Interaktive Computergrafik

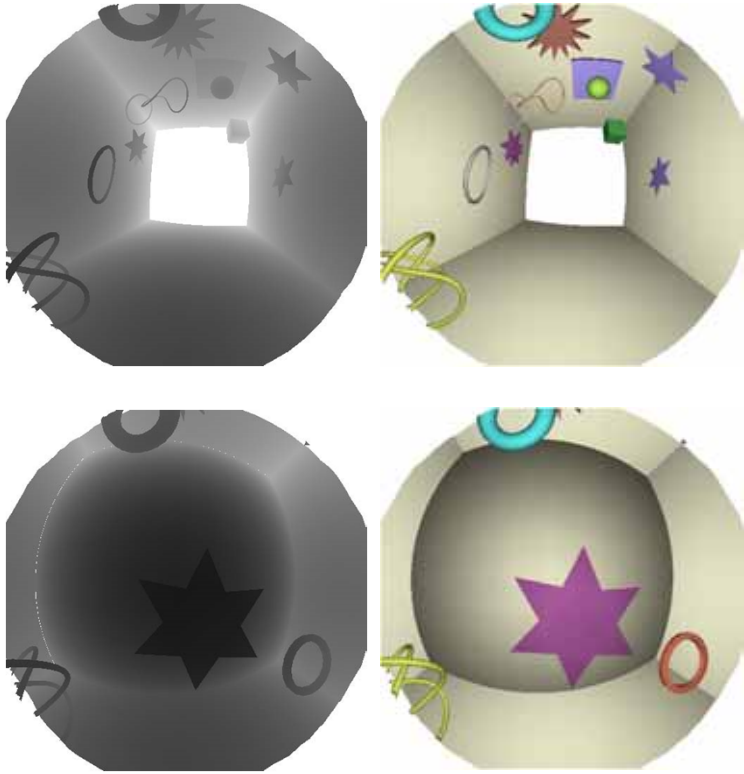
Vorlesung im Sommersemester 2014
Kapitel 2: (Echtzeit-)Schattenverfahren (Teil 3)

Prof. Dr.-Ing. Carsten Dachsbacher
Lehrstuhl für Computergrafik
Karlsruher Institut für Technologie

Lichtquellen und Shadow Maps

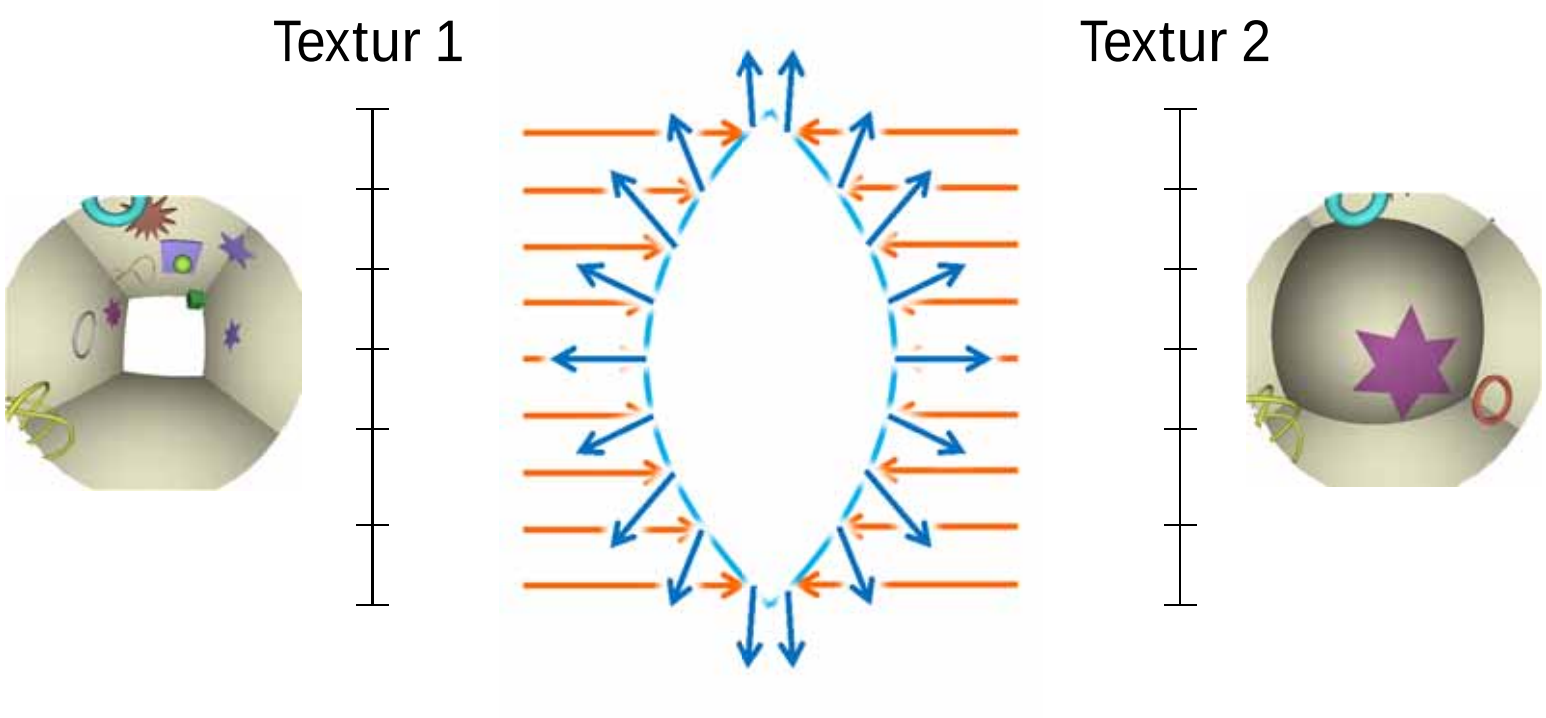
(Omnidirektionale) Punktlichtquellen

- Dual-Paraboloid Shadow Maps:
  - verwende zwei hemisphärische (nicht-lineare) Projektionen
  - Anm. kann ebenfalls für Environment Mapping verwendet werden



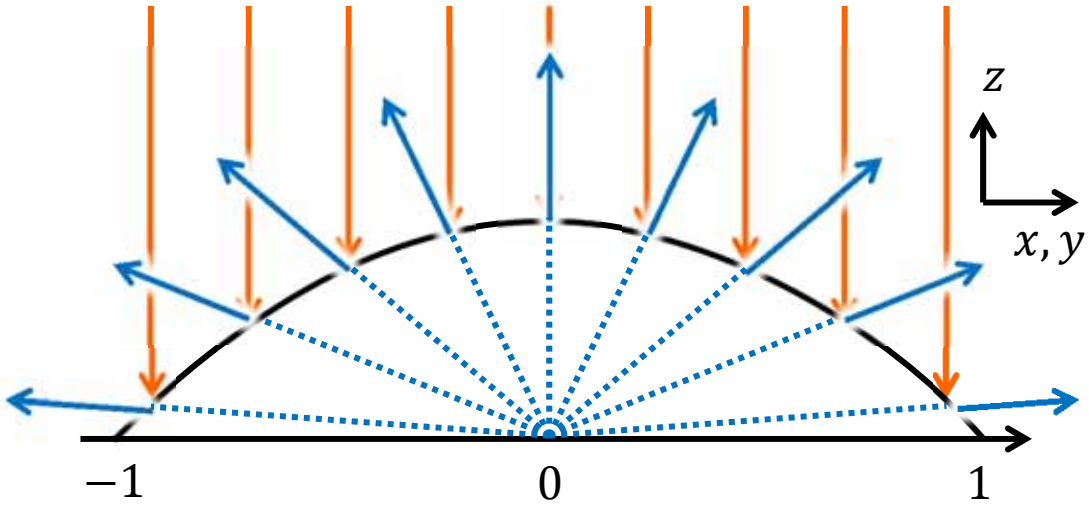
Parabolische Environment/Shadow Maps

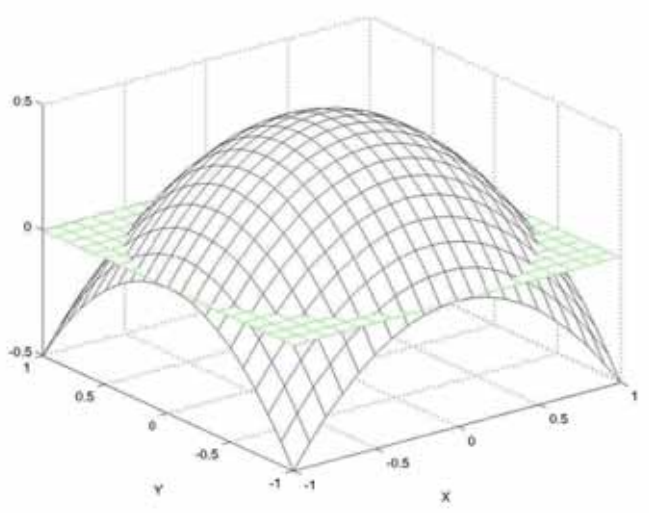
- erinnern wir uns nochmal an Environment Mapping…
- ähnlich wie Sphere-Mapping betrachten wir eine Spiegelung an einem virtuellen Objekt, jetzt aber an zwei Paraboloiden [Heidrich98]
- wir stellen uns vor jeden Spiegel mit einer orthogonalen Kamera zu „betrachten“, also mit parallelen Primärstrahlen
- die Reflexionsstrahlen decken jeweils eine Hemisphäre ab



Parabolische Environment/Shadow Maps

- der Paraboloidspiegel wird beschrieben durch $f(x,y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2)$
  - gemäß dieser Konvention: Einfallsrichtung entlang der z-Achse
  - alle folgenden Betrachtungen in diesem Koordinatensystem!
  - die Funktion ist geschickt gewählt: die Reflexionsstrahlen decken die ganze Hemisphäre ab und besitzen den gemeinsamen Ursprung bei $(0,0)$





Parabolische Environment/Shadow Maps

Berechnung der Normale des Paraboloid

- als Paraboloidfunktion wird gewählt $f(x,y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2)$
  - die Fläche des Spiegels ist $P(x,y) = \big(x, y, f(x,y)\big)^T$
  - Normale (hier $|\boldsymbol{n_{xy}}| \neq 1$) berechnet über partielle Ableitungen:

$$\frac{\partial P}{\partial x} = \left(1,0,\frac{\partial f(x,y)}{\partial x}\right)^T = (1,0,-x)^T$$

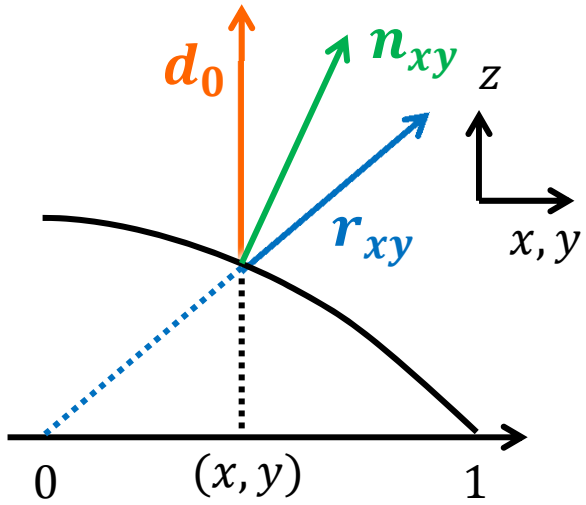
$$\frac{\partial P}{\partial y} = \left(0,1,\frac{\partial f(x,y)}{\partial y}\right)^T = (0,1,-y)^T$$

$$\boldsymbol{n_{xy}} = \frac{\partial P}{\partial x} \times \frac{\partial P}{\partial y} = (x,y,1)^T$$

Parabolische Environment/Shadow Maps

Zusammenhang zwischen Normale und (Reflexions-)Richtung

- „gesucht“: Ausfallsrichtung $\boldsymbol{r_{xy}}$ an der Stelle (x, y)
  - diese Richtung kennen wir eigentlich schon: $\frac{P(x,y)}{|P(x,y)|}$
  - Zweck dieser Herleitung: Abbildung von $\boldsymbol{r}$ auf (x, y)
- für die Reflexion gilt allgemein: $\boldsymbol{r_{xy}} = 2\langle \boldsymbol{n_{xy}}, \boldsymbol{d_0} \rangle \boldsymbol{n_{xy}} - \boldsymbol{d_0}$
  - mit der „Aufnahmerichtung“ $\boldsymbol{d_0} = (0,0,1)^T$ und
  - $\boldsymbol{n_{xy}} = \frac{1}{\sqrt{x^2+y^2+1}} (x, y, 1)^T$



