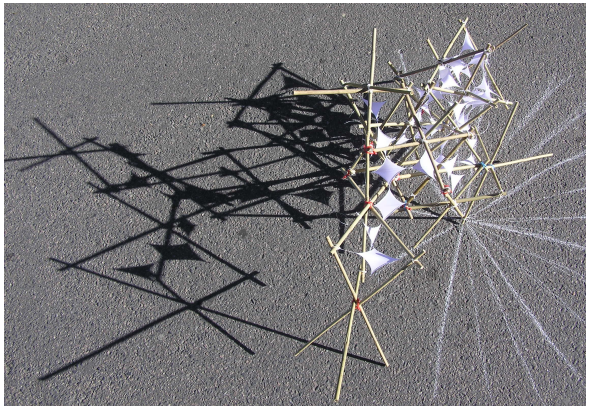
Bremen

CG VR

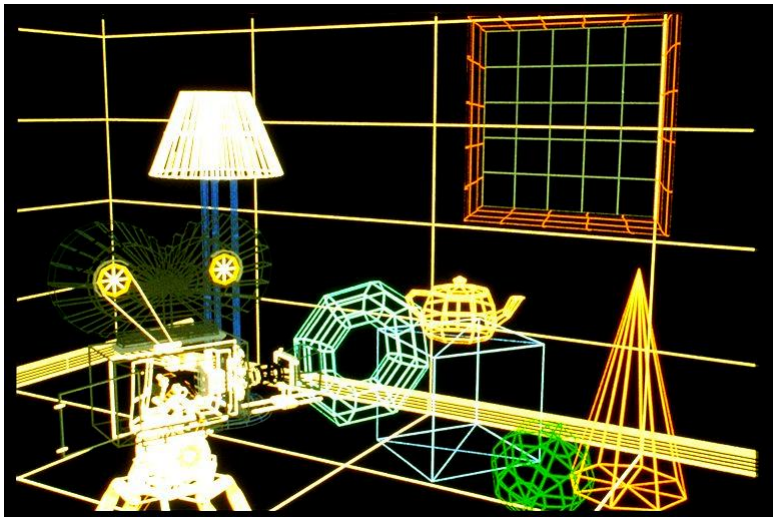
Computer-Graphik 1
Visibility Computations I - Hidden Surfaces, Shadows, and Frame Buffers

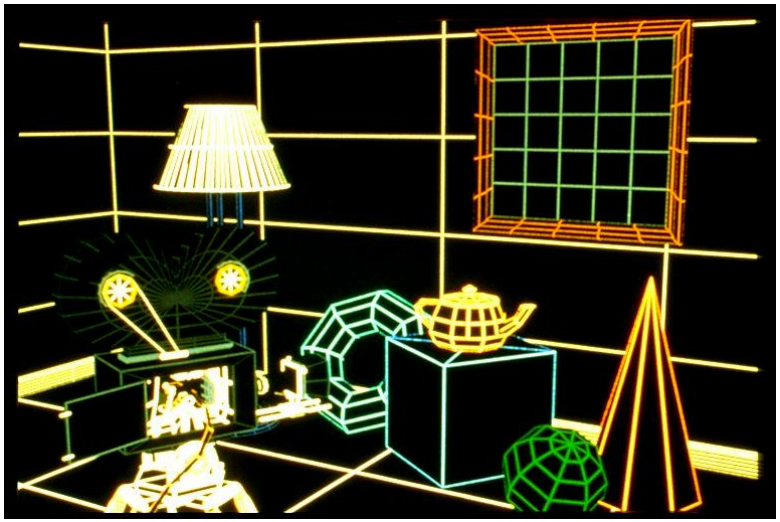


G. Zachmann
University of Bremen, Germany
cgvr.informatik.uni-bremen.de

Motivation

- Verdeckung entsteht, wenn mehrere Objekte bei der Projektion von 3D nach 2D (teilweise) die gleichen Bildschirmkoordinaten aufweisen (*Projektionsäquivalenz*)
- Sichtbar ist das dem Auge am nächsten liegende Objekt
- Ist dieses Objekt (halb-)durchsichtig (transparent), wird der dahinter liegende Punkt auch sichtbar, usw.



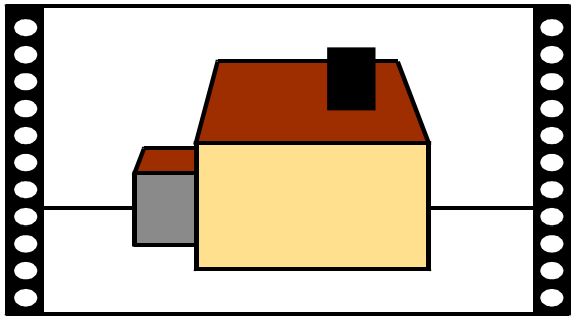


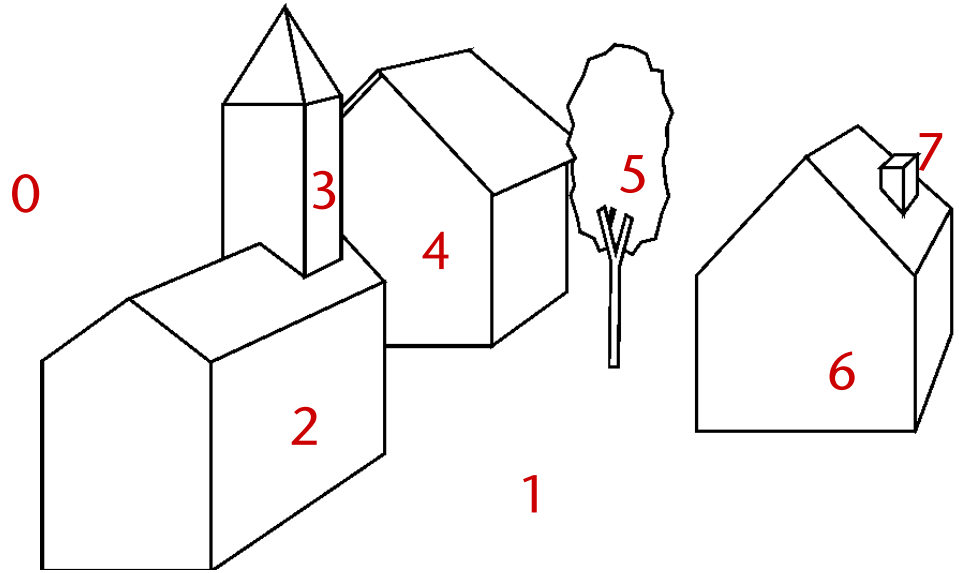
Pixar "Shutterbug"

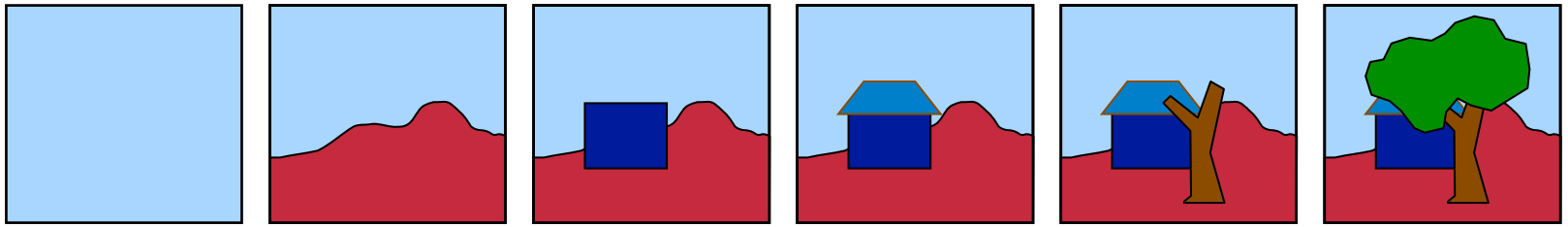
- Es gibt 2 große Problemklassen innerhalb des Bereichs "Visibility Computations"
  1. Verdeckungsberechnung: welche "Pixel" eines Polygons werden von anderen verdeckt?
     - Bezeichnungen: *Hidden Surface Elimination* (früher auch *Hidden Line Elimination*), *Visible Surface Determination*
  2. Culling: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

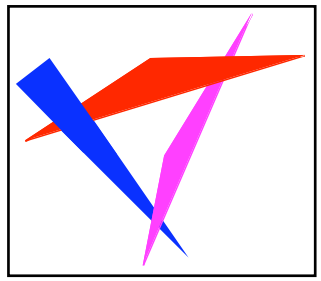


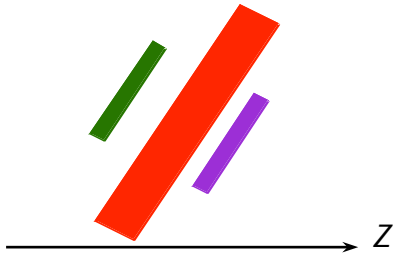




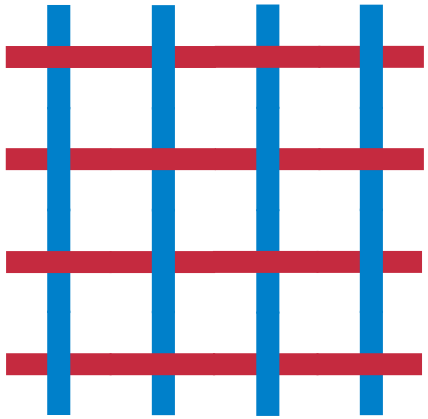
Probleme

- Es gibt Fälle, in denen eine korrekte Sortierung nicht existiert!
  - Oder nicht klar ist ...
- Eine Lösung wäre evtl. eine Zerlegung der Polygone — aber ...
- Diese Zerlegung ist (im Prinzip) abhängig vom Viewpoint; und ...



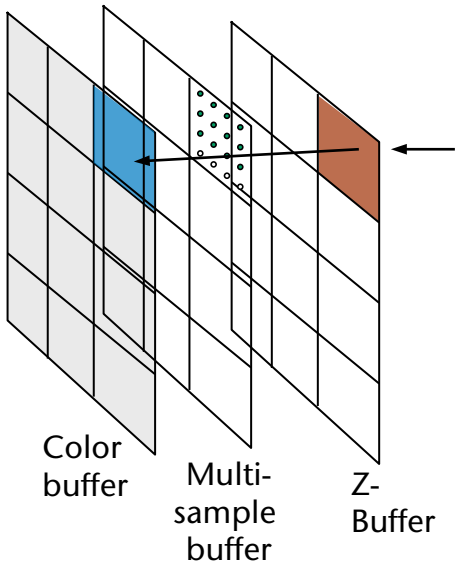


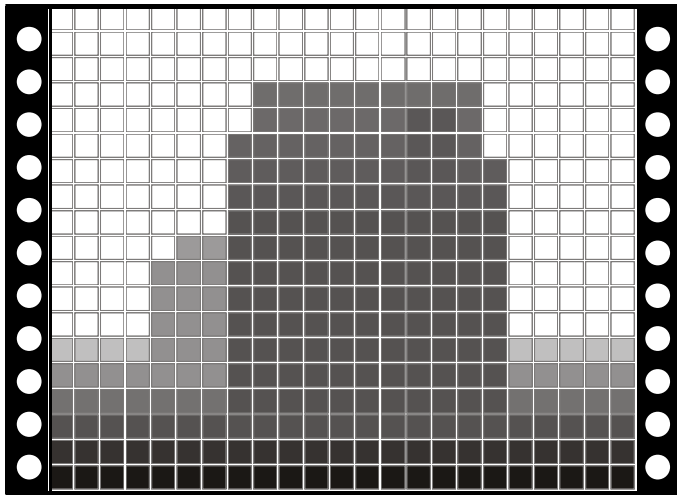
- Bei einer Szene mit n Polygonen können $O(n^2)$ sichtbare Fragmente entstehen



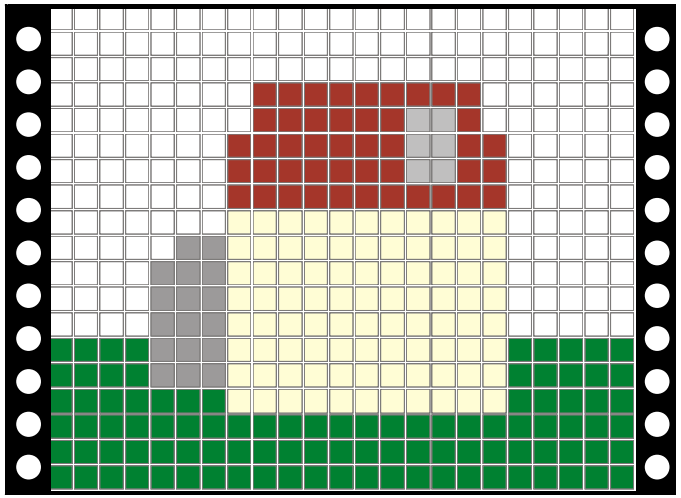
Die Standard-Lösung heute: der Z-Buffer

- Zusätzlich zum Color Buffer
- Speichert pro Pixel den Abstand *z* zur Kamera
- Pixel wird geschrieben, wenn *z* kleiner ist als der Wert im Z-Buffer
- Historische Randnotiz: "ridiculously expensive"



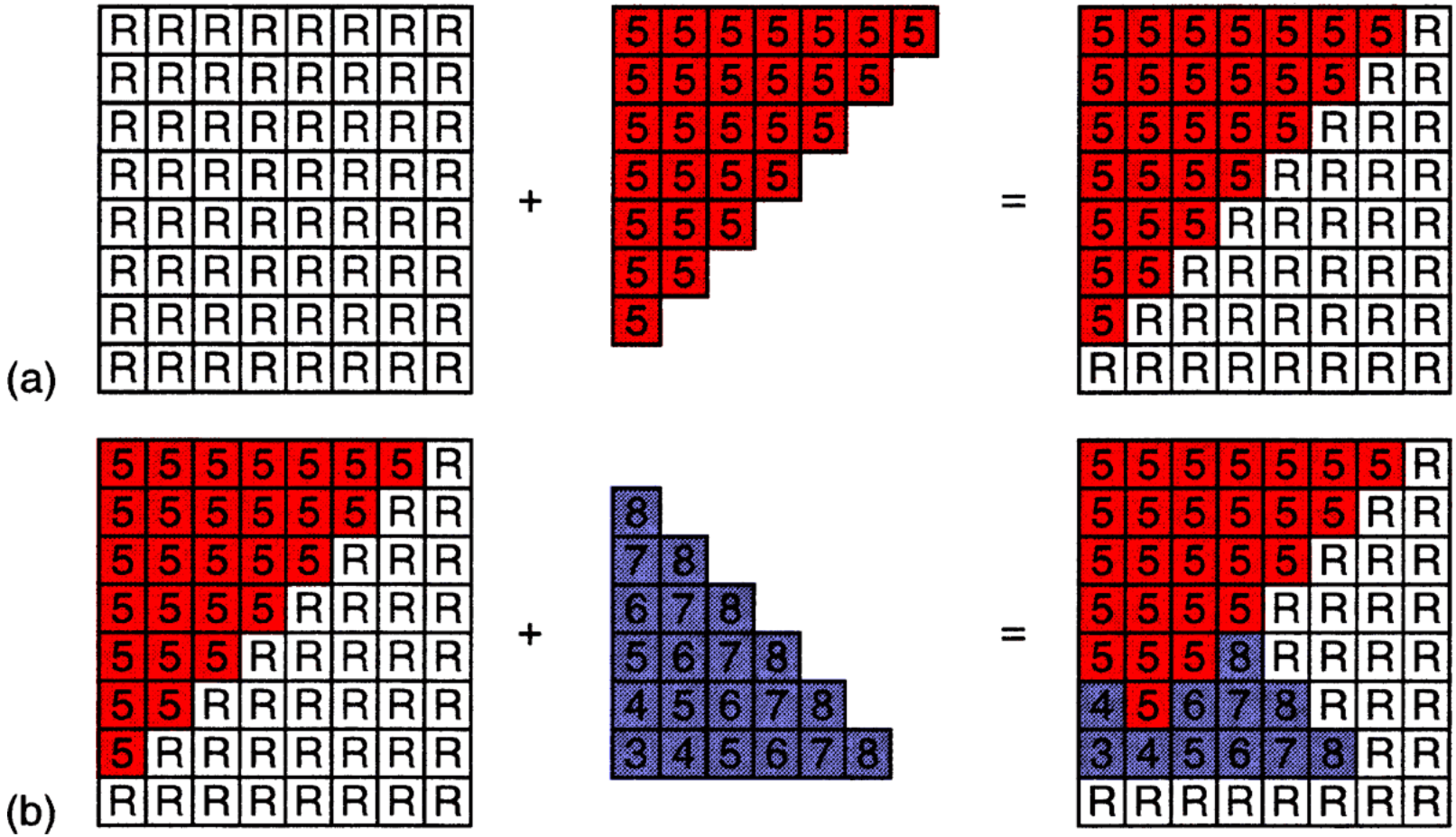


Z-Buffer



Color Buffer

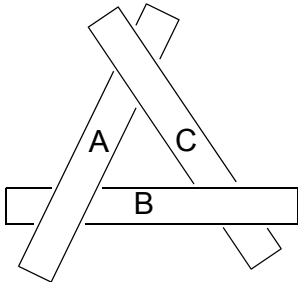
Beispiel

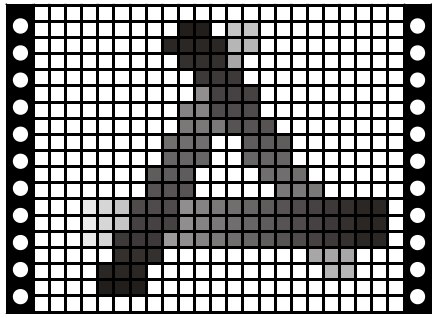


Z-Buffer Pseudo-Code

```
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:

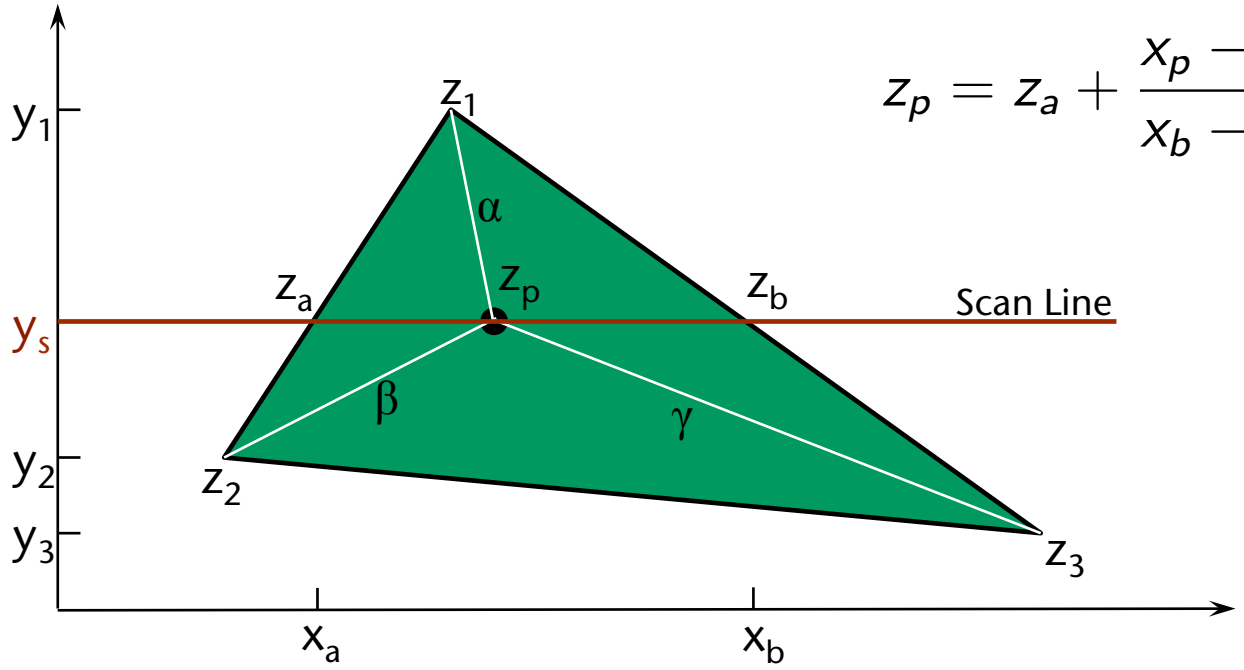




Berechnung des Z-Wertes bei der Scan-Conversion

$$z_a = z_1 + \frac{y_s - y_1}{y_2 - y_1}(z_2 - z_1) \qquad z_b = z_1 + \frac{y_s - y_1}{y_3 - y_1}(z_3 - z_1)$$

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



- 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 );
```

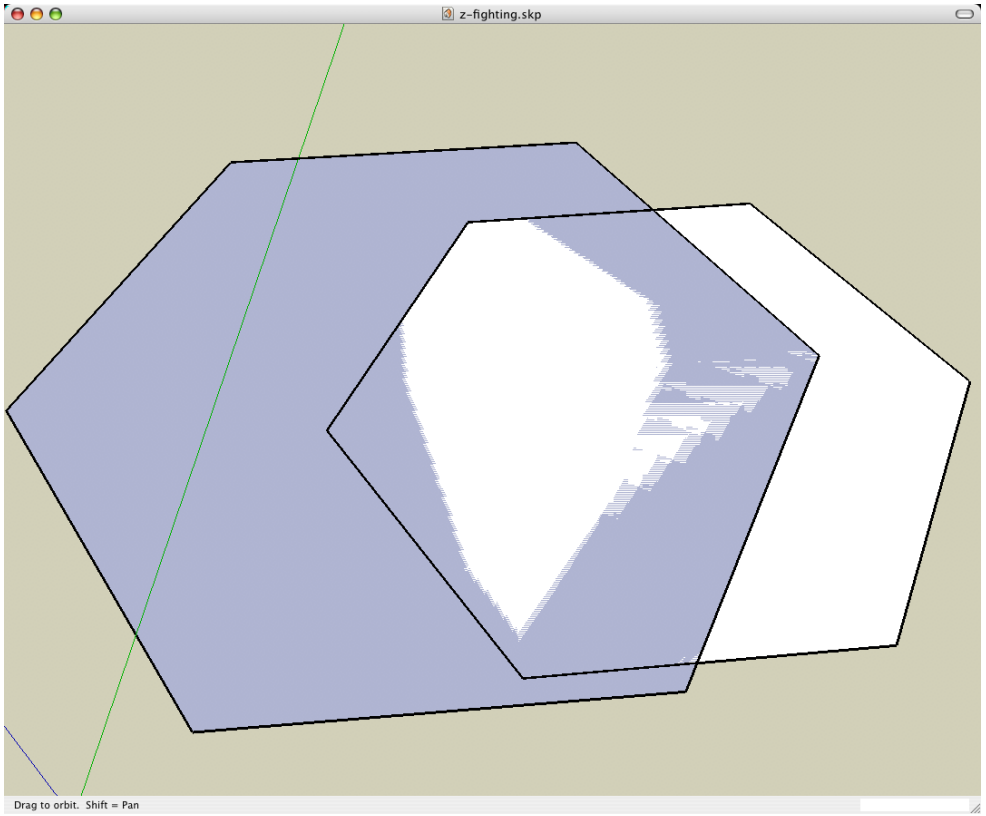
3. Wichtig: nicht nur Bildspeicher, sondern auch Z-Buffer löschen!

