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:

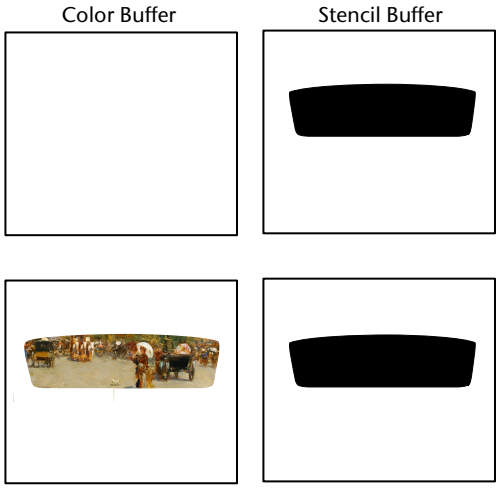
1. **`glStencilFunc(GLenum func, GLint ref, GLuint mask)`**: legt fest, wie und ob in den *Color-Buffer* geschrieben wird (der *Stencil-Test*)
   - Form des Tests: **`s func ref`**
   - Dabei ist s = aktueller Wert im Stencil-Buffer an der Pixelstelle, mask = Maske, ref = ein Referenzwert;
   - Mögliche 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



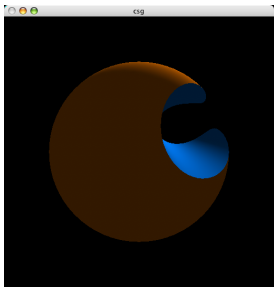

Typisches, einfaches Beispiel

- 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

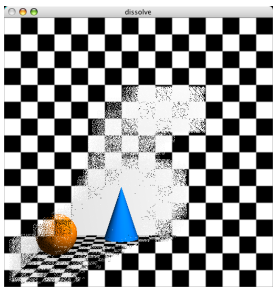


Beispiele für komplexere Operationen/Effekte

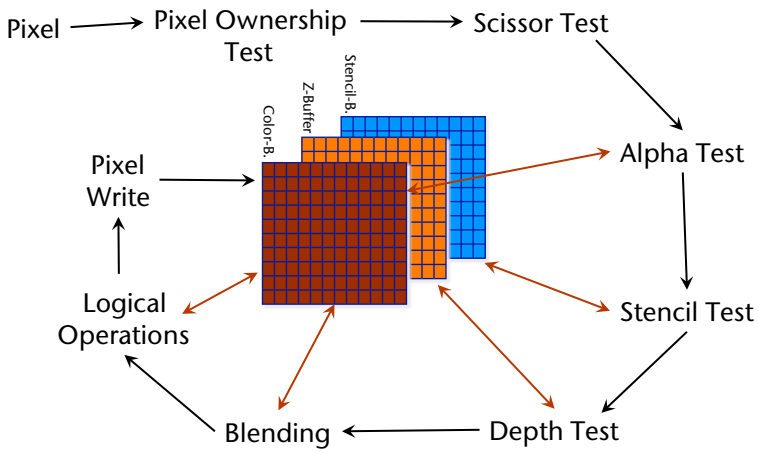
- Beispiel: CSG-Operationen (Schnitt, Differenz, ...)



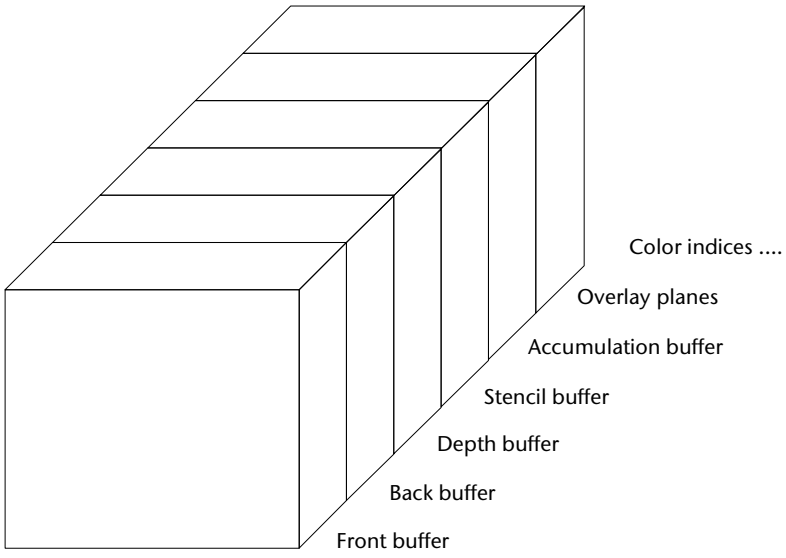
- Beispiel: "Dissolve"



