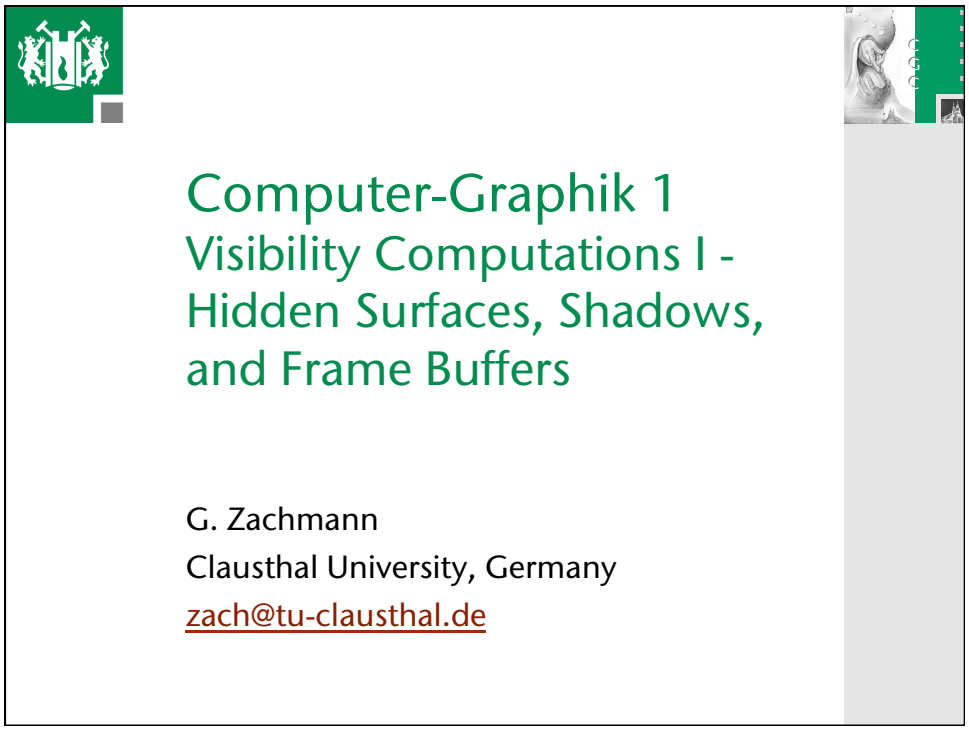
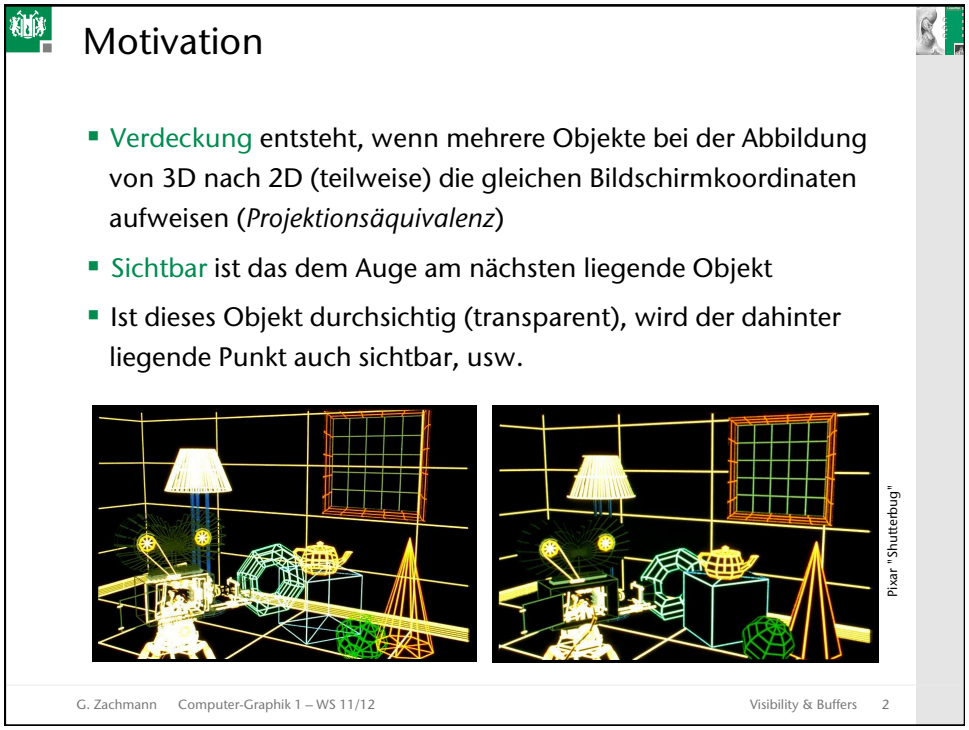