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

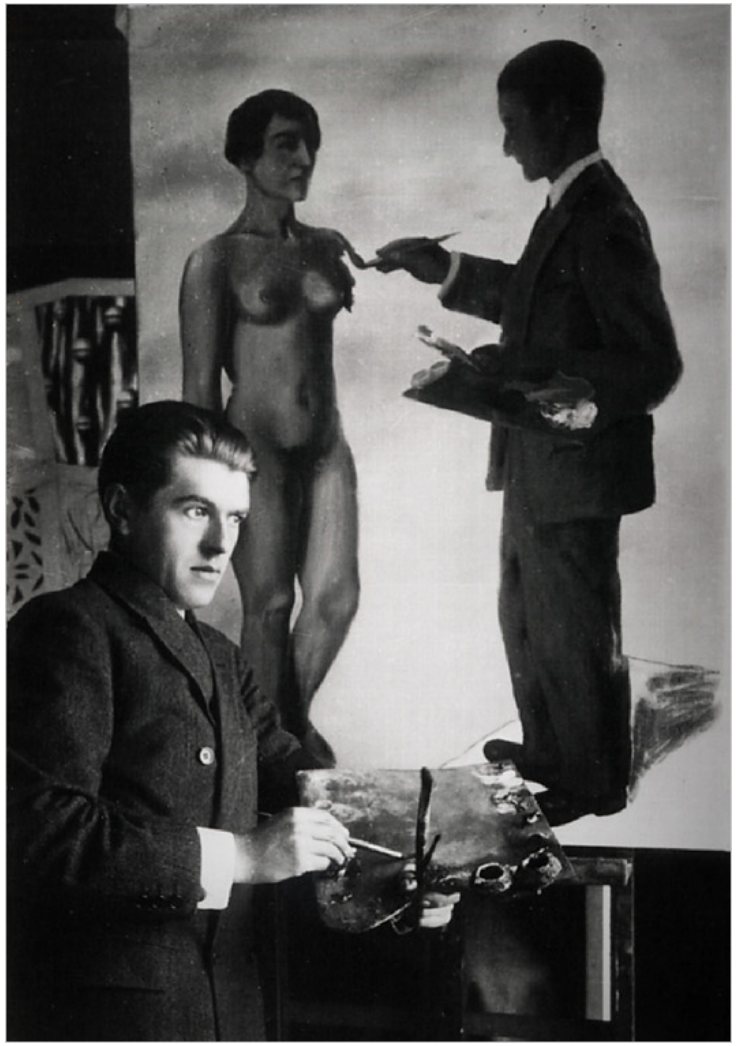
- Achtung: unter Qt ist (1) und (2) per Default angeschaltet
  - Mehr Info unter
    http://www.qt-project.org/doc/qt-4.8/QGLFormat.html
  - Beispiel zu QGLFormat im "OpenGL/Qt-Programmbeispiel" auf der Homepage der Vorlesung

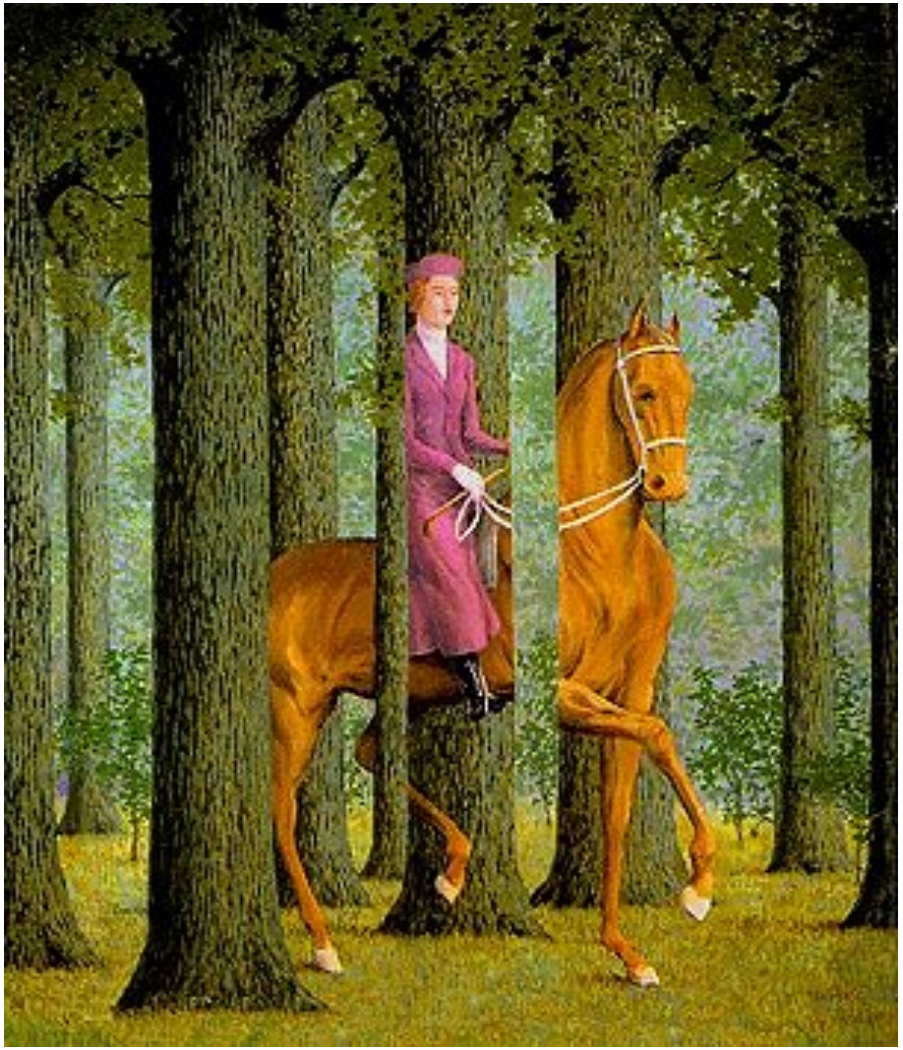
Z-Fighting

- Wegen der begrenzten Auflösung des Z-Buffers kommt es bei koplanaren oder fast koplanaren Polygonen zum sog. Z-Fighting:



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





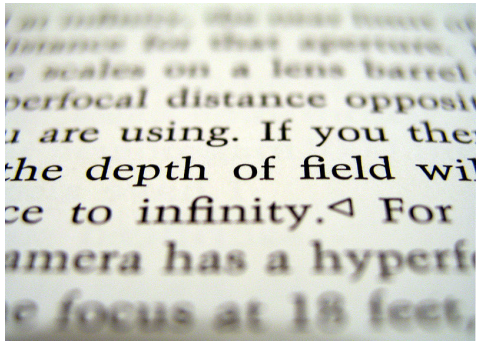
René Magritte, Self-portrait in front of his painting
"Attempting the Impossible", Le Perreux-sur-Mame, 1928

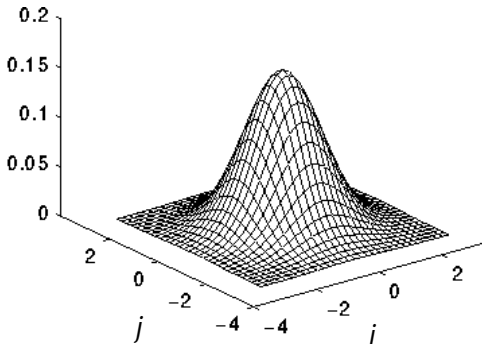
Weitere Anwendung des Z-Buffers: Depth-of-Field

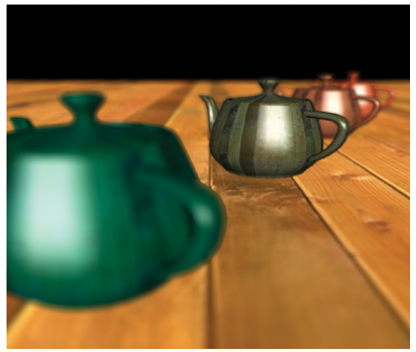
- Tiefenunschärfe (Schärfentiefe, *depth-of-field*) = je weiter ein Objekt von der Fokusebene (*focal plane*) entfernt ist, desto "verschmierter" das Bild
- Die Idee: Image Post-Processing
  - Sei z_0 = Abstand der Fokusebene vom Viewpoint
  - Wende auf unscharfe Pixel einen Blur-Filter mit Radius r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

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

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



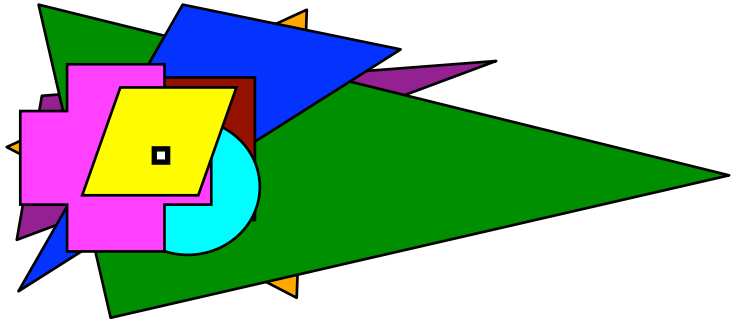




Bewertung des Z-Buffers zur Verdeckungsrechnung

- 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 = # geschriebene Pixel (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

- Frage: 10 Dreiecke überdecken ein Pixel — wie groß ist die *erwartete* Anzahl Schreib-Operationen in den Color-Buffer ?



- Lösung:

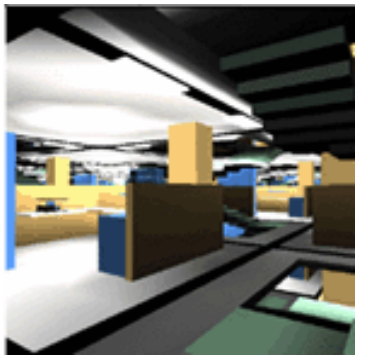
$$1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{10} \approx \ln(10) + 0.55... = 2.9289\ldots$$

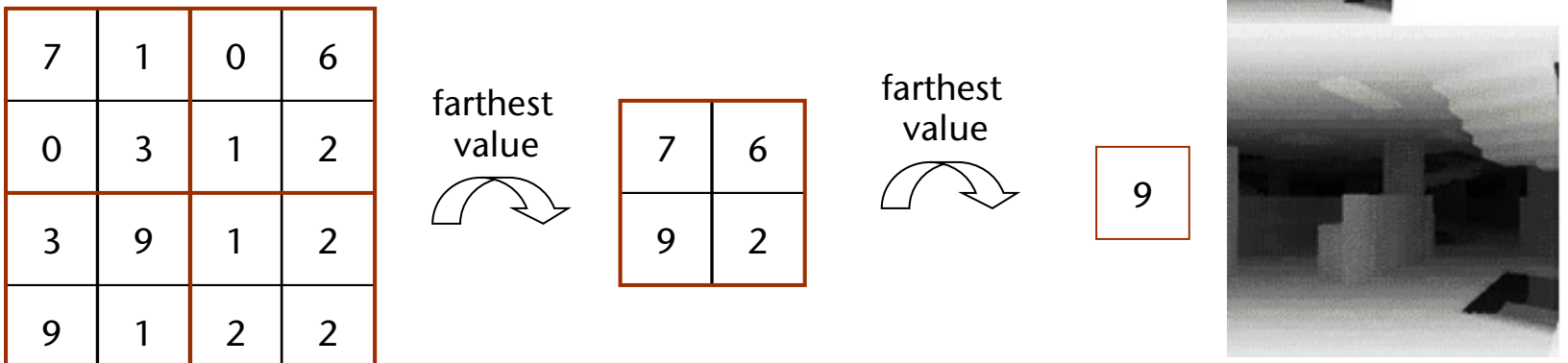
- Definition Depth-Complexity:
  - Anzahl Polygone der Szene, die "hinter" einem Pixel liegen
  - Manchmal auch diese Def.: Anzahl Z-Tests pro Pixel
- Definition Over-Drawing = Maß dafür, wie oft ein Pixel tatsächlich überschrieben wird

Der Hierarchische Z-Buffer (HZB)

[Greene, 1993]

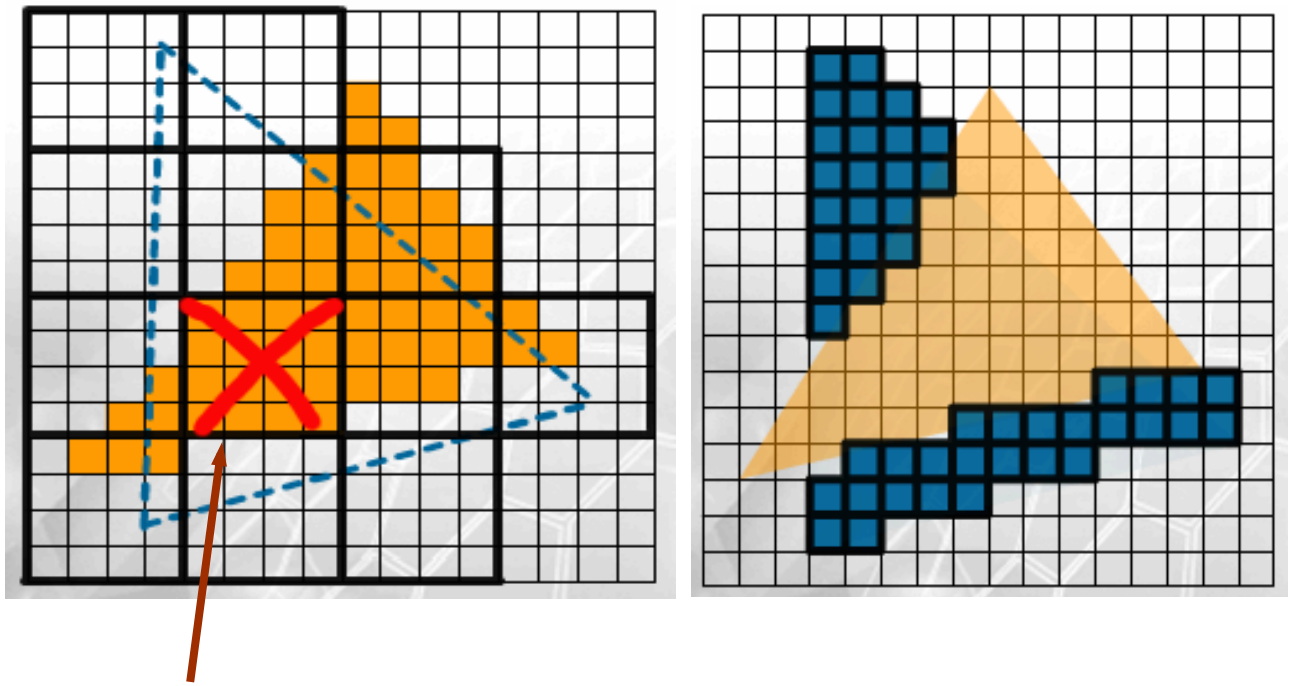
- Frage: wie kann man auch bei großer Depth-Complexity den Overdraw gering halten? (auch im worst case)
- Idee: „Z-Pyramide“
  - einfacher Z-Buffer = höchste Auflösung
  - weitere Levels durch Zusammenfassen von jeweils 4 Pixel
  - Z-Wert auf max. Z-Wert setzen





Beispiel

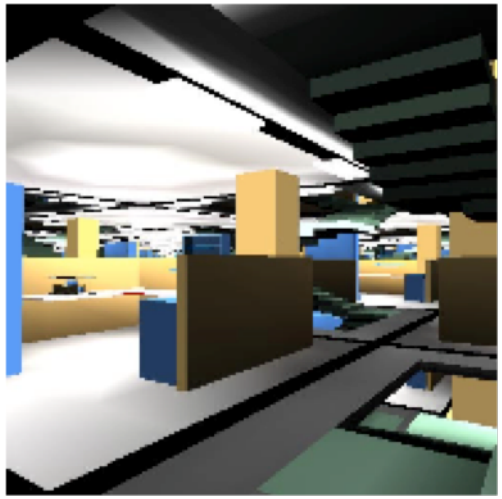
- Sei orangenes Dreieck bereits gezeichnet
- Blaues Dreieck soll (dahinter) gezeichnet werden



vollständig verdeckt →
verwerfe gesamten Block

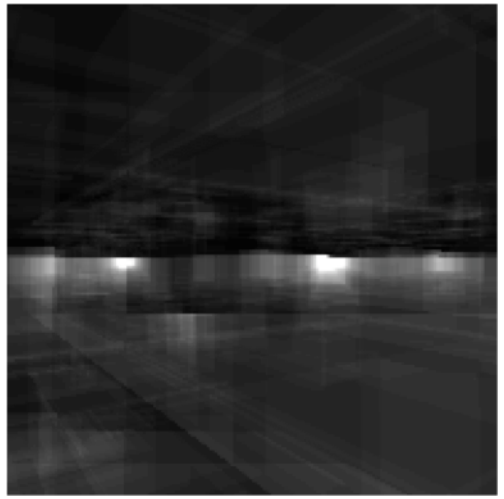
Vergleich

540 Mio Polygone



Overdraw mit
einfachem Z-Buffer

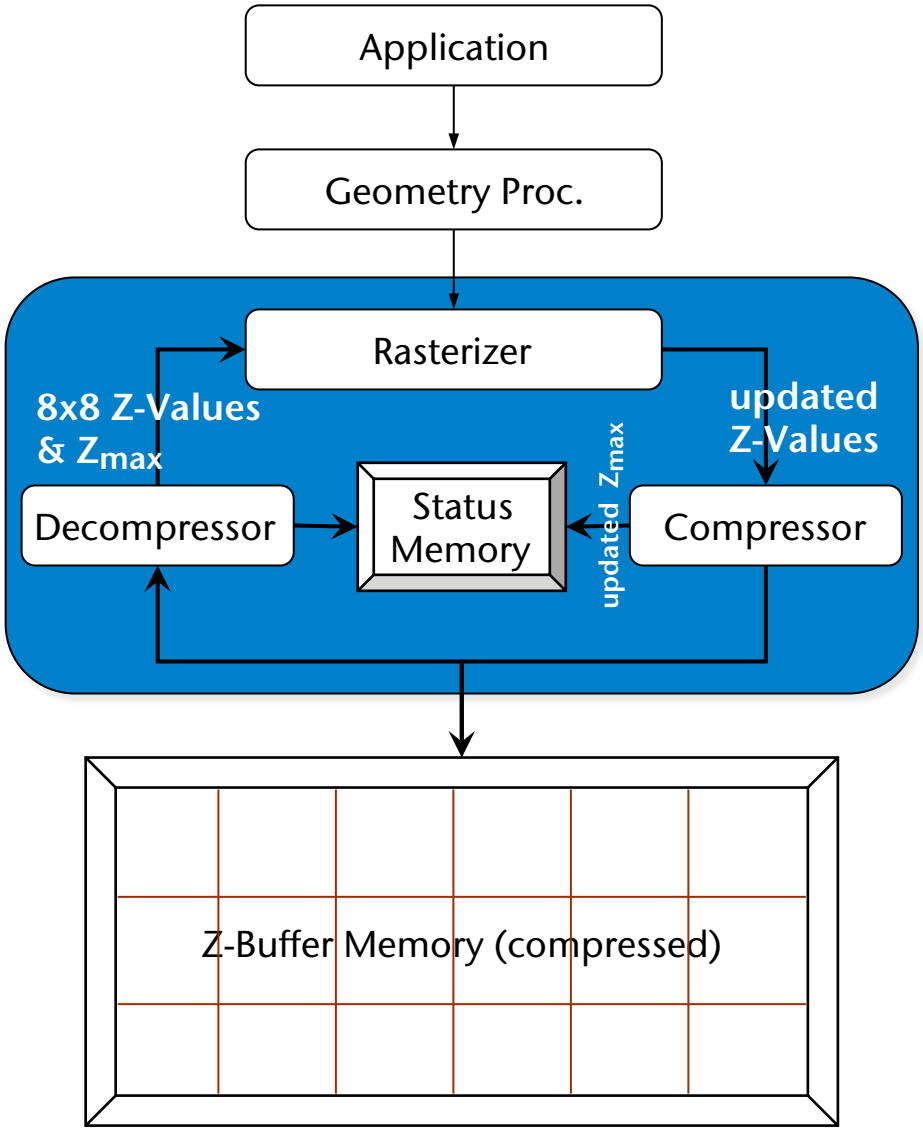
Overdraw mit HZB



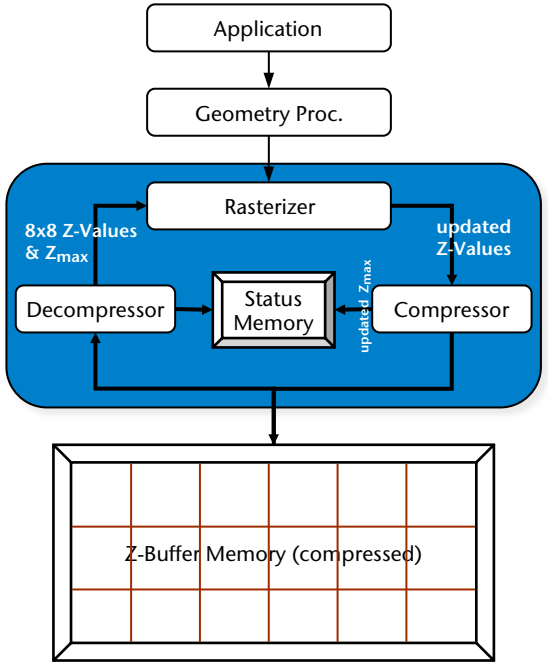
Implementierung in aktueller Graphik-Hardware

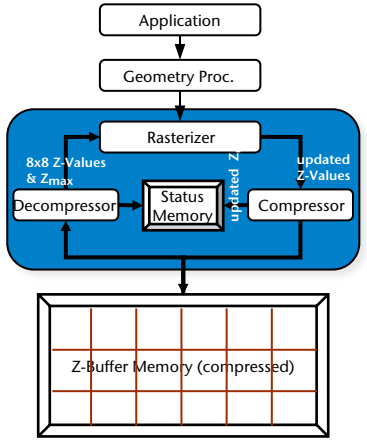
- Problem: Bandbreite zwischen Rasterizer und Speicher ist limitiert
  - Annahmen: Auflösung 1280x1024, 4x depth complexity pro Pixel
  - I/O-Aufwand pro Fragment: 1x Z-Buffer-Read + 1x Z-Buffer-Write + 1x Color-Buffer-Write + 2x Texture-Read, pro Read/Write 32 Bit
  - ➢ Ergibt ca. 18 GByte/sec!
  - Aktueller Speicher erlaubt ca. 10 GByte/sec [2002]
- Wie implementiert man schnell `glClear(DEPTH_BUFFER_BIT)`?
- Wie implementiert man den HZB?
- Lösung:
  - 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
  - Versucht, Z-Werte der Kachel zu komprimieren
  - Falls klappt: setze Status auf "compressed", sonst "uncompressed"
  - Schreibe un-/komprimierte Kachel in Z-Buffer

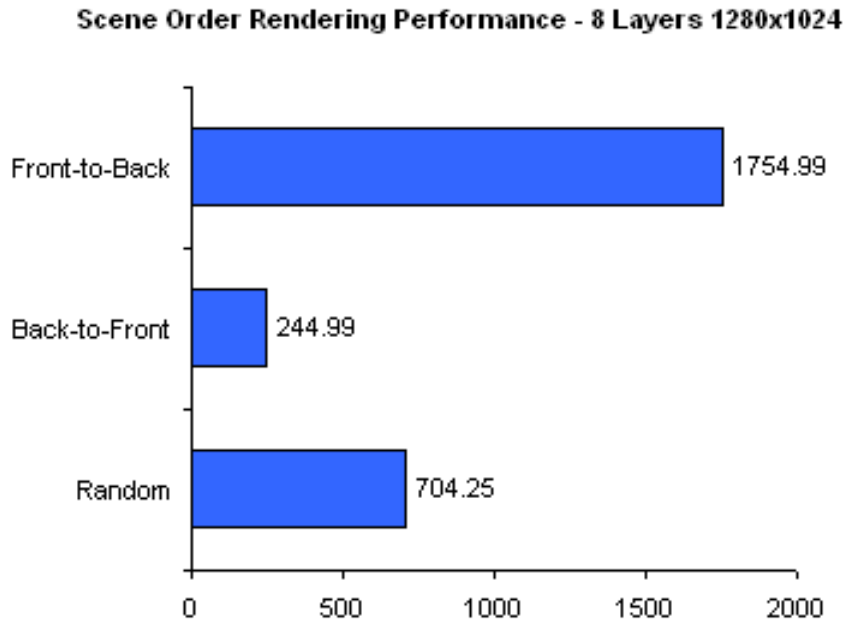




- Z-Buffer lesen:
  - Decompressor liest zuerst Z_{max} aus dem Status Memory
  - Verschiedene Tests möglich ("early z rejection"):
    - Teste die Z-Werte der 4 Ecken der Kachel gegen Z_{max}
      - Bemerkung: diese 4 Z-Werte kann man in den Nachbarkacheln wiederverwenden
    - Teste die Z-Werte der 3 Ecken des Dreiecks gegen Z_{max}
    - Berechne alle Z-Werte der Pixel 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
- Nennt sich *"HyperZ"* oder "*Lightspeed Memory Architecture*" bei den Graphikkartenherstellern
- Kompression (z.B. DPCM) erreicht bis zu Faktor 4:1

Performance-Gewinn in einer Graphikkarte

- Beispiel: ATI RADEON 9700 PRO [2003]
  - 3 Levels: 1. 8x8 Z-Block , 2. 4x4-Block, 3. "Early Z"-Test
- Performance-Gewinn:



Erinnerung: an welcher Stelle der Graphik-Pipeline befinden wir uns

Modell Transformation

Illumination (Shading)

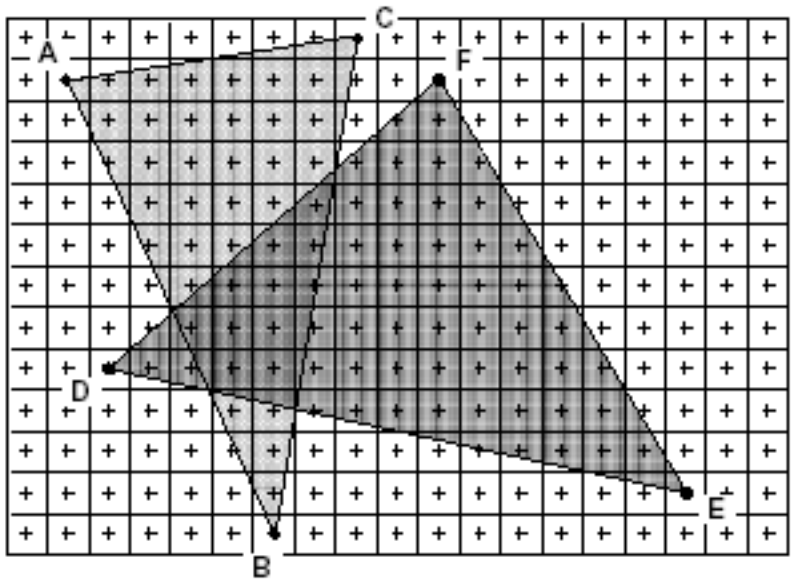
Viewing Transformation (Perspective / Orthographic)

