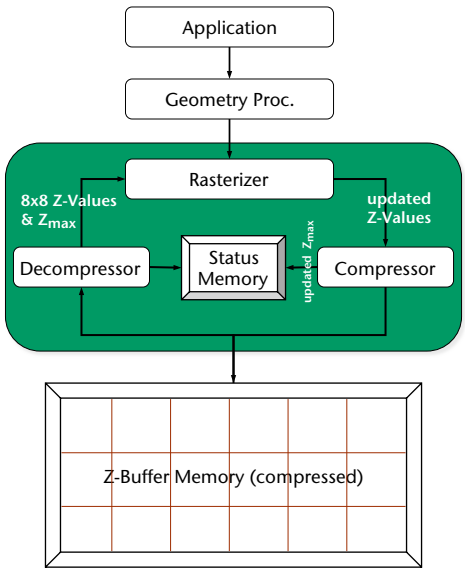
# Implementierung in aktueller Graphik-Hardware

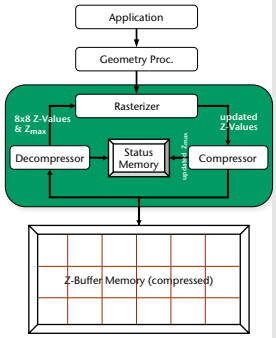
- Problem: Bandbreite zwischen Rasterizer und Speicher beträgt ca. 18 Gbyte/sec!
  - Annahmen: Auflösung 1280x1024, 4x depth complexity pro Pixel, pro Fragment: 1x Z-Buffer-Read + 1x Z-Buffer-Write + 1x Color-Buffer-Write + 2x Texture-Read, pro Read/Write 32 Bit
  - Aktueller Speicher erlaubt ca. 10 Gbyte/sec [2002]
- Wie implementiert man schnell `glClear(DEPTH_BUFFER_BIT)`?
- Wie implementiert man den HZB?
- Lösung: Z-Buffer in Kacheln aufteilen und komprimieren



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

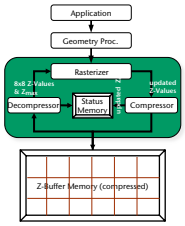


- Z-Buffer löschen:
  - Bei `glClear()` wird der Status jeder Kachel auf "cleared" gesetzt
  - Beim Lesen einer Kachel: Decompressor checkt Status, sieht "cleared", schickt $Z_{far}$ an Rasterizer
  - Kein Datenfluß auf dem Bus
- Z-Buffer schreiben:
  - Compressor berechnet neues $Z_{max}$ der Kachel und schreibt 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



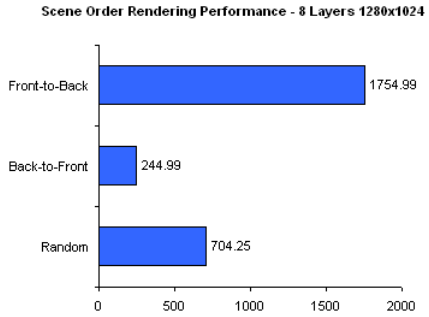



- Z-Buffer lesen:
  - Decompressor liest zuerst $Z_{max}$ aus dem Status Memory
  - Verschiedene Tests möglich:
    - Teste die Z-Werte der 4 Ecken der Kachel gegen $Z_{max}$
      - Bemerkung: diese 4 Z-Werte kann man in den Nachbarkacheln wiederverwenden
    - Teste die Z-Werte der 3 Ecken des Dreiecks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



## Performance-Gewinn in einer Graphikkarte

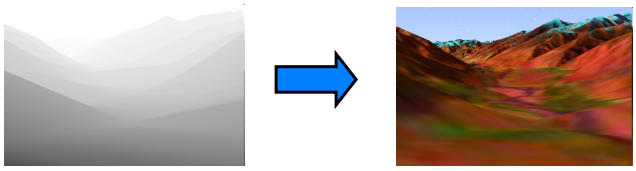
- Beispiel: ATI RADEON 9700 PRO [2003]
  - 3 Levels: 1. 8x8 Z-Block , 2. 4x4-Block, 3. "Early Z"-Test
- Performance-Gewinn:






## 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)
- 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 & kein Pixel im Color-Buffer wird überschrieben