Computer-Graphik 1
Visibility Computations I - Hidden Surfaces, Shadows, and Frame Buffers

G. Zachmann
Clausthal University, Germany
zach@tu-clausthal.de



Motivation

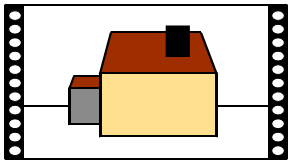
- Verdeckung entsteht, wenn mehrere Objekte bei der Abbildung von 3D nach 2D (teilweise) die gleichen Bildschirmkoordinaten aufweisen (*Projektionsäquivalenz*)
- Sichtbar ist das dem Auge am nächsten liegende Objekt
- Ist dieses Objekt durchsichtig (transparent), wird der dahinter liegende Punkt auch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

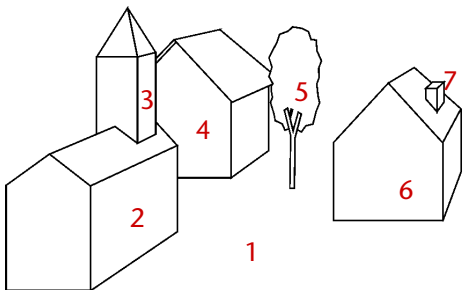
Pixar "Shutterbug"

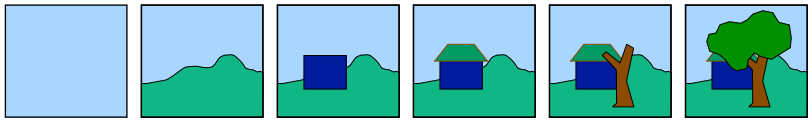
- Es gibt 2 große Problemklassen innerhalb des Bereichs "Visibility Computations"
  1. Verdeckungsberechnung: welche Polygone (oder Teile) werden von anderen verdeckt?
     1. Bezeichnungen: *Hidden Surface Elimination* (früher auch *Hidden Line Elimination*), *Visible Surface Determination*
  2. Culling: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

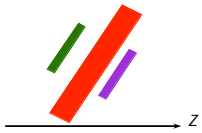


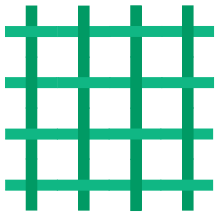




Probleme

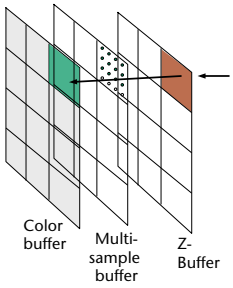
- Es gibt Fälle, in denen eine korrekte Sortierung nicht existiert!
  - Oder nicht klar ist ...
- Eine Lösung wäre evtl. eine Zerlegung der Polygone — aber ...
- Diese Zerlegung ist (im Prinzip) abhängig vom Viewpoint; und ...
- Bei einer Szene mit n Polygonen können $O(n^2)$ sichtbare Fragmente entstehen

