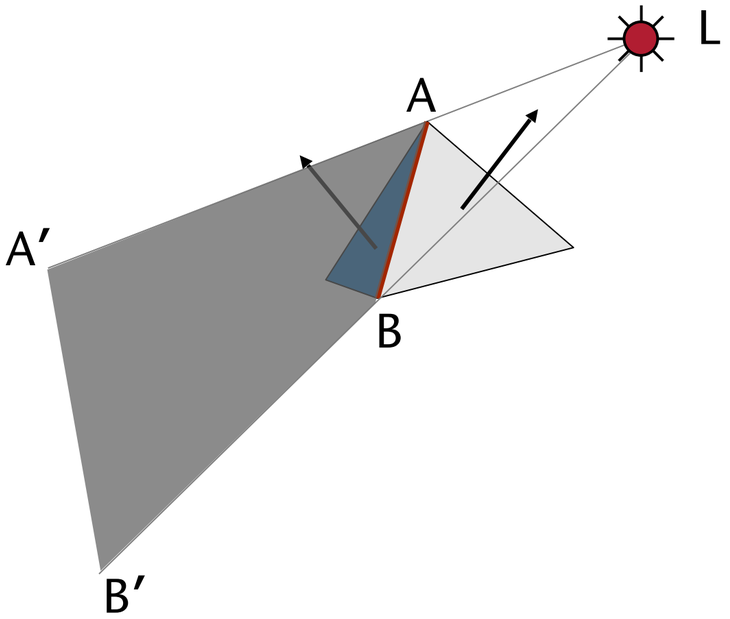
Bemerkungen zu Details

- Berechnung der Silhouettenkanten:
  - 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:
  - Verlängere die Eckpunkte der Silhouettenkante
  - Kann man mit speziellen Shadern während des Renders der Szene erledigen (CG2)