Parabolische Environment/Shadow Maps

Berechnung der Koordinate für eine geg. (Reflexions-)Richtung

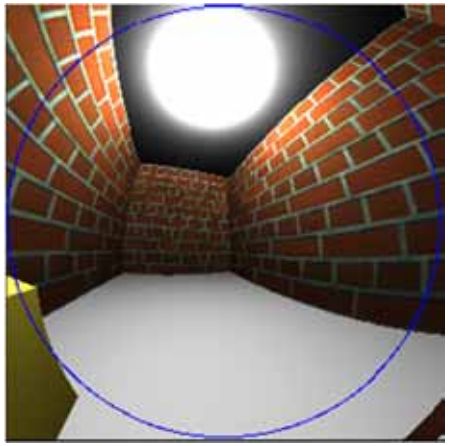
- um festzustellen welcher Koordinate in der Paraboloid-Map eine geg. Richtung $\boldsymbol{r}$ ($|\boldsymbol{r}| = 1$) zugewiesen ist nutzt man diesen Zusammenhang:
  - $\boldsymbol{r} = 2\langle \boldsymbol{n_{xy}}, \boldsymbol{d_0} \rangle \boldsymbol{n_{xy}} - \boldsymbol{d_0}$
    $\Rightarrow \boldsymbol{r} + \boldsymbol{d_0} = 2\langle \boldsymbol{n_{xy}}, \boldsymbol{d_0} \rangle \boldsymbol{n_{xy}}$
- wir sehen, was wir eigentlich bereits wissen:
  die Normalenrichtung $\boldsymbol{n_{xy}}$ liegt genau zw. $\boldsymbol{r}$ und $\boldsymbol{d_0}$
  - d.h. $\boldsymbol{r} + \boldsymbol{d_0}$ ist ein k-faches der nicht-normalisierten Normale bei (x, y)
  - $\boldsymbol{r} + \boldsymbol{d_0} = (r_x, r_y, 1 + r_z) = k(x, y, 1)^T$ mit $k \in \mathbb{R}$
- wir können die Lösung direkt ablesen:
  - $k = 1 + r_z$
  - $\Rightarrow x = \frac{r_x}{1+r_z} \wedge y = \frac{r_y}{1+r_z}$ mit $x, y \in [-1; 1]$
  - die Texturkoordinaten für den Zugriff erhält man durch Abbildung auf das Intervall $[0; 1]$

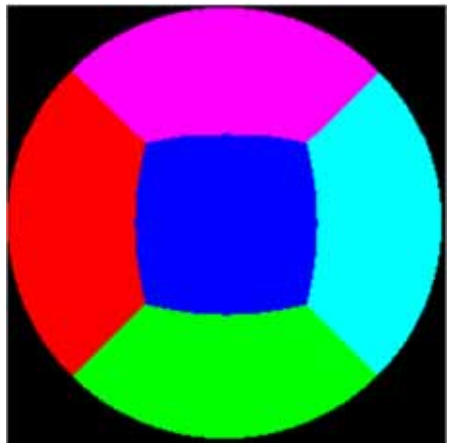
Parabolische Environment/Shadow Maps

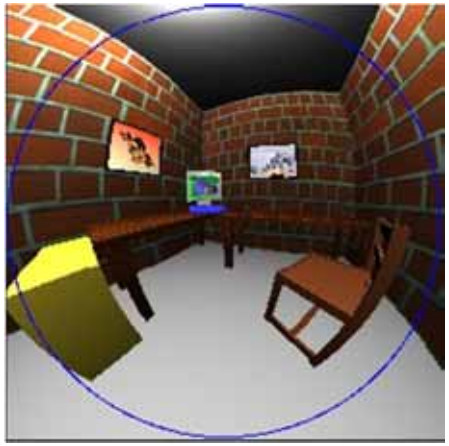
- wir können jetzt eine Richtung auf eine Paraboloid-Koordinate abbilden
- damit ist es möglich eine Paraboloid-Map zu rendern
  - Vertex-Position in lokalen Koordinaten (bzgl. des Paraboloiden) P
  - Position der Lichtquelle (bzw. „Paraboloid-Kamera“) L
  - Richtung $\boldsymbol{r}$ zum Vertex $\boldsymbol{r} = (P - L)/|P - L|$
  - Vertex-Transformation mit der Matrix $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{pmatrix} r_x \\ r_y \\ r_z \\ 1 \end{pmatrix}$
  - zweiter Halbraum: analog
- Achtung: es handelt sich um eine nicht-lineare Abbildung
  - Linien werden auf Kurven abgebildet!
  - Approximationsfehler bei grober Tessellierung
- Nachschlagen in Paraboloid-Maps mit derselben Berechnung (Fallunterscheidung nach Halbraum)

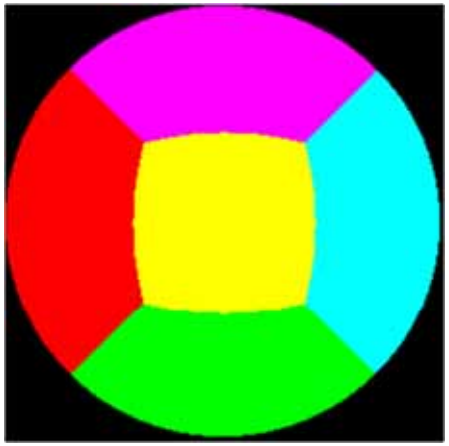
Parabolische Environment/Shadow Maps

- effektiv genutzter Texturraum $\frac{\pi}{4} < 80\%$

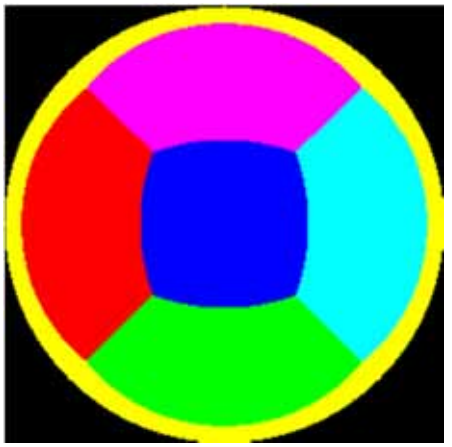








zum Vergleich:
sphärische EnvMap



- nur 2 Texturen um alle Richtungen abzutasten
- relativ geringe Verzerrungen

Aliasing und Shadow Mapping

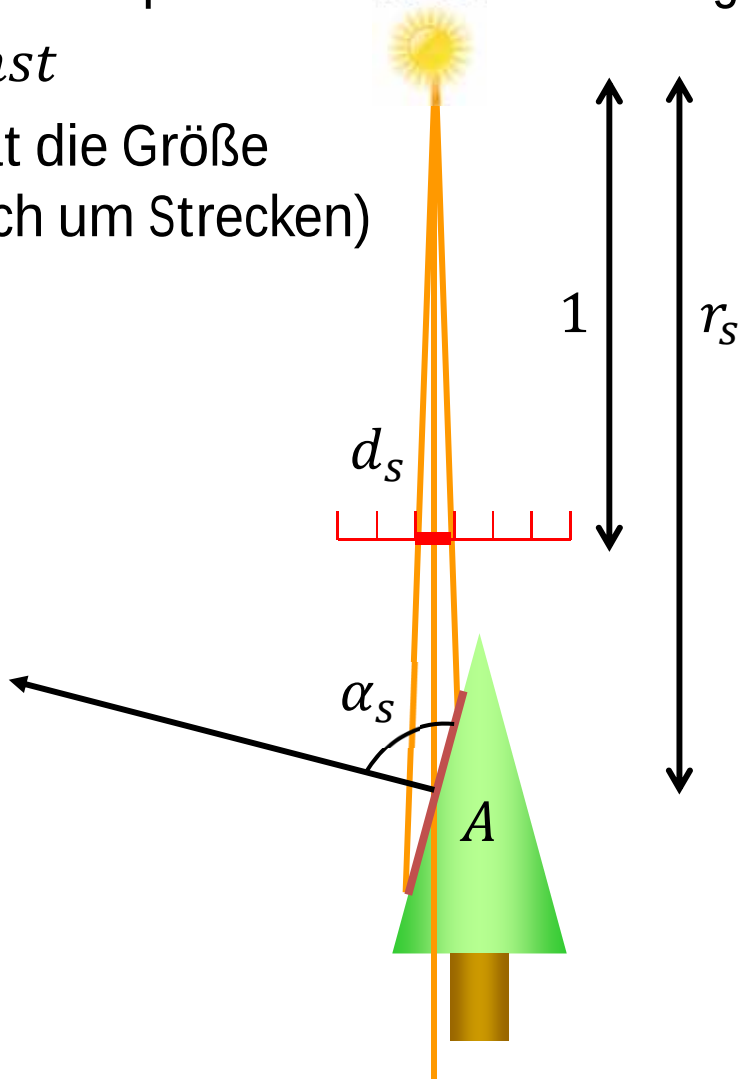
(Zwischen-)Fazit

- jeder Lichtquellentyp erfordert eine bestimmte Projektion/Abbildung
- Shadow Maps haben eine endliche Auflösung
  - Aliasing reduzieren wir mit PCF, stochastischem Filter oder VSM
  - Surface Acne benötigt einen Tiefenoffset (Bias)
  - es gibt günstige (Miner‘s Lamp) und ungünstige Konstellationen
    - kann man Letztere vermeiden? ein Ausblick…

Aliasing und Shadow Mapping

Analyse der beteiligten Größen

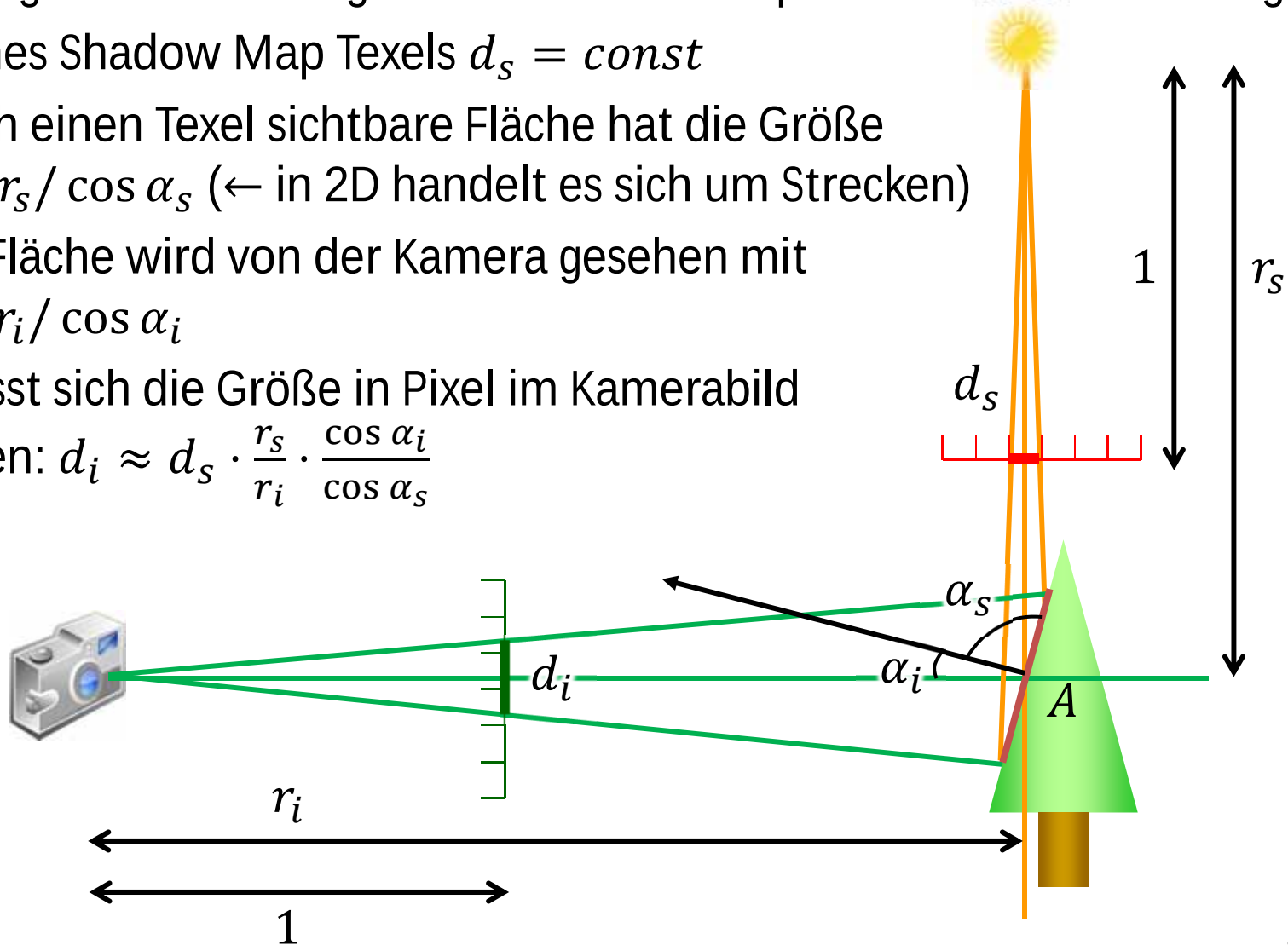
- wir haben gesehen: Aliasing entsteht durch Perspektive und Orientierung
- Größe eines Shadow Map Texels $d_s = const$
- eine durch einen Texel sichtbare Fläche hat die Größe
  $A \approx d_s \cdot r_s / \cos \alpha_s$ (← in 2D handelt es sich um Strecken)