Die Folge von Tests in der Graphik-Pipeline



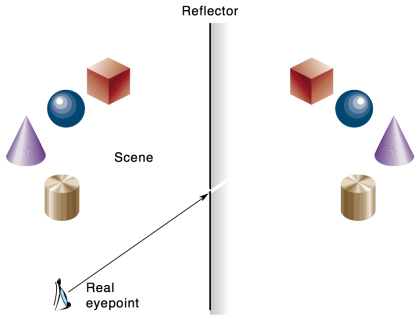
Bemerkung: es gibt viele weitere Buffer in OpenGL



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

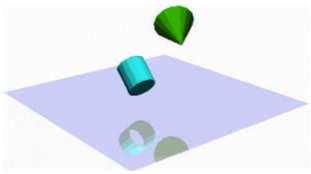


- Dies ist ein Beispiel für einen multi-pass Rendering-Algo
- Achtung: rendere in Schritt 2 nur Polygone *hinter* der Spiegelebene (mittels Clipping-Plane in Spiegelebene; → später)

- 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);
      ```
      ```

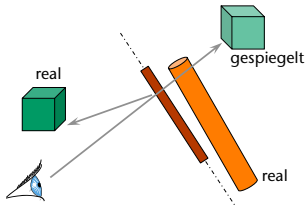


- Disable drawing into the color buffer
  - `glColorMask(0, 0, 0, 0)`
- Draw the geometry of the mirror, with blending if desired
  - This sets stencil bits & fills z buffer with depth value of mirror geometry
- Clear color buffer outside mirror geometry:
  - Configure the stencil test to pass outside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

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

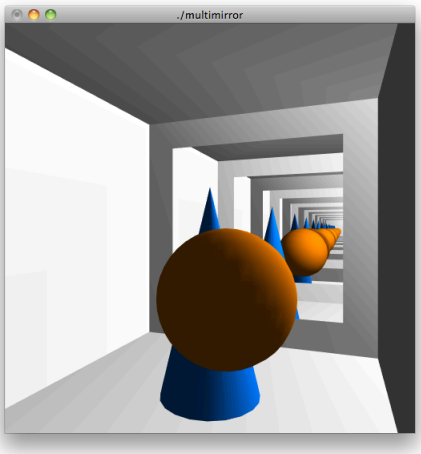
  - Clear color buffer, so that pixels outside the mirror return to the background color: `glClear(GL_COLOR_BUFFER_BIT)`

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



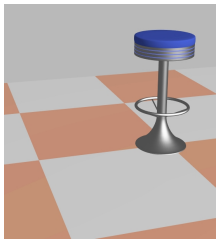
Demo

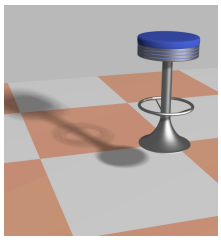
- Das Ganze kann man rekursiv machen:

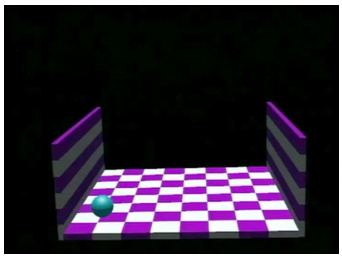


Schatten

- 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



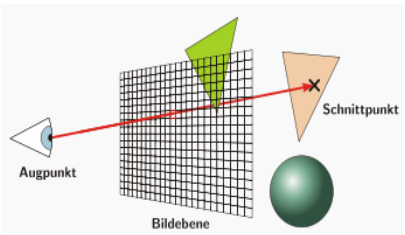


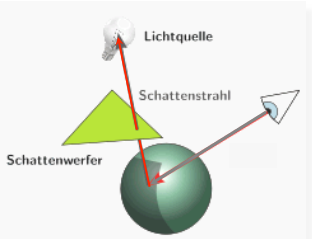


Die Trajektorie des Balls im Bild ist in beiden Video-Segmenten genau dieselbe!