## Object-Space vs. Image-Space

- Image Space Algorithmus: 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 Algorithmus: 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 den Aufbau komplexer Hilfsdatenstrukturen
  - Funktioniert besser bei statischen Szenen



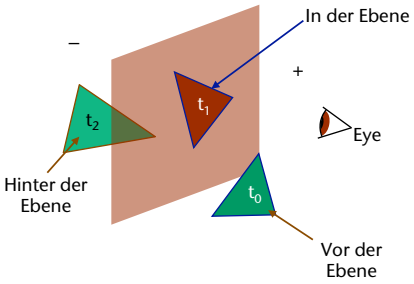
## Binary Space Partition (BSP) Tree [ca. 1982]

- Ein Object-Space-Algorithmus
- Rekursive Unterteilung des Raumes zur Tiefensortierung
- 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! (http://en.wikipedia.org/wiki/Quake_engine)

## 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 Ft1(eye) > 0 :
   draw t2
   draw t1
   draw t0
else:
   draw t0
   draw t1
   draw t2
```

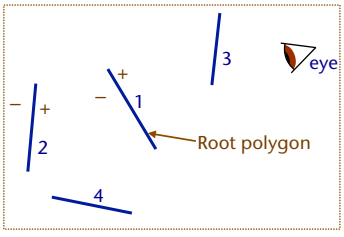



- Funktioniert für beliebigen Viewpoint!

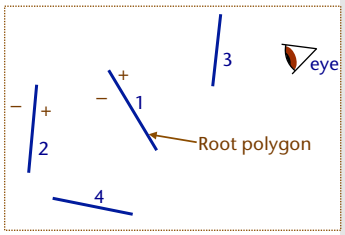


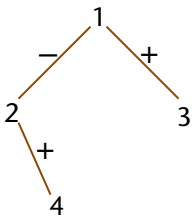
## Rendering-Algorithmus mit BSPs

- BSP Tree: unterteilt Raum rekursiv in positive und negative Teile
  - Knoten = Zeiger auf Polygon(e) und 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
  - Zeichne Polygone rekursiv auf der Gegenseite vom Viewpoint
  - Zeichne Polygon(e) im Knoten
  - Zeichne rekursiv Polygone auf der selben Seite wie Viewpoint