Clipping

Projektion (in Screen Space)

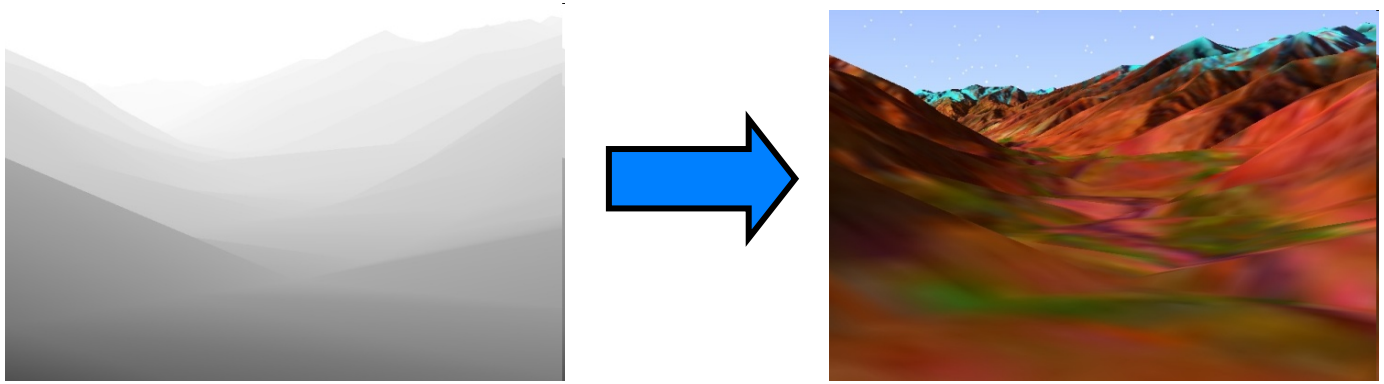
Scan Conversion (Rasterization)

Visibility / Display



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

- Spezielles Feature aktueller Graphikkarten: Durchsatz ist *doppelt* so hoch, falls **nur** der Z-Buffer geschrieben wird (nicht Color-Buffer)
- Trick:
  - Schalte Color-Buffer aus, nur Z-Buffer an
  - Rendere Szene 1x "ohne alles" (keine Lichtquellen, keine Texturen, keine Farben), schreibt nur Z-Buffer = *"lay down depth"*
  - Rendere Szene noch 1x "mit allem" → HZB kann voll wirken
    → 0 Overdraw



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 Objekte/Polygone andere verdecken
  - Berechnung basiert oft auf dem Aufbau komplexer Hilfsdatenstrukturen
  - Funktioniert besser bei statischen Szenen

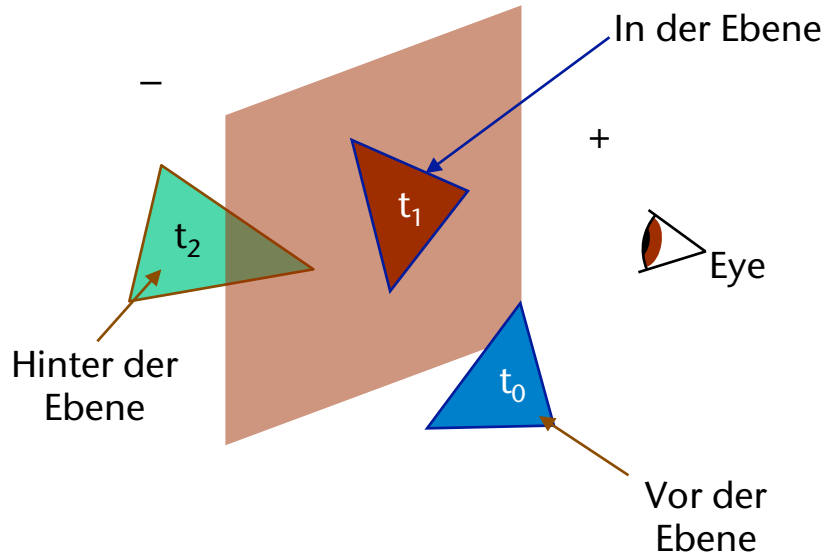
Binary Space Partition (BSP) Tree [ca. 1982]

- Ein Object-Space-Algorithmus/-Datenstruktur
  - Generell für Polygon-Sortierung bzgl. eines bestimmten Punktes im Raum
- Ansatz: rekursive Unterteilung des Raumes entlang Polygon-Ebenen (*for depth sorting*)
- 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 2 Polygone schneiden sich
- F_p sei die implizite Gleichung der Ebenengleichung die das Polygon p enthält
- 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
```

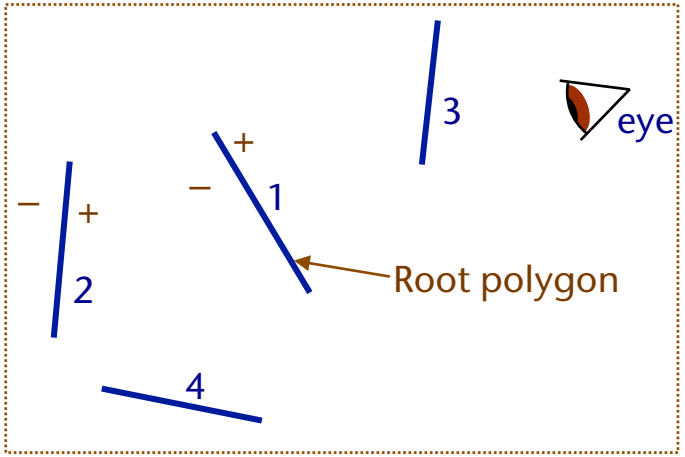


- Funktioniert für beliebigen Viewpoint!

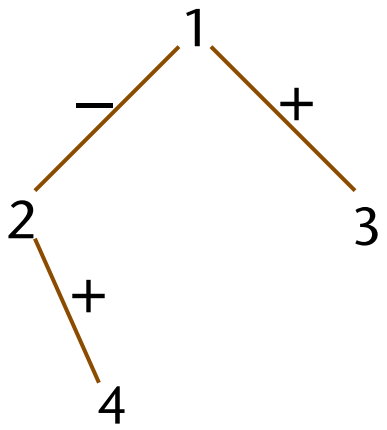
Der Rendering-Algorithmus mit BSPs

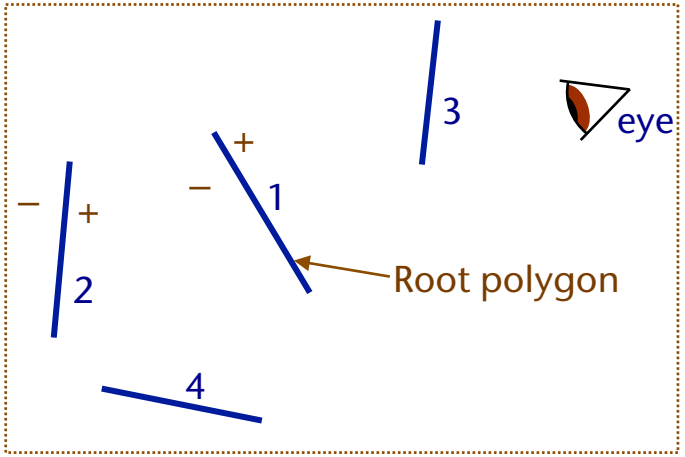
- BSP Tree: unterteilt Raum rekursiv in positive und negative Teile
  - Knoten = speichert Zeiger auf Polygon(e) und die Ebenengleichung, in der diese Polygone liegen
  - Linker / rechter Sohn = Unterbaum, der all Polygone enthält, die im negativen / positiven Halbraum der Ebene liegen
  - Polygone, die auf beiden Seiten liegen, werden gesplittet
- Rendering (von hinten nach vorne):
  - Beginne bei der Wurzel

1. Zeichne Polygone rekursiv auf der Gegenseite vom Viewpoint
2. Zeichne Polygon(e) im Knoten
3. Zeichne rekursiv Polygone auf der selben Seite wie Viewpoint



```
draw(node, eye):
  if node.empty():
    return;
  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 )
```





Reihenfolge beim Zeichnen: 2, 4, 1, 3

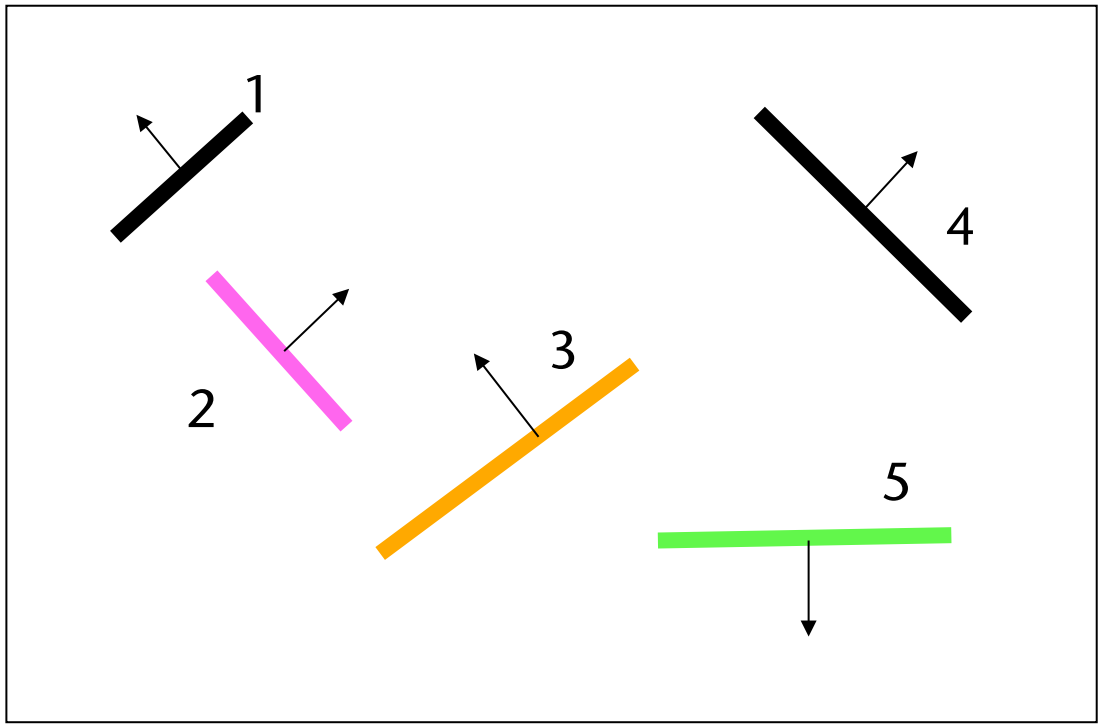
Rekursiver Aufbau eines BSP Tree

1. Wähle ein Polygon und setze es als Wurzelelement
2. Partitioniere die Menge der restlichen Polygone in zwei Teilmengen, je nachdem auf welcher Seite sie liegen
3. Schneidet ein Polygon die Ebene, dann unterteile es in zwei Polygone, jeweils ein Teil auf einer Seite (*splitting*)
4. Baue rekursiv je einen BSP für alle Polygone auf der negativen bzw. positiven Seite und hänge diese als Kinder an die Wurzel
5. Stoppe, wenn ein Unterbaum nur noch ein Polygon enthält

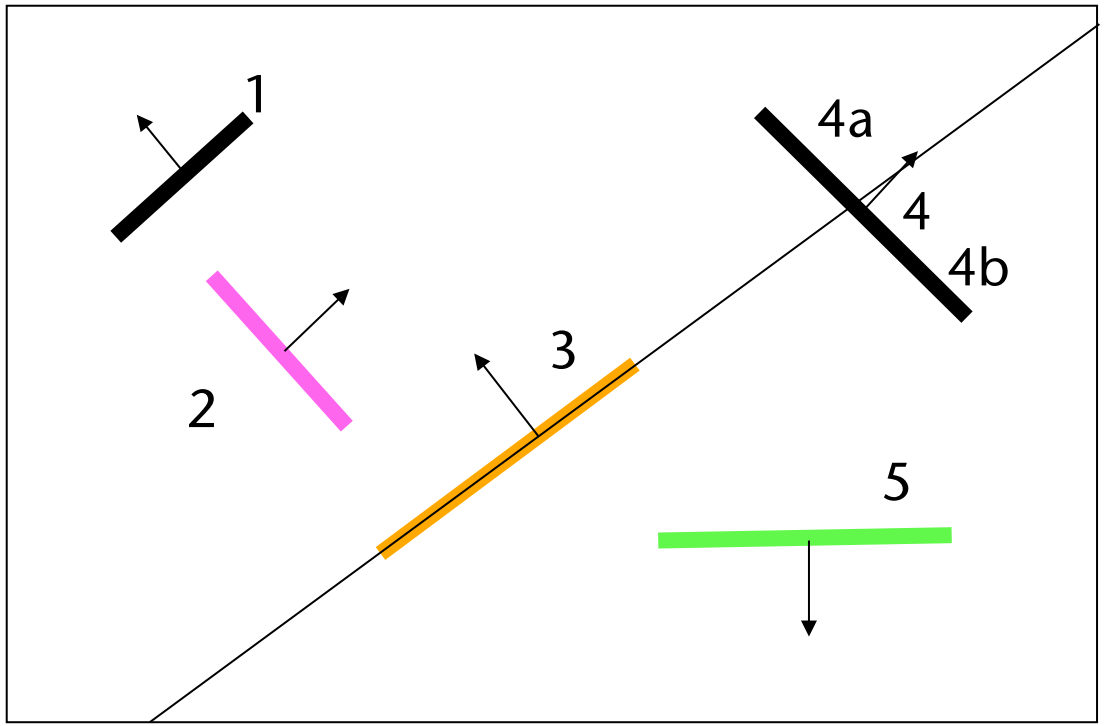
- NB: diese Art BSP heißt Auto-Partition

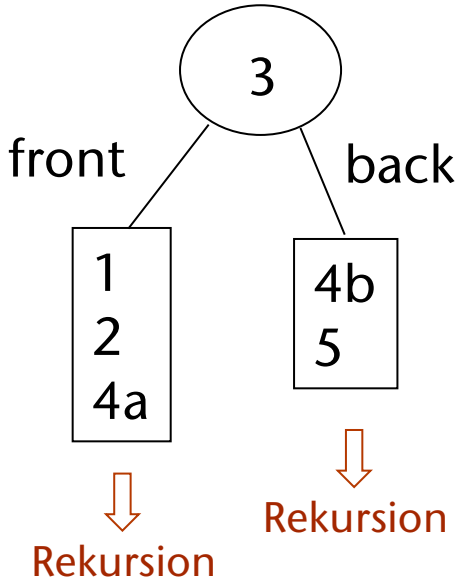
Beispiel

- Ausgangsszene:

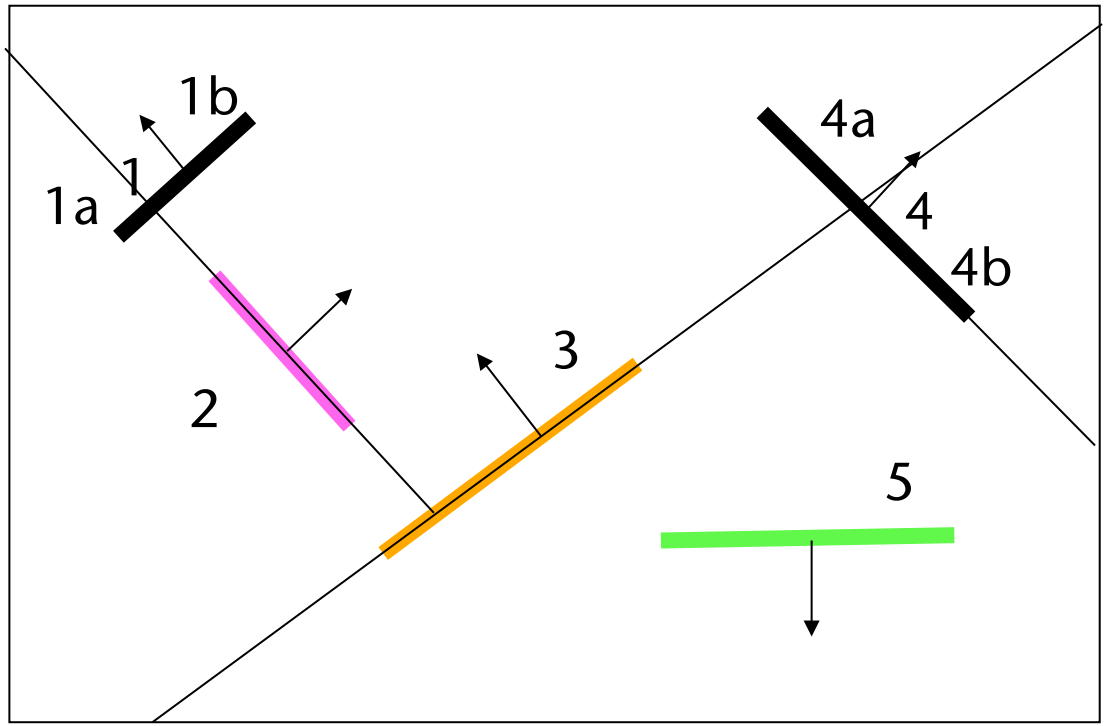


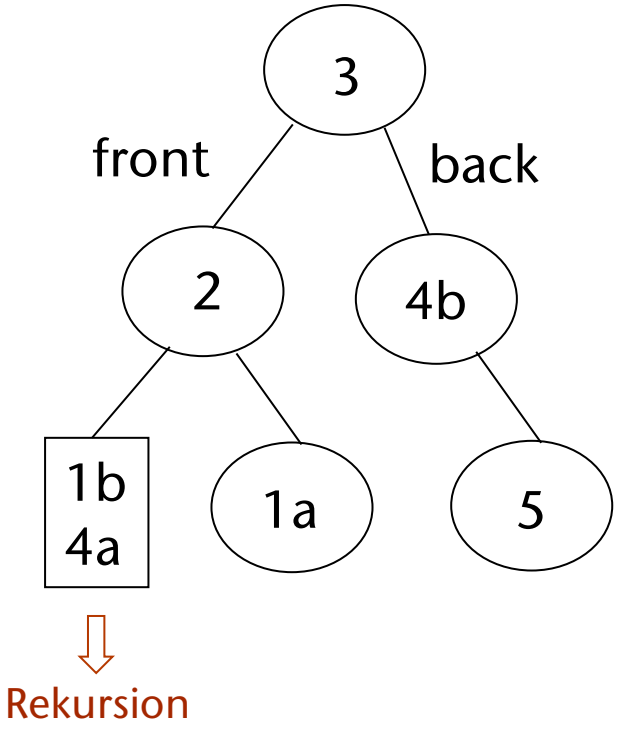
- Wähle z.B. Polygon 3 als Wurzelelement



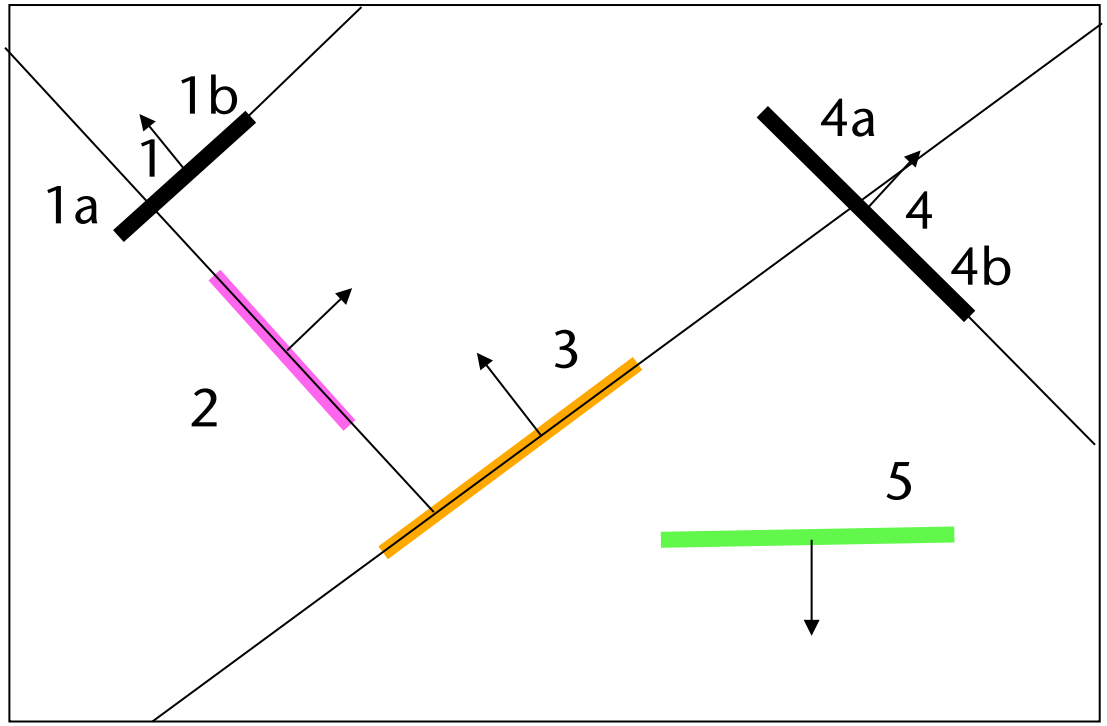


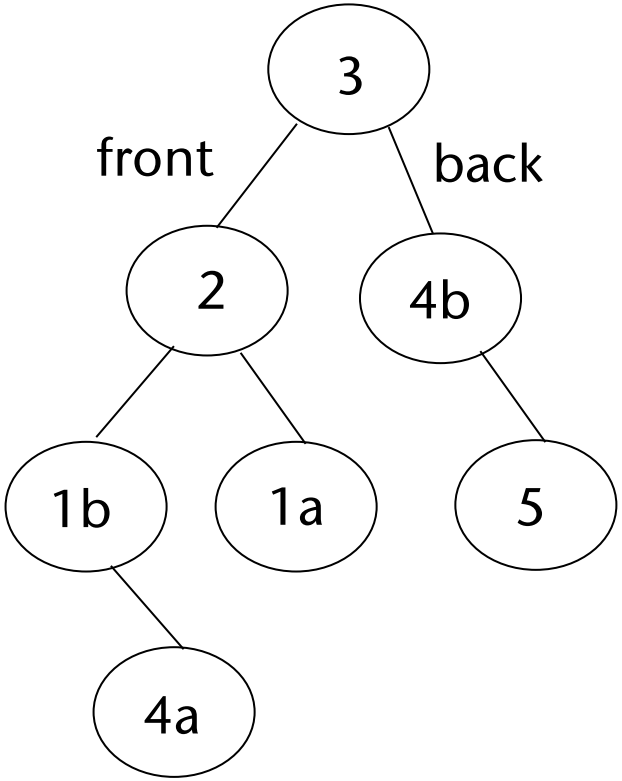
- Wähle Polygon 2 und 4b als nächste Knoten





- Nun wähle Polygon 1b als Knoten





Der Pseudo-Code zur Konstruktion

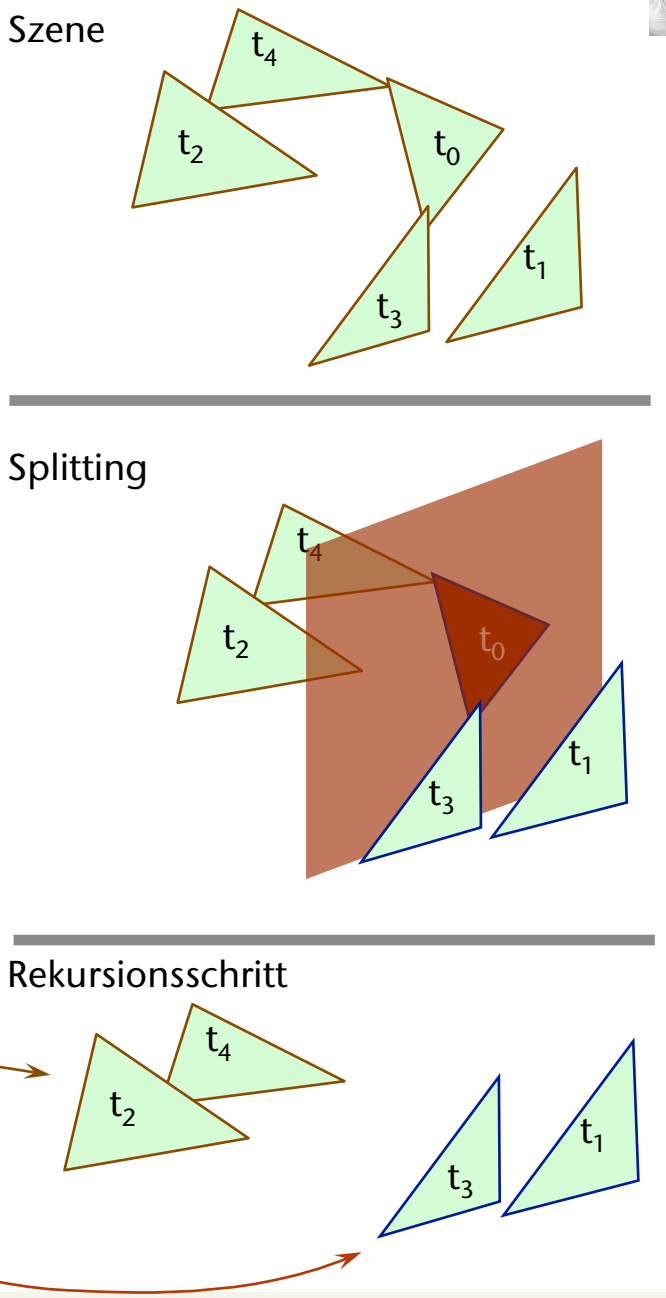
```
buildBSP( scene ):

  node = new Node
  node.t = scene[0]
  front = new Scene
  back  = new Scene

  if scene.size > 1:
    for i in scene[1..len-1]:
      if scene[i] intersects node.plane
        split scene[i]
        do following test with each part
      if scene[i] on back-side of node.plane:
        back.push( scene[i] )
      else
        front.push( scene[i] )