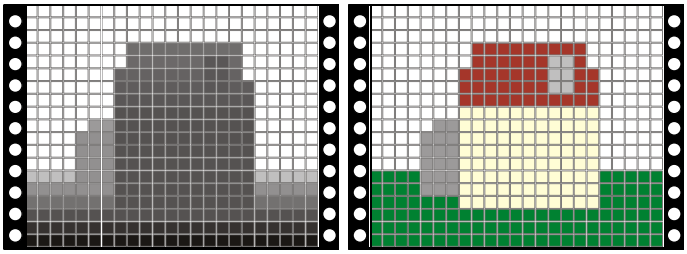




Die Standard-Lösung heute: der Z-Buffer

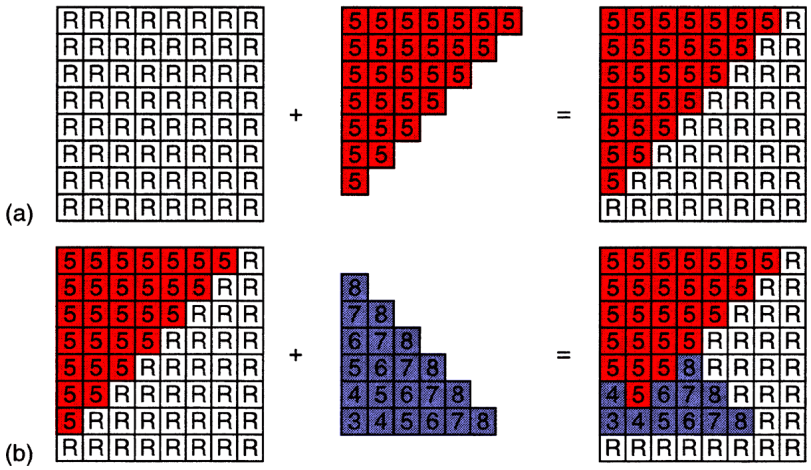
- Zusätzlich zum Color Buffer (R, G, B)
- Speichert pro Pixel den Abstand z zur Kamera
- Pixel wird geschrieben, wenn Z kleiner ist als der Wert im Z-Buffer





Z-Buffer Color Buffer

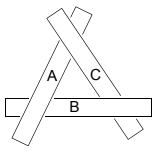
Beispiel

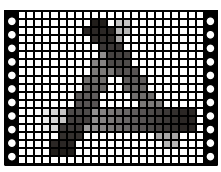


Z-Buffer Pseudo-Code

```
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:

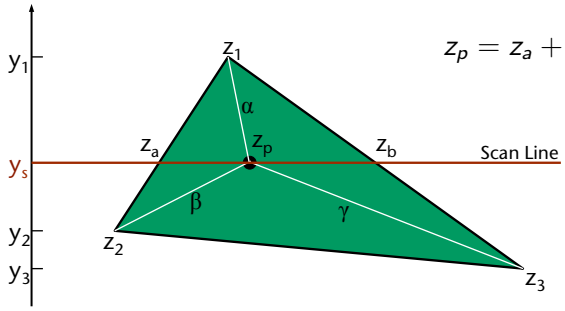




Berechnung des Z-Wertes bei der Scan-Conversion

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

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



- 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 );
```

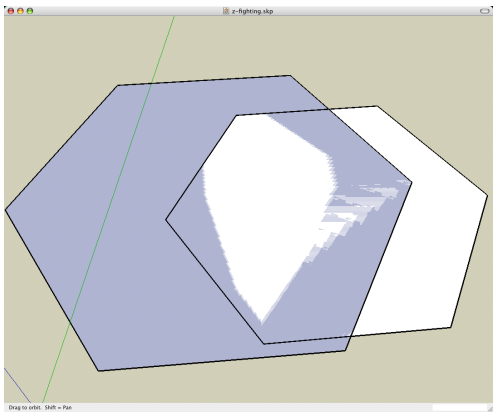
3. Wichtig: nicht nur Bildspeicher, sondern auch Z-Buffer löschen!

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

- Achtung: unter Qt ist (1) und (2) per Default angeschaltet
  - Mehr Info unter http://doc.trolltech.com/4.2/qglformat.html
  - Beispiel zu QtFormat 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:



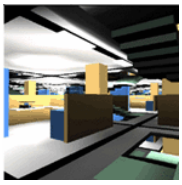
Bewertung

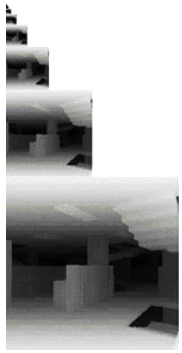
- Komplexität des Algorithmus': $O(n)$, mit n = Anzahl Polygone
  - Kein zusätzlicher Aufwand, z.B. durch Sortieren (z.B. $O(n \log n)$)
- Eigentlich: $O(n+p)$, wobei p = # geschriebene Pixel (kann unter Umständen viel größer als Anzahl sichtbarer Pixel sein!)
- Läßt sich ideal in Hardware implementieren:
  - 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, um Speicherplatz zu sparen