```
draw(node, eye):
  if node.empty():
    return;
  if node.plane(eye) < 0 :
    draw( node.plus, eye )
    rasterize( node.triangle )
    draw( node.minus, eye )
  else
    draw( node.minus, eye )
    rasterize( node.triangle )
    draw( node.plus, eye )
```




Reihenfolge beim Zeichnen: 2, 4, 1, 3



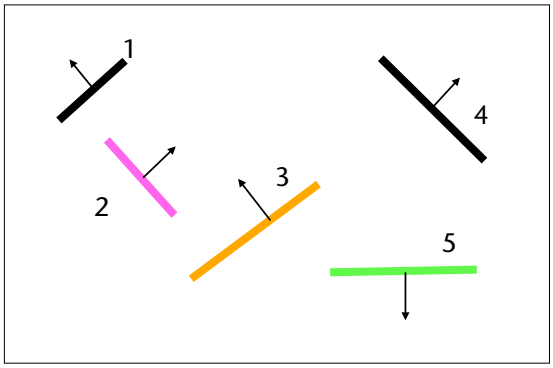
## Aufbau eines BSP Tree

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
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

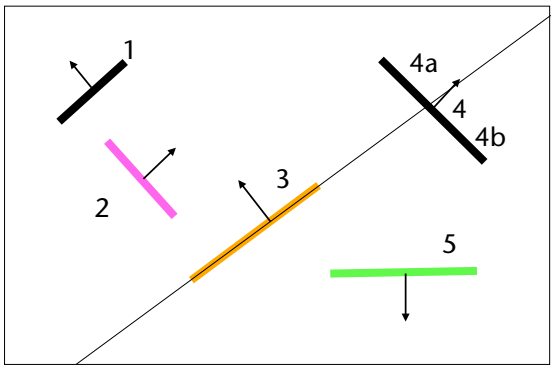
## Beispiel

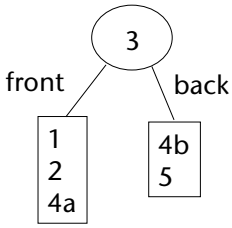
- Ausgangsszene:



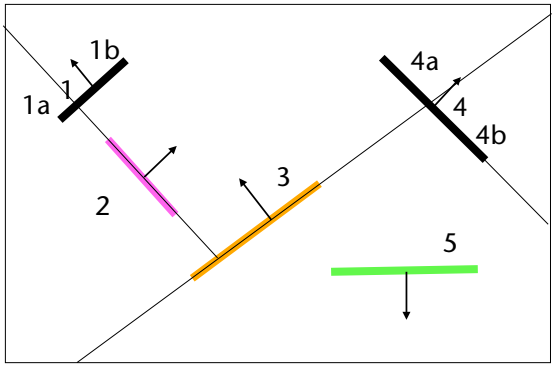


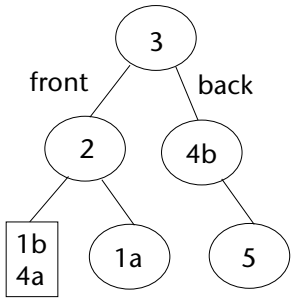

- Wähle z.B. Polygon 3 als Wurzelelement



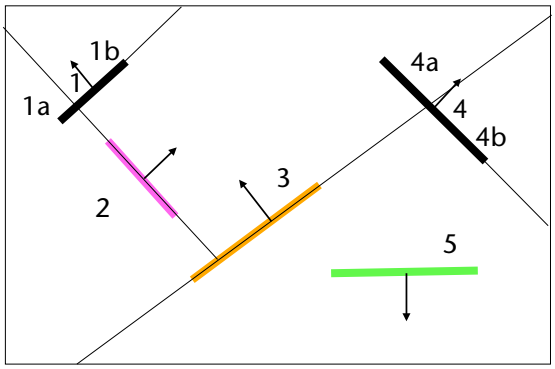
- Wähle Polygon 2 und 4b als nächste Knoten

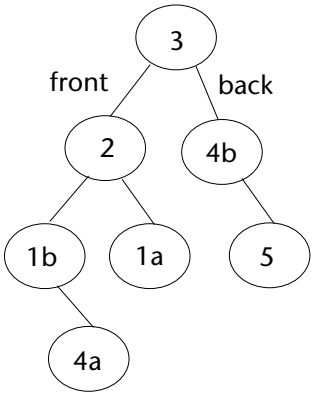




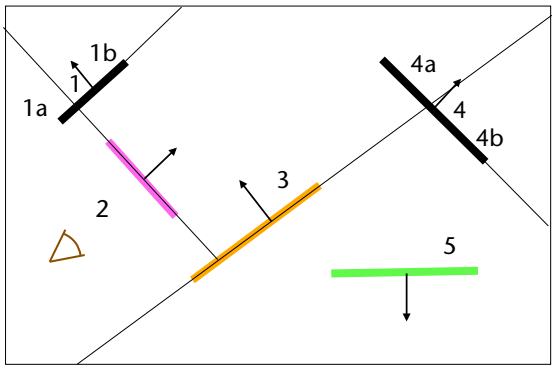

- Nun wähle Polygon 1b als Knoten

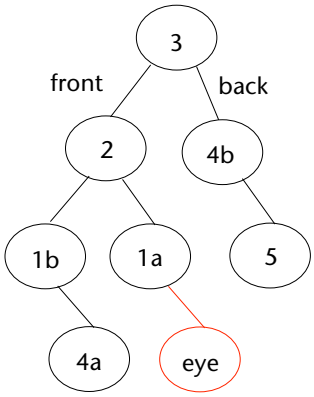


## Beispiel fürs Rendering mittels BSP

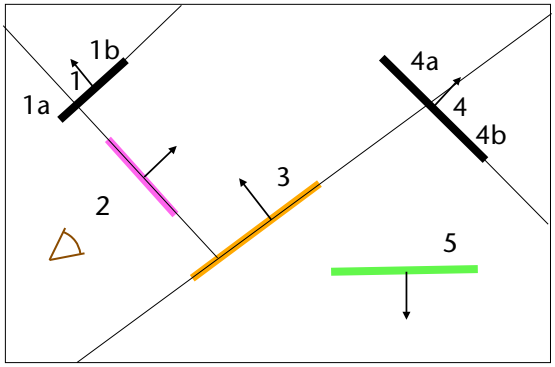
- Angenommen, der Viewpoint befindet sich wie hier dargestellt