Rendering Shadows using Shadow Volumes

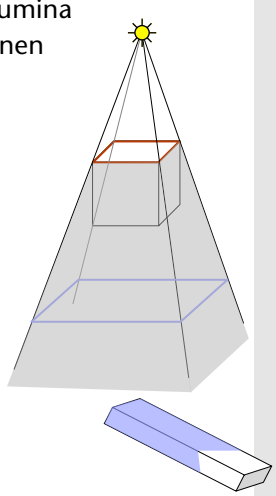
- Zusammenhang zwischen Visibility und Shadows:
  - Visibilitätsberechnung = welche Objekte sind vom Betrachter aus sichtbar
  - Schattenberechnung = welche Objekte sind von der Lichtquelle aus sichtbar



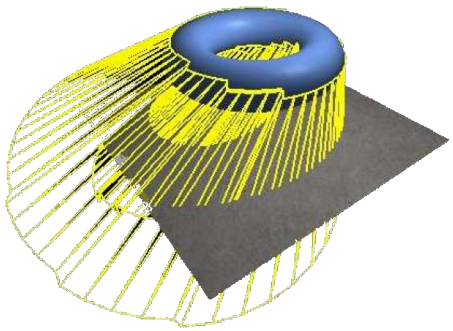


Das Schattenvolumen

- Ansatz im Folgenden: modelliere die (Teil-)Volumina des Universums, die kein Licht von der gegebenen Lichtquelle erhalten
- Das Schattenvolumen:
  - 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*"

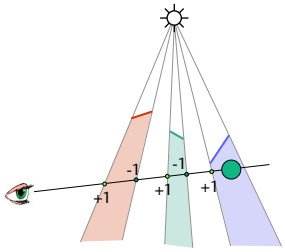


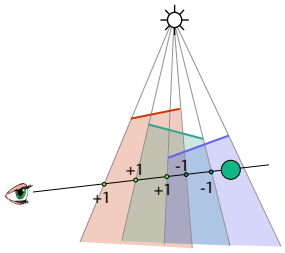
- Beispiel für ein komplexeres Shadow Volume:



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 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 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

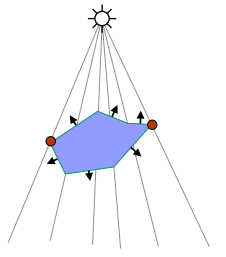
```
glEnable(GL_LIGHTING);                   // switch off light source 0
glDisable(GL_LIGHT0);                    //    but keep all others
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_EQUAL);                   // must match from 1st step
glDepthMask(0);                          // no need to update z buffer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

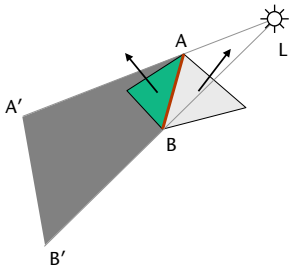
- Dieser Algorithmus heißt "z-pass algorithm", weil in Pass 2 nur Shadow-Volume-Pixel den Stencil-Wert verändern, die den Z-Test passieren

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)



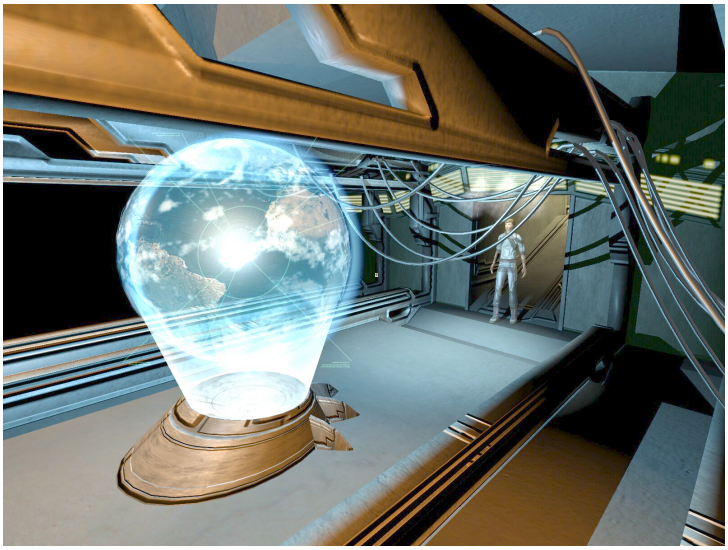


Beispiele

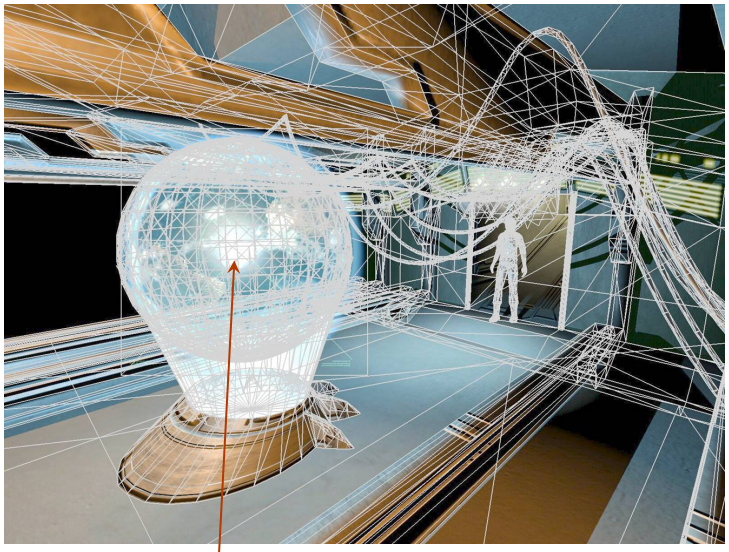
Shadowed scene

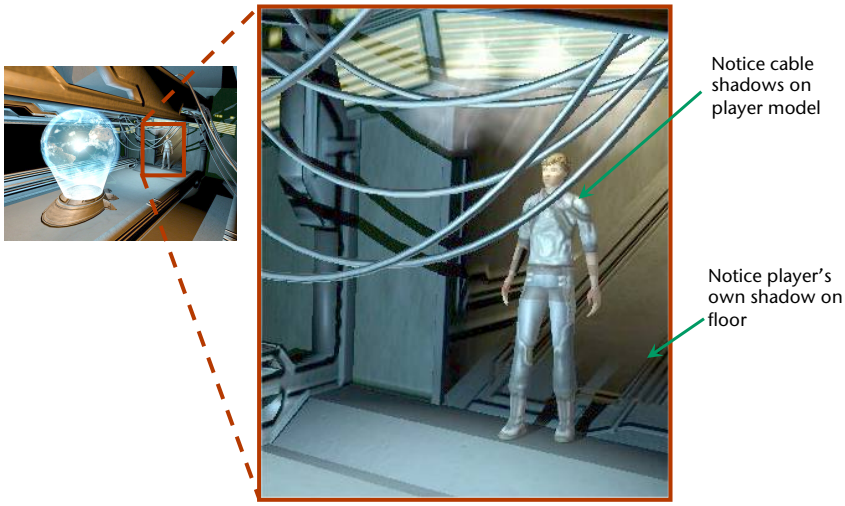
Stencil buffer contents

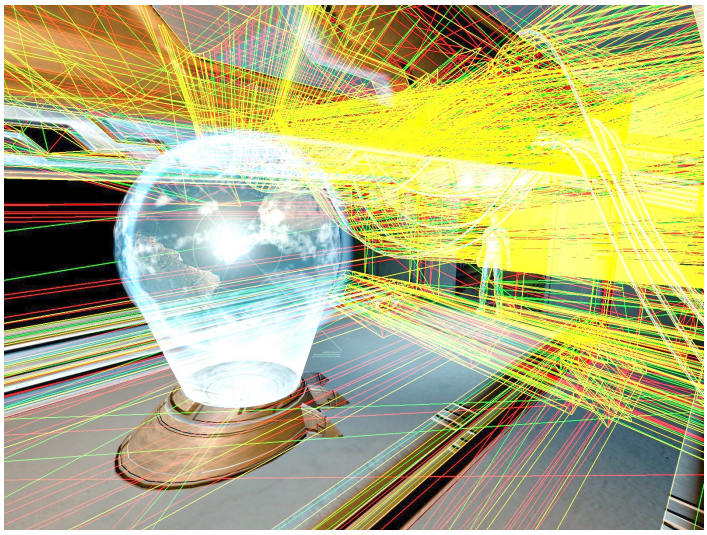
green = stencil value of 0
red = stencil value of 1