Aliasing und Shadow Mapping

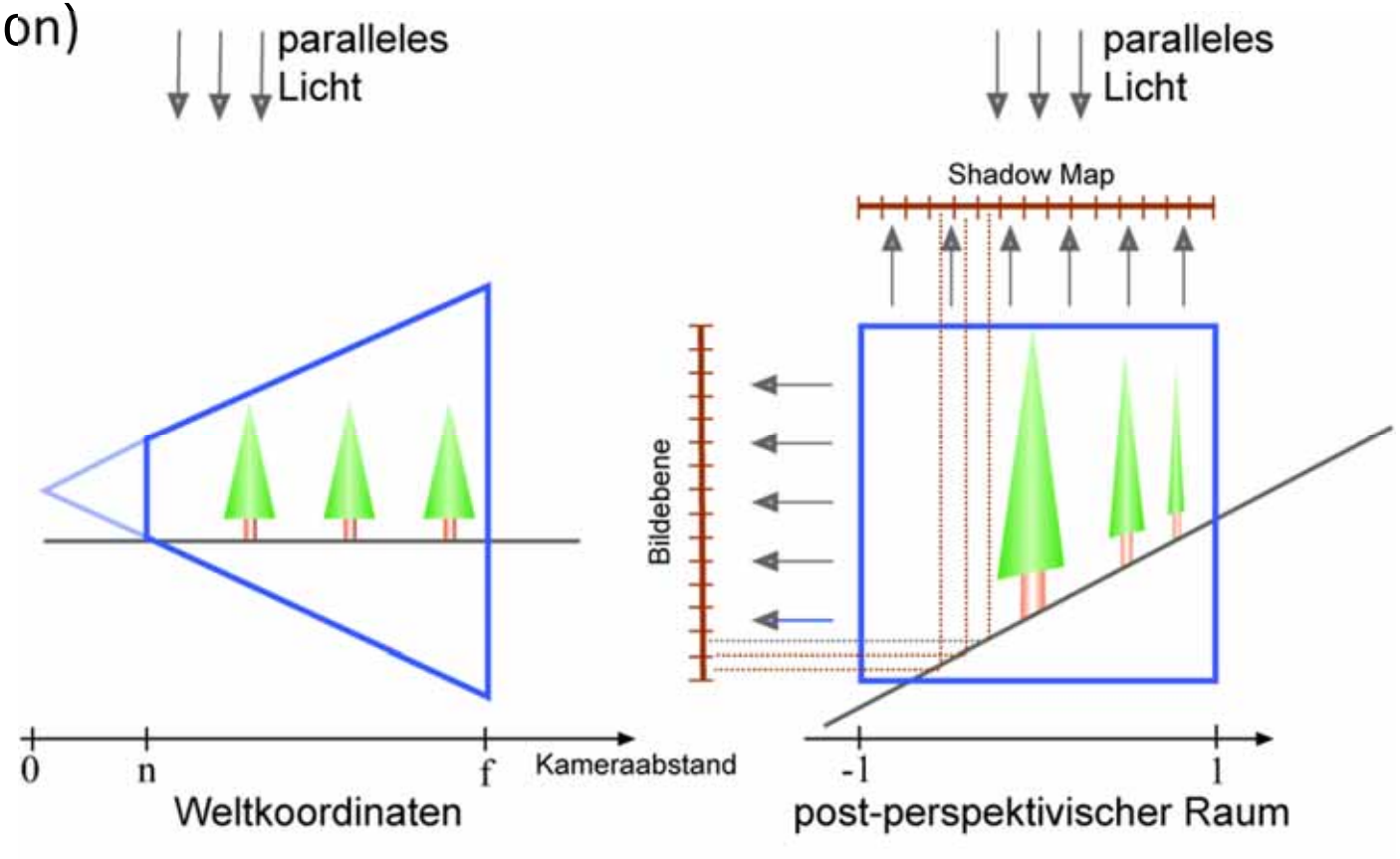
Analyse der beteiligten Größen

- wir haben gesehen: Aliasing entsteht durch Perspektive und Orientierung
- Größe eines Shadow Map Texels $d_s = const$
- eine durch einen Texel sichtbare Fläche hat die Größe
  $A \approx d_s \cdot r_s / \cos \alpha_s$ (← in 2D handelt es sich um Strecken)
- dieselbe Fläche wird von der Kamera gesehen mit
  $A \approx d_i \cdot r_i / \cos \alpha_i$
- daraus lässt sich die Größe in Pixel im Kamerabild bestimmen: $d_i \approx d_s \cdot \frac{r_s}{r_i} \cdot \frac{\cos \alpha_i}{\cos \alpha_s}$



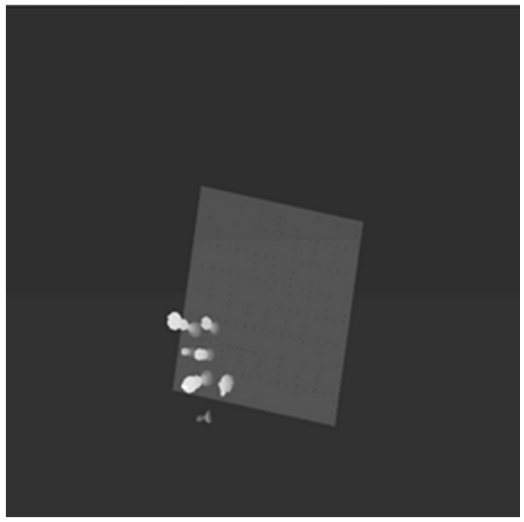
Perspective Shadow Maps (PSMs)

- Ziel der sog. Perspective Shadow Maps ist die Reduzierung von Aliasing aufgrund der Perspektive, d.h. des $\frac{r_s}{r_i}$ Terms
- Idee: wende zuerst die perspektivische Transformation der Kamera auf die Szene an und berechne danach die Shadow Map
  - in diesem Beispiel spielt r_s aufgrund der parallelen LQ keine Rolle (r_i schon)



Perspective Shadow Maps (PSMs)

