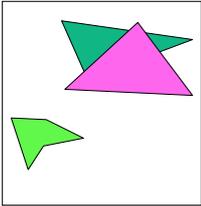


## Warnock's Algorithmus [1996]



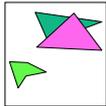
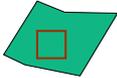
- Ein Image-Space-Verfahren, das auf einer rekursiven Unterteilung des Bildschirms beruht, bis die einzelnen Gebiete "homogen" sind
- Heute nicht mehr relevant (im Moment)
- Zeigt aber sehr schön folgendes algorithmisches Prinzip:
  - Kann man eine geometrische Entscheidung nicht für den ganzen Bereich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



G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 44

## 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

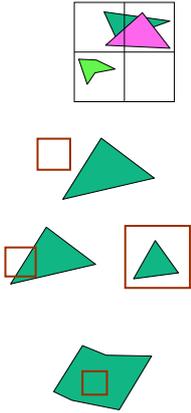


G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 45

- 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

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 46

- 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

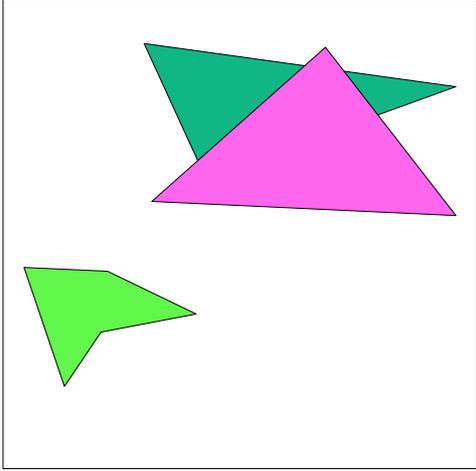


G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 47

- 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

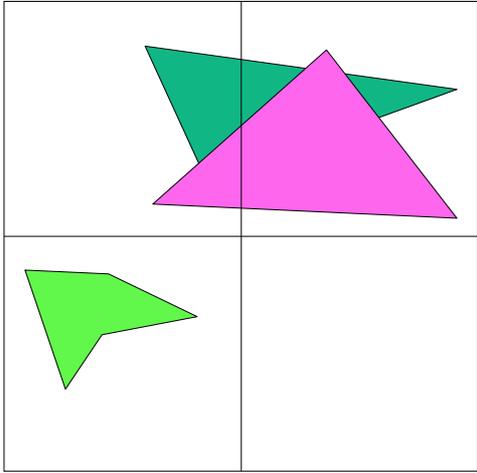
G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 48

### Beispiel



Ausgangsszene

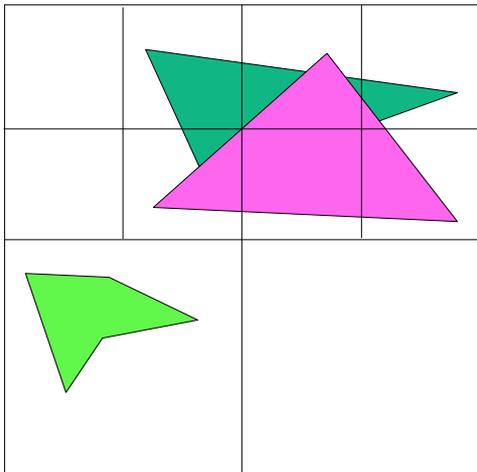
G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 49



Erste Unterteilung

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 50

The diagram illustrates the first subdivision of a scene. A 2x2 grid is shown. In the top-left quadrant, there is a cyan triangle pointing right. In the top-right quadrant, there is a pink triangle pointing down. In the bottom-left quadrant, there is a green triangle pointing right. The bottom-right quadrant is empty. The text 'Erste Unterteilung' is centered below the grid.



zweite Unterteilung

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 51

The diagram illustrates the second subdivision of the scene. A 4x4 grid is shown. The cyan triangle is now in the top-left quadrant of the 2x2 grid. The pink triangle is in the top-right quadrant of the 2x2 grid. The green triangle is in the bottom-left quadrant of the 2x2 grid. The bottom-right quadrant of the 2x2 grid is empty. The text 'zweite Unterteilung' is centered below the grid.