darker reds = stencil value > 1



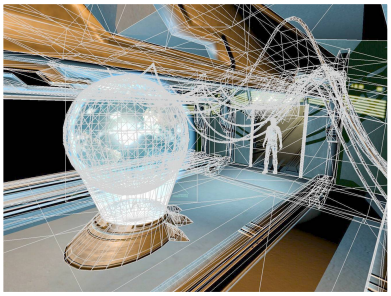
Abducted game images courtesy Joe Riedel at Contraband Entertainment


Primary light source location
Wireframe shows geometric complexity of visible geometry

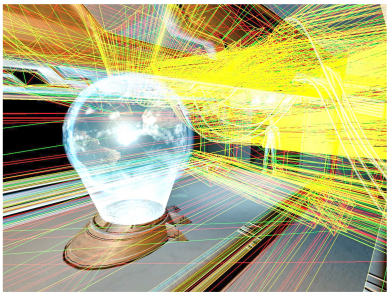

Notice cable shadows on player model
Notice player's own shadow on floor



Wireframe shows geometric *complexity of shadow volume* geometry



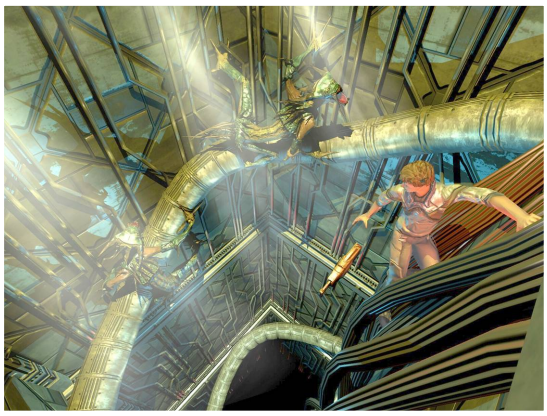
<<



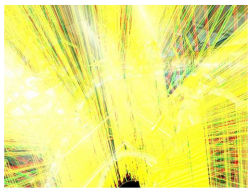
Visible geometry

Shadow volume geometry

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



Visible geometry



Shadow volume geometry

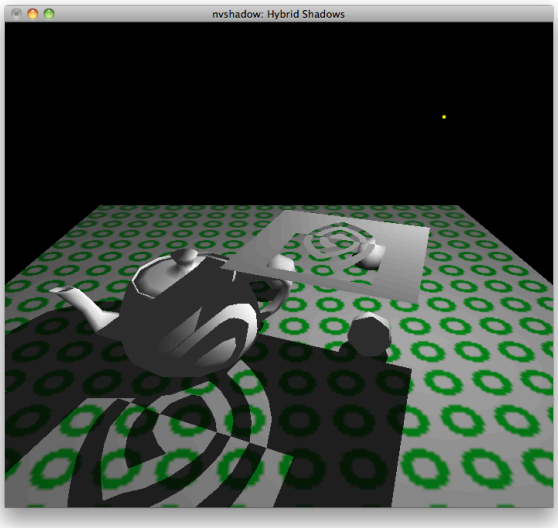
Situations When Shadow Volumes Are Too Expensive

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

Demos

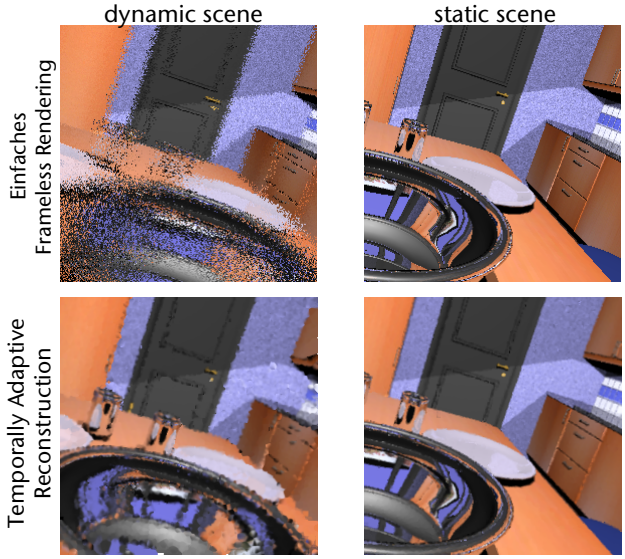


http://www.opengl.org/resources/features/StencilTalk/

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