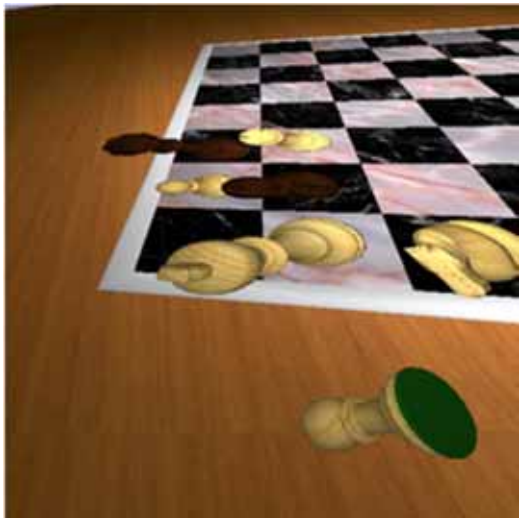
normale Shadow Maps



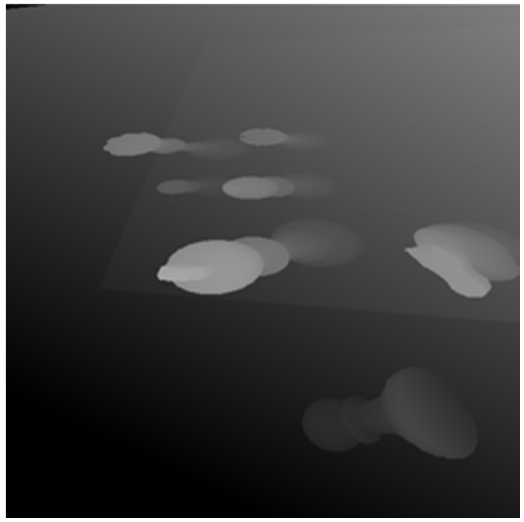
Szene von Lichtquelle

Shadow Map

Szene von Kamera

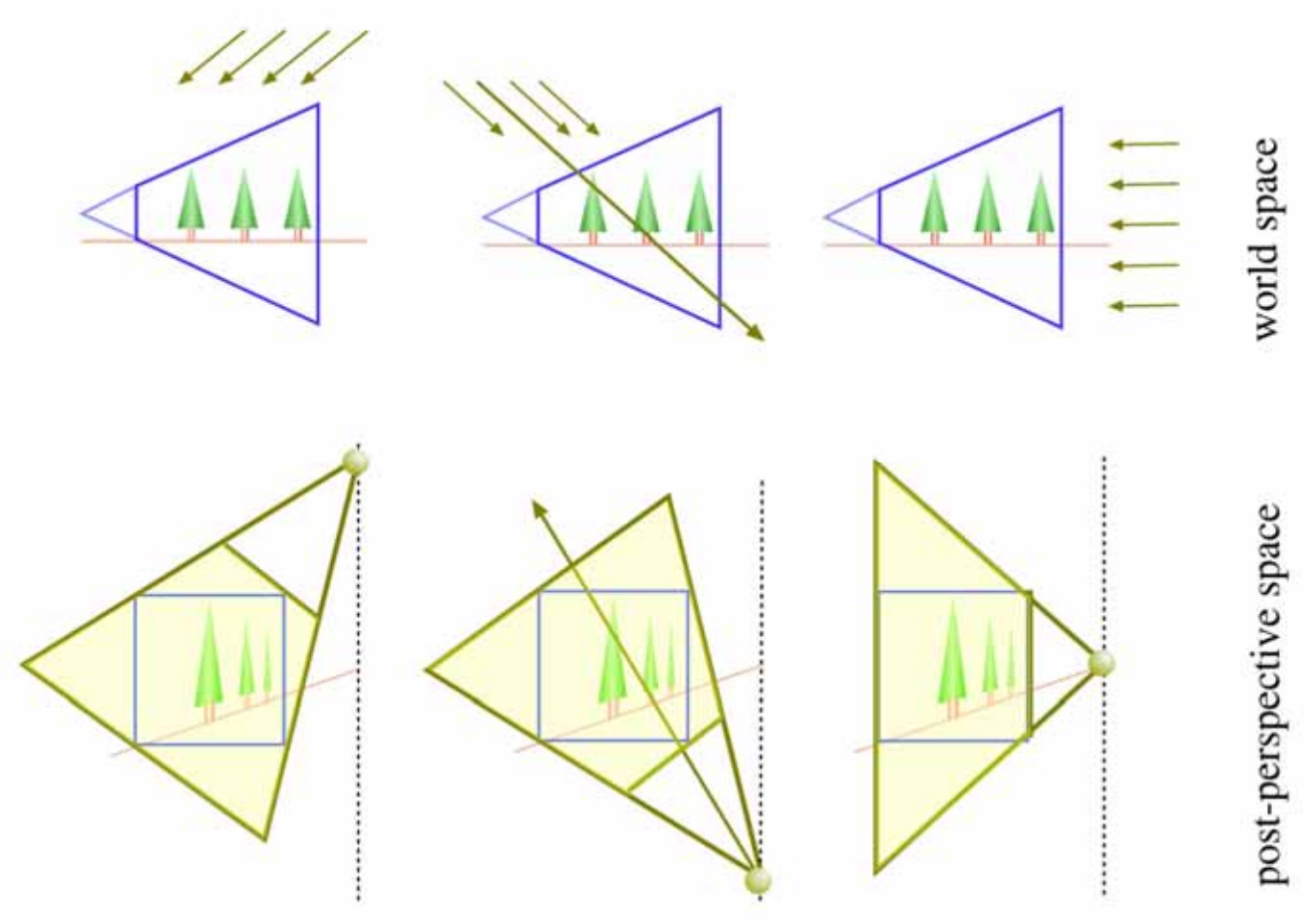


perspektivische Shadow Maps



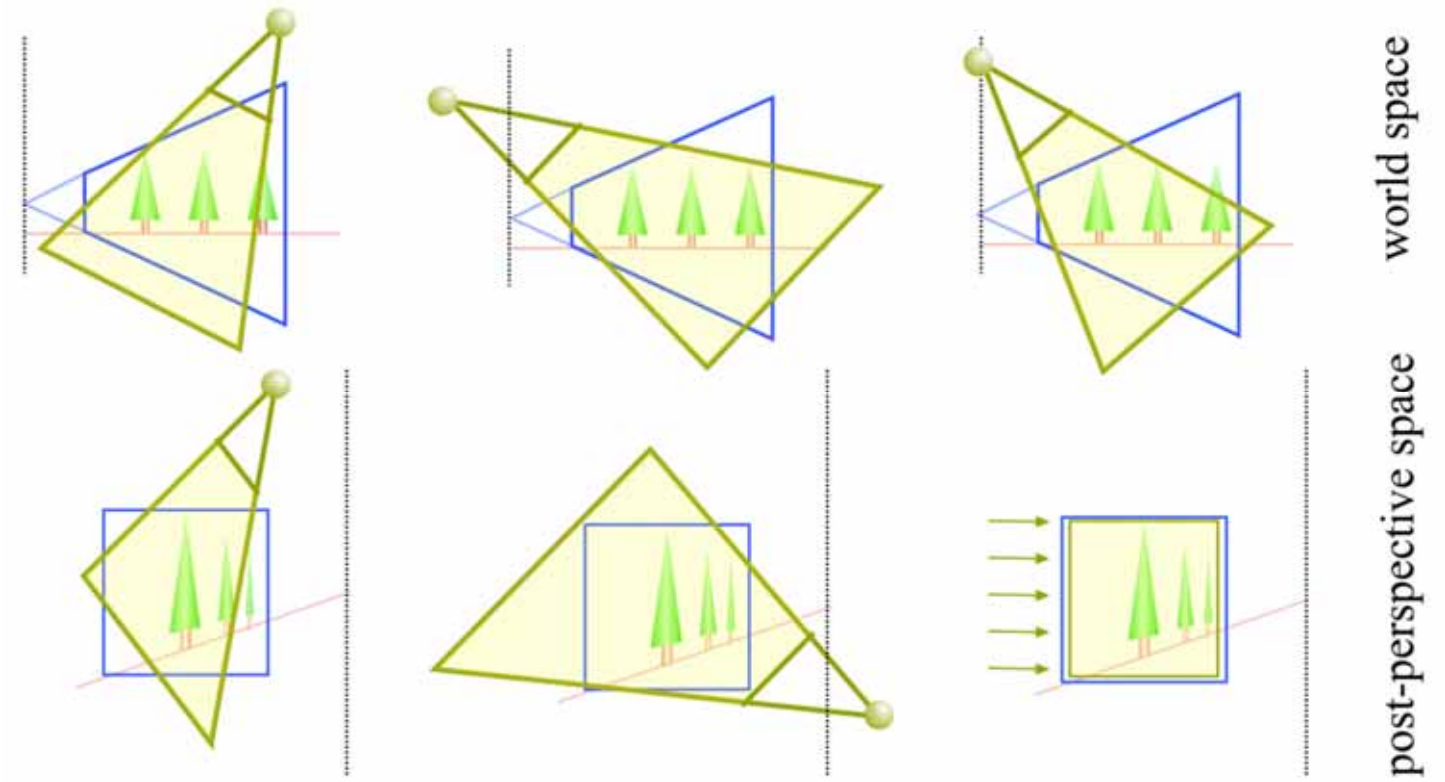
Perspective Shadow Maps (PSMs)

- Eigenschaften projektiver Abbildungen erfordern Fallunterscheidungen
  - parallele Geraden bleiben i.A. nicht parallel
  - die Lichtstrahlen einer Parallellichtquelle werden i.A. zu einer Punktlichtquelle im post-perspektivischen Raum



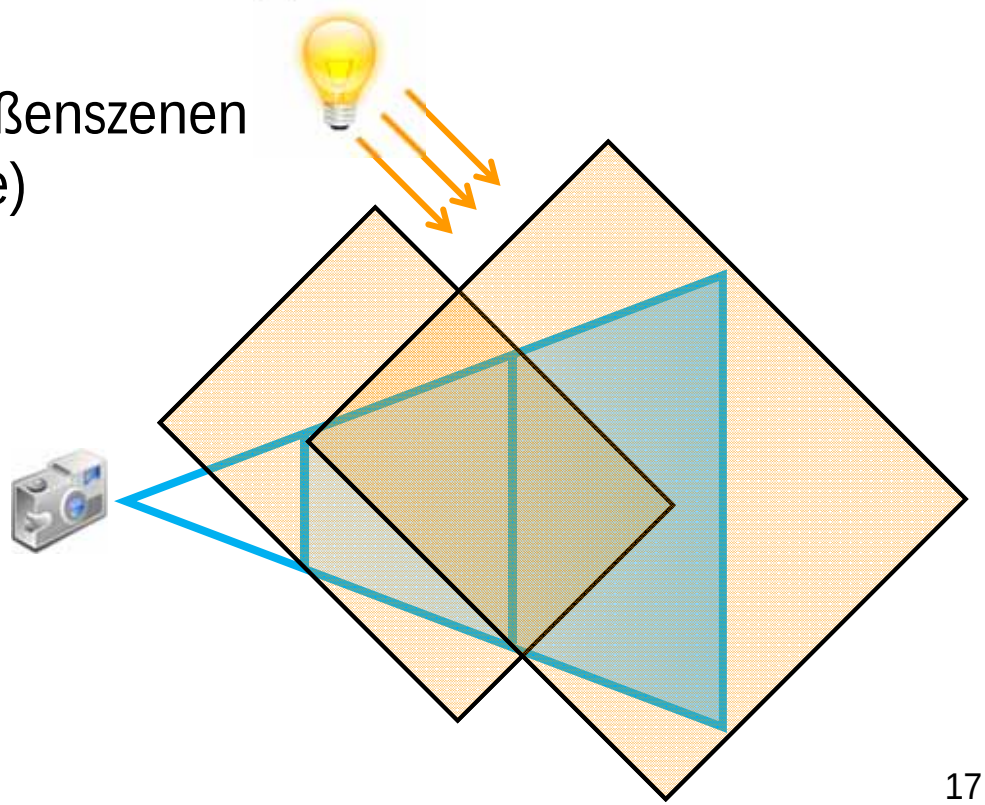
Perspective Shadow Maps (PSMs)

- Eigenschaften projektiver Abbildungen erfordern Fallunterscheidungen
  - parallele Geraden bleiben i.A. nicht parallel
  - Punktlichtquellen können zu parallelen Lichtquellen im post-perspektivischen Raum werden



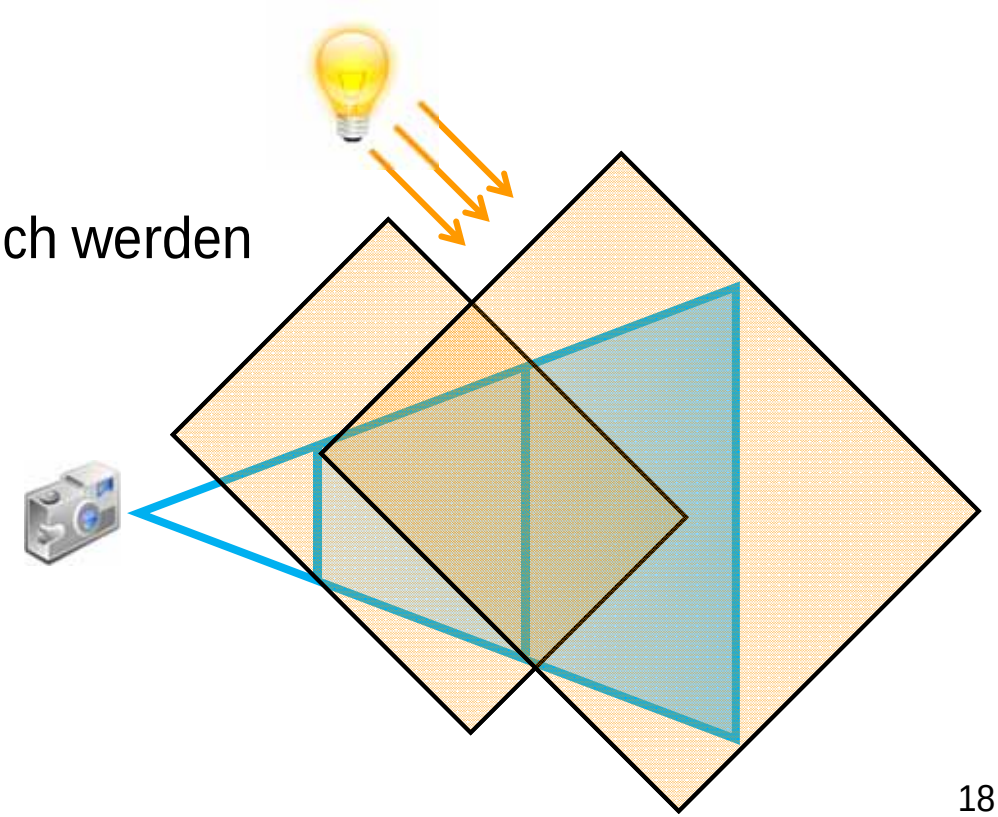
Cascaded Shadow Maps

- Perspective Shadow Maps (und davon abgeleitete Verfahren) sind nicht ganz einfach in den Griff zu kriegen: Spezialfälle, noch schwierigere Parameterwahl (z.B. Bias) usw.
- Alternative zu PSMs
  - berechne mehrere (hier 2) Shadow Maps mit gleicher Auflösung
  - eine SM ist für den vorderen Teil der Sichtpyramide, die andere für den hinteren Teil bestimmt
- typischerweise verwendet für Außenszenen (Sonnenlicht, parallele Lichtquelle)



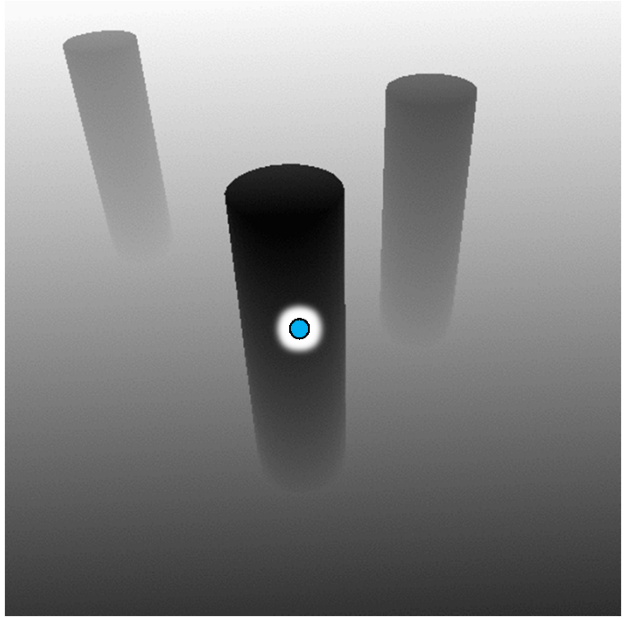
Cascaded Shadow Maps

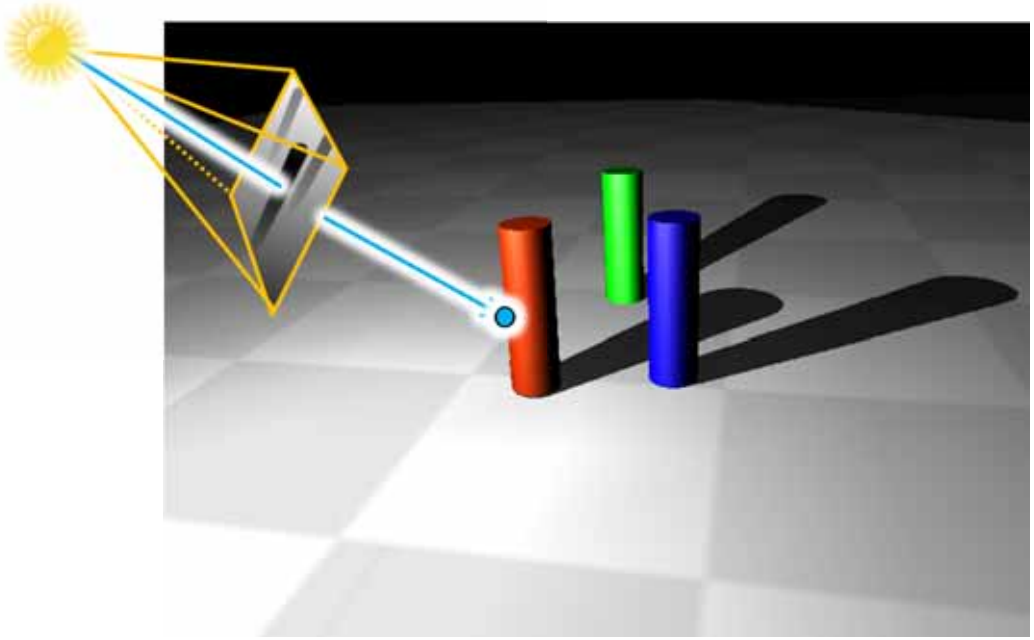
- Achtung:
  - jede SM muss natürlich die schattenwerfenden Objekte erfassen
  - im Überlappungsbereich werden meist die Resultate aus beiden SMs kombiniert
- Vorteile
  - einfache Implementation
  - robust und schnell
- Nachteile
  - Objekte im Überlappungsbereich werden mehrfach gezeichnet



Shadow Maps: Zusammenfassung

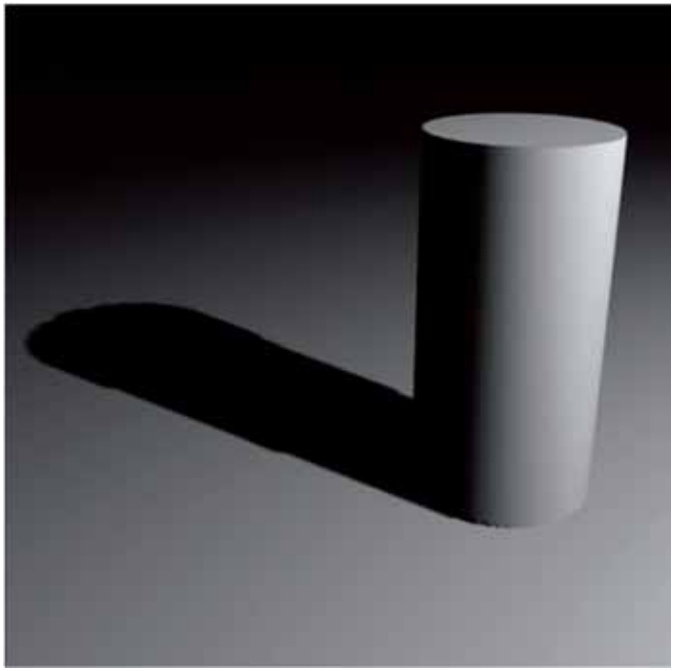
- das Shadow Map-Verfahren ist
  - relativ einfach, schnell und einigermaßen robust (Bias, ...)
  - inhärentes Problem: es handelt sich um eine diskrete Abtastung
  - das meistverwendete Verfahren für Schatten mit Grafik-HW
- sehr viele Erweiterungen und Varianten, z.B.
  - Cascaded Shadow Maps für räumlich große Szenen
  - PSMs und darauf aufbauende Arbeiten
  - Adaptive Shadow Maps für besseres Abtasten, u.v.m.





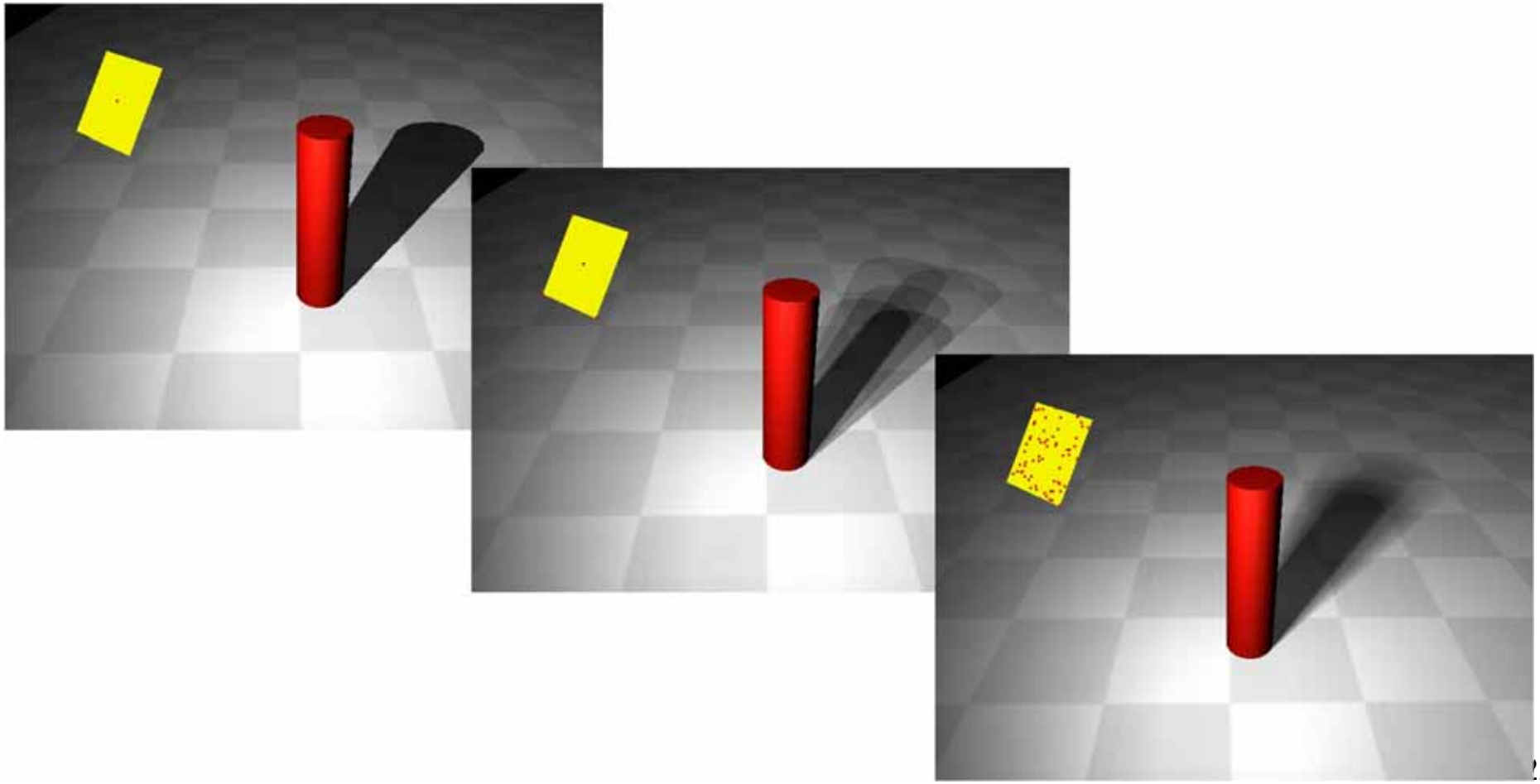
Flächenlichtquellen

- bisher: Verfahren, die Sichtbarkeit von einer Punktlq. aus bestimmen
- reale LQ besitzen eine endliche Größe: weiche Schatten entstehen durch partielle Sichtbarkeit der LQ
  - Achtung: weiche Kanten durch Filterung (z.B. mit VSM) ≠ weiche Schatten aufgrund einer Flächenlichtquelle



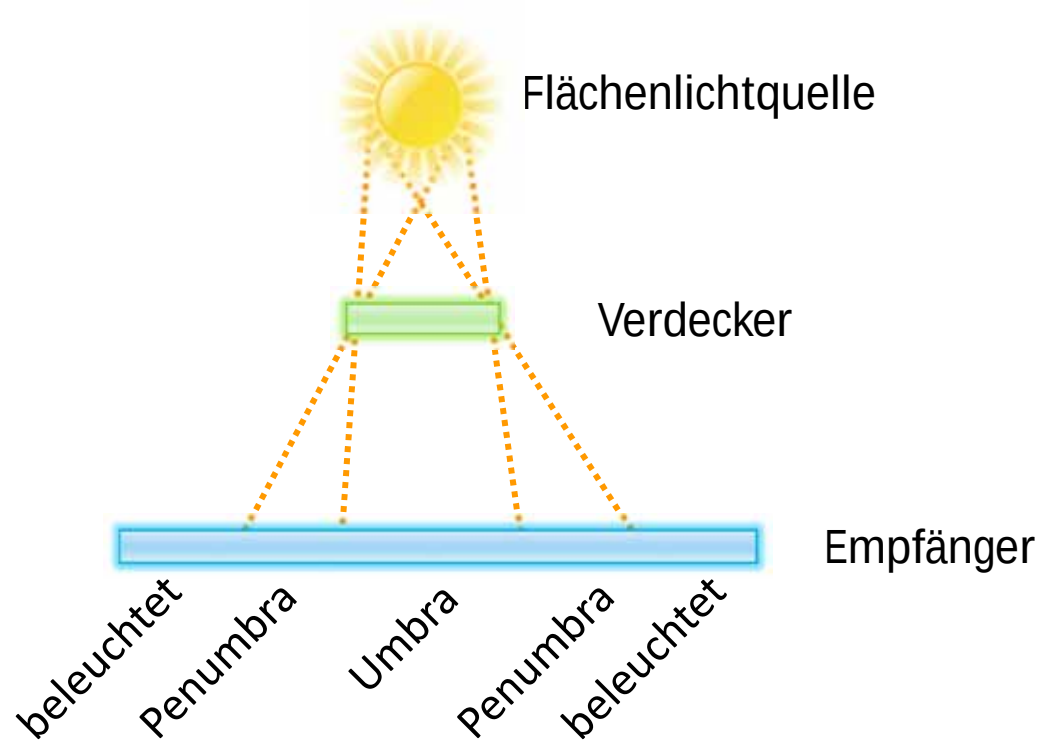
Flächenlichtquellen

- weiche Schatten erhalten wir (einfach, aber langsam) durch
  - Abtasten der Flächenlichtquelle (wie bei MC-Integration)
  - in diesem Kontext: Erzeugen von mehreren Punktlichtquellen mit Shadow Maps/Shadow Volumes und Mittelung der Beiträge



Flächenlichtquellen

- unterscheide 3 Regionen:
  Licht – Halbschatten (Penumbra) – Schatten (Umbra)

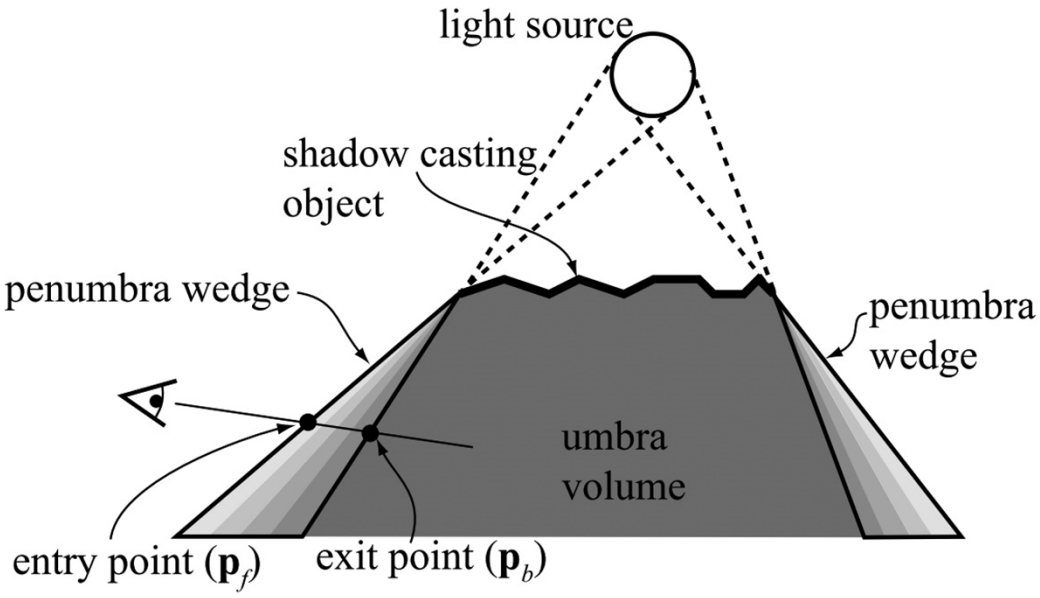


- für große Lichtquellen/kleine Verdecker kann der Vollschatten auch verschwinden

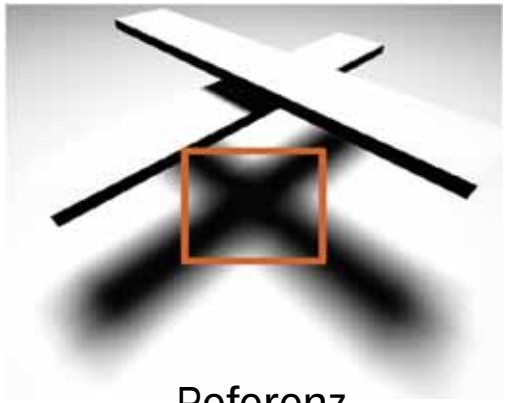
Flächenlichtquellen und Shadow Volumes

Penumbra Wedges

- erzeuge Schattenvolumen die den Halbschattenbereich einschließen
- schätze die Verschattung einer Oberfläche in diesen Volumen ab
- keine exakte Verdeckungsberechnung: die Abschätzung kann zu falschen Resultaten führen



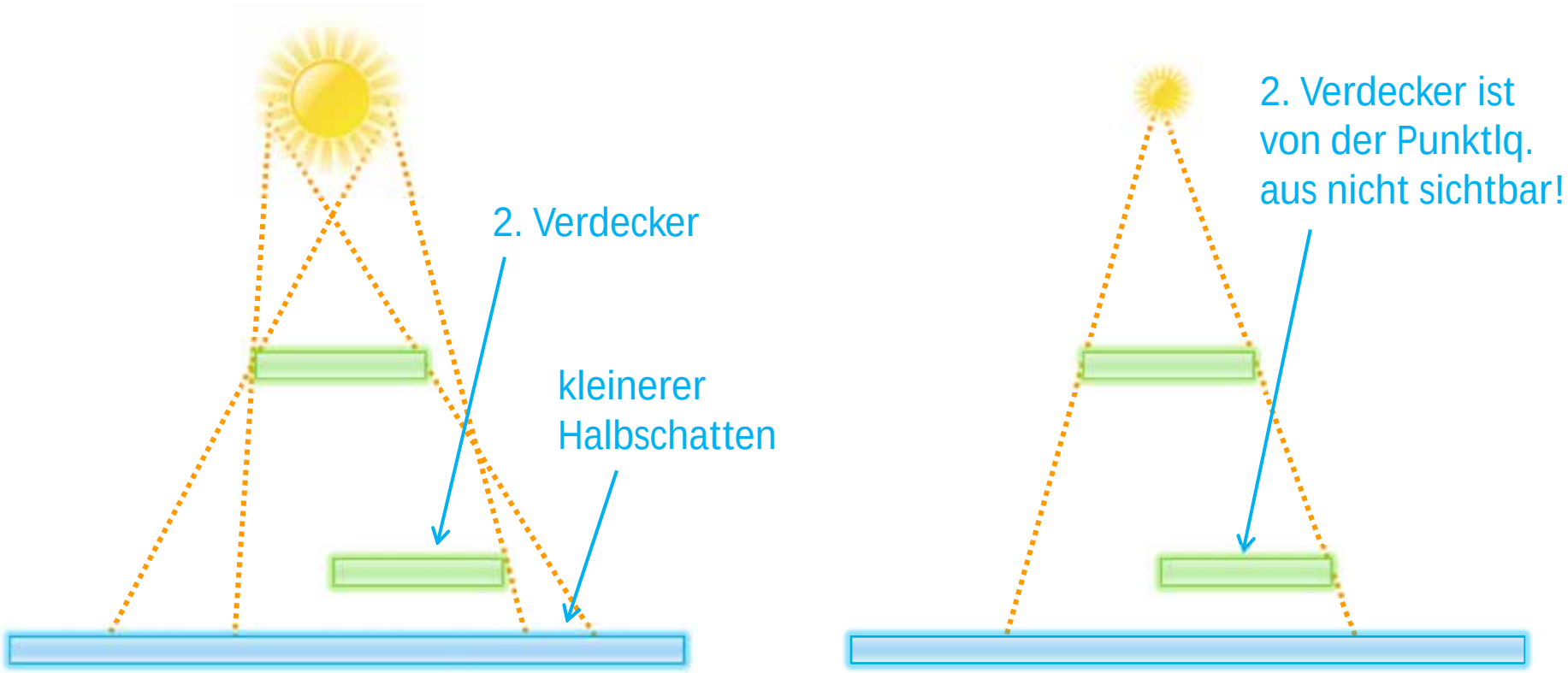
Approximation



Referenz

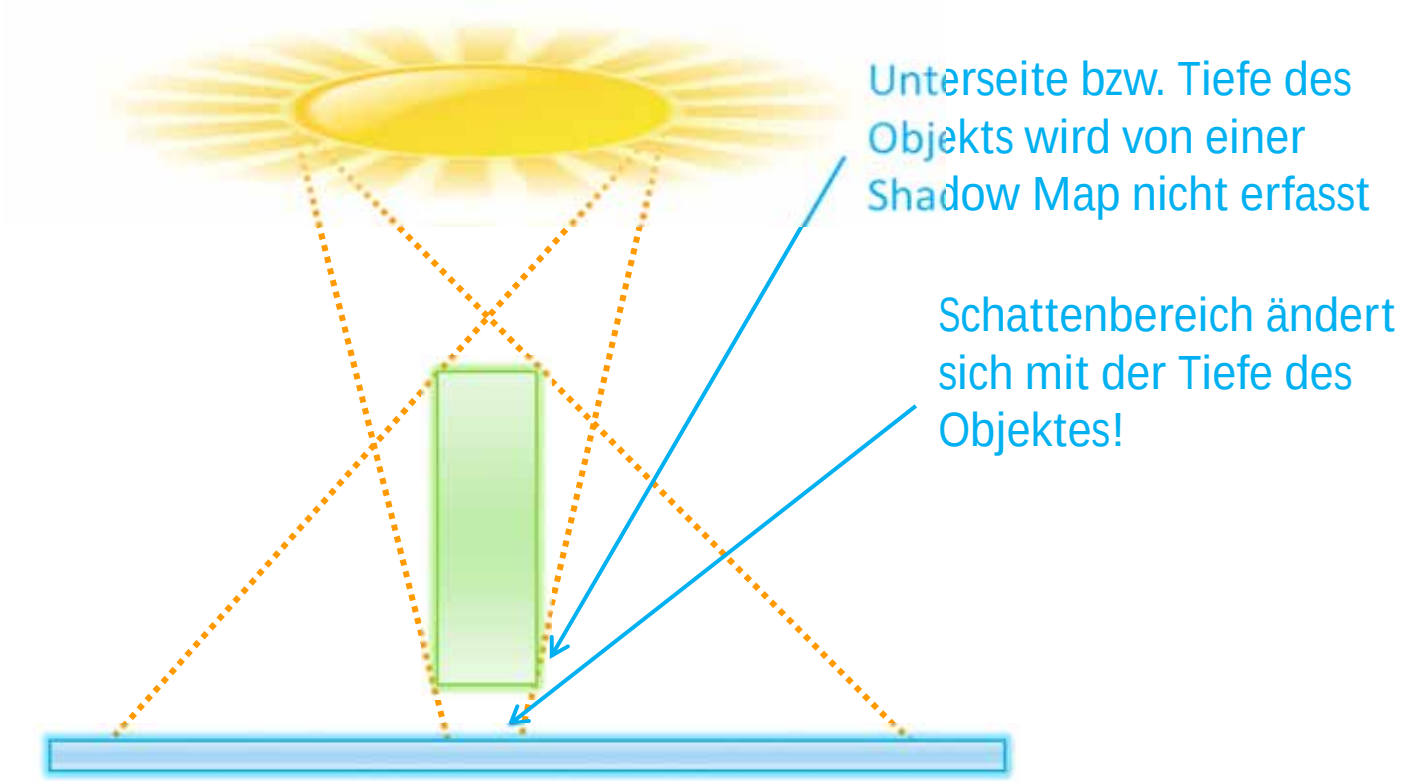
Flächenlichtquellen und Shadow Maps

- viele Ansätze versuchen die Verschattung von Flächenlichtquellen mit nur einer Shadow Map zu approximieren
- es kann nur bei einer Approximation bleiben, denn nicht alle Informationen sind in der SM enthalten



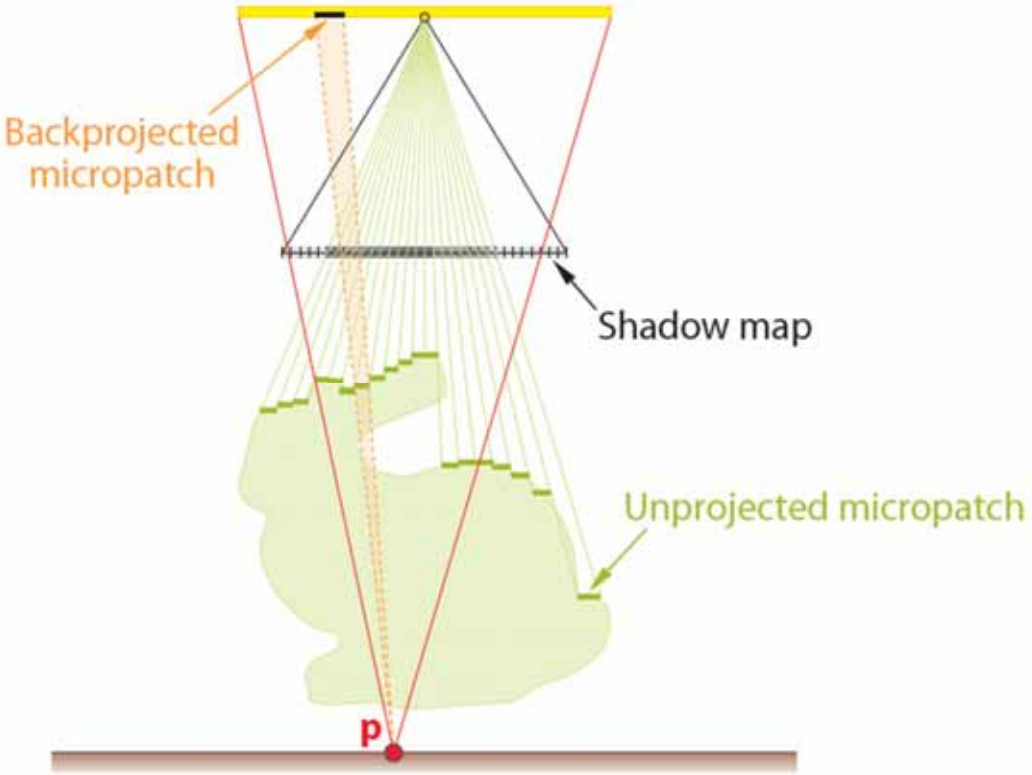
Flächenlichtquellen und Shadow Maps

- viele Ansätze versuchen die Verschattung von Flächenlichtquellen mit nur einer Shadow Map zu approximieren
- es kann nur bei einer Approximation bleiben, denn nicht alle Informationen sind in der SM enthalten



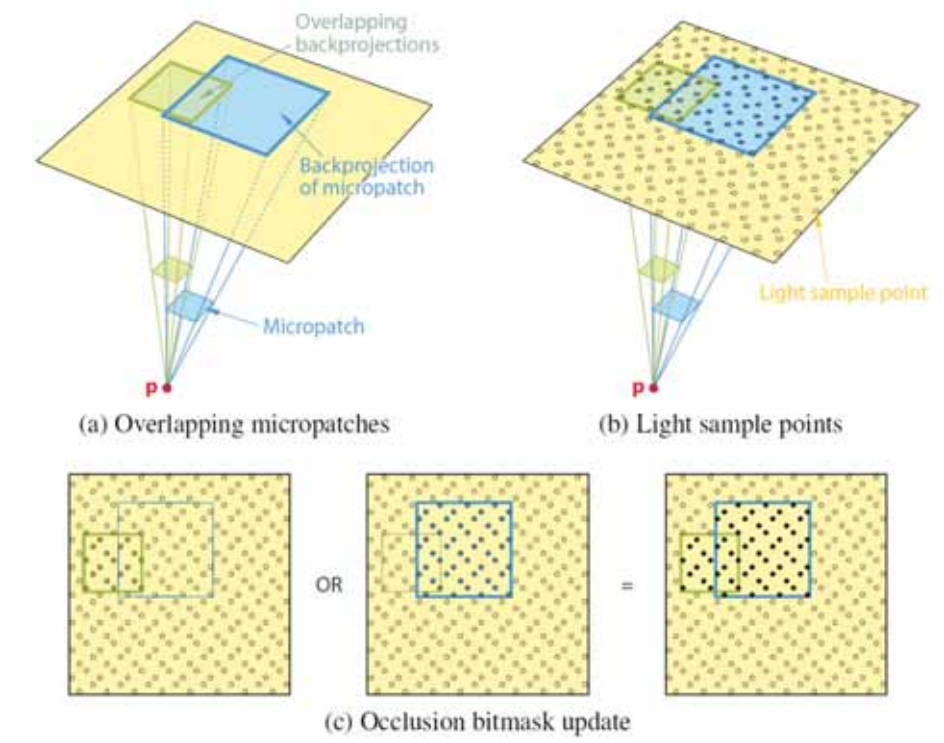
Flächenlichtquellen und Shadow Maps

- genereller Ansatz: Rückprojektion
  - jeder Texel der Shadow Map repräsentiert ein kleines Flächenstück
  - Rückprojektion dieser „Micropatches“ auf die Fläche der Lichtquelle
  - Abschätzung wie viel der Fläche der Lichtquelle bedeckt ist
- Hauptunterschied der Verfahren, die diese Idee verfolgen, ist die Verwaltung der rückprojizierten Fläche



Flächenlichtquellen und Shadow Maps

- Verwaltung der rückproj. Fläche, z.B. über Bitmasken [Schwarz2007]



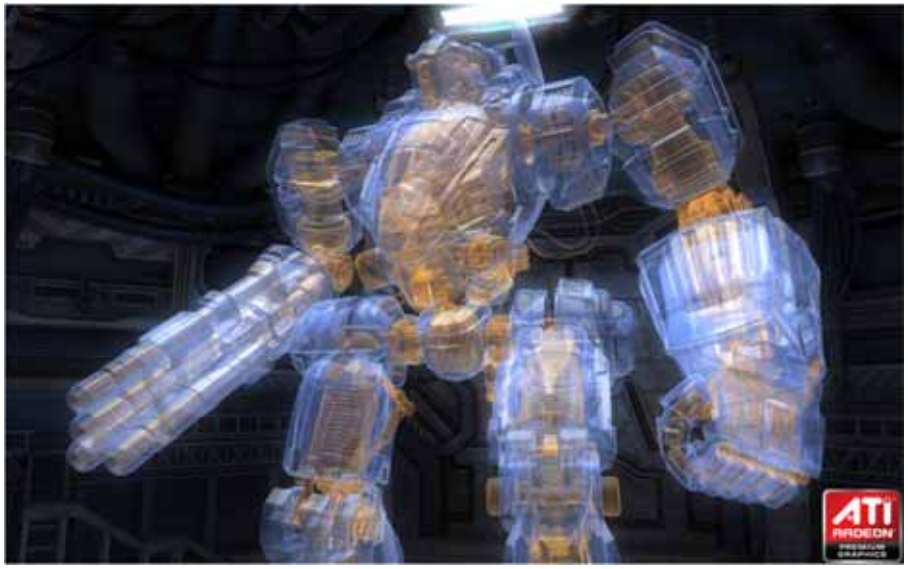
- weitere Verfahren:
  - Suche nach Schattenkanten und Abschätzen der Verdeckung ähnlich Penumbra Wedges, u.v.m.
  - siehe SIGGRAPH-Kurs zu Schatten: http://www.realtimeshadows.com/

Exkurs: Per-Pixel Linked Lists

Beispielanwendung: Transparenz ohne Sortierung der Geometrie

- Ziel: vermeide Sortierung von transparenten Polygonen
- Schritt 1: zeichne opaque Geometrie
- Schritt 2: Rendering transparenter Objekte
- Schritt 3: Pro-Pixel Compositing

- direkte Anwendung:
  Shadow Maps mit semi-transparenten Flächen…
- … anschließend:
  weiche Schatten!



Quelle der Folien: Advances in Real-Time Rendering
(http://advances.realtimerendering.com/s2010/index.html)

Implementationsdetails:
http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html

Exkurs: Per-Pixel Linked Lists

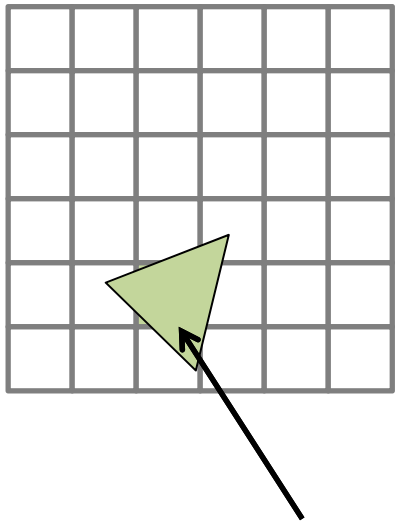
Beispielanwendung: Transparenz ohne Sortierung der Geometrie

- Schritt 1: zeichne opaque Geometrie (wie üblich)
- Schritt 2: Rendering transparenter Objekte
  - speichere Fragmente mit einer verketteten Listen
  - speichere pro Fragment: Farbe, Alpha, Tiefe
  - hierzu werden 2 Puffer benötigt
    - Head Pointer Buffer: ein Integer pro Pixel
    - Node Buffer: Puffer zum Speichern aller Fragmente
- Schritt 3: Pro-Pixel Compositing