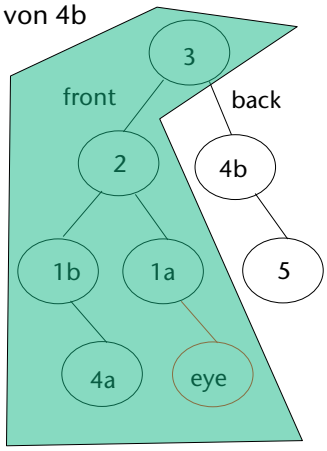




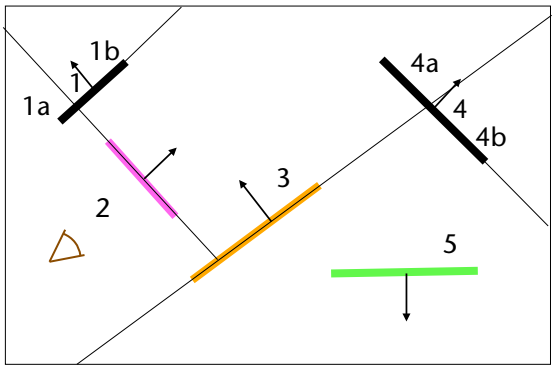

- Viewpoint liegt auf der Rückseite von 4b, somit zeichnen wir zuerst die Polygone auf der Vorderseite von 4b

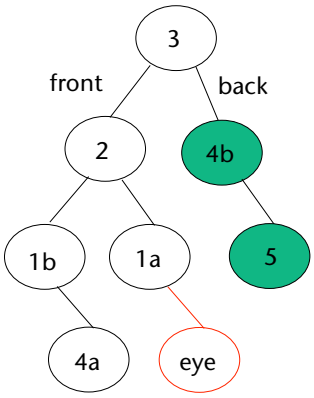



- Zeichne 4b und im Anschluss die Polygone auf der Rückseite von 4b, also 5

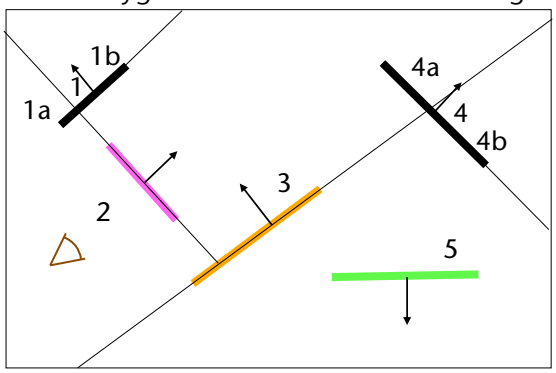
- Nun zeichnen wir die Vorderseite von 3

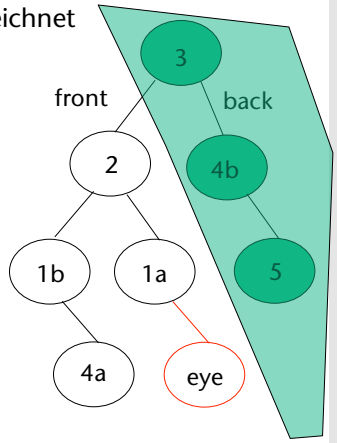


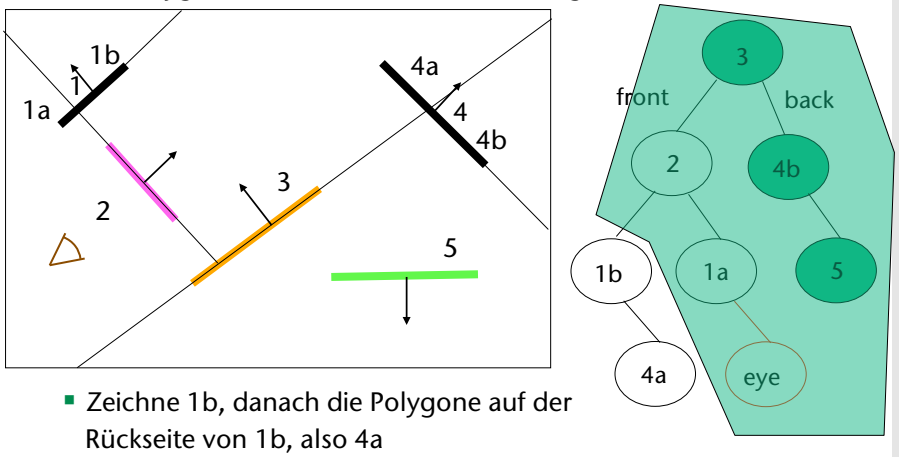



- Der Viewpoint liegt auf der Rückseite von 2, somit werden erst die Polygone auf der Vorderseite von 2 gezeichnet



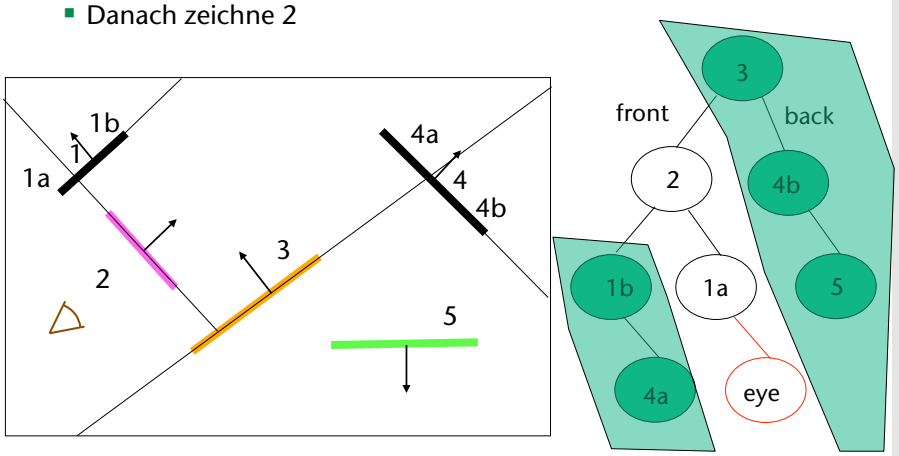
- Der Viewpoint liegt auf der Rückseite von 1b, somit werden erst die Polygone auf der Vorderseite von 1b gezeichnet




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



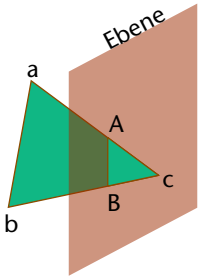
- Danach zeichne 2

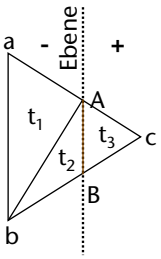



- Im Anschluss die Polygone auf der Rückseite von 2, also 1a
- Ergibt Gesamtreihenfolge: 4b, 5, 3, 1b, 4a, 2, 1a

## Zerschneiden von Dreiecken

- Dreieck schneidet die Ebene → unterteilen