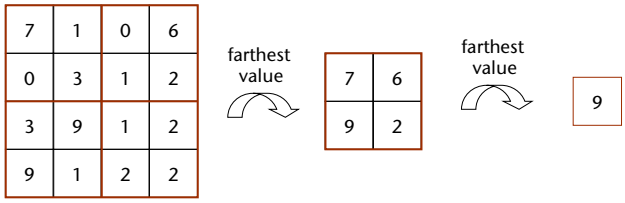
Hierarchischer Z-Buffer (HZB)

[Greene, 1993]

- Idee: „Z-Pyramide“
  - einfacher Z-Buffer = höchste Auflösung
  - weitere Levels durch Zusammenfassen von jeweils 4 Pixel
  - z-Wert auf max. z-Wert setzen

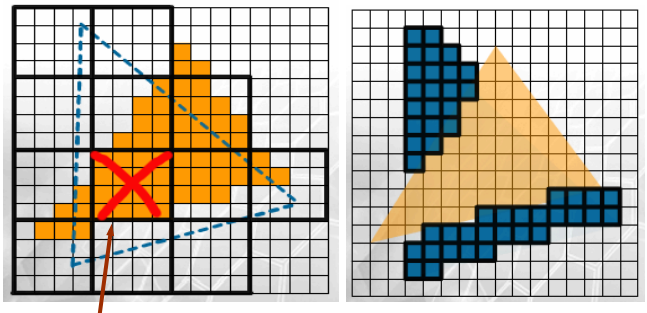






Beispiel

- Orangenes Dreieck bereits gezeichnet
- Blaues Dreieck soll dahinter gezeichnet werden

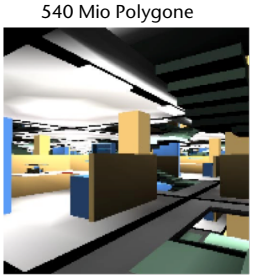


vollständig verdeckt → verwerfe gesamten Block

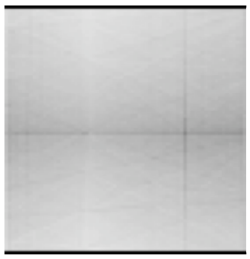
Vergleich

- Definition Depth-Complexity:
  - Eigentlich: Anzahl Polygone, die "hinter" einem Pixel liegen
  - Hier: Anzahl z-Tests pro Pixel

540 Mio Polygone



Depth complexity mit einfachem Z-Buffer



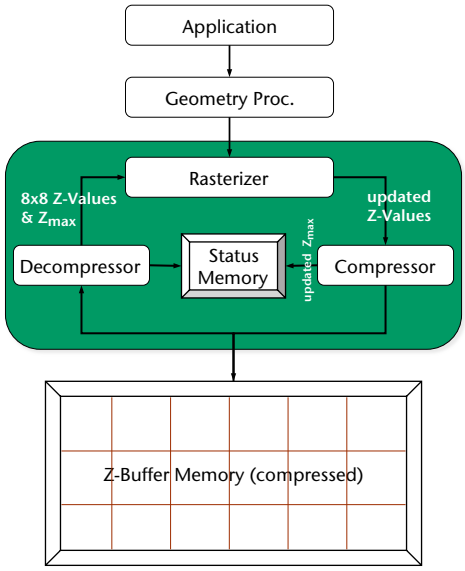
Depth complexity mit HZB

- Definition Over-Drawing = Maß dafür, wie oft ein Pixel tatsächlich überschrieben wird

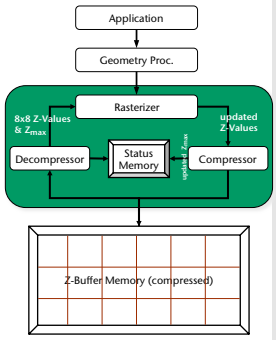
Implementierung in aktueller Graphik-Hardware

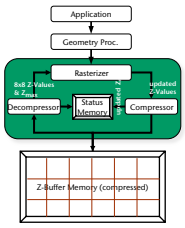
- Problem: Bandbreite zwischen Rasterizer und Speicher beträgt ca. 18 Gbyte/sec!
  - Annahmen: Auflösung 1280x1024, 4x depth complexity pro Pixel, pro Fragment: 1x Z-Buffer-Read + 1x Z-Buffer-Write + 1x Color-Buffer-Write + 2x Texture-Read, pro Read/Write 32 Bit
  - Aktueller Speicher erlaubt ca. 10 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: