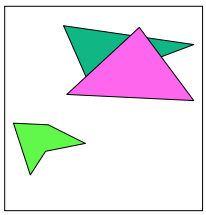
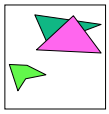
Warnock's Algorithmus [1996]

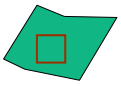
- Ein Image-Space-Verfahren, das auf einer rekursiven Unterteilung des Bildschirms beruht, bis die einzelnen Gebiete "homogen" sind
- Heute nicht mehr relevant (im Moment)
- Zeigt aber sehr schön folgendes algorithmisches Prinzip:
  - Kann man eine geometrische Entscheidung nicht für den ganzen Bereich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

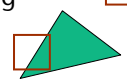


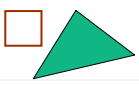
Idee

- 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



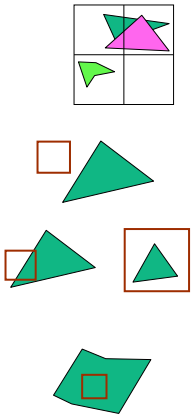






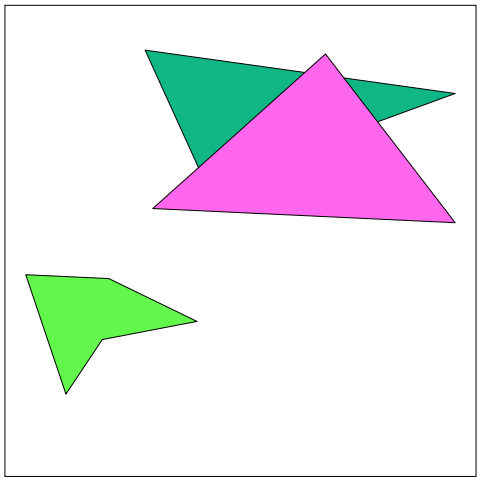
- Nicht vom Gebiet geschnittene Polygone beeinflussen das Gebiet nicht
- Schneidet ein Polygon das Gebiet, so beeinflusst der außerhalb liegende Teil das Gebiet nicht
- In jedem Schritt berechnen wir die Farbe des Gebietes; ist die Berechnung nicht eindeutig, dann unterteile es erneut

- 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

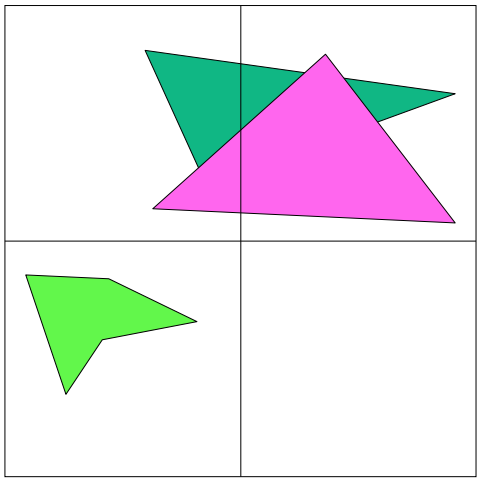


- Unterteilung wird fortgeführt bis:
  - Alle Gebiete entsprechen einer der vier Kriterien
  - Die Größe des Gebietes entspricht einem Pixel
    - In diesem Fall wird die Farbe irgendeines Polygons zum Füllen gewählt; oder ...
    - Man füllt mit dem Mittelwert aller Polygonfarben; oder ...
    - Man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