**dritte Unterteilung**

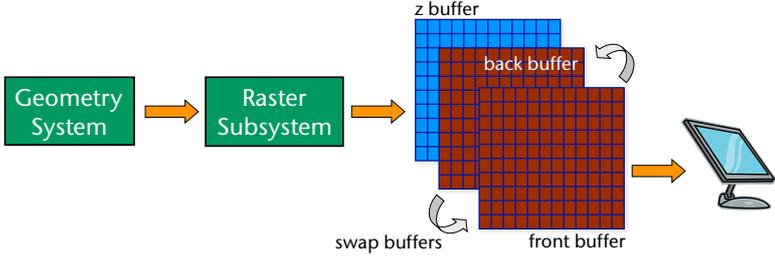
G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 52

**vierte Unterteilung**

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 53

## 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



G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 54

## 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"  $2 \cdot 16 - 22 = 10$  Millisek. warten!

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 55

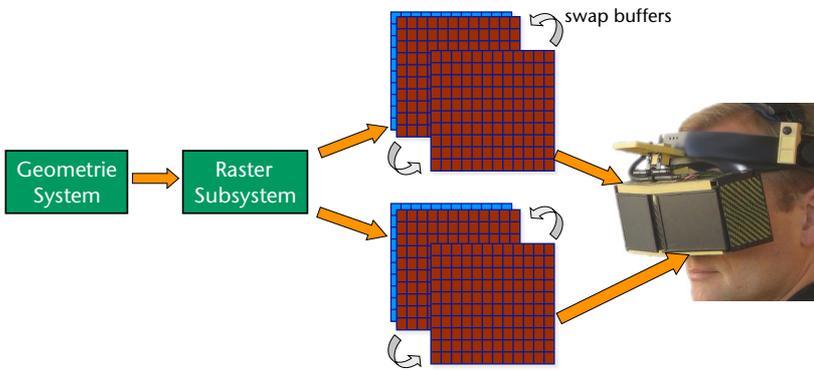
## Weitere Synchronisationen

- 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

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 56

## 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!)



The diagram illustrates the quad buffer architecture for stereo rendering. It starts with a 'Geometrie System' (Geometry System) which feeds into a 'Raster Subsystem'. The Raster Subsystem outputs two separate rasterized images, one for the left eye and one for the right eye. Each image is rendered into a pair of buffers: a front buffer and a back buffer. The back buffers are used for depth testing (Z-Buffering). After rendering, the front and back buffers for each eye are swapped, as indicated by the 'swap buffers' label and arrows. The resulting two stereo images are then displayed to a user wearing a VR headset.

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 57

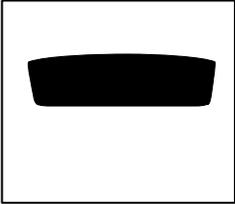
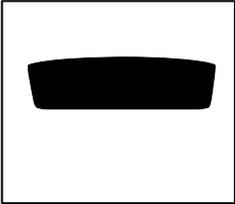
## Der Stencil Buffer

- 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  $s \text{ func } ref$
    - 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 fest, wie und ob in den *Stencil-Buffer* geschrieben 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. ...

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 61

## Typisches, einfaches Beispiel

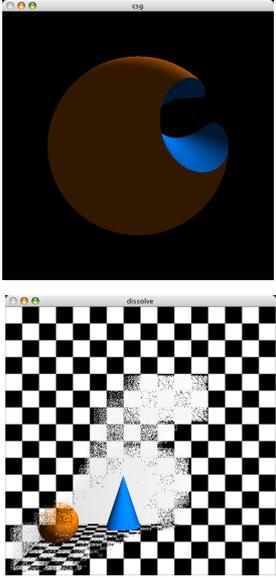
- Szene durch ein Objekt maskieren:
  1. Alle Buffer inkl. Stencil-Buffer löschen
  2. Objekt A rendern, dabei Stencil-Buffer überall dort auf 1 setzen, aber Color-Buffer unverändert lassen(!)
  3. Rest der Szene zeichnen, aber nur dort, wo Stencil-Wert = 1

	Color Buffer	Stencil Buffer
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		

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 62

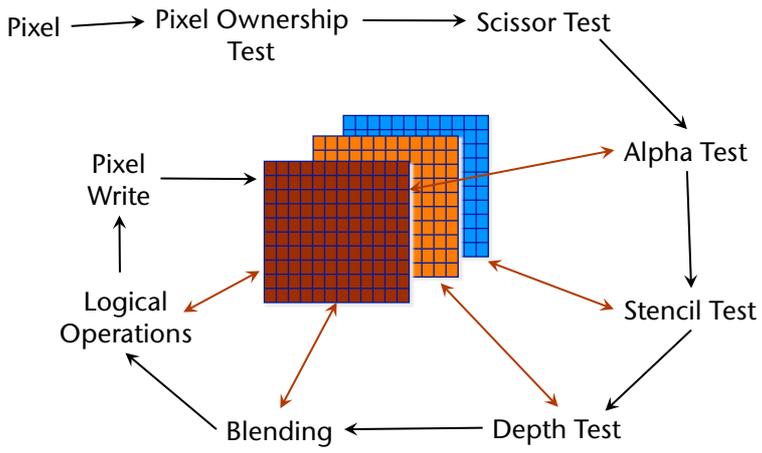
### Beispiele für komplexere Operationen/Effekte

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



G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 63

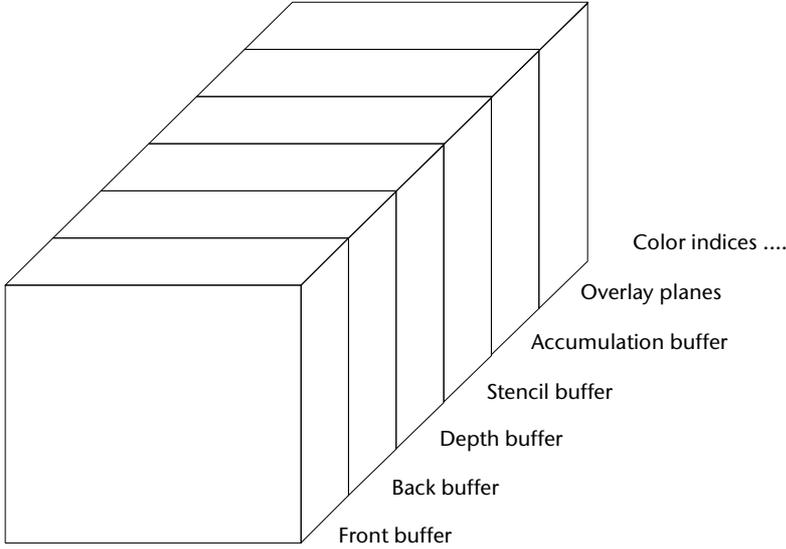
### Die Folge von Tests in der Graphik-Pipeline



Pixel → Pixel Ownership Test → Scissor Test → Alpha Test → Stencil Test → Depth Test → Blending → Logical Operations → Pixel Write

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 64

■ Bemerkung: es gibt viele weitere Buffer in OpenGL

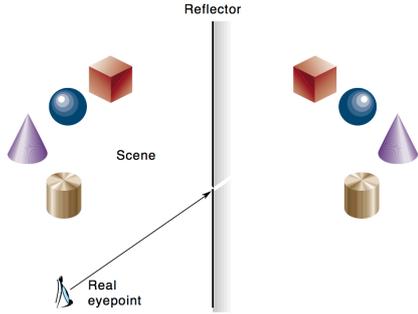


Color indices ....  
 Overlay planes  
 Accumulation buffer  
 Stencil buffer  
 Depth buffer  
 Back buffer  
 Front buffer

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 65

■ 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)



G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 66

- 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

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 67

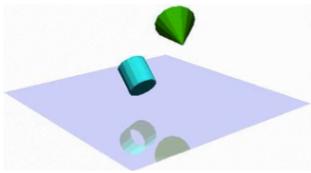
### Der 2-pass Algorithmus im Detail

- Clear color & z buffer; set up viewpoint, etc.
- Pass 1:
  - Set clipping plane, so objs *in front* of mirror are *not* rendered
  - Compute reflection transformation and apply to all polygons
  - Render scene without mirror geometry
- 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
 

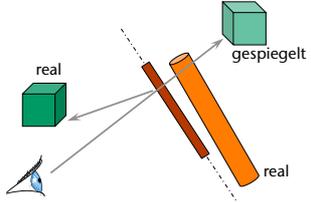
```
-glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
          glStencilFunc (GL_ALWAYS, 1, 1);
          glEnable (GL_STENCIL_TEST);
```

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 68



- Disable drawing into the color buffer
  - `glColorMask(0, 0, 0, 0)`
- Draw the mirror geometry, 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:
 

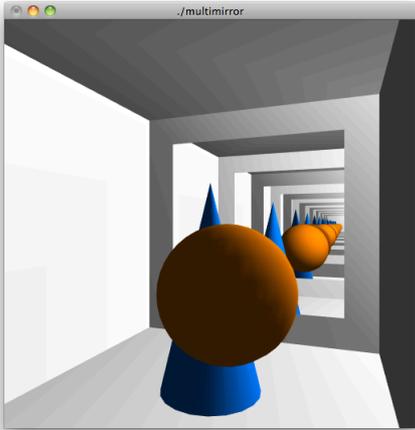
```
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glStencilFunc(GL_NOTEQUAL, 1, 1);
```
  - Clear color buffer, so that pixels outside the mirror return to the background color: `glClear(GL_COLOR_BUFFER_BIT)`
- Pass 2:
  - Disable stencil test
  - Disable clipping plane
  - Render scene as usual



G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 69

## Demo

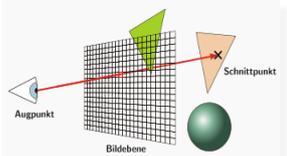
- Das Ganze kann man rekursiv machen:

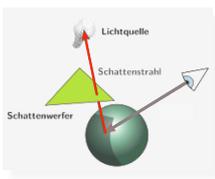


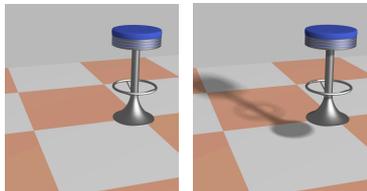
G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 70

## 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
- 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



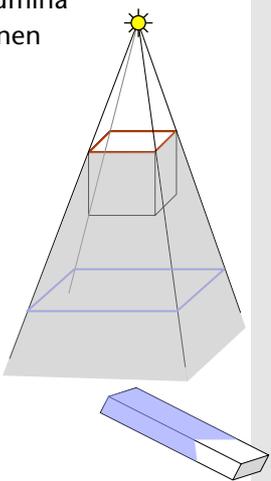




G. Zachmann Computer-Graphik 1 – WS 10/11
Visibility & Buffers 71

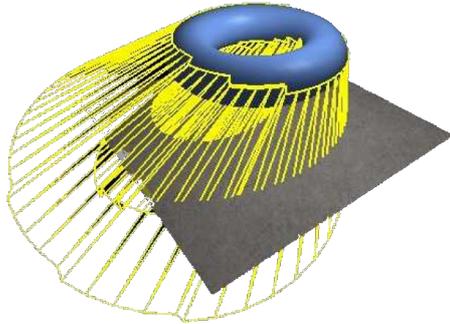
## Das Schattenvolumen

- Ansatz im Folgenden: modelliere die (Teil-)Volumina des Universums, die kein Licht von der gegebenen Lichtquelle erhalten
- Das **Schattenvolumen**:
  - Ein Kegelstumpf, mit der Lichtquelle als Spitze
  - Erzeugt durch einen "*shadow caster*"
  - Pro **Silhouettenkante** (*silhouette edge*), von der Lichtquelle aus(!), des Casters gibt es genau ein Quad im Shadow Volume
  - Der Kegelstumpf ist (im Prinzip) unendlich
- Bemerkung: die Silhouettenkanten liegen nicht notwendigerweise in einer Ebene!
- Liegt ein Objekt (teilweise) im Inneren des Schattenvolumens, so heißt dieses "*shadow receiver*"



G. Zachmann Computer-Graphik 1 – WS 10/11
Visibility & Buffers 72

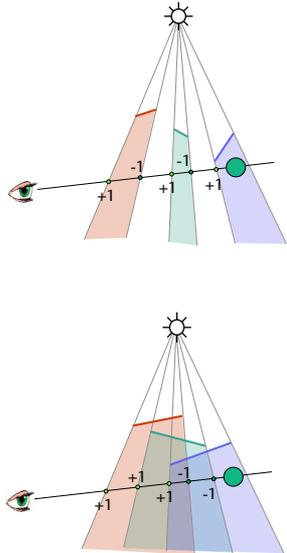
▪ Beispiel für ein komplexeres Shadow Volume:



G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 73

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**



G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 74

## 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 Volume; 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 val
glCullFace(GL_FRONT); // for all back-facing pgons ..
glStencilOp(GL_KEEP, GL_KEEP, GL_DECR); // .. passing the depth test
renderShadowVolumePolygons(); // .. decrease stencil val

```

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);        // must match from 1st step
glDepthFunc(GL_EQUAL);          // no need to update z buffer
glDepthMask(0);                 // only render pixels that are
glEnable(GL_STENCIL_TEST);       // inside the shadow
glStencilFunc(GL_EQUAL, 1, ~0u); // no need to update stencil
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP); // do modify color buffer
glColorMask(1,1,1,1);
renderScene();

```

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

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 77

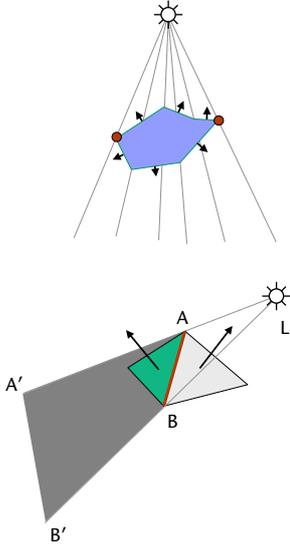
Bemerkungen

- 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 klar
- 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)

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 78

■ Berechnung der Silhouettenkanten:
 

- Kante (mit genau 2 adjazenten Polygonen) ist Silhouettenkante  $\Leftrightarrow$  ein Polygon zeigt zur Lichtquelle und ein Polygon zeigt weg von der Lichtquelle (Skalarprodukt)



G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 79

■ Beispiele

Shadowed scene

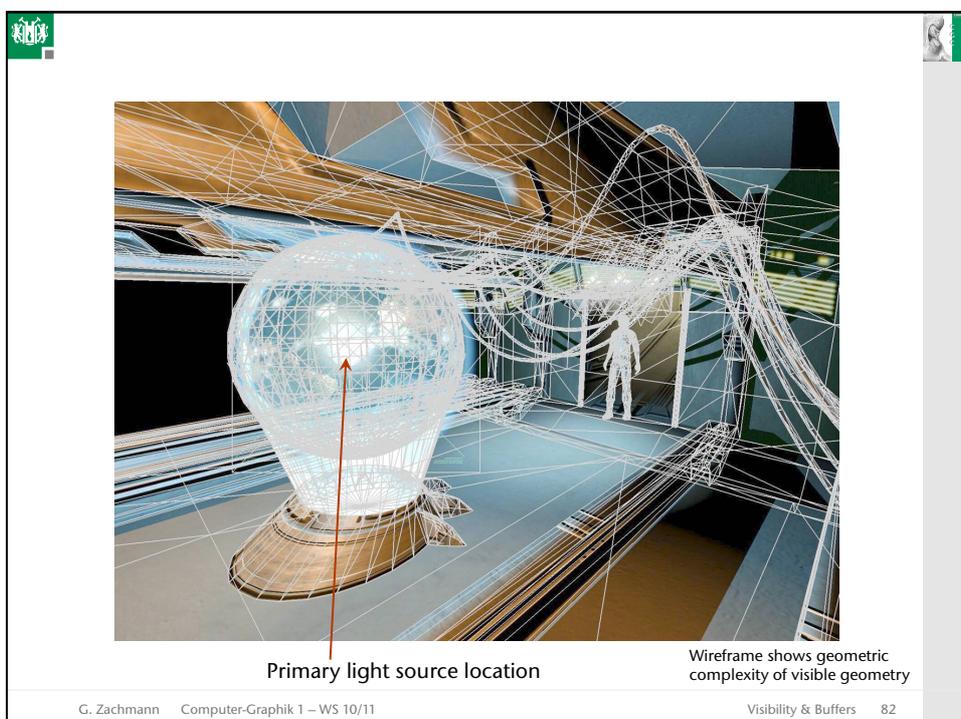
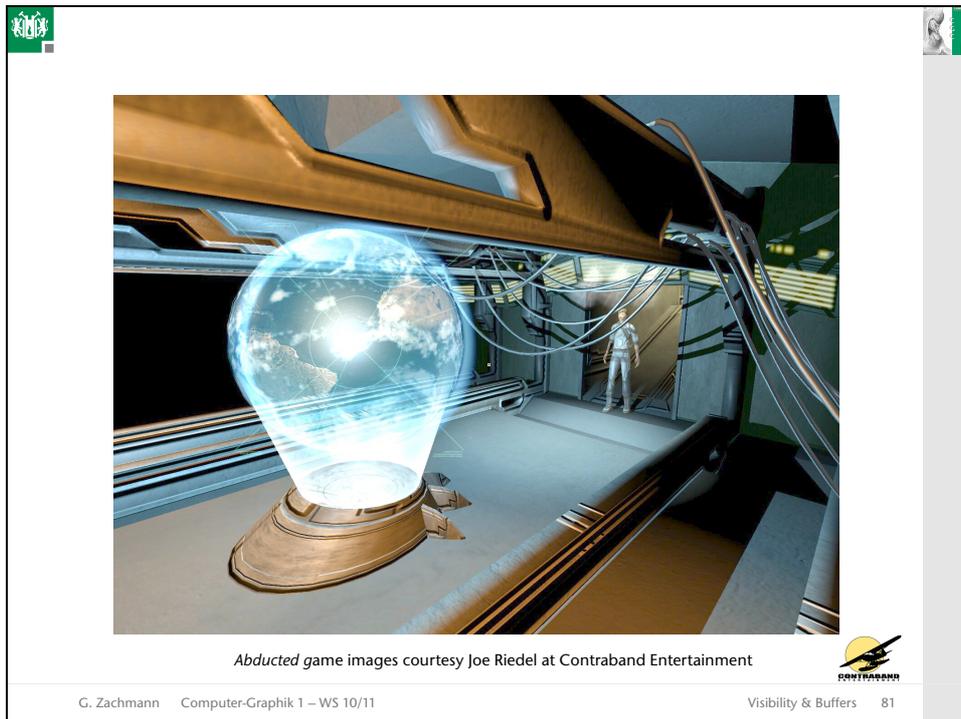


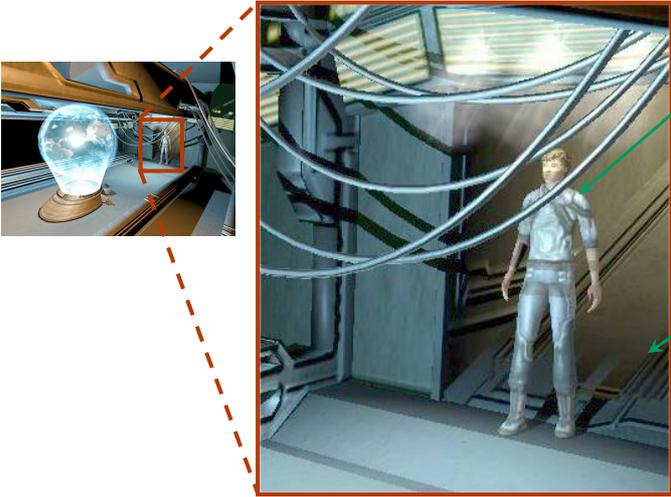
Stencil buffer contents



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

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 80





Notice cable shadows on player model

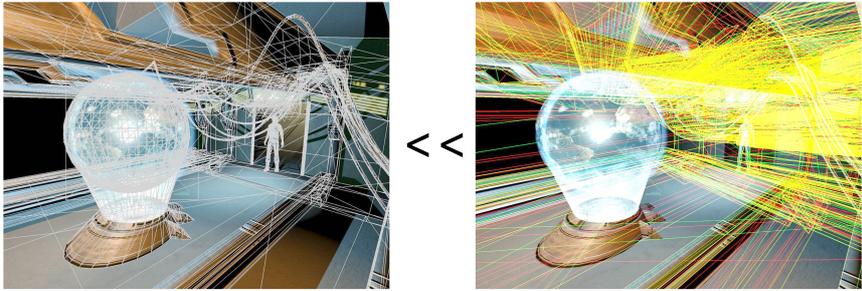
Notice player's own shadow on floor

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 83



Wireframe shows geometric complexity of shadow volume geometry

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 84

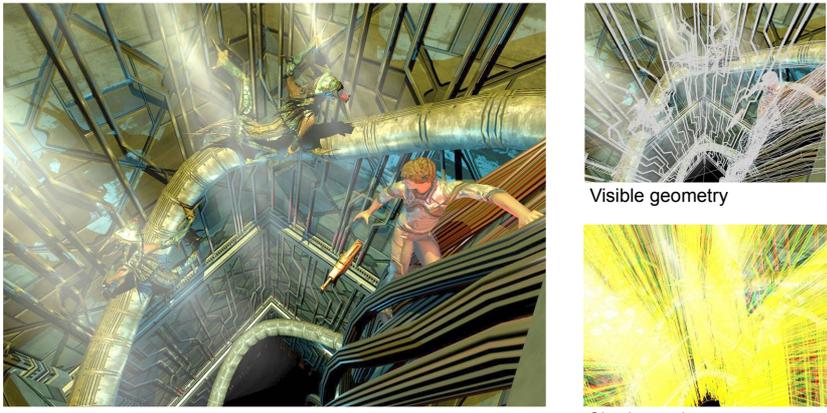


Visible geometry << Shadow volume geometry

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

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 85

This slide compares two rendering techniques for a scene featuring a glowing light bulb on a stand. The left image, labeled 'Visible geometry', shows the scene with a dense network of white lines representing the geometry of the environment and the light bulb. The right image, labeled 'Shadow volume geometry', shows the same scene but with a much denser network of yellow and red lines, representing the shadow volumes cast by the geometry. A double less-than sign (<<) is placed between the two images, indicating that the shadow volume geometry is significantly more complex than the visible geometry. Below the images, a text block states: 'Typically, shadow volumes generate considerably more pixel updates than visible geometry'. The footer contains the text 'G. Zachmann Computer-Graphik 1 – WS 10/11' and 'Visibility & Buffers 85'.

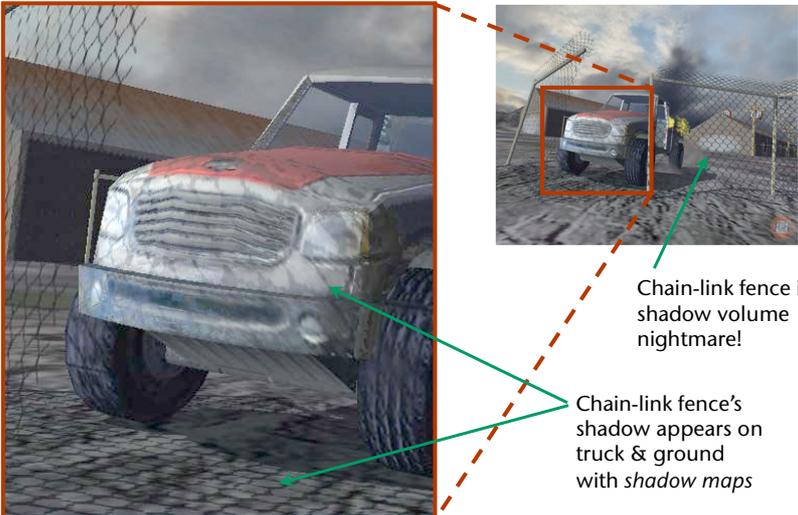


Visible geometry << Shadow volume geometry

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 86

This slide compares two rendering techniques for a complex industrial scene. The left image, labeled 'Visible geometry', shows a detailed 3D rendering of a character in a futuristic, industrial environment with various pipes, beams, and structures. The right image, labeled 'Shadow volume geometry', shows the same scene but with a dense network of yellow and red lines representing the shadow volumes cast by the geometry. A double less-than sign (<<) is placed between the two images, indicating that the shadow volume geometry is significantly more complex than the visible geometry. Below the images, a text block states: 'Visible geometry << Shadow volume geometry'. The footer contains the text 'G. Zachmann Computer-Graphik 1 – WS 10/11' and 'Visibility & Buffers 86'.

### Situations When Shadow Volumes Are Too Expensive



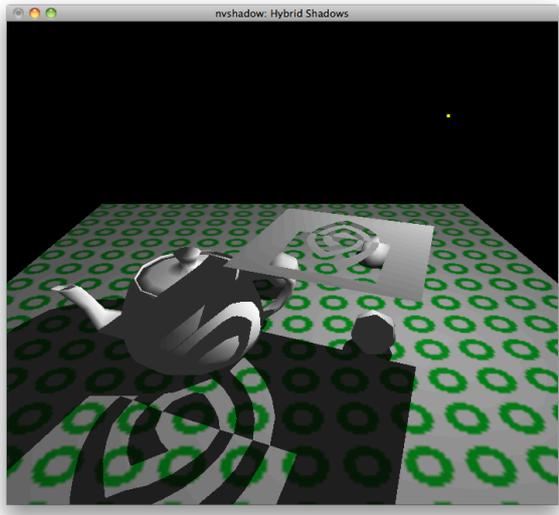
Chain-link fence is shadow volume nightmare!

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

*Fuel* game image courtesy Nathan d'Obrenan at Firetoad Software

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 87

### Demos



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

G. Zachmann Computer-Graphik 1 – WS 10/11 Visibility & Buffers 88