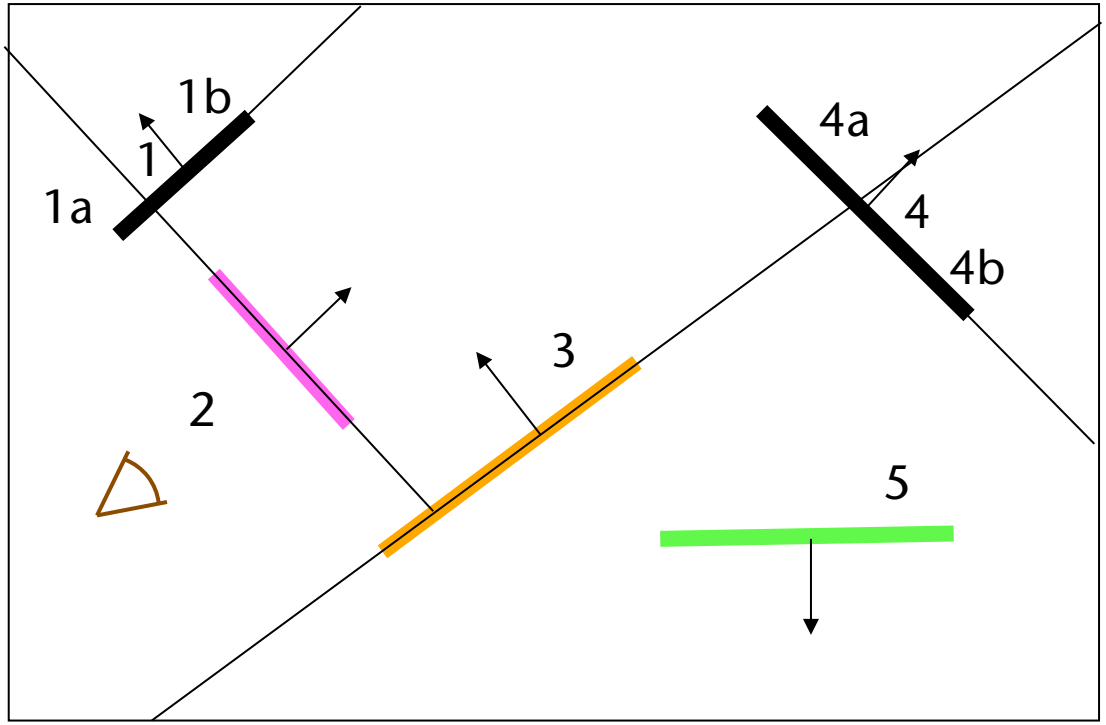
  node.front = buildBSP( front )
  node.back = buildBSP( back )

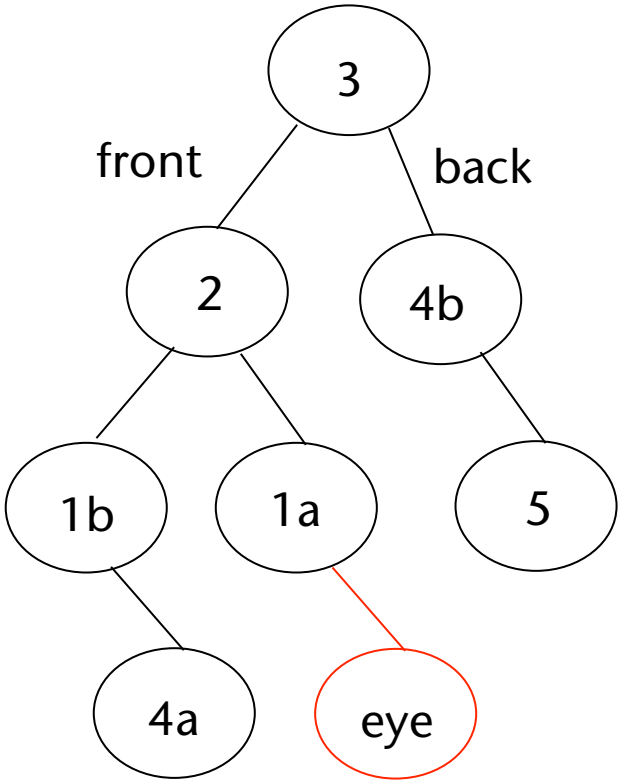
  return node
```



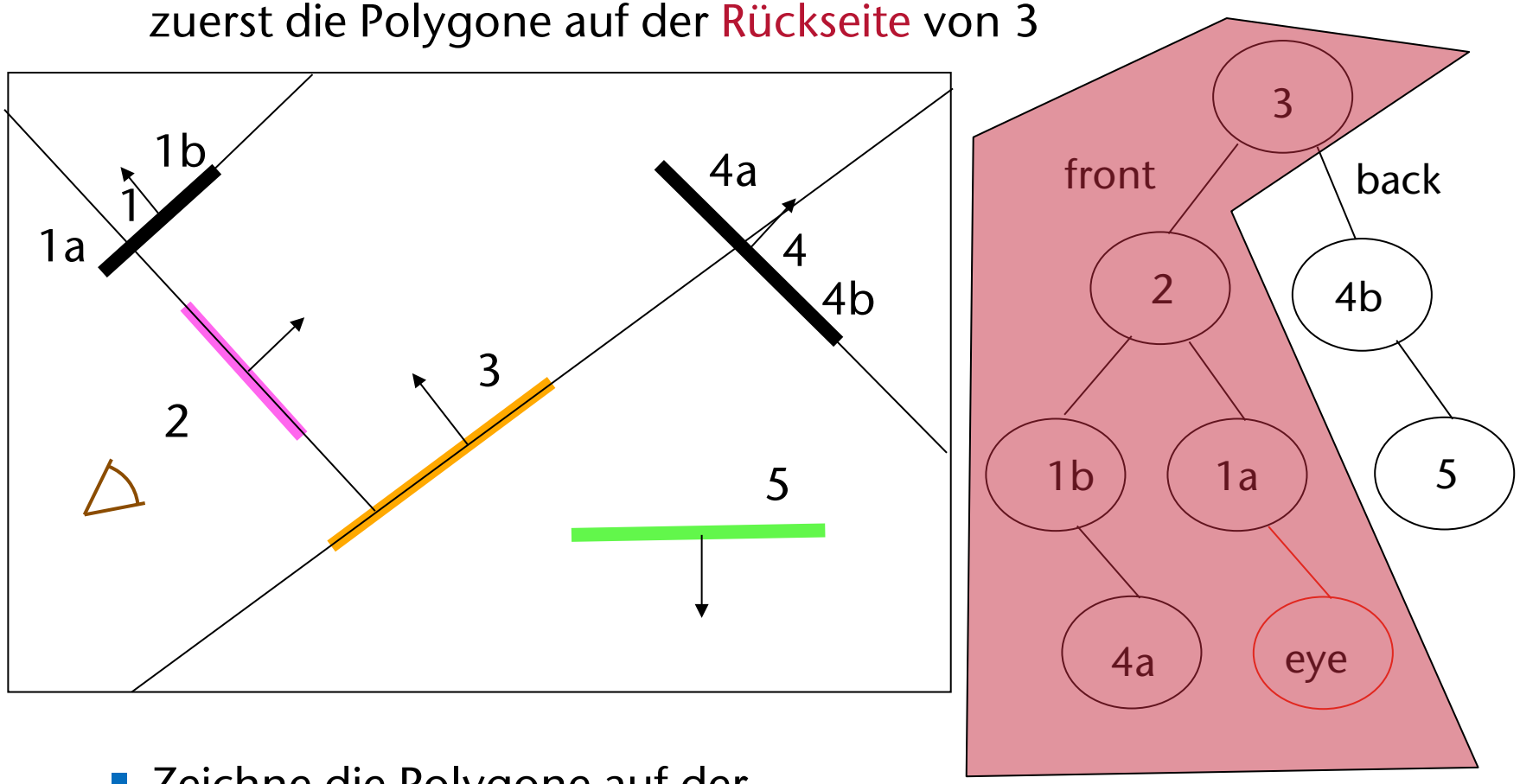
Beispiel fürs Rendering mittels BSP

- Angenommen, der Viewpoint befindet sich wie hier dargestellt



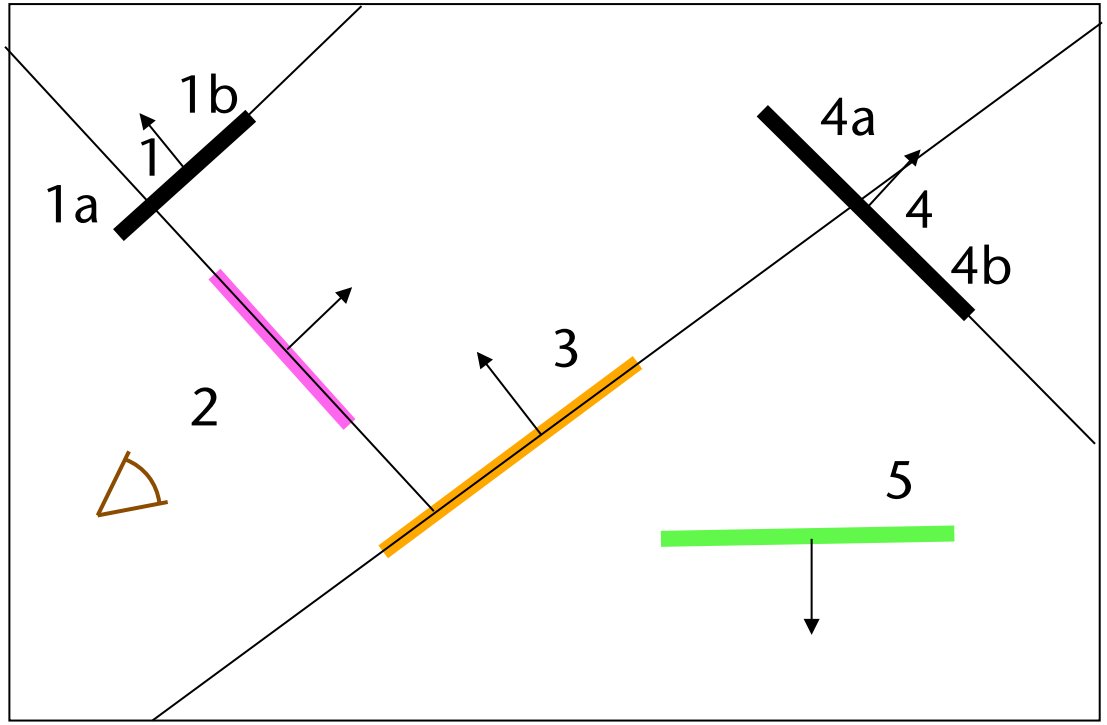


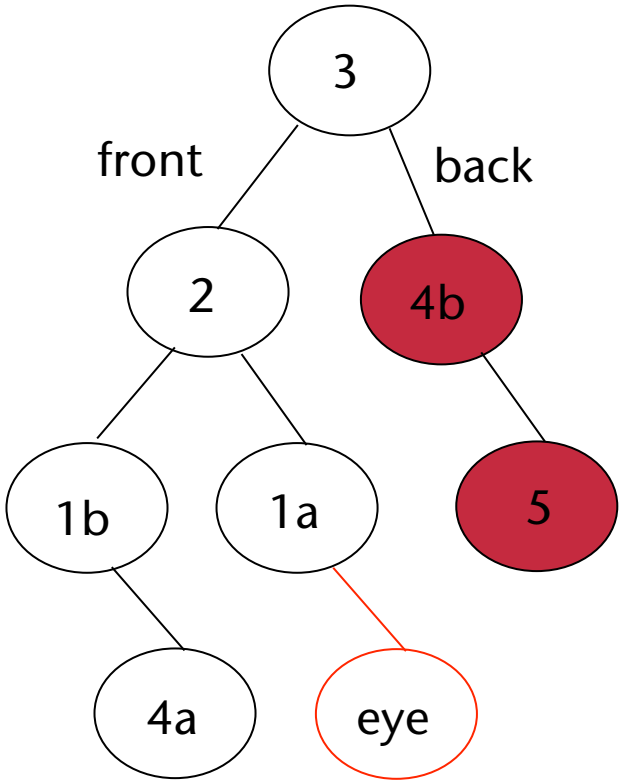
- Viewpoint liegt auf der Vorderseite von 3, somit zeichnen wir zuerst die Polygone auf der Rückseite von 3



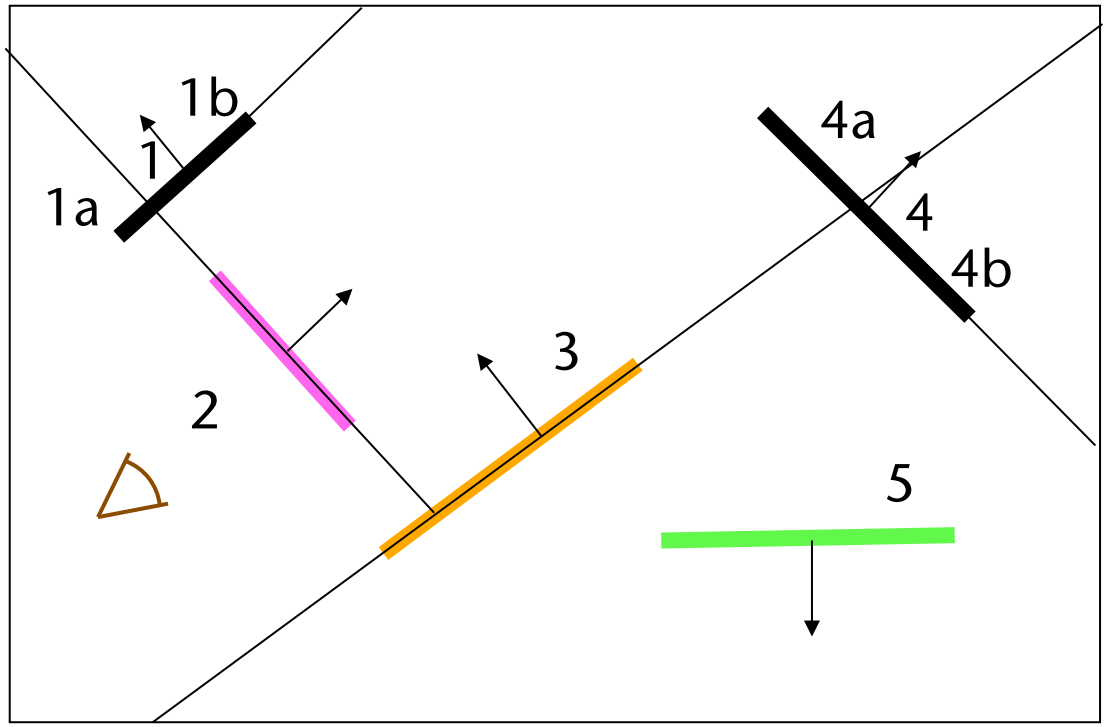
- Zeichne die Polygone auf der Rückseite von 4b, also 5, und im Anschluss 4b

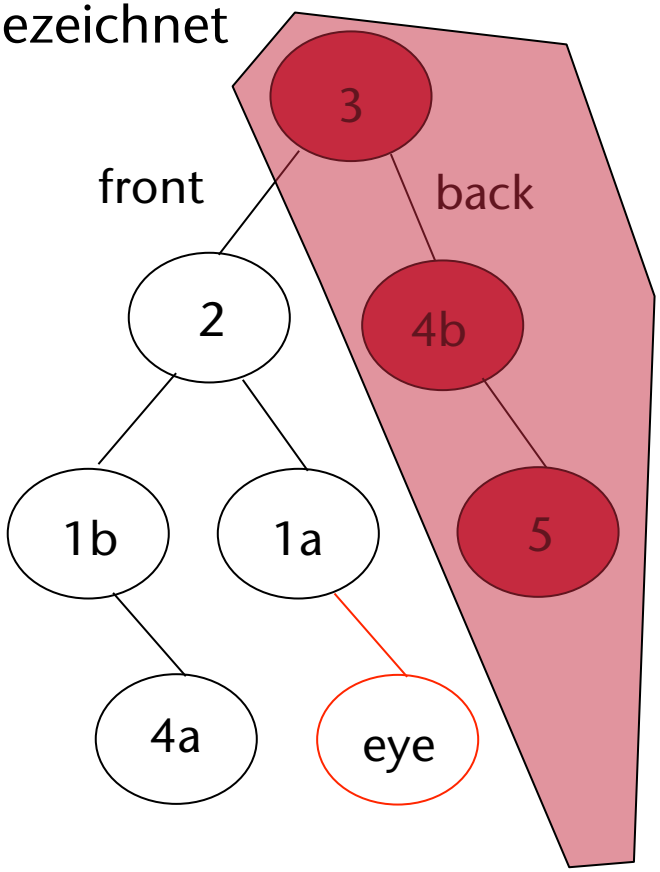
- Nun zeichnen wir Polygon 3 und dann die Vorderseite von 3



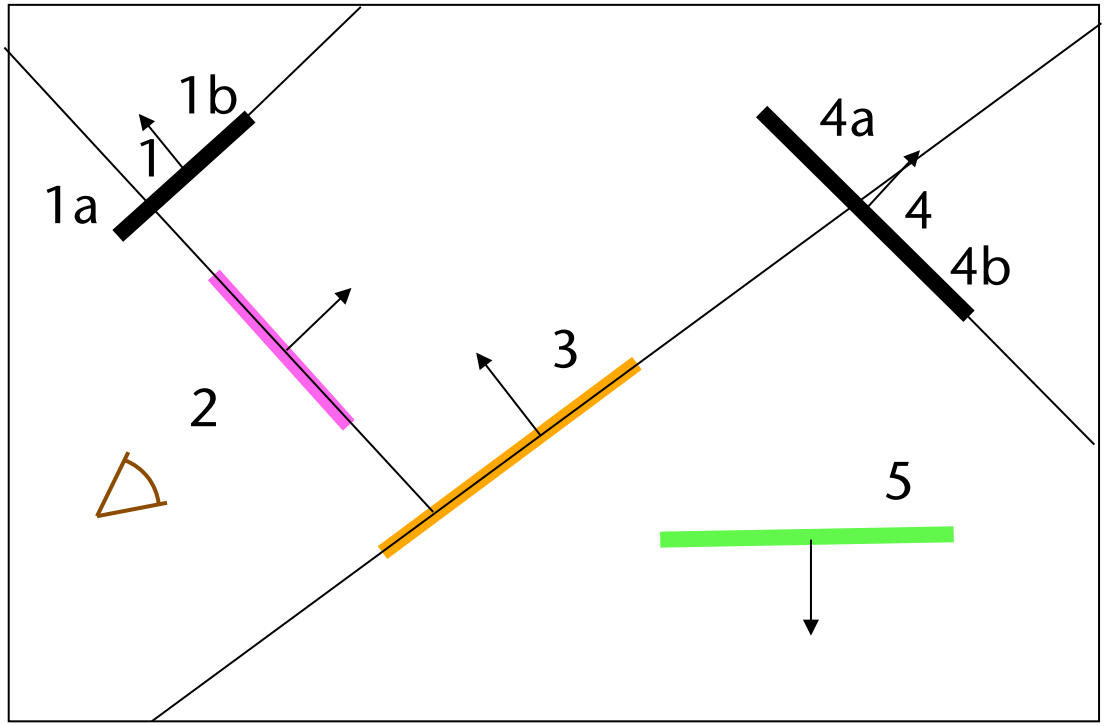


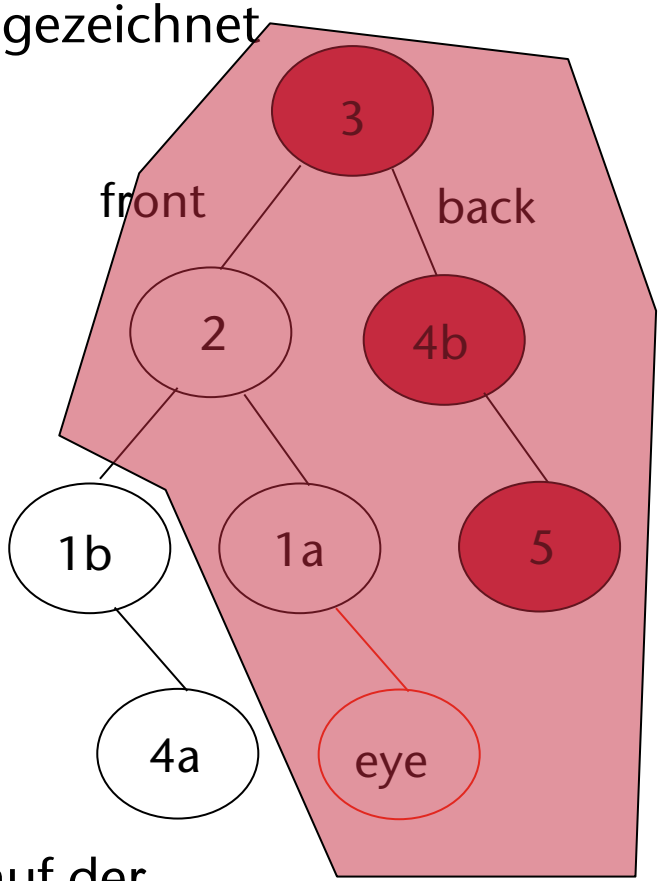
- Der Viewpoint liegt auf der Rückseite von 2, somit werden erst die Polygone auf der Vorderseite von 2 gezeichnet





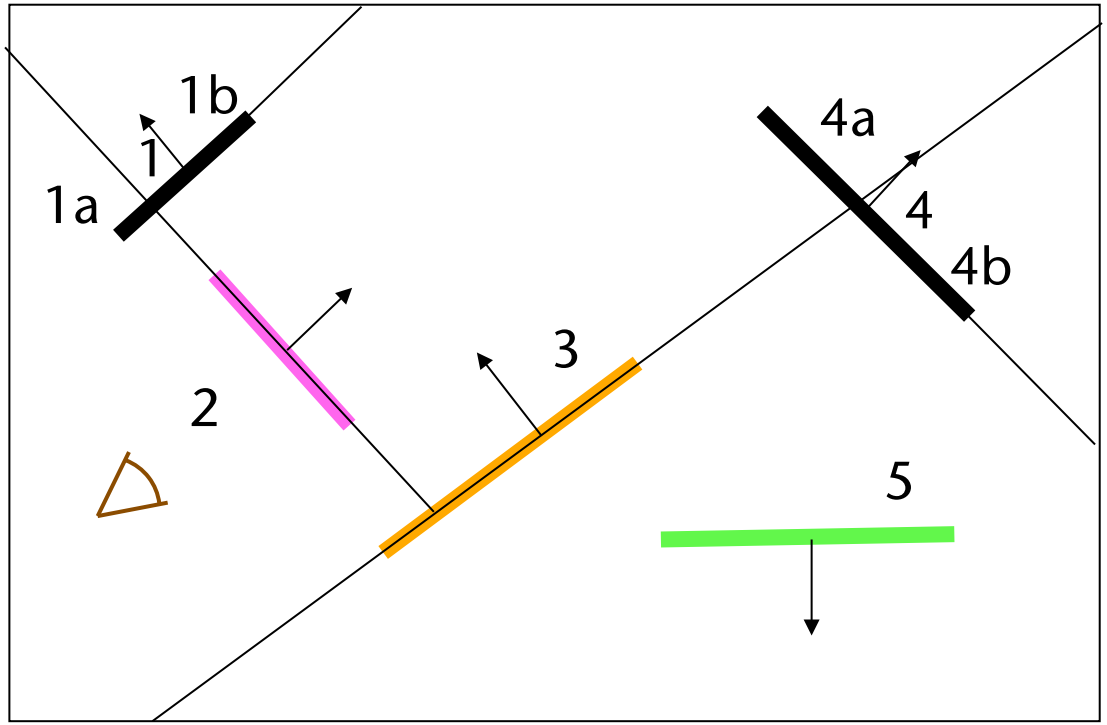
- Der Viewpoint liegt auf der Rückseite von 1b, somit werden erst die Polygone auf der Vorderseite von 1b gezeichnet

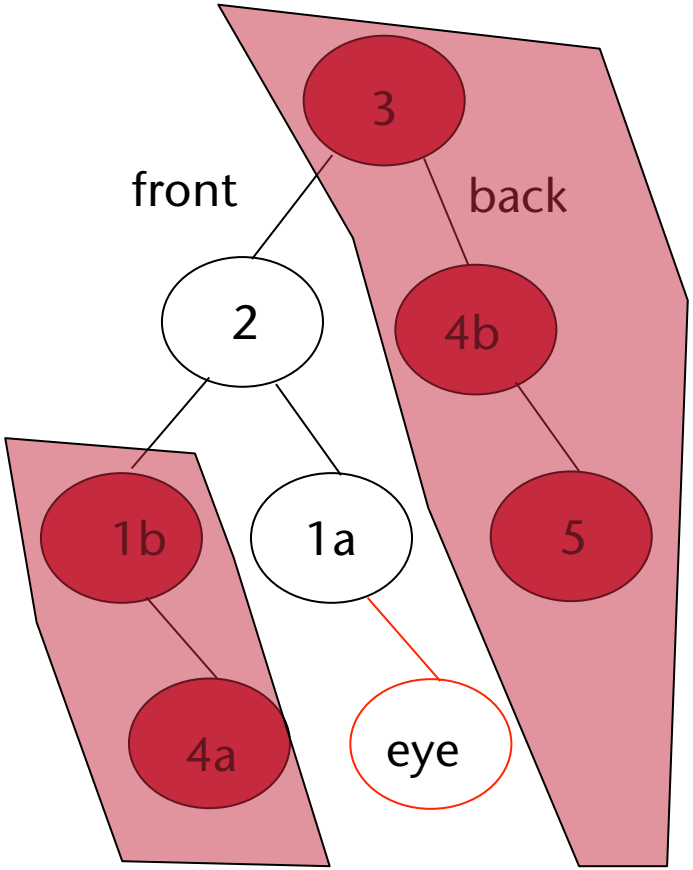




- Zeichne dann 1b, danach die Polygone auf der Rückseite von 1b, also 4a

- Danach zeichne 2

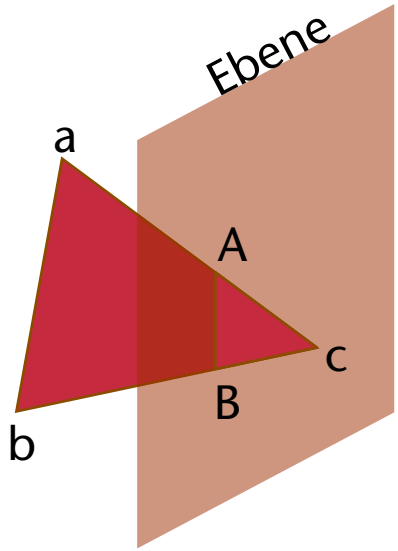


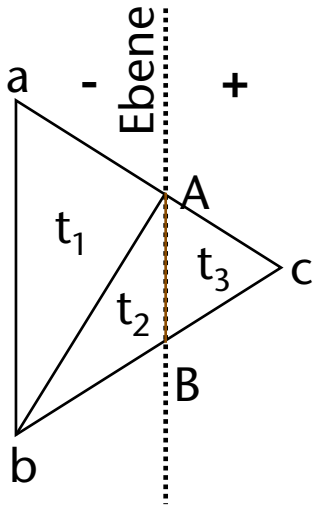


- Im Anschluss die Polygone auf der Rückseite von 2, also 1a
- Ergibt Gesamtreihenfolge: 4b, 5, 3, 1b, 4a, 2, 1a

Zerschneiden von Dreiecken

- Dreieck schneidet die Ebene → unterteilen

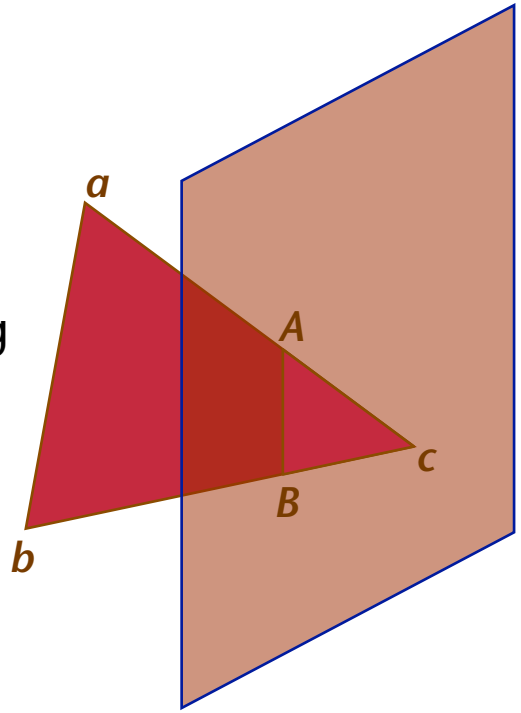




$t_1 = (a, b, A)$
$t_2 = (b, B, A)$
$t_3 = (A, B, c)$

- Achtung: Reihenfolge der Eckpunkte muß beibehalten werden, damit sich Normale nicht ändert
- Angenommen c liegt allein auf einer Seite der Ebene und es gilt $f_{plane}(c) > 0$, dann:
  - Füge t_1 und t_2 in den negativen Unterbaum ein
  - Füge t_3 in den positiven Unterbaum ein

- Wie bestimmt man A und B ?
  - A: Schnittpunkt der Gerade zwischen a und c und der Ebene f_{plane}
  - Verwende Parameterform der Geradengleichung p(t) = a + t(c - a)
  - Setze p in die Ebenengleichung ein:



$$f_{plane}(p) = (n \cdot p) - d$$

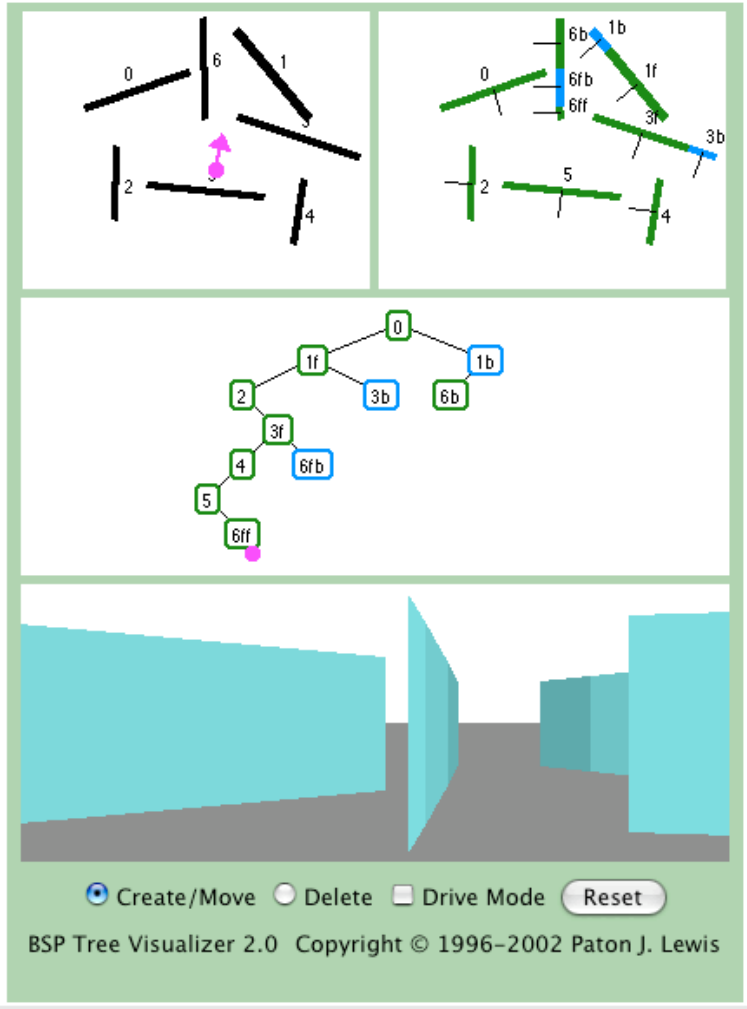
$$= n \cdot (a + t(c - a)) - d \stackrel{!}{=} 0$$

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

$$t = \frac{d - (n \cdot a)}{n \cdot (c - a)}$$

  - Wiederhole dies zur Berechnung von B

Demo



Quelle: Paton J. Lewis - http://symbolcraft.com/graphics/bsp/

Zusammenfassung

Vorteile

- Sehr effiziente Datenstruktur um Polygone bzgl. eines Punktes zu sortieren!
- Unabhängig vom Viewpoint
- Wird auch für andere Aufgaben benötigt

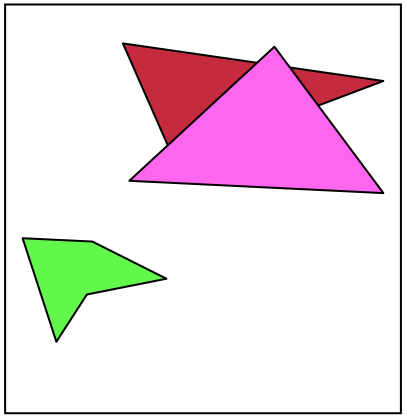
Nachteile

- Viele kleine Polygonteile (wg Splitting)
- Starkes Over-drawing
  - Viele Pixel werden „umsonst“ geschrieben (wg. Back-to-front–Sortierung)
- Schwierig, den Baum ausgeglichen zu halten

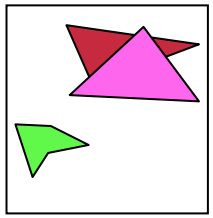
Warnock's Algorithmus [1996]

- Ein Image-Space-Verfahren, das auf einer rekursiven Unterteilung des Bildschirms 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



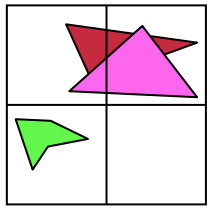
Idee

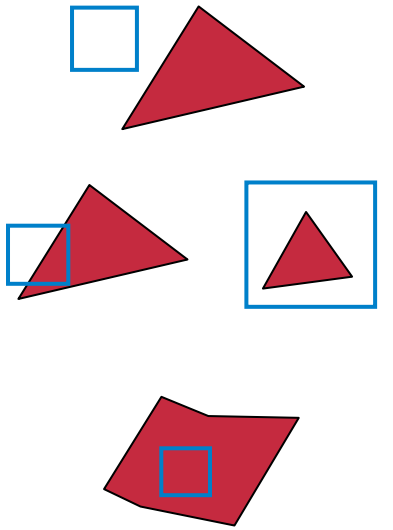


- Unterteile den Bereich in 4 gleiche Gebiete
- Treffe für jedes Teilgebiet die Entscheidung, welches Polygon (vorne) gezeichnet werden soll
- Zwischen einem Polygon und einem Gebiet gibt es folgende 4 Fälle:

- Beobachtungen:
  - Nicht vom Gebiet geschnittene Polygone beeinflussen das Gebiet nicht
  - Schneidet ein Polygon das Gebiet, so beeinflusst der außerhalb liegende Teil das Gebiet nicht
  - In jedem Schritt berechnen wir die Farbe des Gebietes; ist die Berechnung nicht eindeutig, dann unterteile es erneut

- Der Algo unterteilt nun 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

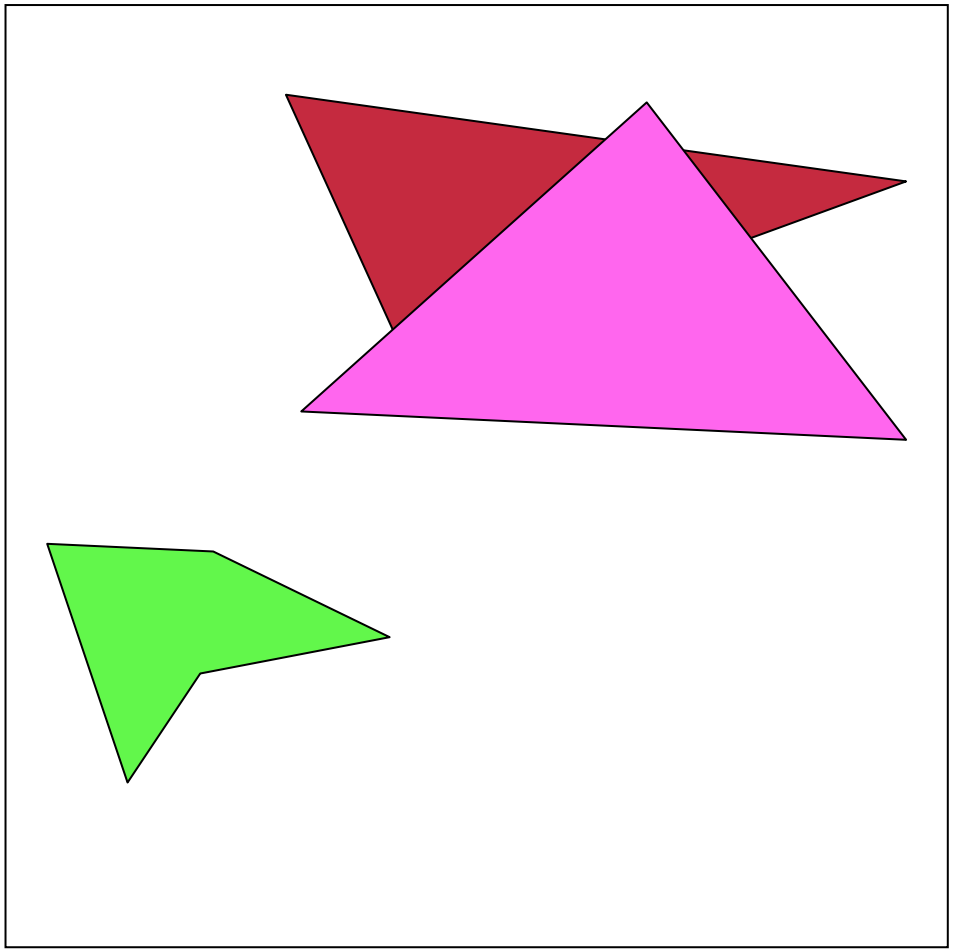




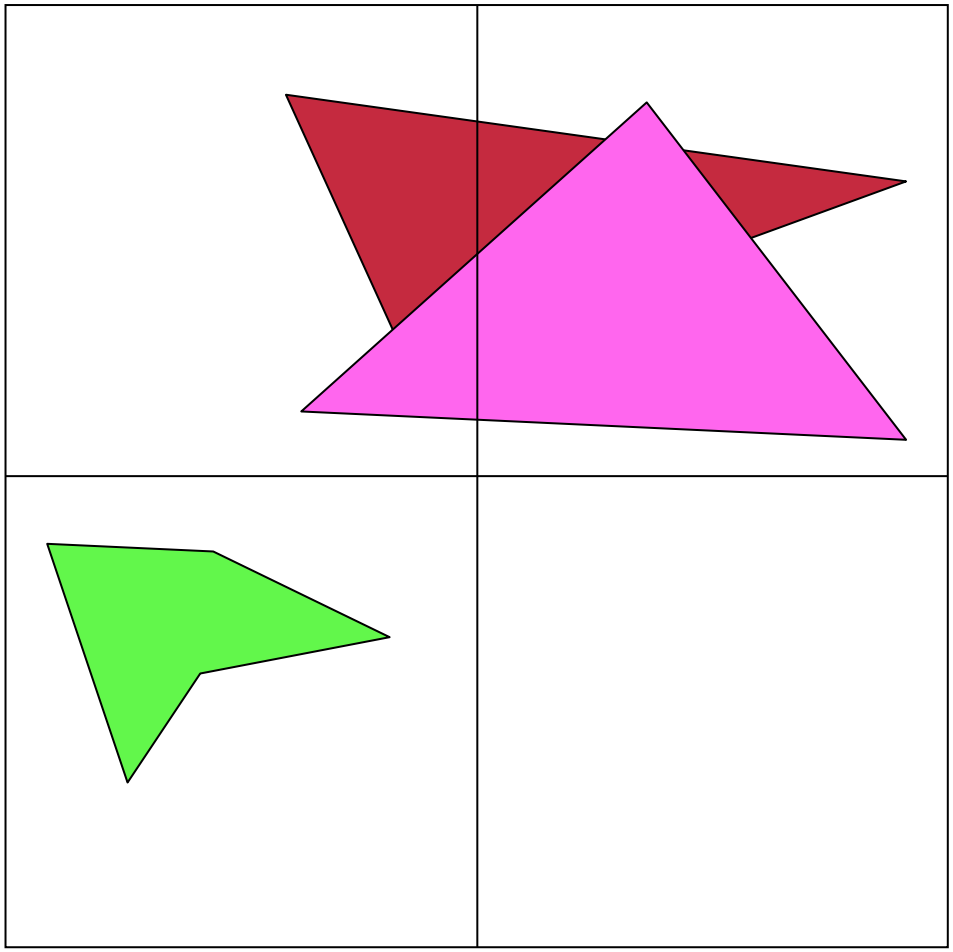
- Unterteilung wird fortgeführt bis:
  - Alle Gebiete entsprechen einer der vier Kriterien
  - Die Größe des Gebietes entspricht einem Pixel
    - In diesem Fall wird die Farbe irgendeines Polygons zum Füllen gewählt; oder ...
    - Man füllt mit dem Mittelwert aller Polygonfarben; oder ...
    - Man macht Anti-Aliasing zwischen den Polygonen

- Nebenbemerkung: die Datenstruktur heißt Quadtree
  - Wird aber im Warnock-Algorithmus **nicht explizit** erzeugt!

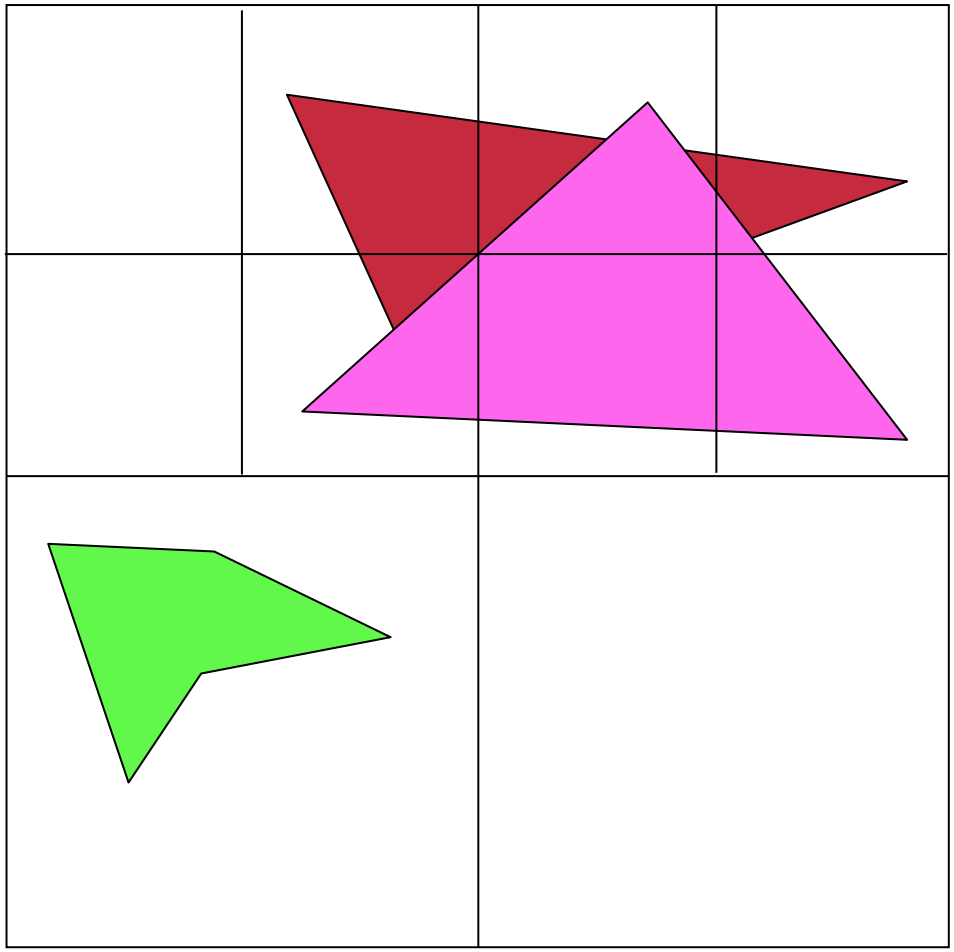
Beispiel



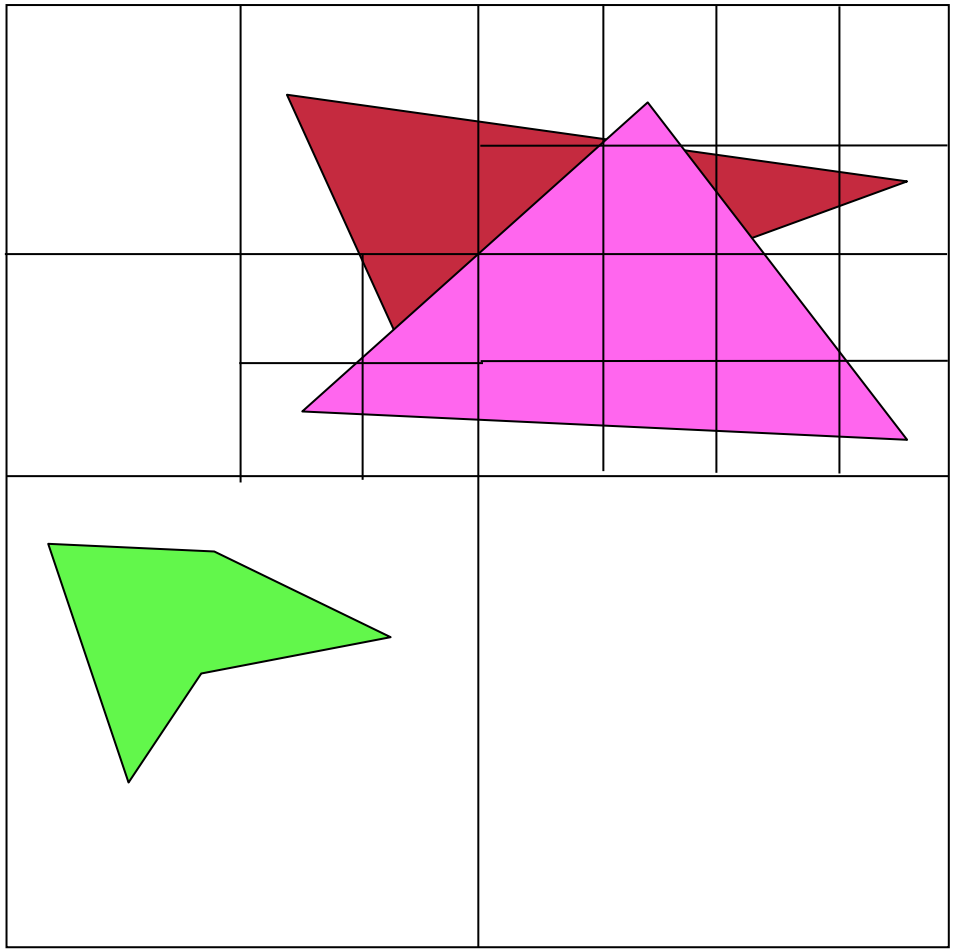
Ausgangsszene



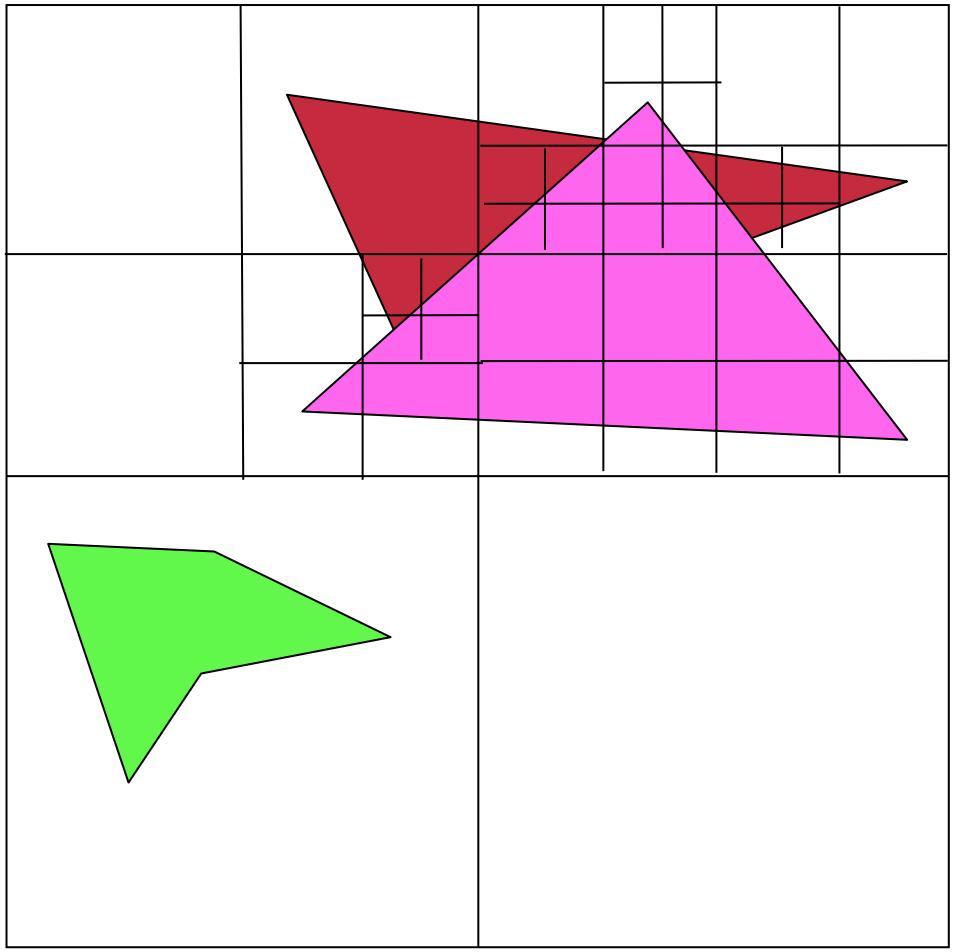
Erste Unterteilung



zweite Unterteilung



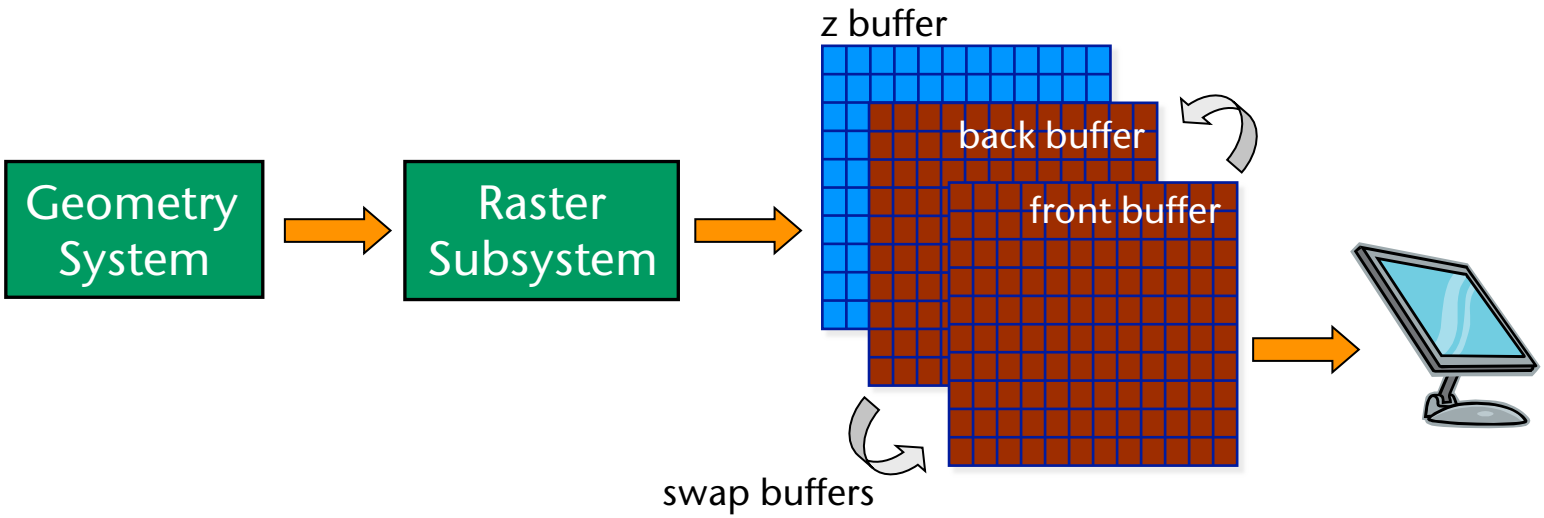
dritte Unterteilung



vierte Unterteilung

Speaking of "Buffers" ...

- Es gibt noch viele weitere Buffer in einem Framebuffer
- Der Double-Buffer:
  - Problem beim Single-Buffering: Flickering
  - Lösung: 2 Buffers
  - Front Buffer = Color-Buffer, der vom Display gerade angezeigt wird
  - Back Buffer = Color-Buffer, in den gerade gezeichnet wird



Ein Wort zu "swap buffers" und Synchronisation allgemein

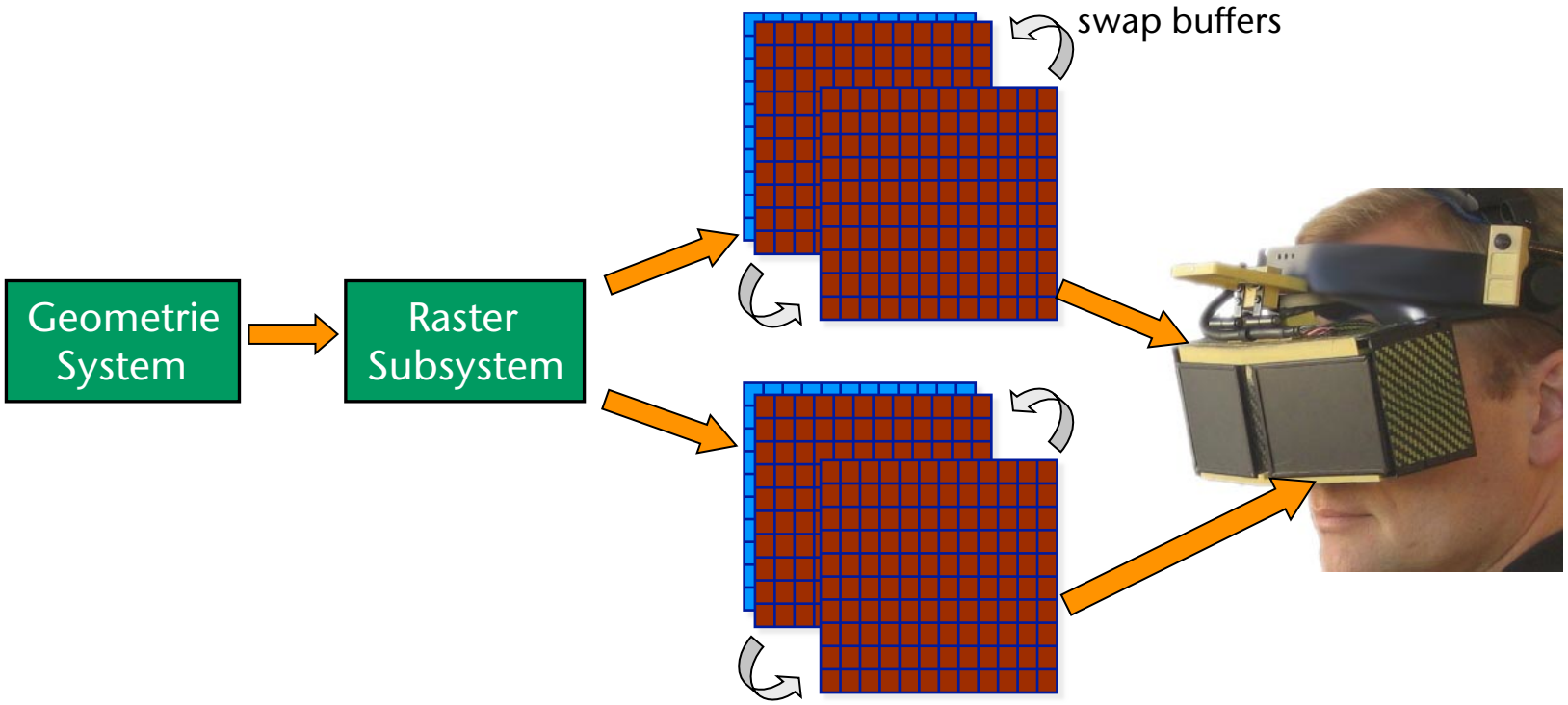
- Funktionsname in OpenGL: **`glSwapBuffers()`**
- Verwendet man praktisch nie "von Hand" bei Einsatz von high-level GUI-Libraries (Qt, GLUT, GLEW, etc.)
  - Dort liegt die *main loop* (und damit die Kontrolle) immer in der GUI-Library
- Der *Buffer-Swap* muß (normalerweise) mit dem *vertical retrace* des Monitors synchronisiert werden
- Konsequenz: es kann hohe Zeitverluste durch Synchronisation geben
  - Beispiel: main loop benötigt 1/45 Sekunde = 22 Millisek., Monitor läuft mit 60 Hz → main loop läuft nur mit 30 Hz → die main loop muß am Ende jedes "Applikations-Frames" 2·16 — 22 = 10 Millisek. warten!

Weitere Synchronisationen

- Bei Setups mit mehreren PCs für 1 Display (z.B. Powerwall) muß der Swap-Buffers aller Renderer auf allen PCs miteinander synchronisiert werden → *Swaplock*
  - Wird typischerweise durch einen *Barrier* implementiert
- Damit dies das gewünschte Resultat produziert, muß der Retrace aller Monitore (oder Projektoren) miteinander synchronisiert werden → *Genlock*
- Fazit: noch mehr Synchronisationsverluste

Quad Buffers

- Für Stereo- (3D-) Rendering muß man 2 unterschiedliche Bilder generieren: je eines für das linke bzw. rechte Auge
- Lösung: 2 Front buffers, 2 back buffers (und 2 Z-Buffer!)



Der Stencil Buffer

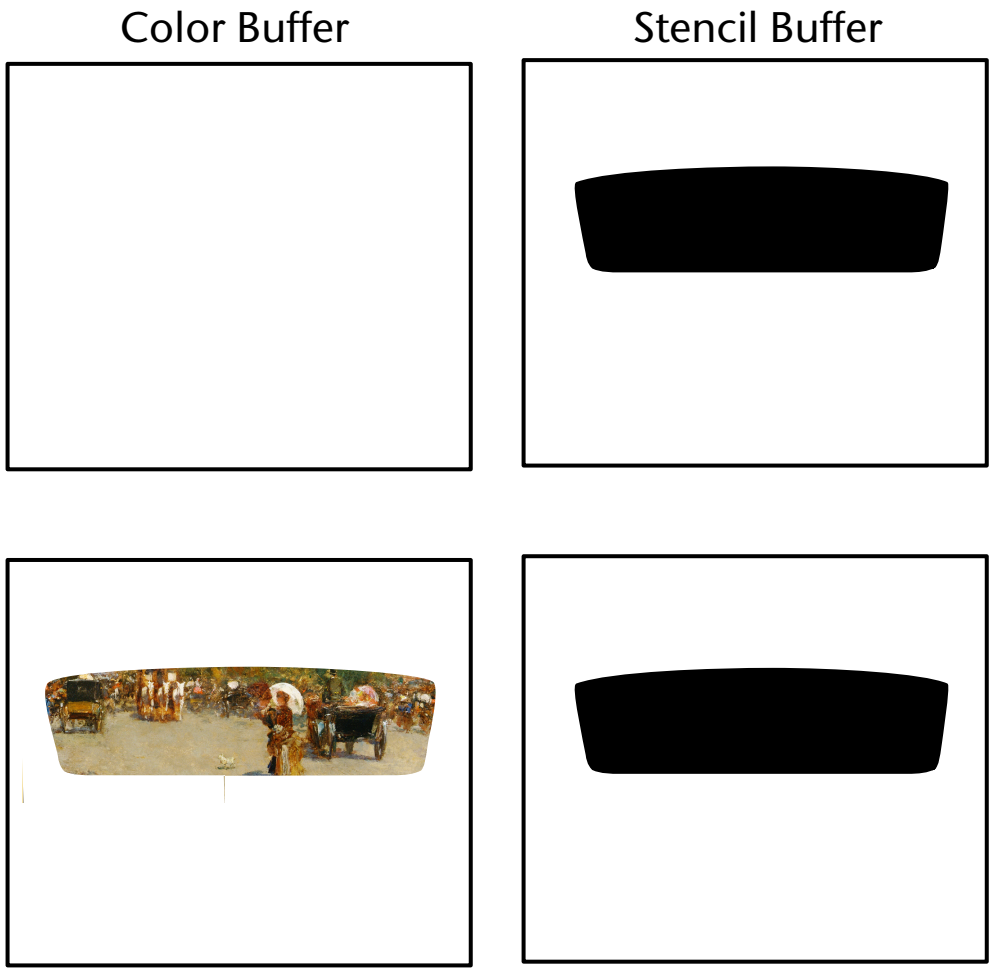
- Der Stencil-Buffer ist eine Art "Vergleichs-Buffer"
  - Ähnlich zu Z-Buffer (*test & pass/kill*) , aber mit anderen Features
- Die zwei Operationen bei eingeschaltetem Stencil-Buffer:

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

"Stencil" im echten Leben

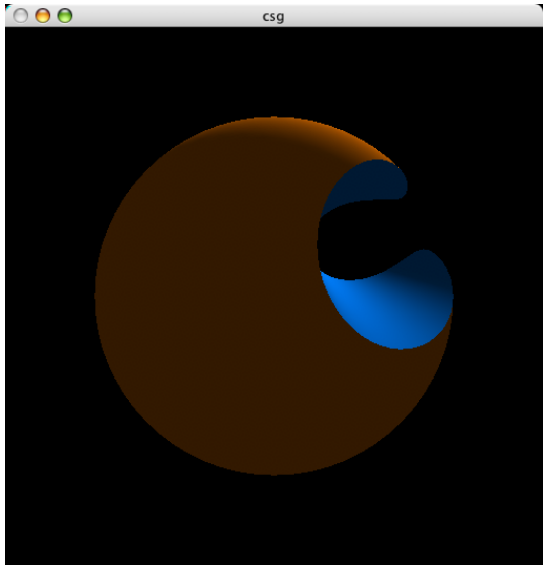
Typisches, einfaches Beispiel

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

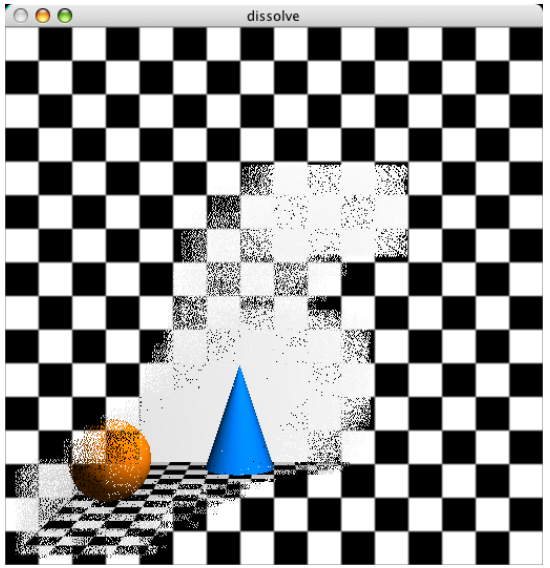


Beispiele für komplexere Operationen/Effekte

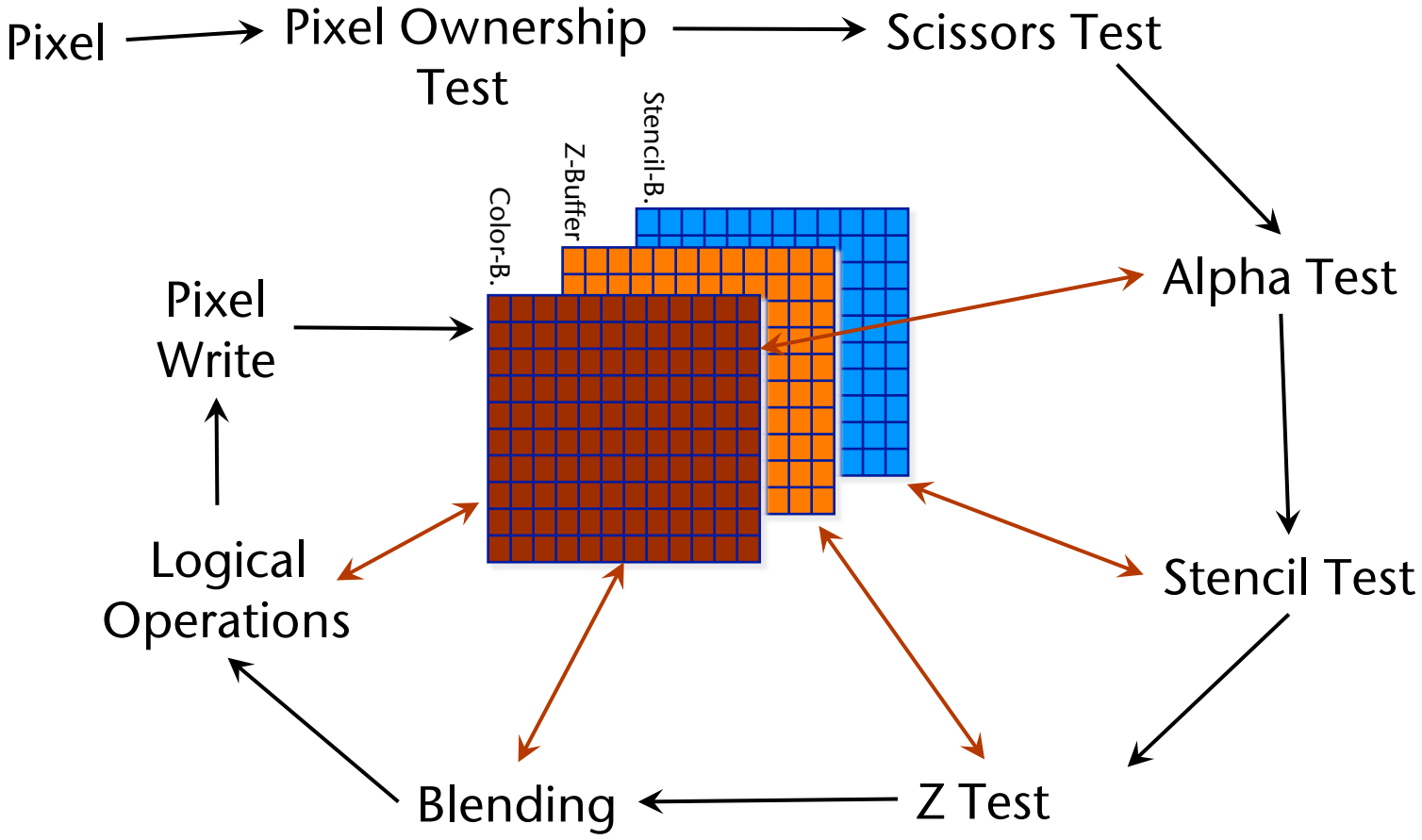
- Beispiel: CSG-Operationen (Schnitt, Differenz, ...)



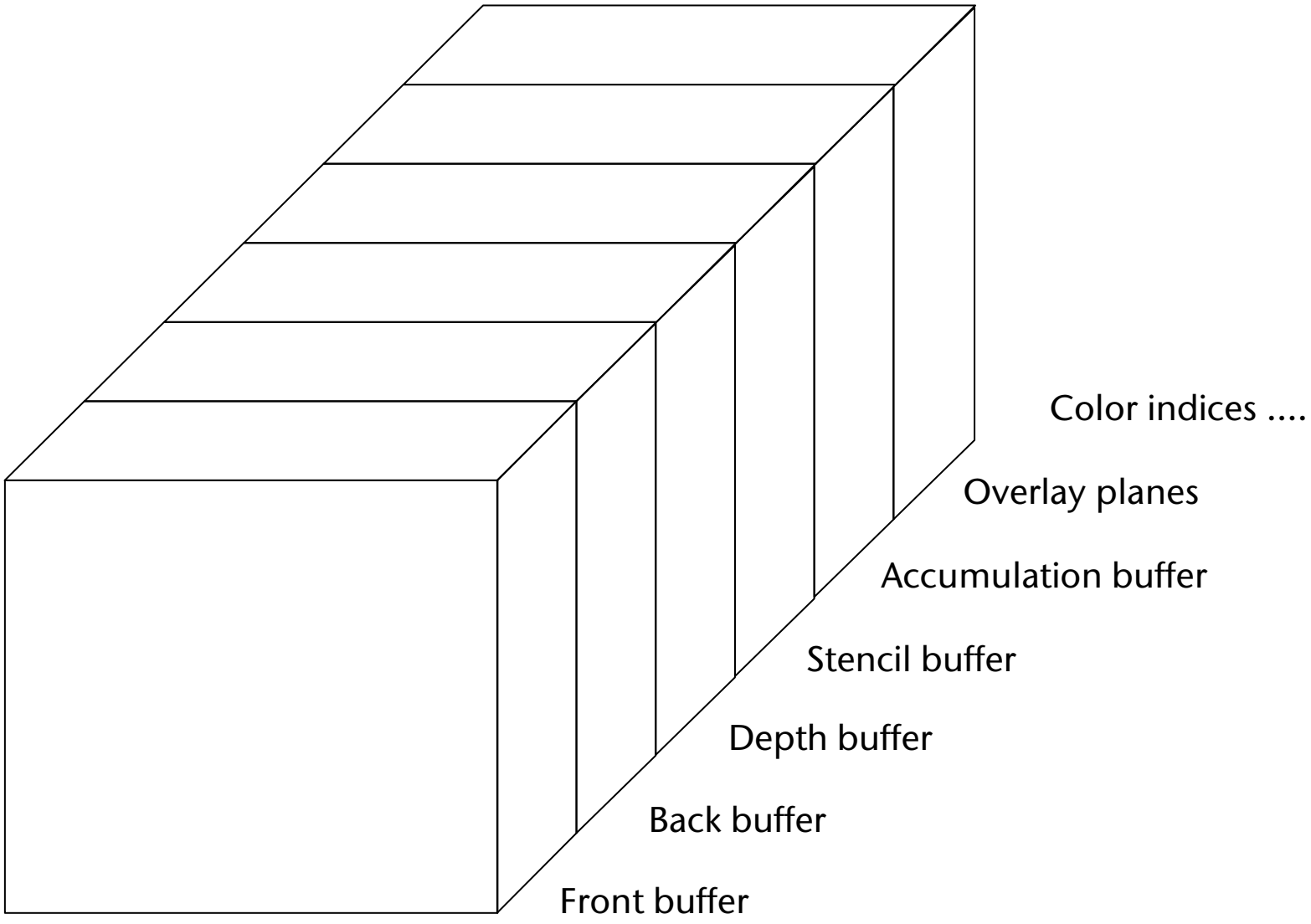
- Beispiel: "Dissolve"



Die Abfolge von Tests in der Graphik-Pipeline



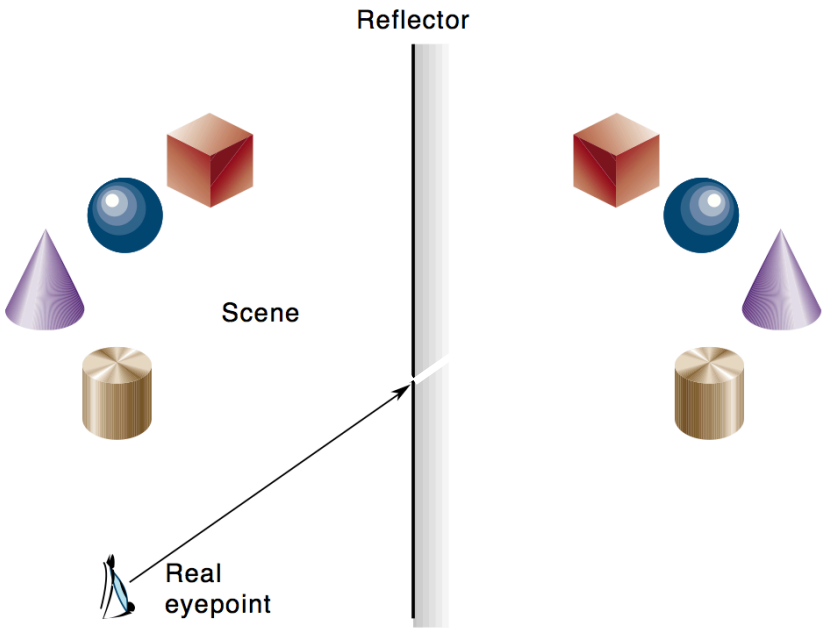
Bemerkung: es gibt viele weitere Buffer in OpenGL



Erste Anwendung: planare Spiegel

Rendering *planar reflections* using the Stencil Buffer

- Grundlegende Idee: erzeuge für jedes Objekt ein "virtuelles" gespiegeltes Objekt
- Der allg. Algorithmus:
  - Betrachte Spezialfall, daß Spiegelebene die Ebene z=0 ist
  1. Setze Viewpoint
  2. Rendere alle Polygone mit z' = -z
  3. Rendere die Szene normal

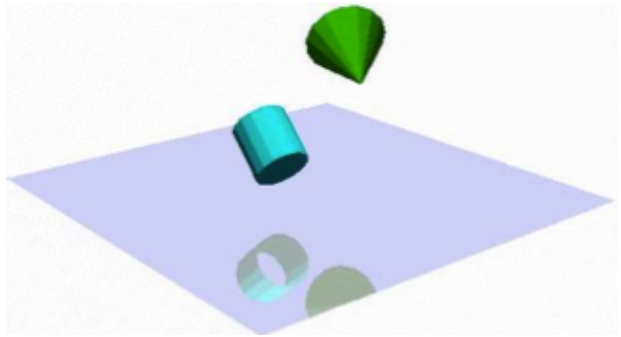


- Dies ist ein Beispiel für einen multi-pass Rendering-Algo
- Achtung: rendere in Schritt 2 nur Polygone *hinter* der Spiegelebene (mittels Clipping-Plane in Spiegelebene; → später)

- 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 clipping plane, so that objs *in front* of mirror are *not* rendered
  - Compute reflection transformation and apply to all polygons
  - Render scene (without geometry of mirror itself)
- 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 the stencil buffer such that 1 will be stored at each pixel touched by a polygon
    - ```
      glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
      glStencilFunc(GL_ALWAYS, 1, 1);
      glEnable(GL_STENCIL_TEST);
      ```
      ```

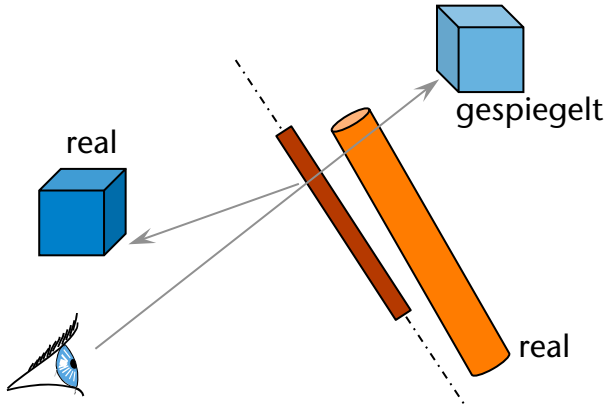


- Disable drawing into the color buffer
  - `glColorMask(0, 0, 0, 0)`
- Draw the geometry of the mirror, with blending if desired
  - This sets stencil bits & fills z buffer with depth value of mirror geometry
- Clear color buffer outside mirror geometry:
  - Configure the stencil test to pass outside the mirror polygon:

    ```
    glStencilFunc(GL_NOTEQUAL, 1, 1);
    glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
    ```

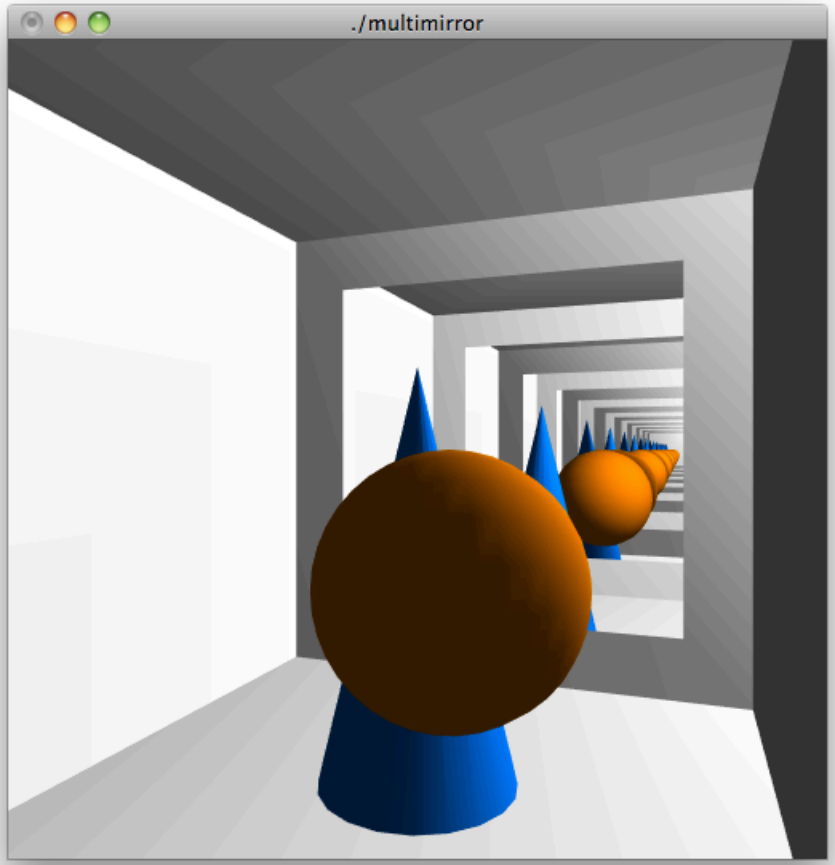
  - Clear color buffer, so that pixels outside the mirror return to the background color: `glClear(GL_COLOR_BUFFER_BIT)`

- Pass 2:
  - Disable stencil test
  - Disable clipping plane
  - Render scene as usual



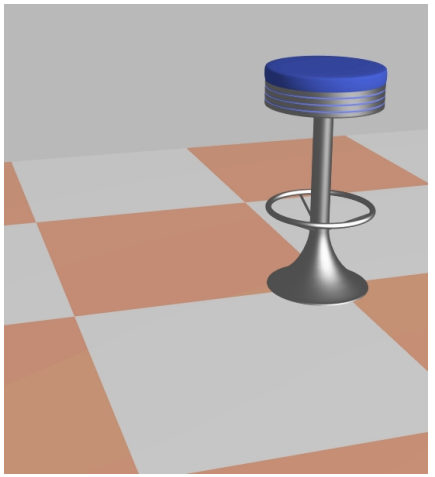
Demo

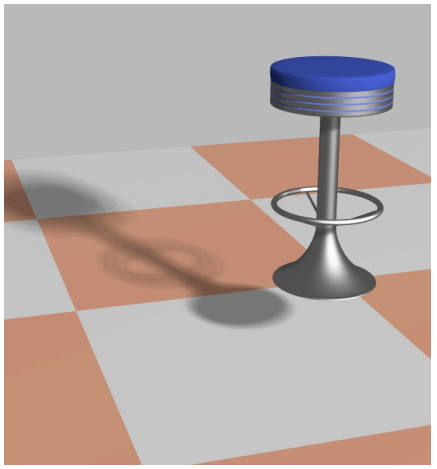
- Das Ganze kann man rekursiv machen:

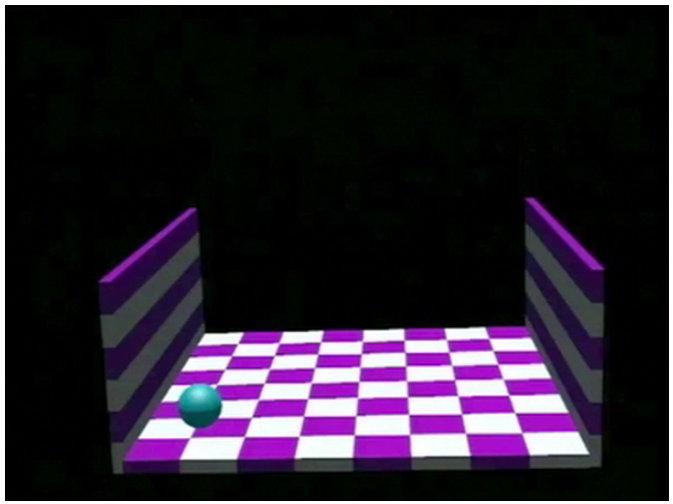


Schatten

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



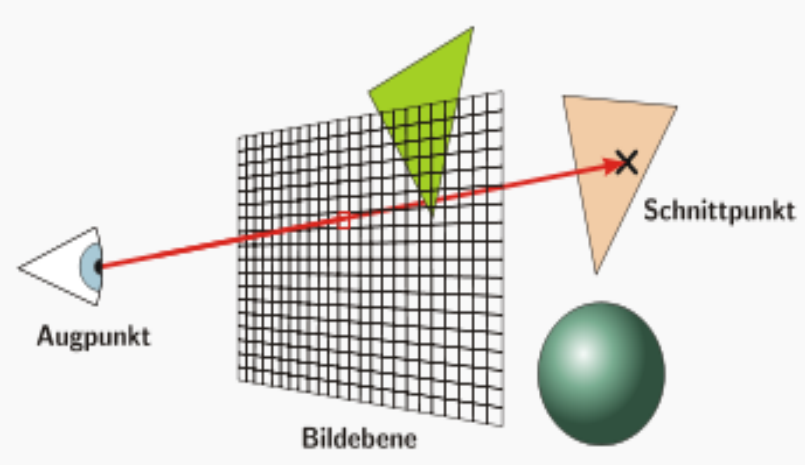


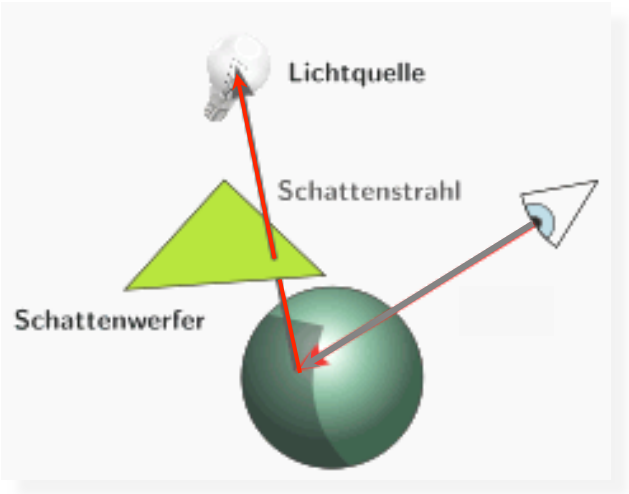


Die Trajektorie des Balls im Bild ist in beiden Fällen genau dieselbe!

Rendering Shadows using Shadow Volumes

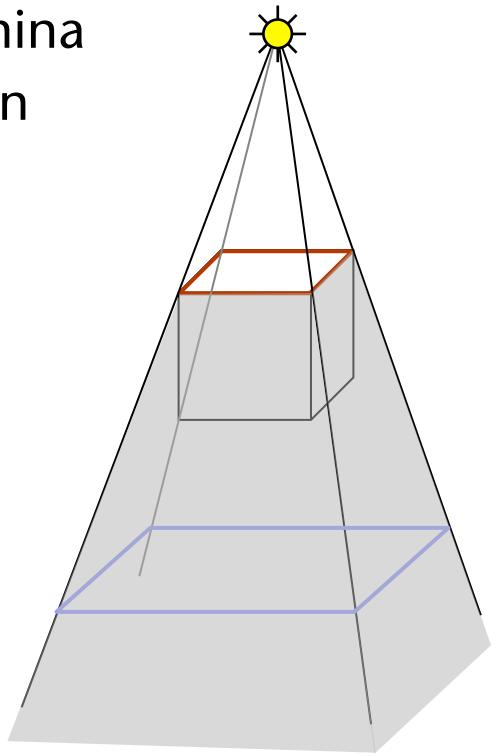
- Zusammenhang zwischen *Visibility* und *Shadows*:
  - Visibilitätsberechnung = welche Objekte sind vom Betrachter aus sichtbar
  - Schattenberechnung = welche Objekte sind von der Lichtquelle aus sichtbar

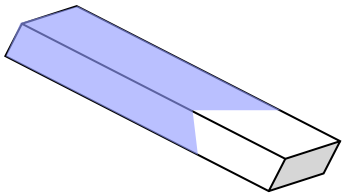




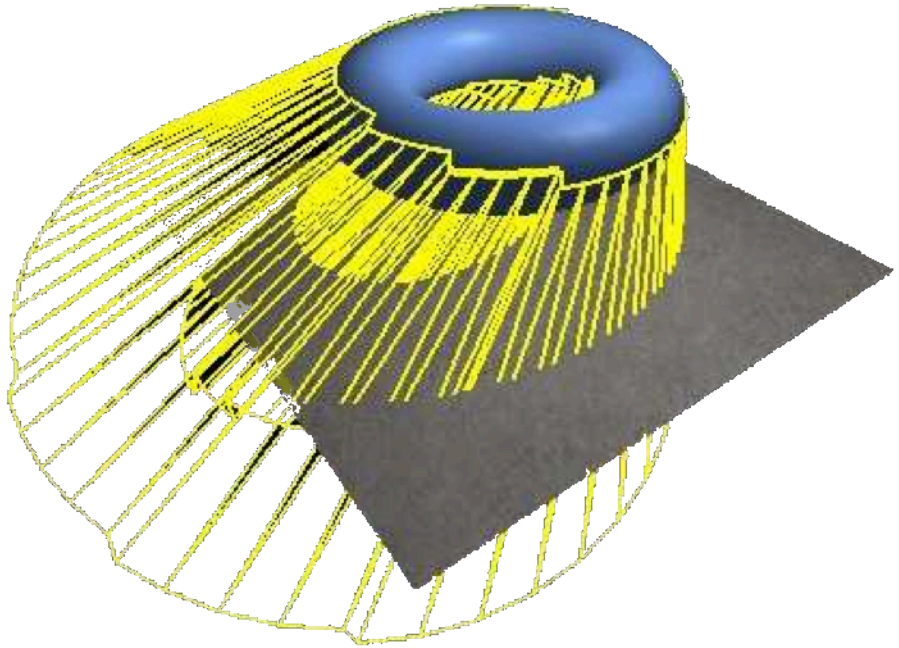
Das Schattenvolumen

- Ansatz im Folgenden: modelliere die (Teil-)Volumina des Universums, die kein Licht von der 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
- Bemerkung: die Silhouettenkanten liegen nicht notwendigerweise alle in einer Ebene!
- Liegt ein Objekt (teilweise) im Inneren des Schattenvolumens, so heißt dieses "*shadow receiver*"



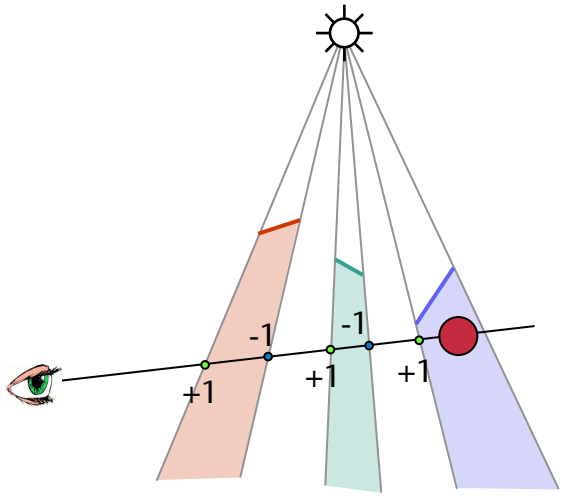


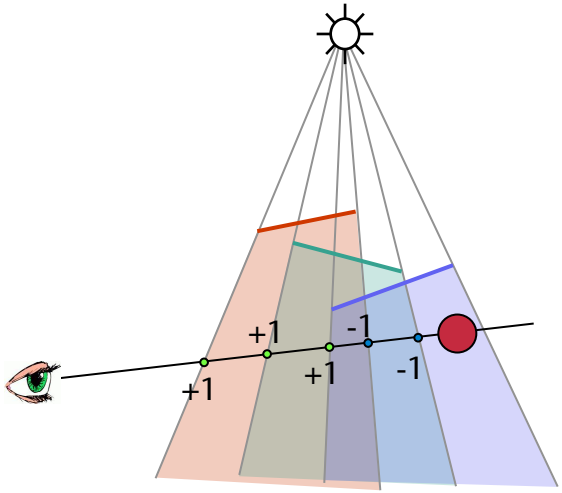
- Beispiel für ein komplexeres Shadow Volume:



Ein Kriterium für "Im Schatten"

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





Der Algorithmus im Detail (der "Z-Pass-Algo")

- Pre-processing: berechne alle Shadow Volumes

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();
```