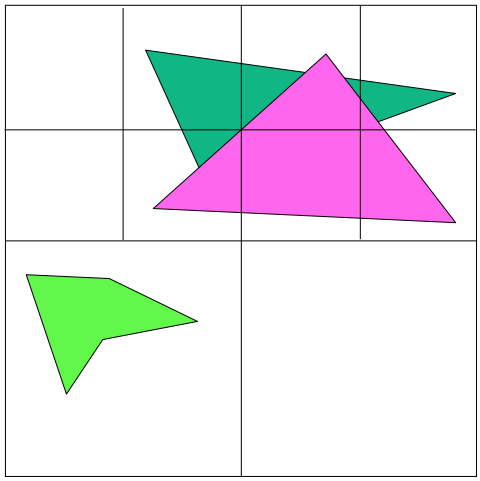
Beispiel



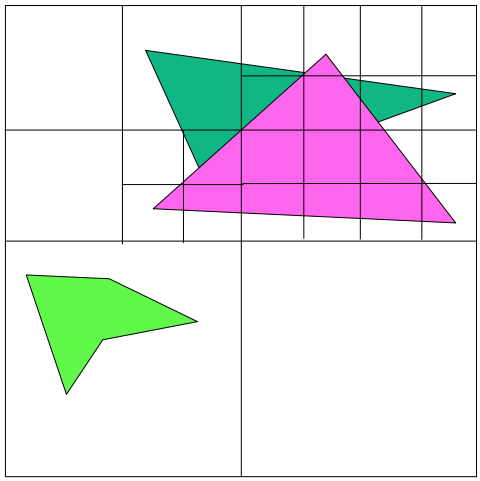
Ausgangsszene



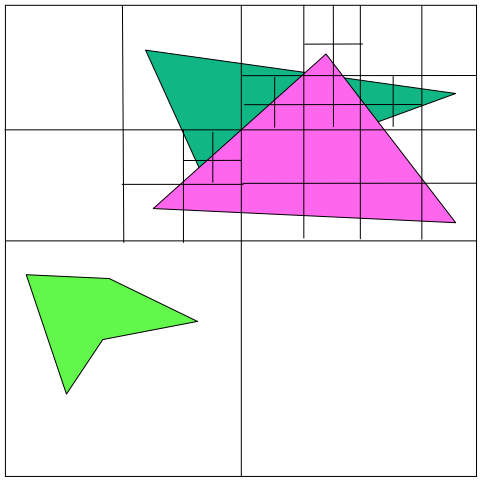
Erste Unterteilung



zweite Unterteilung



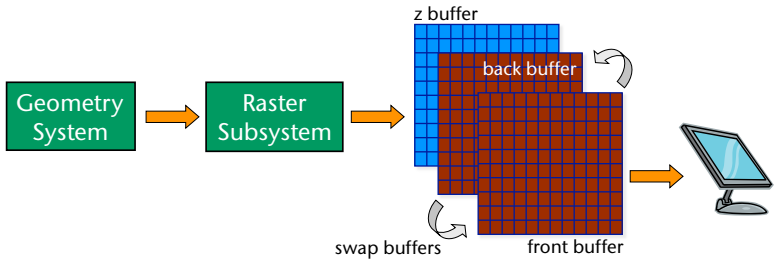
dritte Unterteilung



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

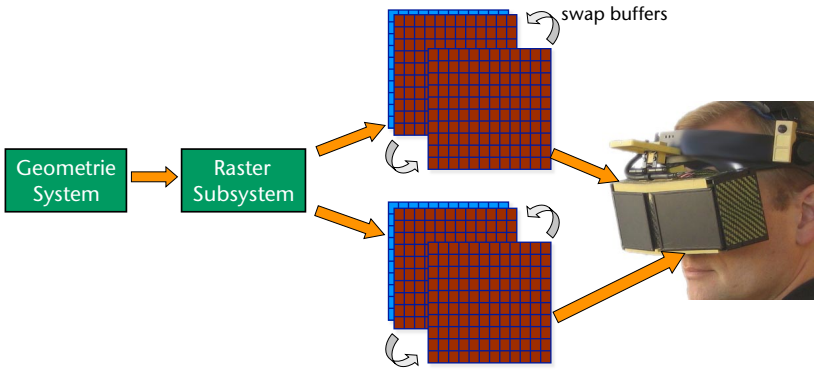
- Funktionsname in OpenGL: `glSwapBuffers()`
- Verwendet man praktisch nie "von Hand" bei Einsatz von high-level GUI-Libraries (Qt, GLUT, GLEW, etc.)
  - Dort liegt die *main loop* (und damit die Kontrolle) immer in der GUI-Library
- Der *Buffer-Swap* muß (normalerweise) mit dem *vertical retrace* des 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!)

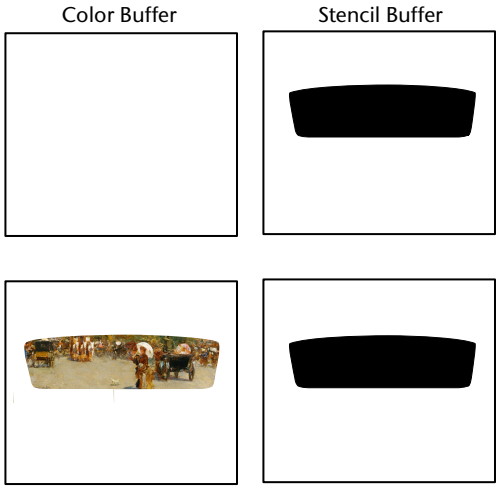


Der Stencil Buffer

- 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 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. ...

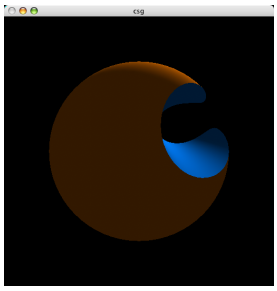
Typisches, einfaches Beispiel

- 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

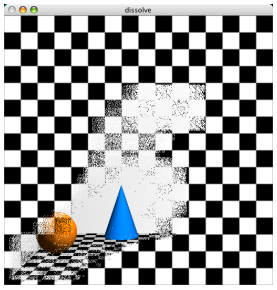


Beispiele für komplexere Operationen/Effekte

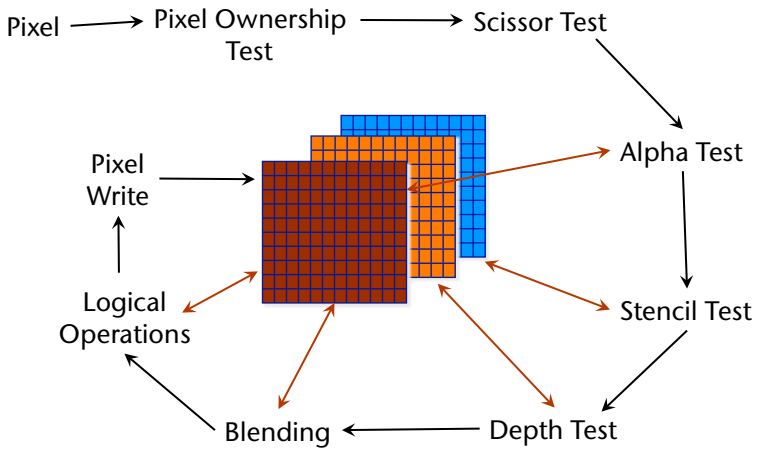
- Beispiel: CSG-Operationen (Schnitt, Differenz, ...)



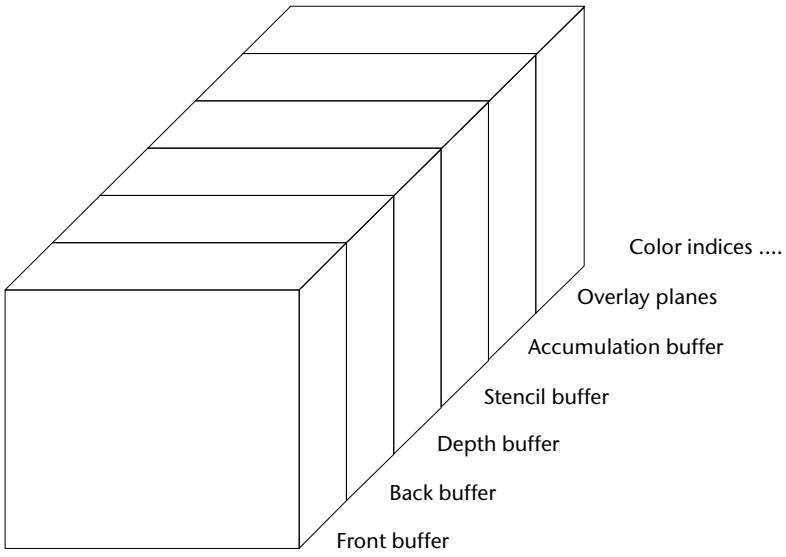
- Beispiel: "Dissolve"



Die Folge von Tests in der Graphik-Pipeline

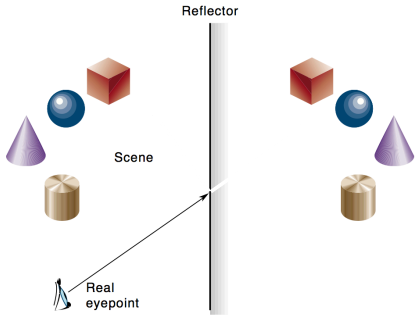


Bemerkung: es gibt viele weiter Buffer in OpenGL



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



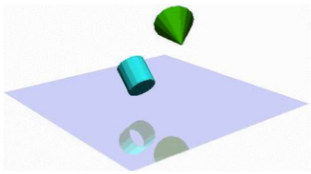
- Dies ist ein Beispiel für einen multi-pass Rendering-Algo
- Achtung: rendere in Schritt 2 nur Polygone hinter der Spiegelebene (mittels Clipping-Plane in Spiegelebene; → später)

- 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

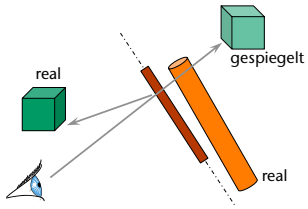
Der 2-pass Algorithmus im Detail

- Clear color & z buffer; set up viewpoint, etc.
- Pass 1:
  - Set clipping plane, so 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);
```



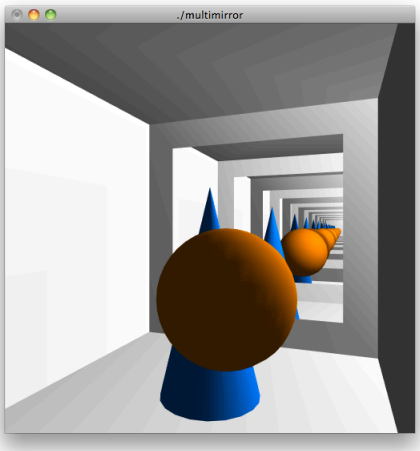
- Disable drawing into the color buffer
  - `glColorMask(0, 0, 0, 0)`
- Draw the mirror geometry, with blending if desired
  - This sets stencil bits & fills z buffer with depth value of mirror geometry
- Clear color buffer outside mirror geometry:
  - Configure the stencil test to pass outside the mirror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

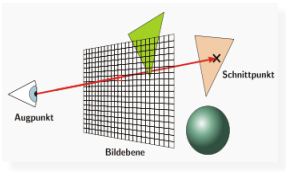
Demo

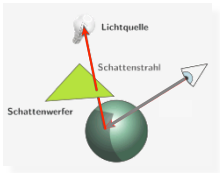
- Das Ganze kann man rekursiv machen:



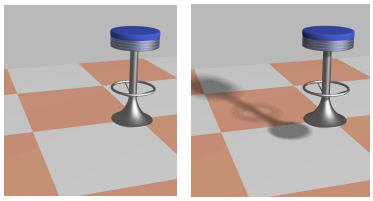
Rendering Shadows using Shadow Volumes

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



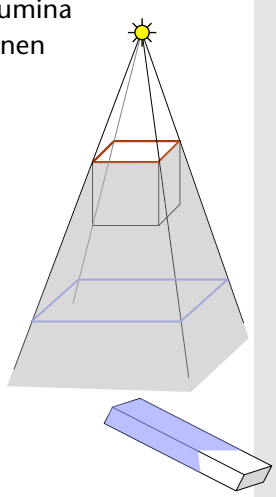


- Warum ist Schatten so wichtig?
  - Erhöhung des Realismus einer Szene
  - Bessere "Verankerung" der Objekte in der Szene:
    - Mehr Information über die relative Lage der Objekte im Raum
    - Tiefeninformation
  - Hervorhebung der Beleuchtungsrichtung

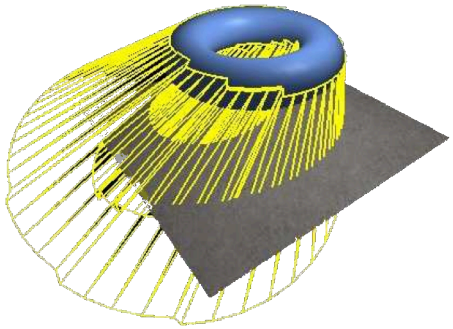


Das Schattenvolumen

- 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*"

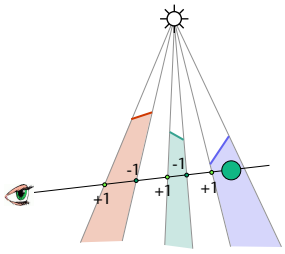


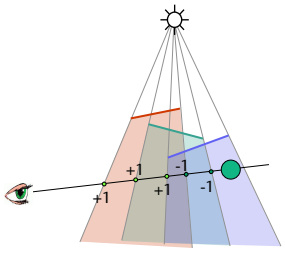
- Beispiel für ein komplexeres Shadow Volume:



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 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

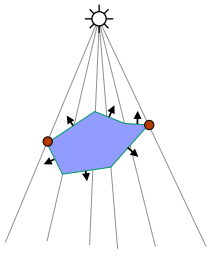
```
glEnable(GL_LIGHTING);                  // switch off light source 0
glDisable(GL_LIGHT0);                   //    but keep all others
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_EQUAL);                  // must match from 1st step
glDepthMask(0);                         // no need to update z buffer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

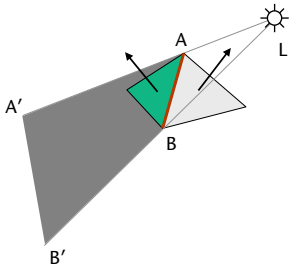
- Dieser Algorithmus heißt "z-pass algorithm", weil in Pass 2 nur Shadow-Volume-Pixel den Stencil-Wert verändern, die den Z-Test passieren

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)

- 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)



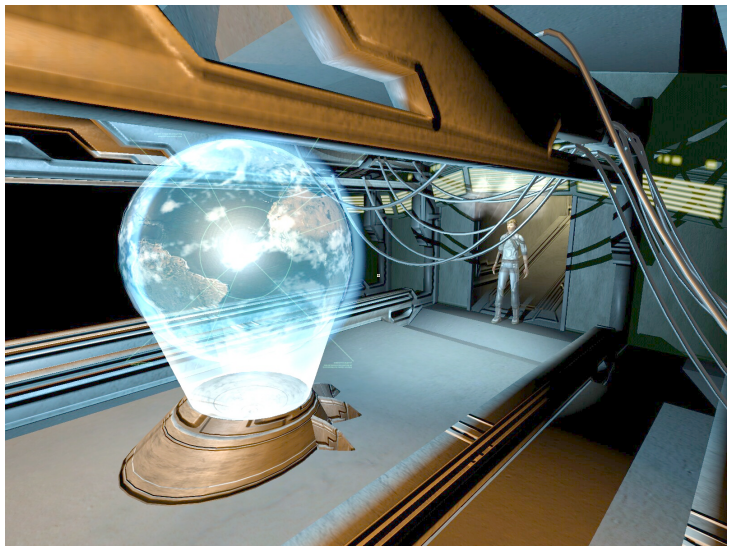


Beispiele

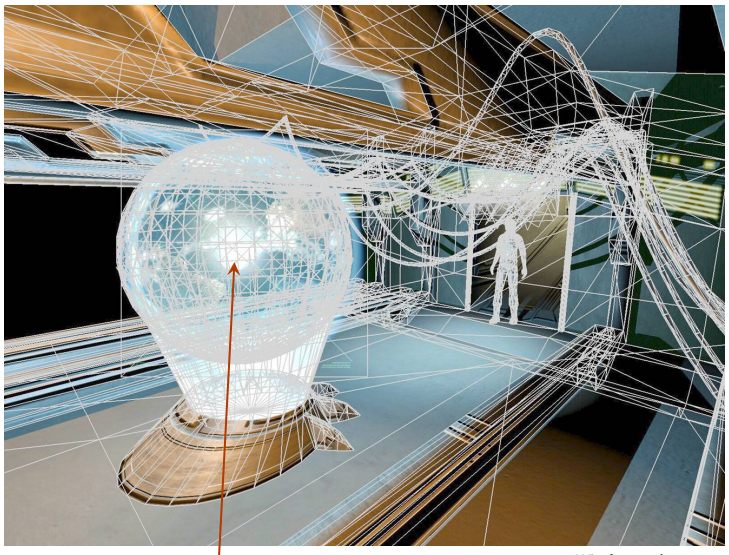
Shadowed scene

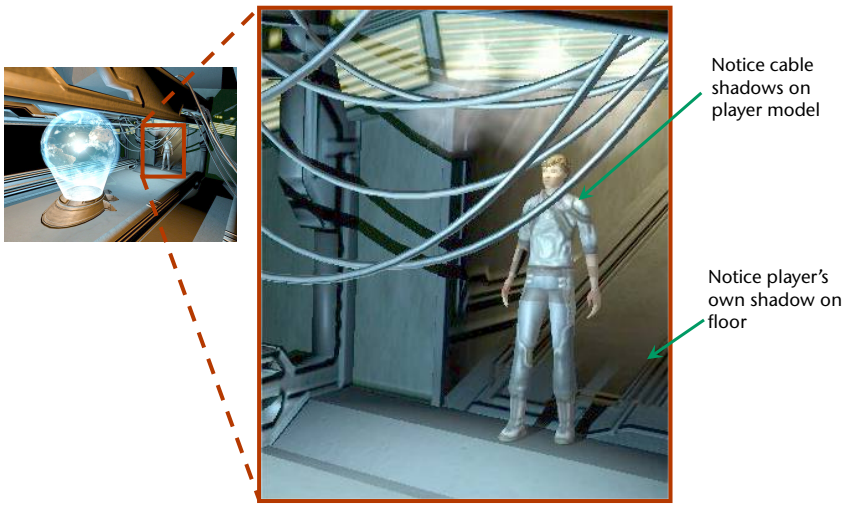
Stencil buffer contents

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

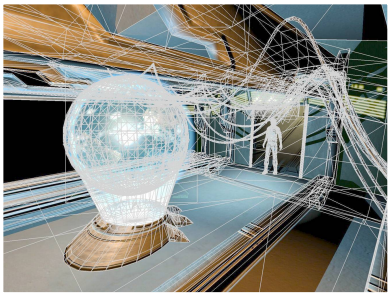


Abducted game images courtesy Joe Riedel at Contraband Entertainment

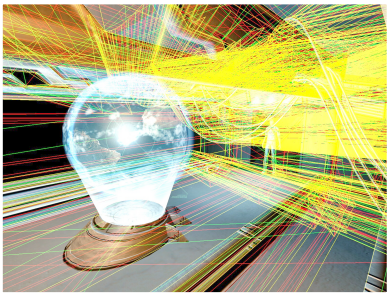




Wireframe shows geometric *complexity of shadow volume* geometry



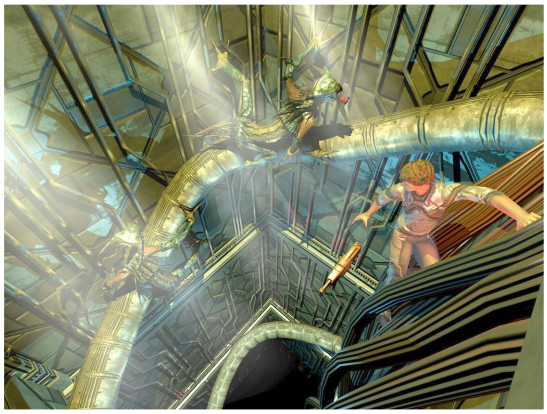
<<

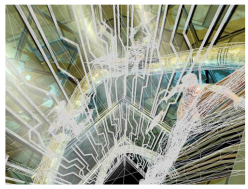


Visible geometry

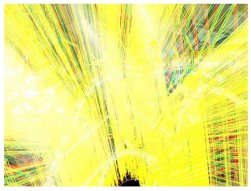
Shadow volume geometry

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



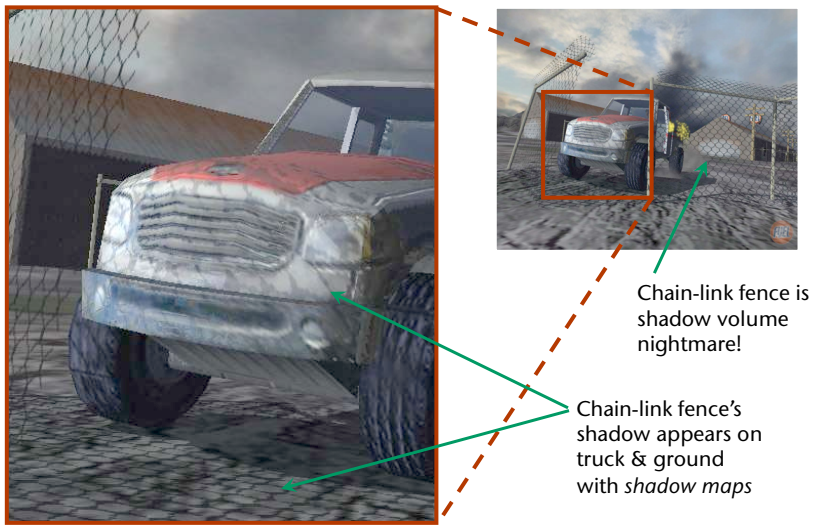


Visible geometry



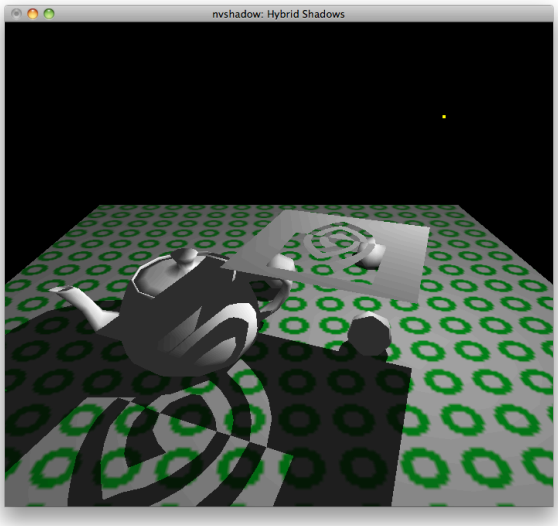
Shadow volume geometry

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/