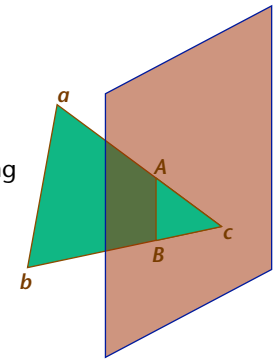




$t_1 = (a, b, A)$
$t_2 = (b, B, A)$
$t_3 = (A, B, c)$

- 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



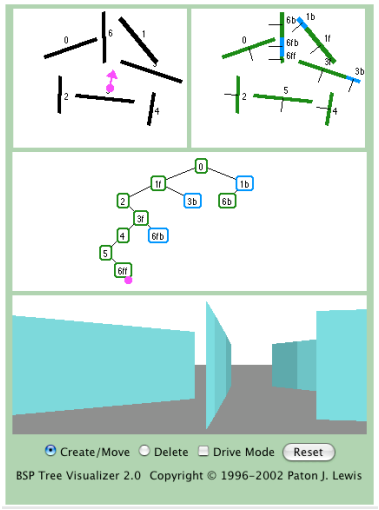

$$f_{plane}(p) = (n \cdot p) - d$$
$$= n \cdot (a + t(c - a)) - d \stackrel{!}{=} 0$$

  - 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

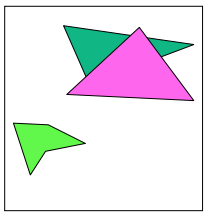
- Sehr effiziente Datenstruktur um Polygone bzgl. eines Punktes zu sortieren!
- Unabhängig vom Viewpoint
- Wird auch für andere Aufgaben benötigt

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]

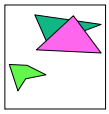
- Ein Image-Space-Verfahren, das auf einer rekursiven Unterteilung des Bildschirms beruht, bis die einzelnen Gebiete "homogen" sind
- Heute nicht mehr relevant (im Moment)
- Zeigt aber sehr schön folgendes algorithmisches Prinzip:
  - Kann man eine geometrische Entscheidung nicht für den ganzen Bereich fällen, so teile diesen erst einmal auf (hier: Bildraum wird aufgeteilt)
  - Ist im Prinzip eine Variante von Divide-and-Conquer

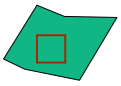


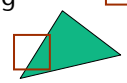


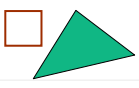
## Idee

- Unterteile den Bereich in 4 gleiche Gebiete
- Treffe für jedes Teilgebiet die Entscheidung, welches Polygon (vorne) gezeichnet werden soll
- In jedem Gebiet liegt bzgl. jedes der Polygone einer der folgenden 3 Fälle vor:
  1. Das Gebiet liegt komplett innerhalb eines Polygons, und dieses befindet sich vor allen anderen Polygonen, die auch innerhalb dieses Gebietes liegen
  2. Genau 1 Polygon liegt im Gebiet (entweder vollständig innerhalb des Gebiets oder dieses schneidend)
  3. Gebiet und Polygon sind disjunkt





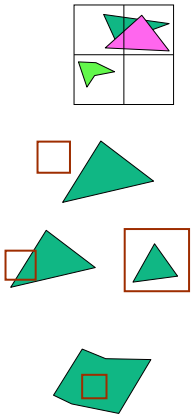




- Nicht vom Gebiet geschnittene Polygone beeinflussen das Gebiet nicht
- Schneidet ein Polygon das Gebiet, so beeinflusst der außerhalb liegende Teil das Gebiet nicht
- In jedem Schritt berechnen wir die Farbe des Gebietes; ist die Berechnung nicht eindeutig, dann unterteile es erneut



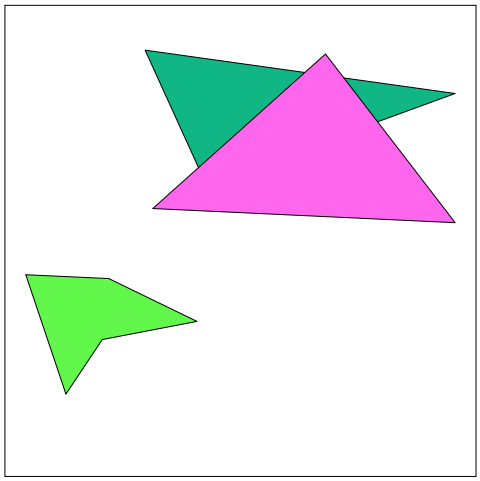
- Der Algo unterteilt nun rekursiv den Bildschirm (und die Menge der Polygone)
- Bei jeder Rekursion wird das Teilgebiet untersucht:
  1. Kein Polygon innerhalb des Gebietes → fülle mit der Hintergrundfarbe
  2. Nur 1 Polygon liegt ganz oder teilweise innerhalb des Gebiets → fülle Gebiet mit der Hintergrundfarbe 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 das Gebiet mit der Farbe dieses Polygons
  - Anderenfalls: unterteile das Gebiet und fahre mit Rekursion fort



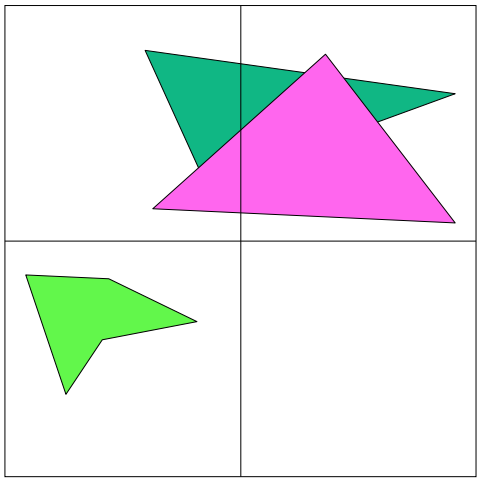
- Unterteilung wird fortgeführt bis:
  - Alle Gebiete entsprechen einer der vier Kriterien
  - Die Größe des Gebietes entspricht einem Pixel
    - In diesem Fall wird die Farbe irgendeines Polygons zum Füllen gewählt; oder ...
    - Man füllt mit dem Mittelwert aller Polygonfarben; oder ...
    - Man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



## Beispiel

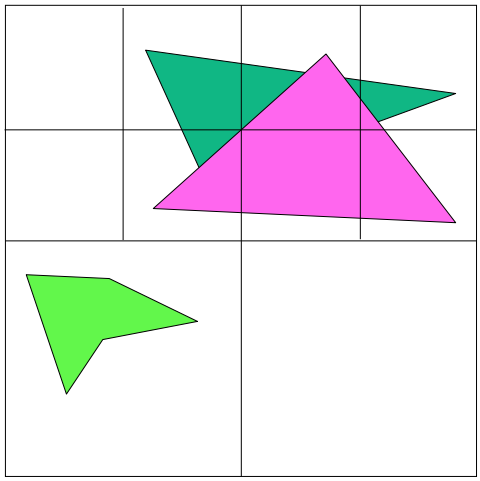


Ausgangsszene

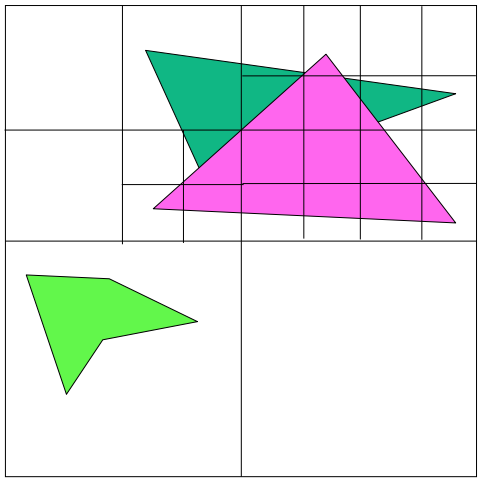


Erste Unterteilung



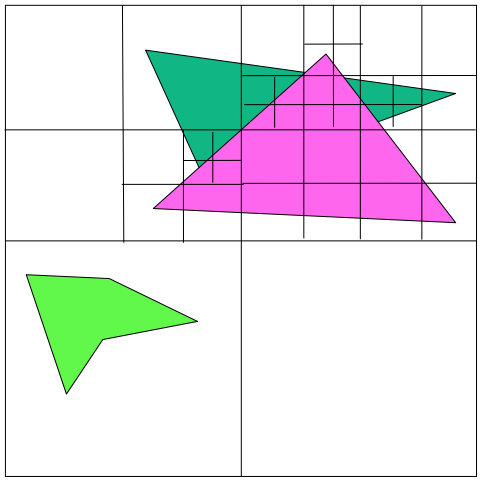


zweite Unterteilung



dritte Unterteilung

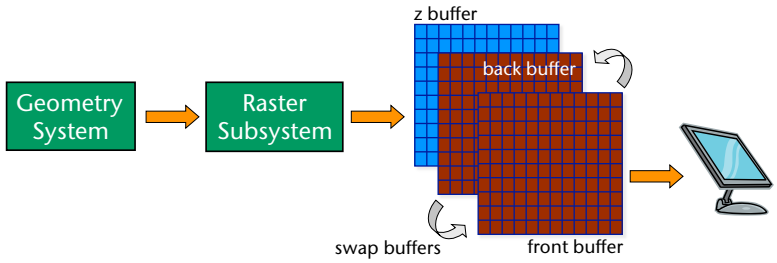




vierte Unterteilung

## 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!)