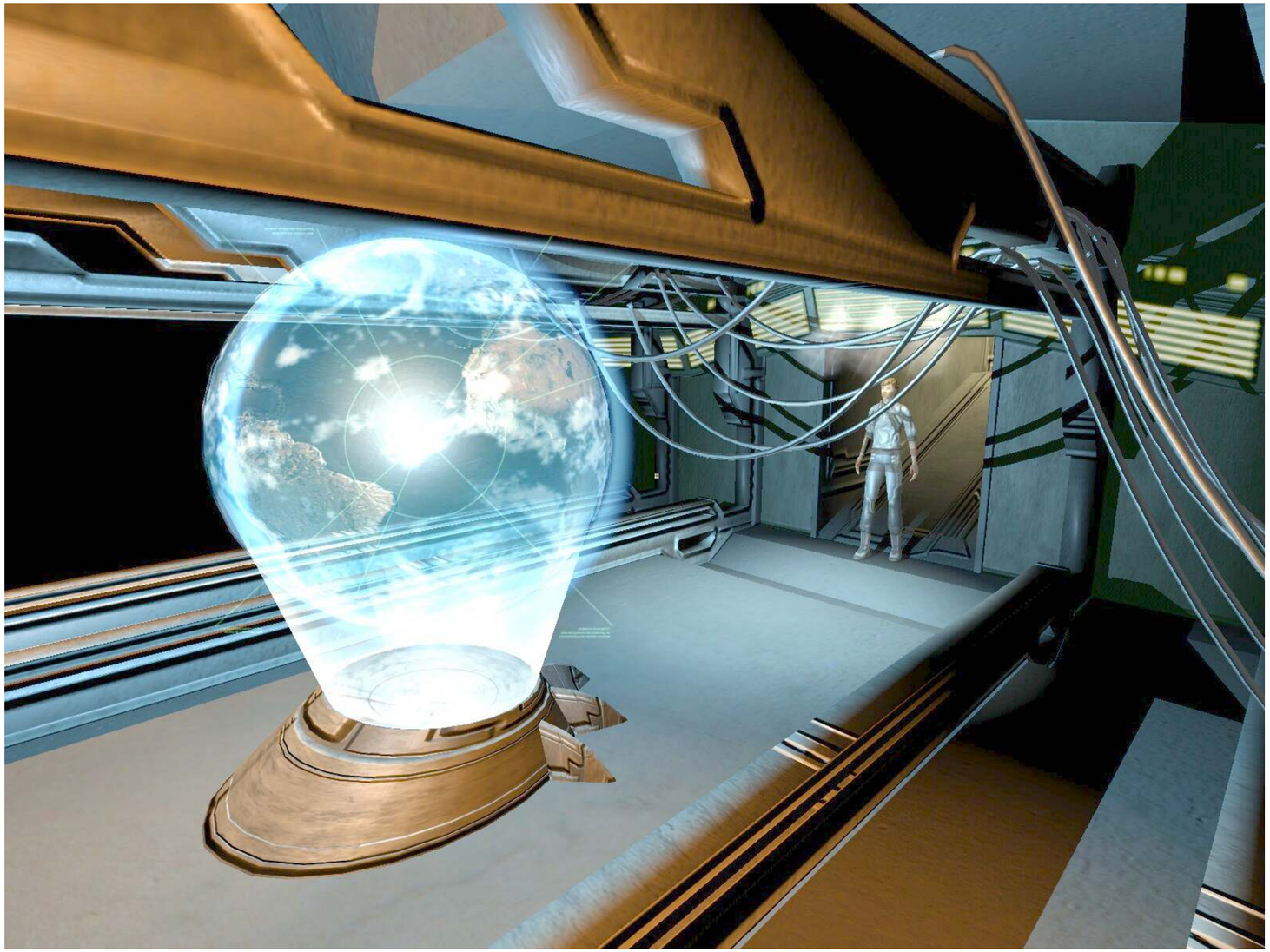


Beispiele

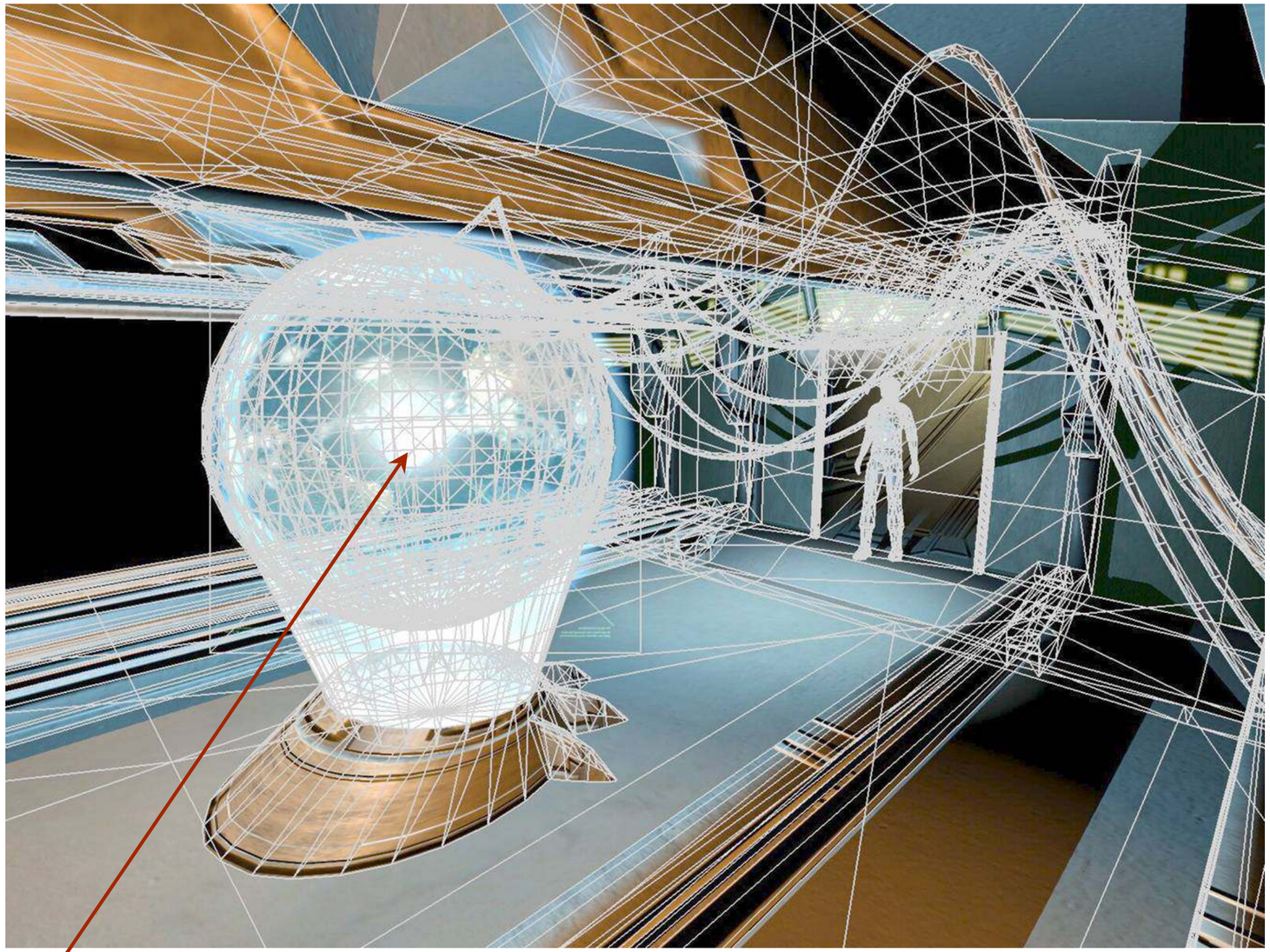
Shadowed scene

Stencil buffer contents

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

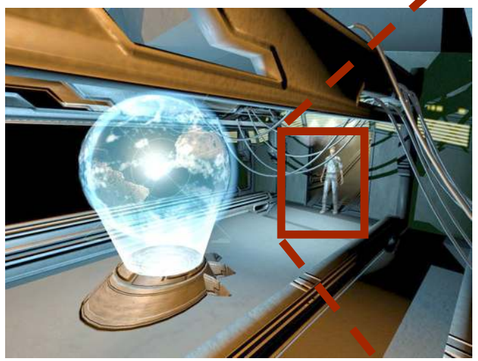


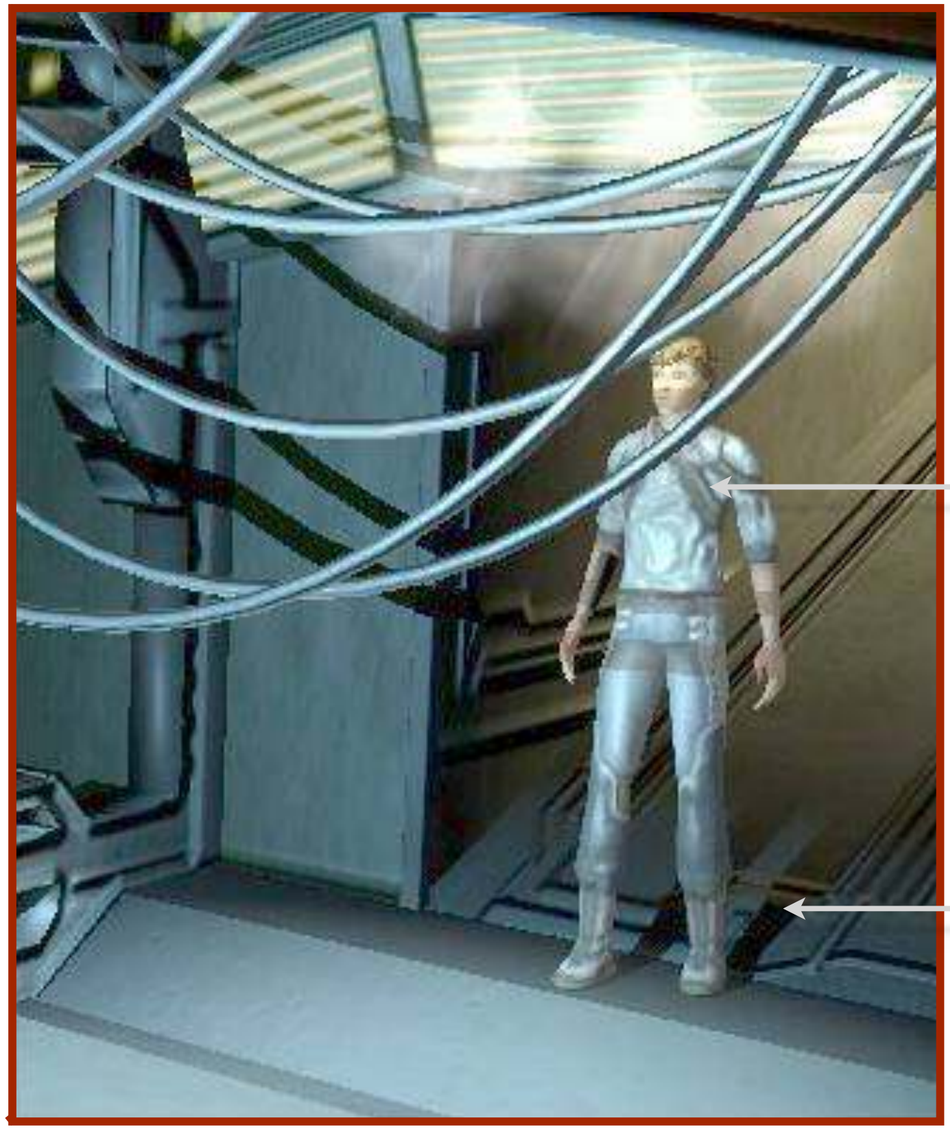
Abducted game images courtesy Joe Riedel at Contraband Entertainment



Primary light source location

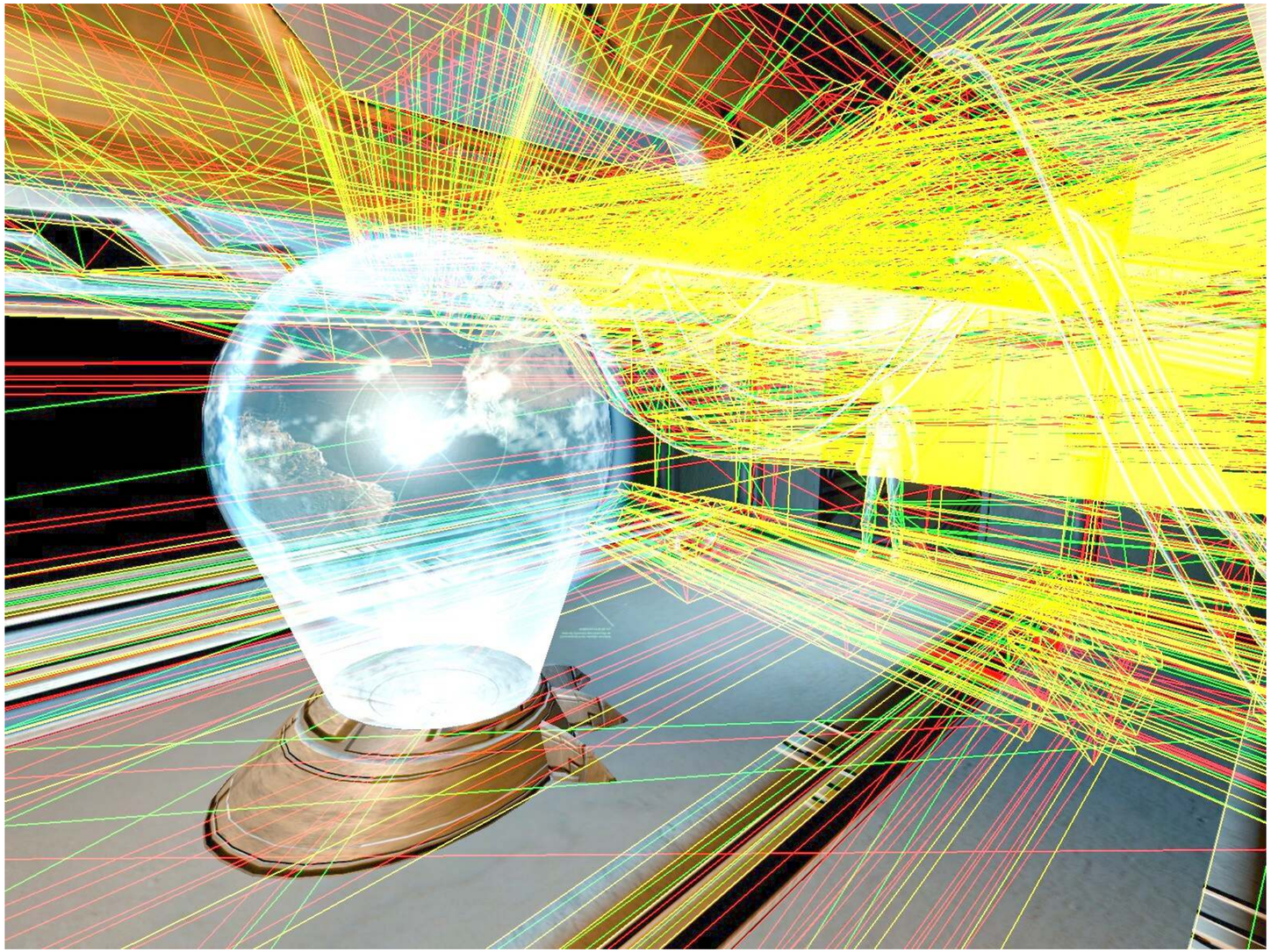
Wireframe shows geometric complexity of visible geometry



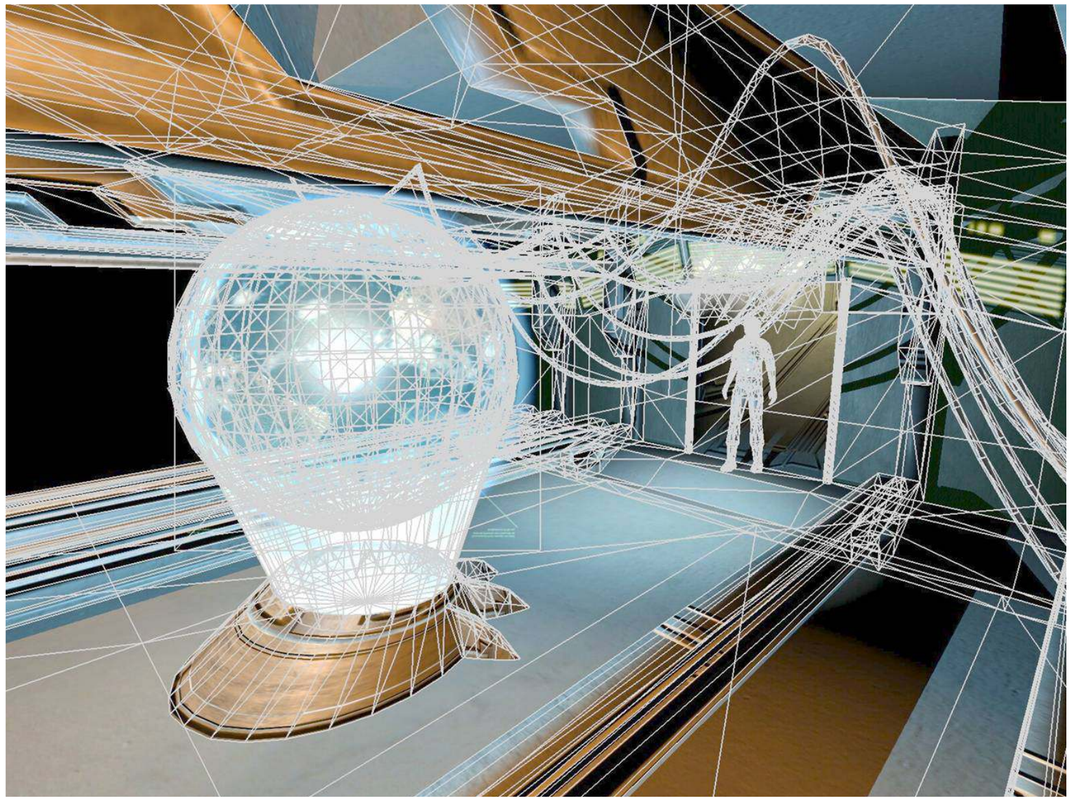


Notice cable shadows on player model

Notice player's own shadow on floor



Wireframe shows geometric *complexity of shadow volume* geometry



Visible geometry

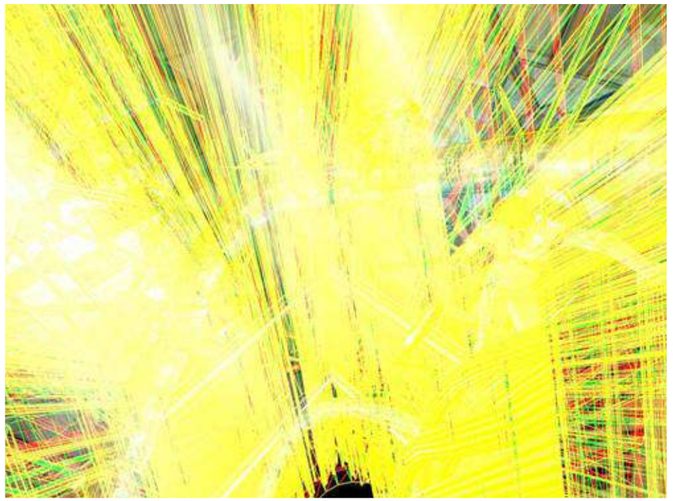
<

<

Shadow volume geometry

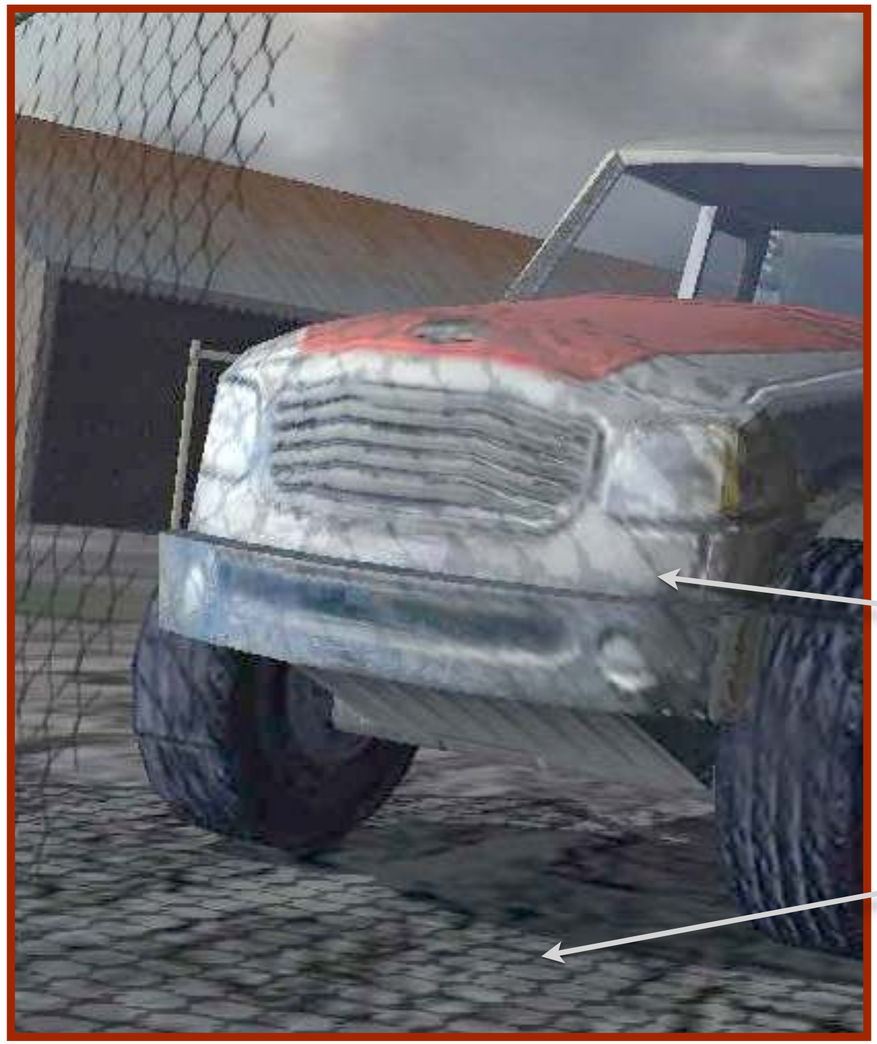
Typically, shadow volumes generate considerably more pixel updates than visible geometry

Visible geometry



Shadow volume geometry

Situations When Shadow Volumes Are Too Expensive

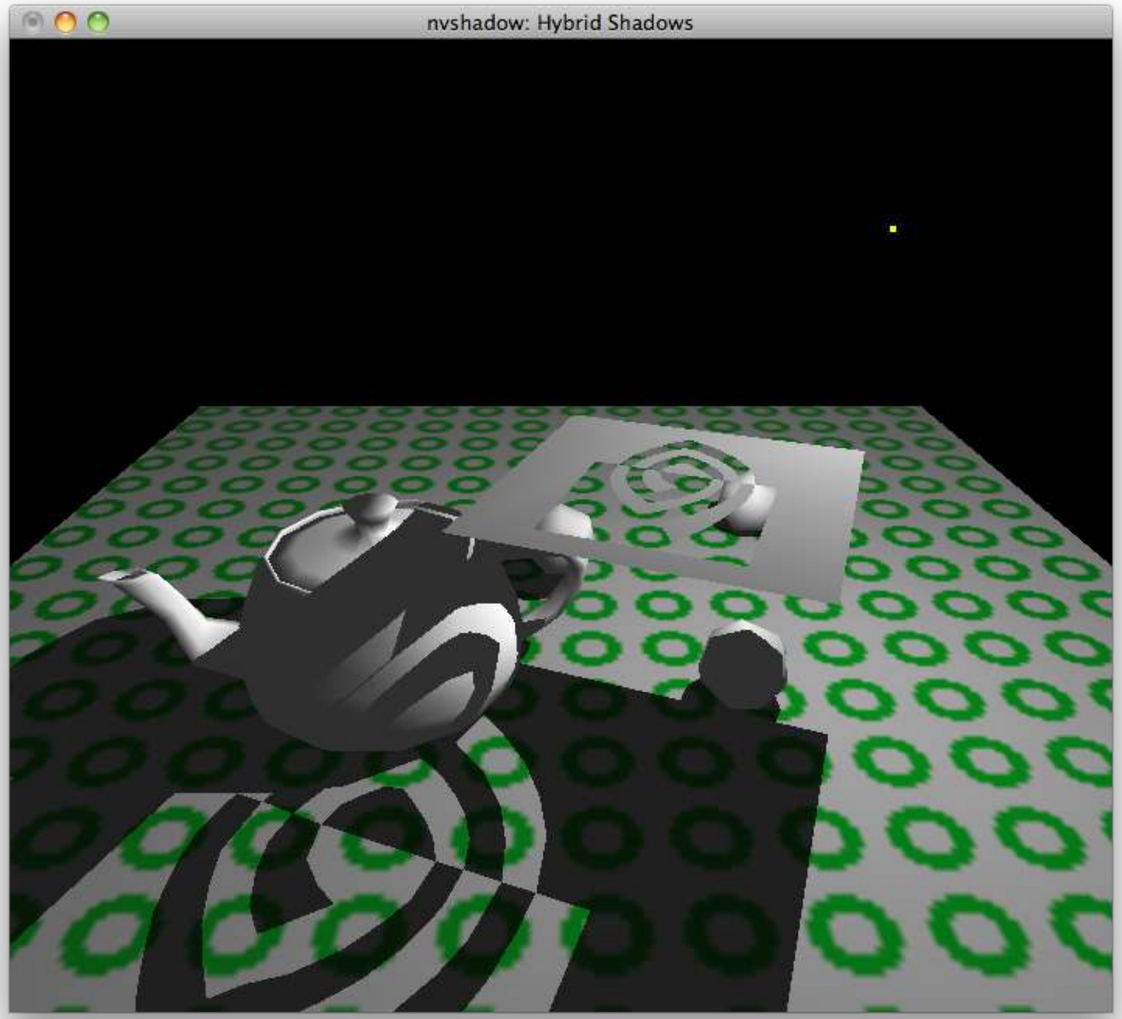


Chain-link fence is shadow volume nightmare!

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

Fuel game image courtesy Nathan d'Obrenan at Firetoad Software

Demos



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

Sichtbare Schattenvolumen in der Realität