2. Pass: rendere 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();              // .. increase stencil value
glCullFace(GL_FRONT);                    // for all back-facing pgons ..
glStencilOp(GL_KEEP, GL_KEEP, GL_DECR);   // .. passing the depth test
renderShadowVolumePolygons();              // .. decrease stencil value
```

3. Pass: rendere die Szene ohne Lichtquelle (= Schatten); schreibe Pixel nur dann in den Color-Buffer, wenn sie im Schatten von Lichtquelle 0 sind

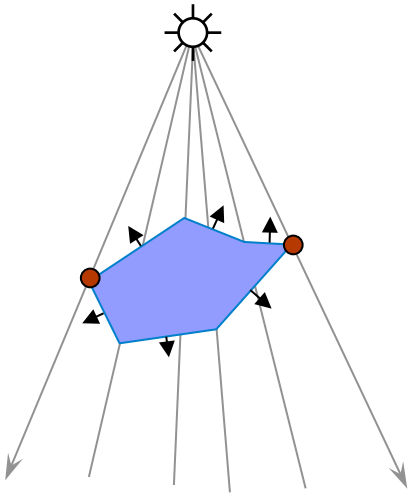
```
glEnable(GL_LIGHTING);                      // switch off light source 0
glDisable(GL_LIGHT0);                       //    but keep all others
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_EQUAL);                      // must match from 1st step
glDepthMask(0);                             // no need to 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();
```

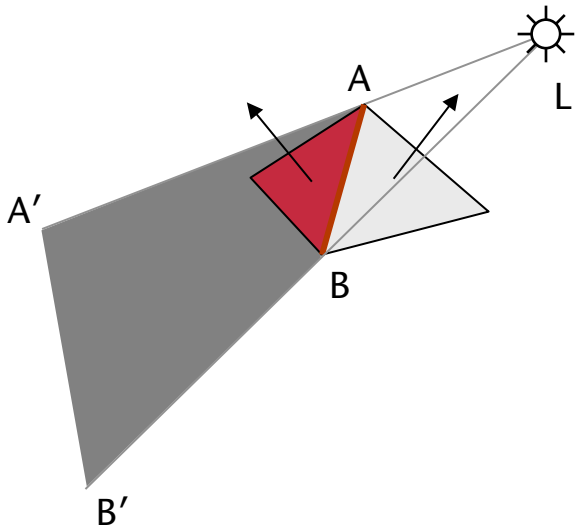
- Dieser Algorithmus heißt "z-pass algorithm", weil in Pass 2 nur Shadow-Volume-Pixel den Stencil-Wert verändern, die den Z-Test passieren

Varianten des Algorithmus'

- Es gibt eine GL-Extension, so daß man eine Stencil-Operation für front-facing, und 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 (kommt später) 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)

- Berechnung der Silhouettenkanten:
  - Kante (mit genau 2 adjazenten Polygonen) 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



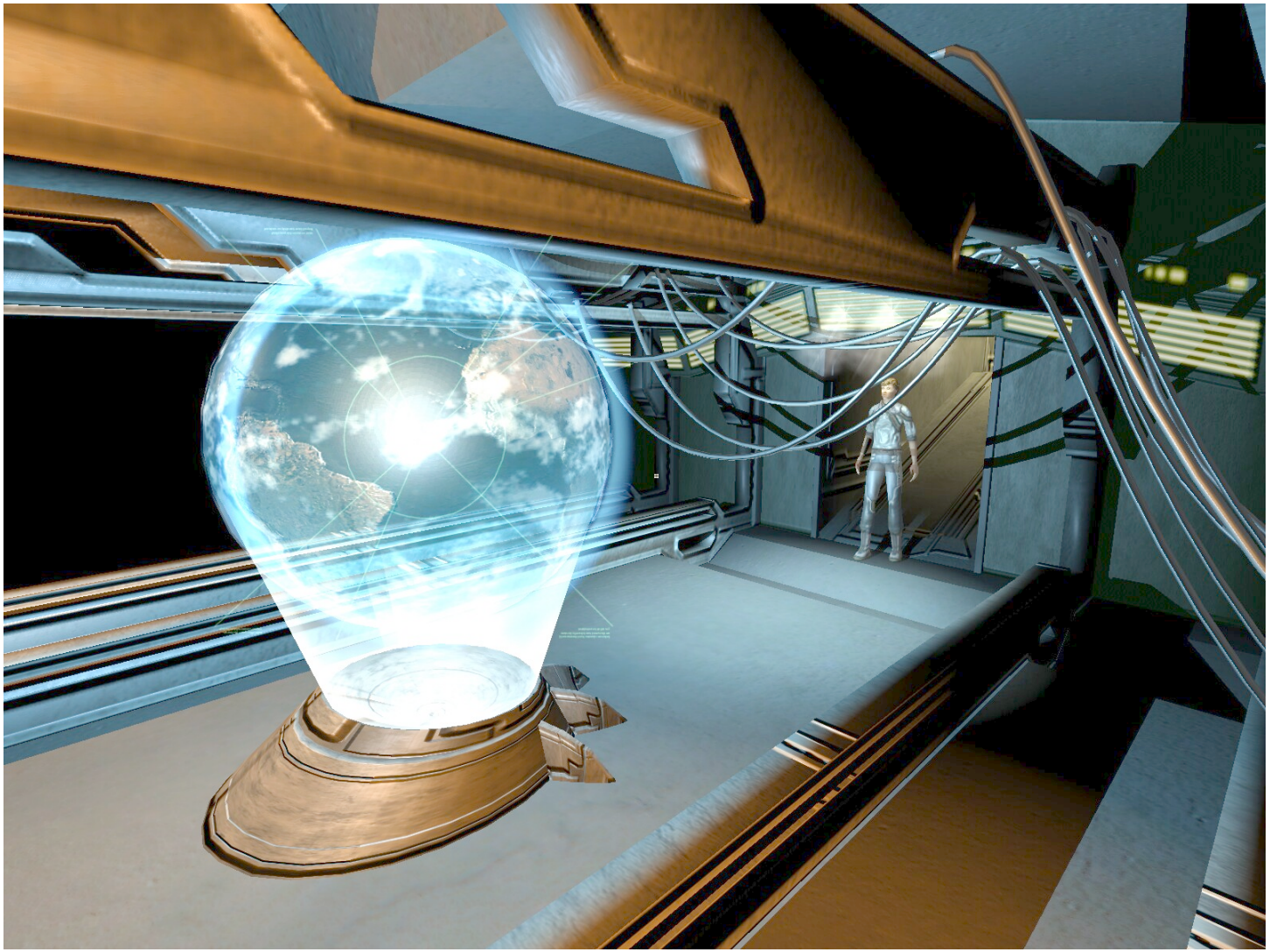


Beispiele

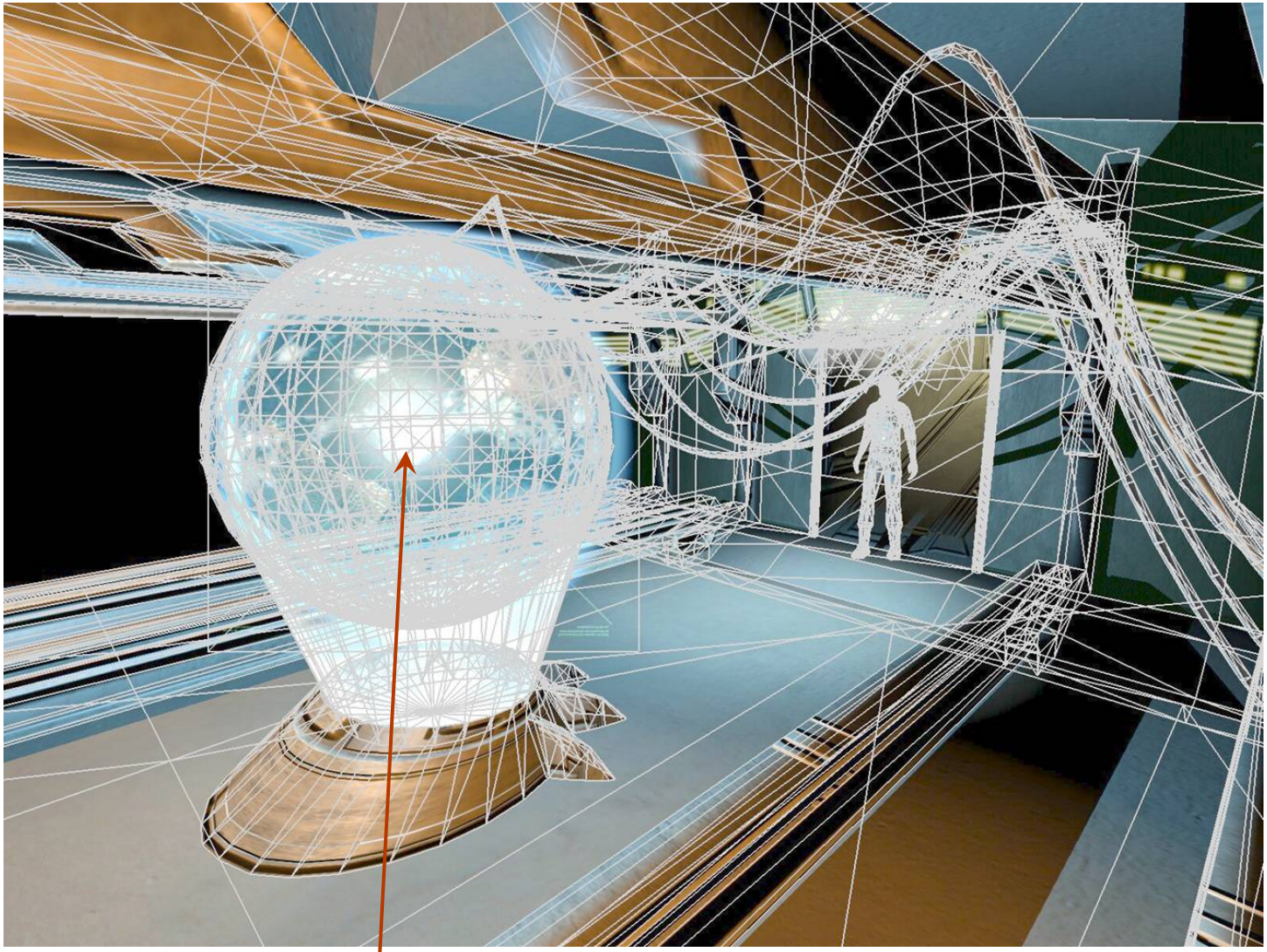
Shadowed scene

Stencil buffer contents

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

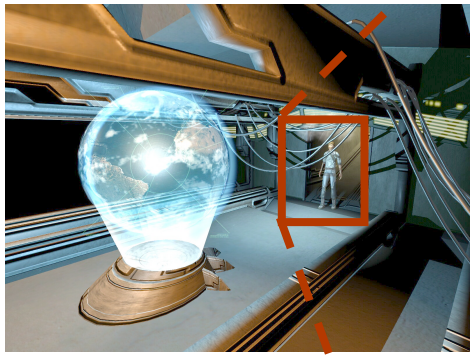


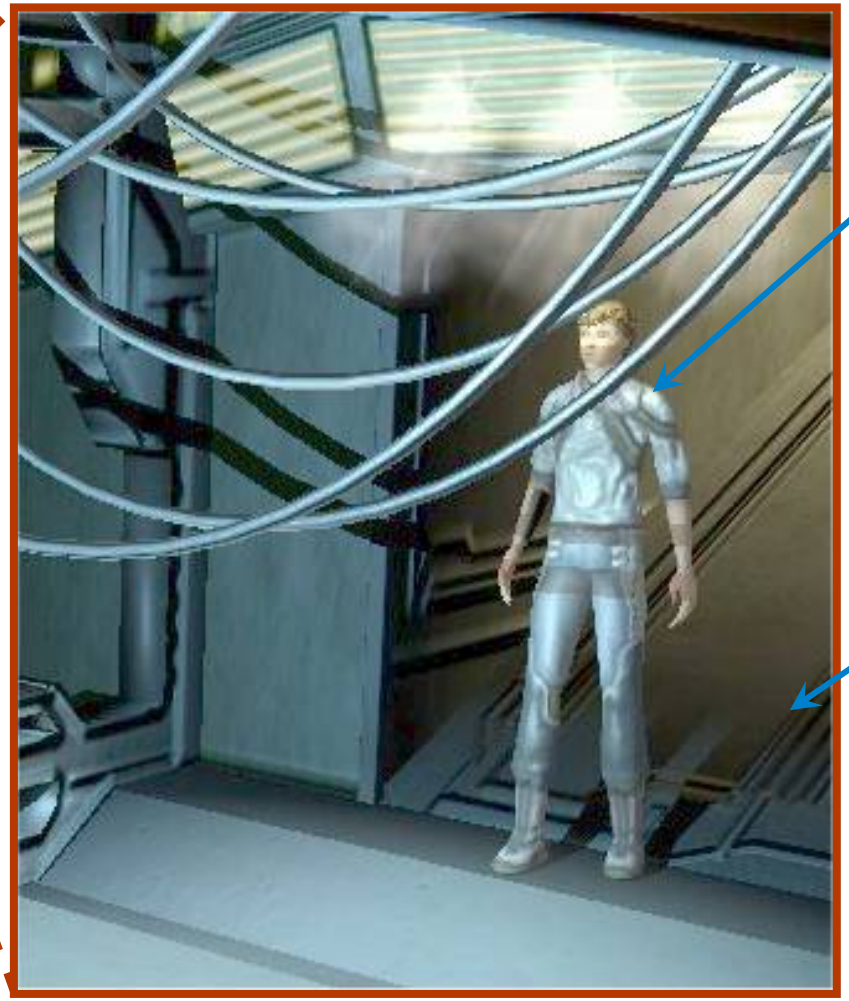
Abducted game images courtesy Joe Riedel at Contraband Entertainment



Primary light source location

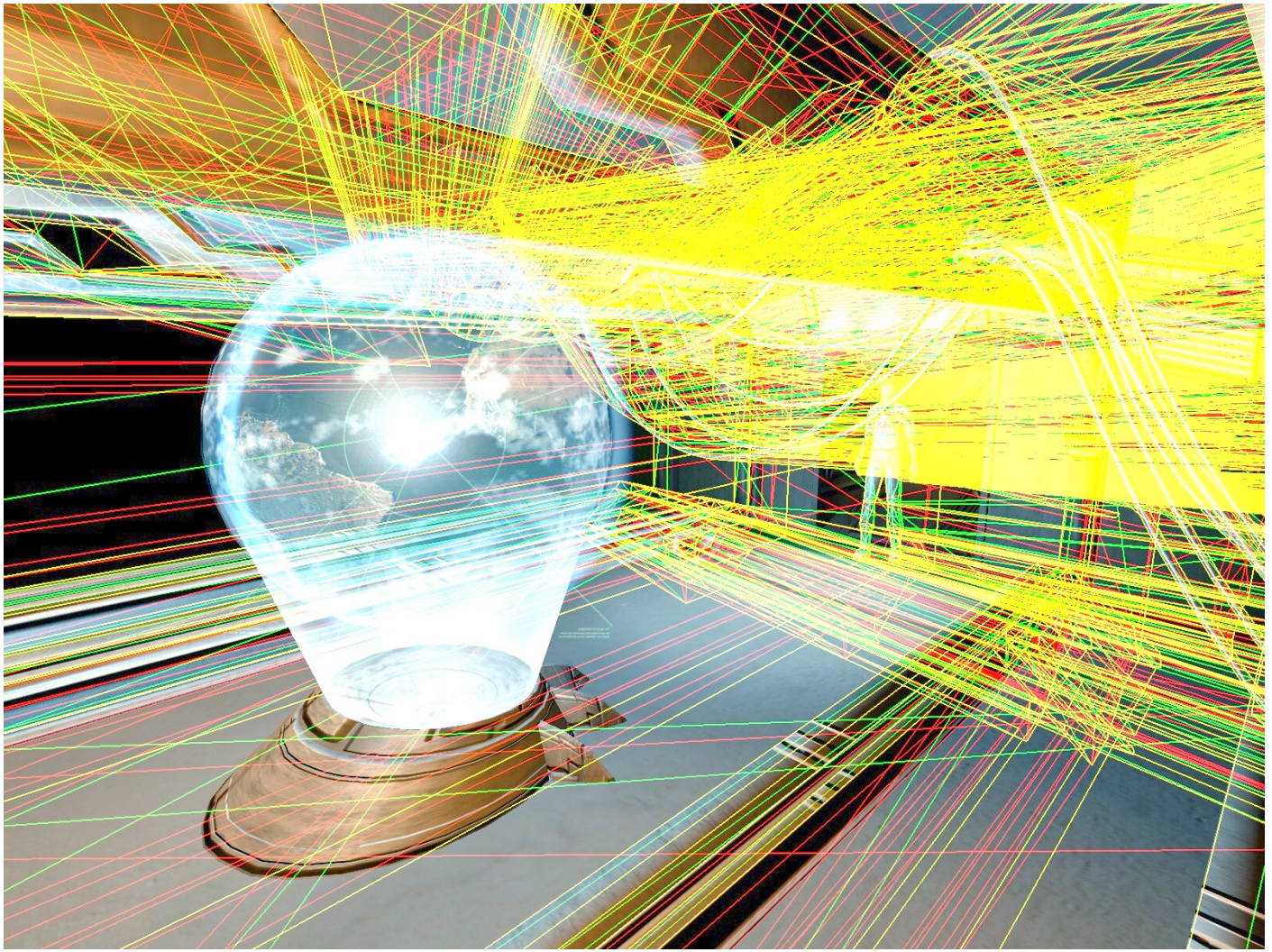
Wireframe shows geometric complexity of visible geometry



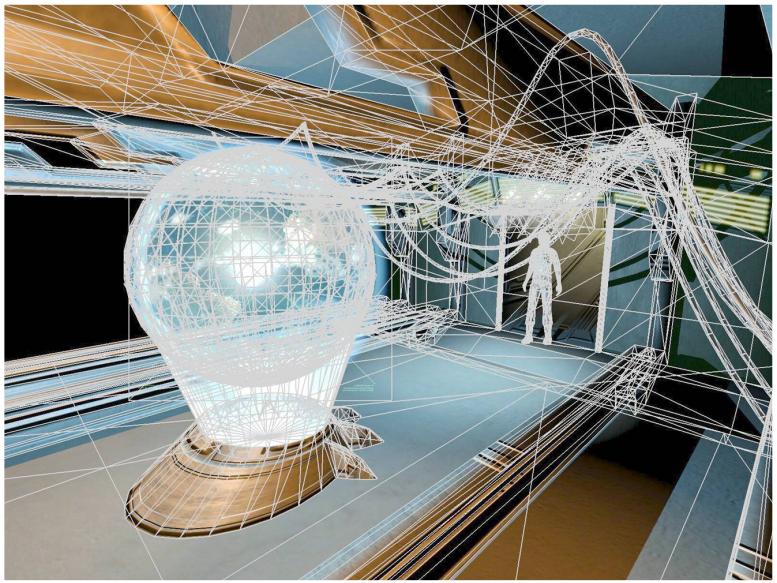


Notice cable shadows on player model

Notice player's own shadow on floor

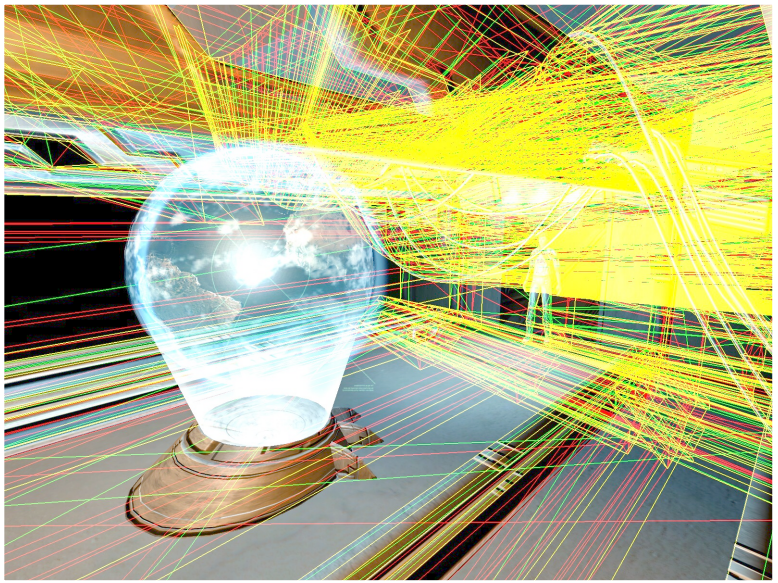


Wireframe shows geometric *complexity of shadow volume* geometry



Visible geometry

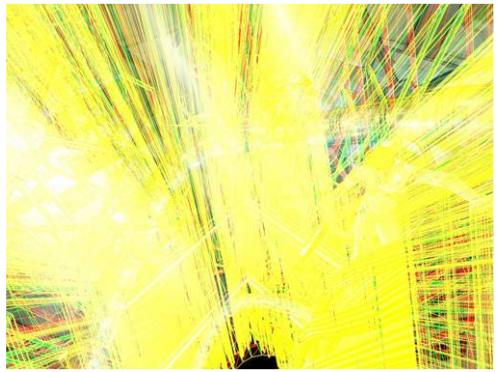
<



Shadow volume geometry

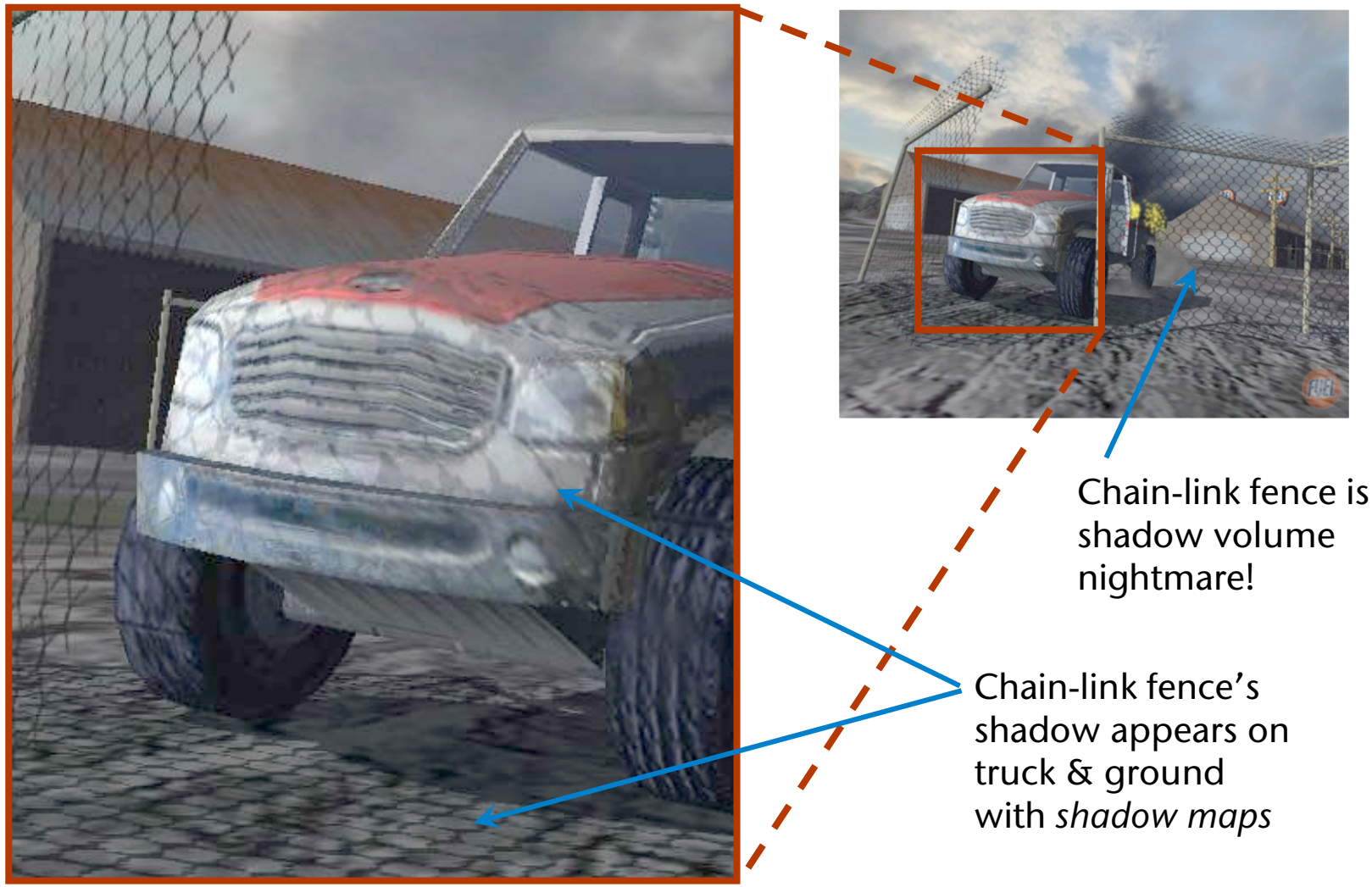
Typically, shadow volumes generate considerably more pixel updates than visible geometry

Visible geometry



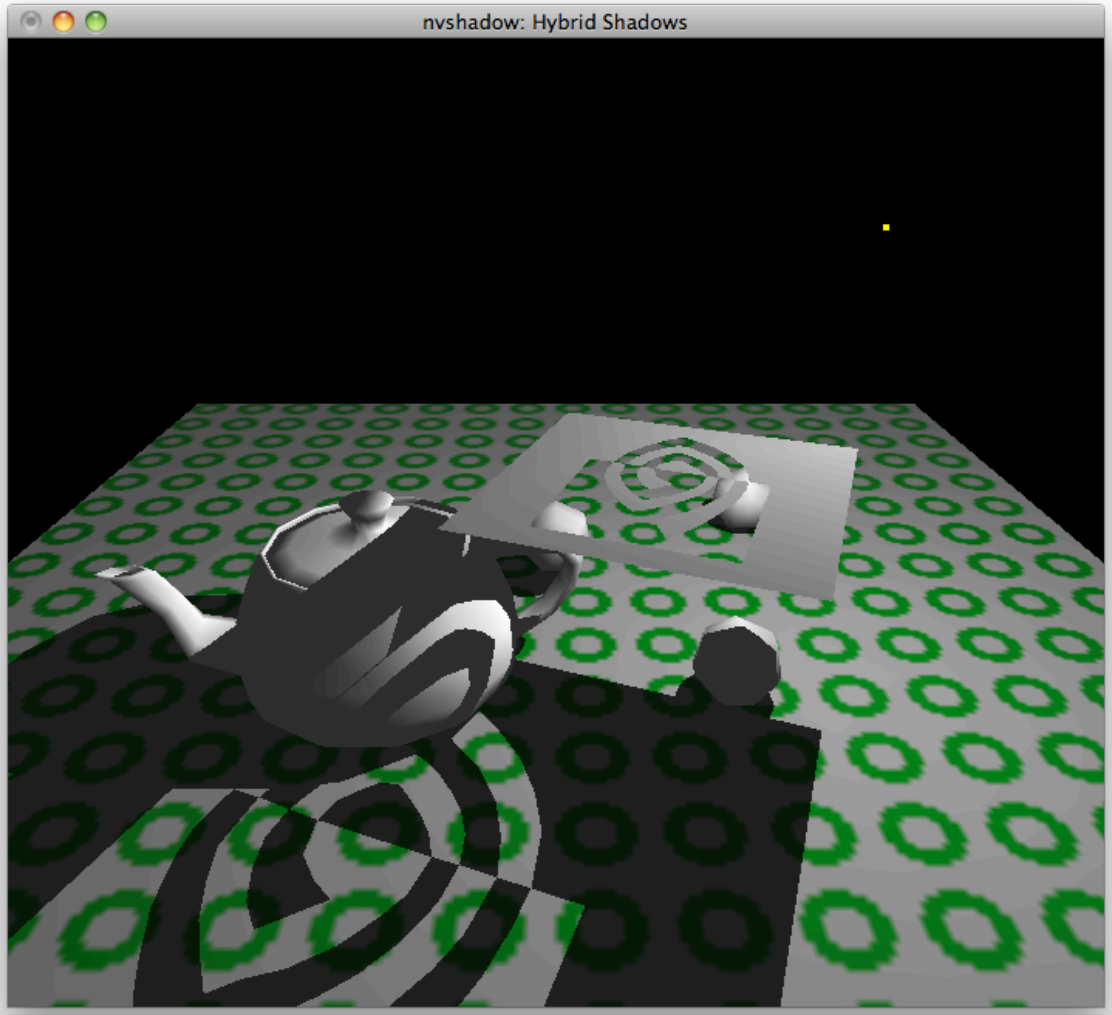
Shadow volume geometry

Situations When Shadow Volumes Are Too Expensive



Fuel game image courtesy Nathan d'Obrenan at Firetoad Software

Demos

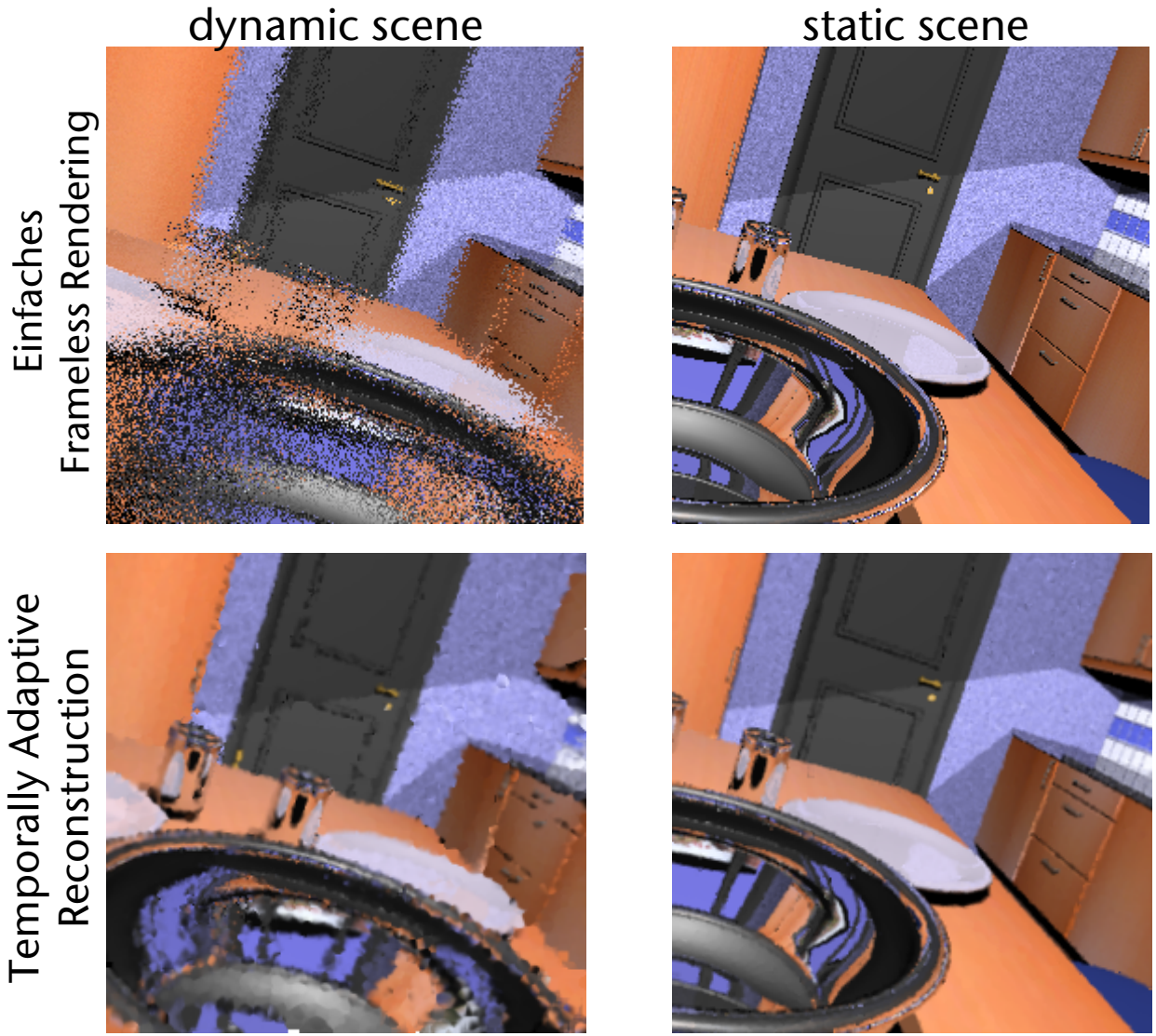


http://www.opengl.org/resources/features/StencilTalk/

Frameless Rendering [1994, 2005]

- Annahme: die Anzahl der Pixel im Frame ist der bestimmende Faktor für die Rendering-Zeit (→ *"fill limited"*)
  - Z.B. der Fall bei wenigen Polygonen und großem Display; oder bei Ray-Tracing (später)
- Idee: verwende das alte Frame wieder, und erneuere nur einige, zufällig ausgewählte Pixel
  - Konsequenz: es gibt keinen Double-Buffer mehr
  - Wenn die Szene dann statisch wird, werden sukzessive alle Pixel erneuert, und das Bild konvergiert zum "klassisch" gerenderten Bild
- Vorteil: wesentlich geringere Latenz zwischen Kamera-Bewegung und Erscheinen eines neuen Frames auf dem Display

Beispiele


dynamic scene
static scene
Einfaches Frameless Rendering
Temporally Adaptive Reconstruction