Exkurs: Per-Pixel Linked Lists

Erzeugung der verketteten Listen

Render Target



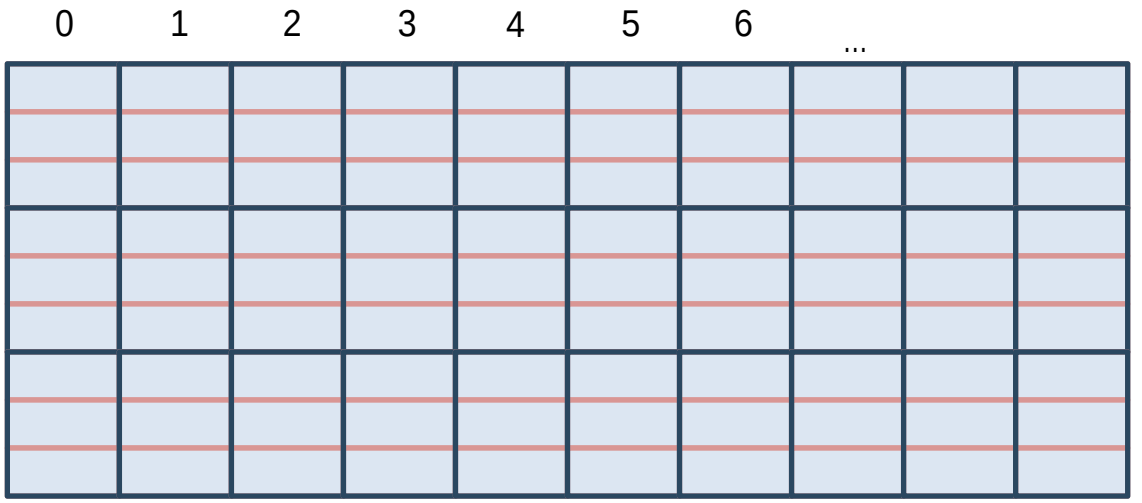
opake Geometrie
aus Schritt 1

Head Pointer Buffer

-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1

Counter = 0

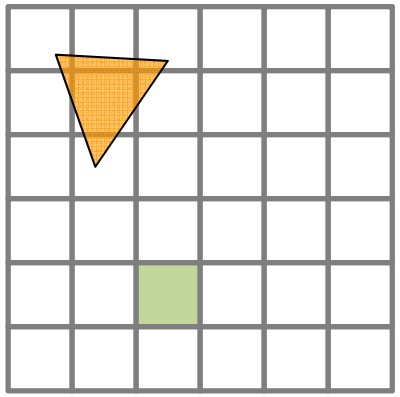
Node Buffer



Exkurs: Per-Pixel Linked Lists

Erzeugung der verketteten Listen

Render Target

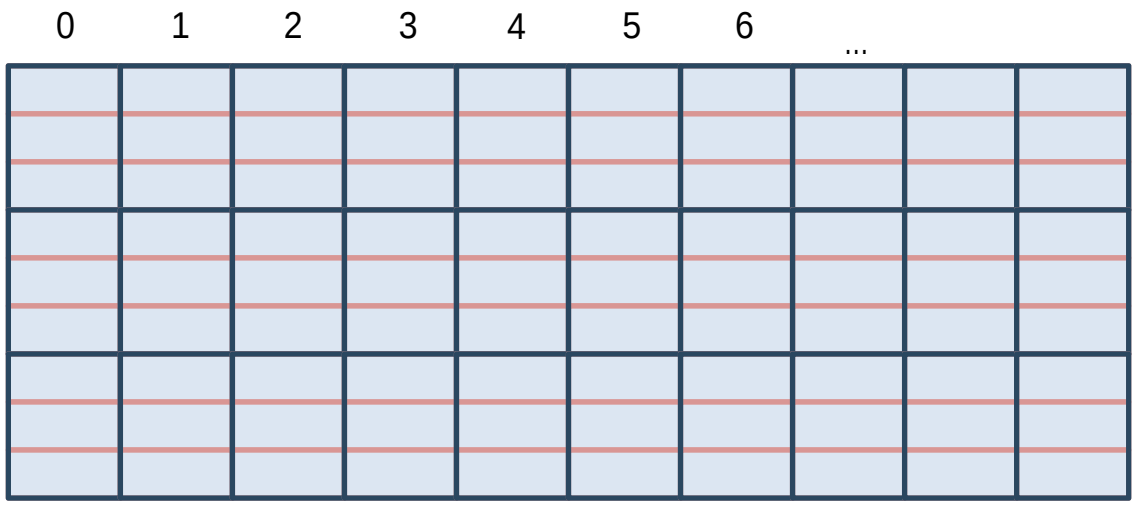


Head Pointer Buffer

-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1

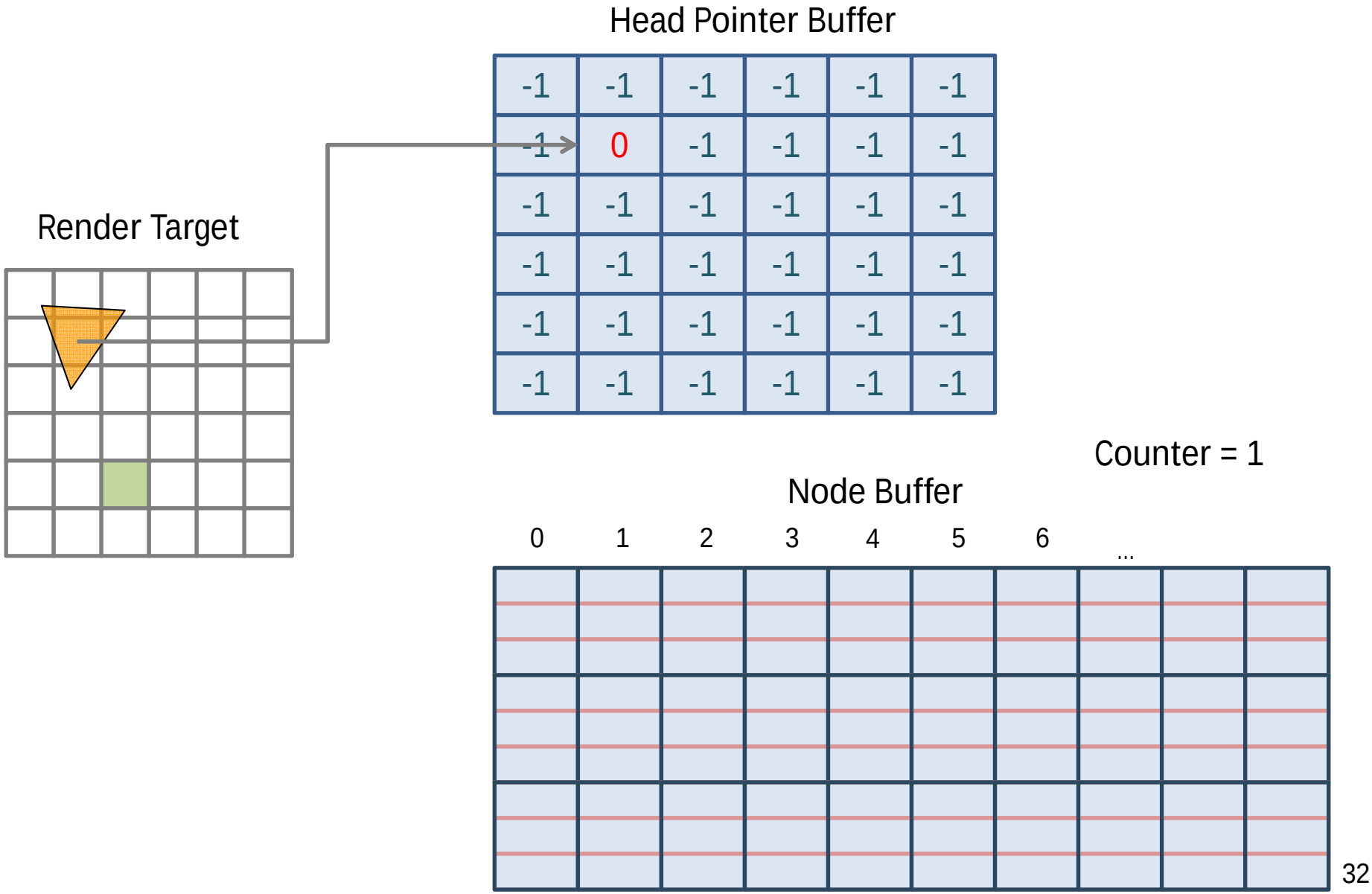
Counter = 0

Node Buffer



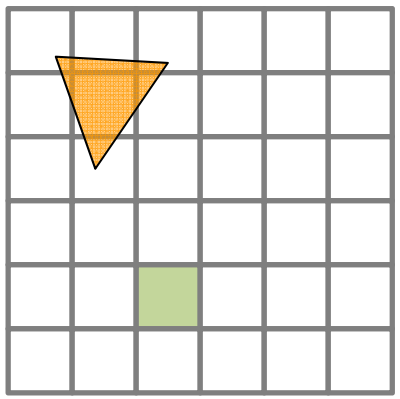
Exkurs: Per-Pixel Linked Lists

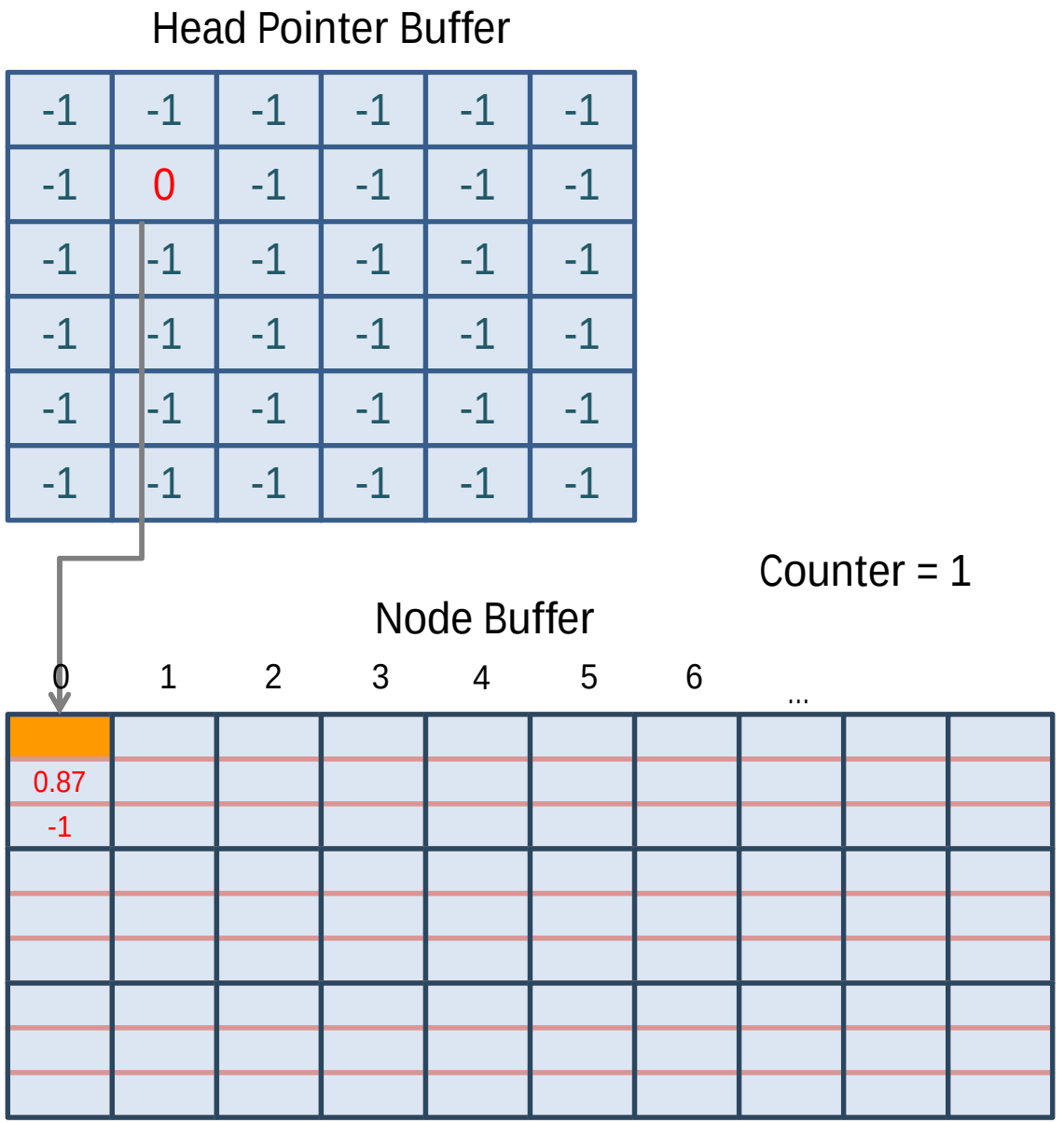
Erzeugung der verketteten Listen



Exkurs: Per-Pixel Linked Lists

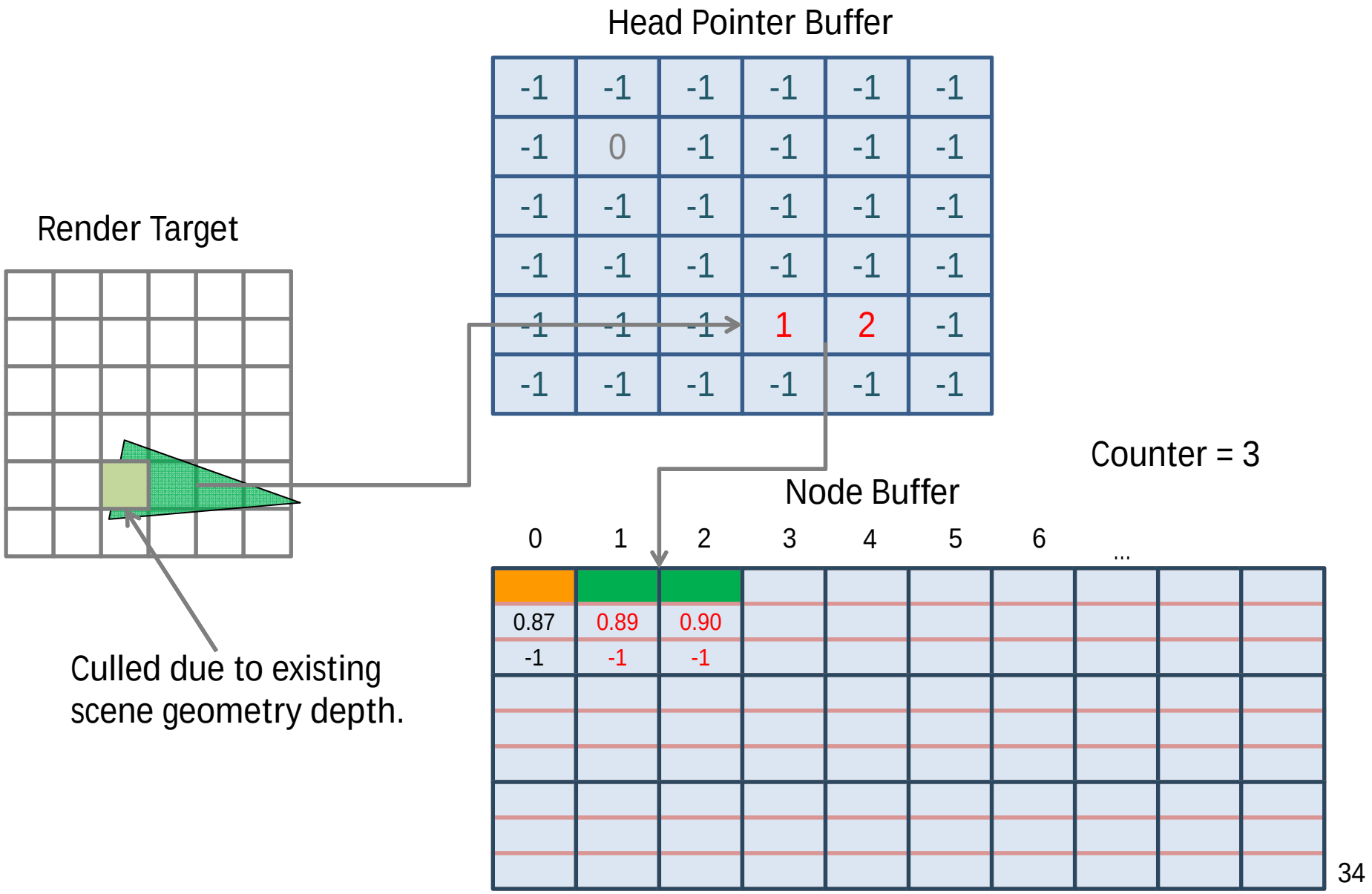
Erzeugung der verketteten Listen





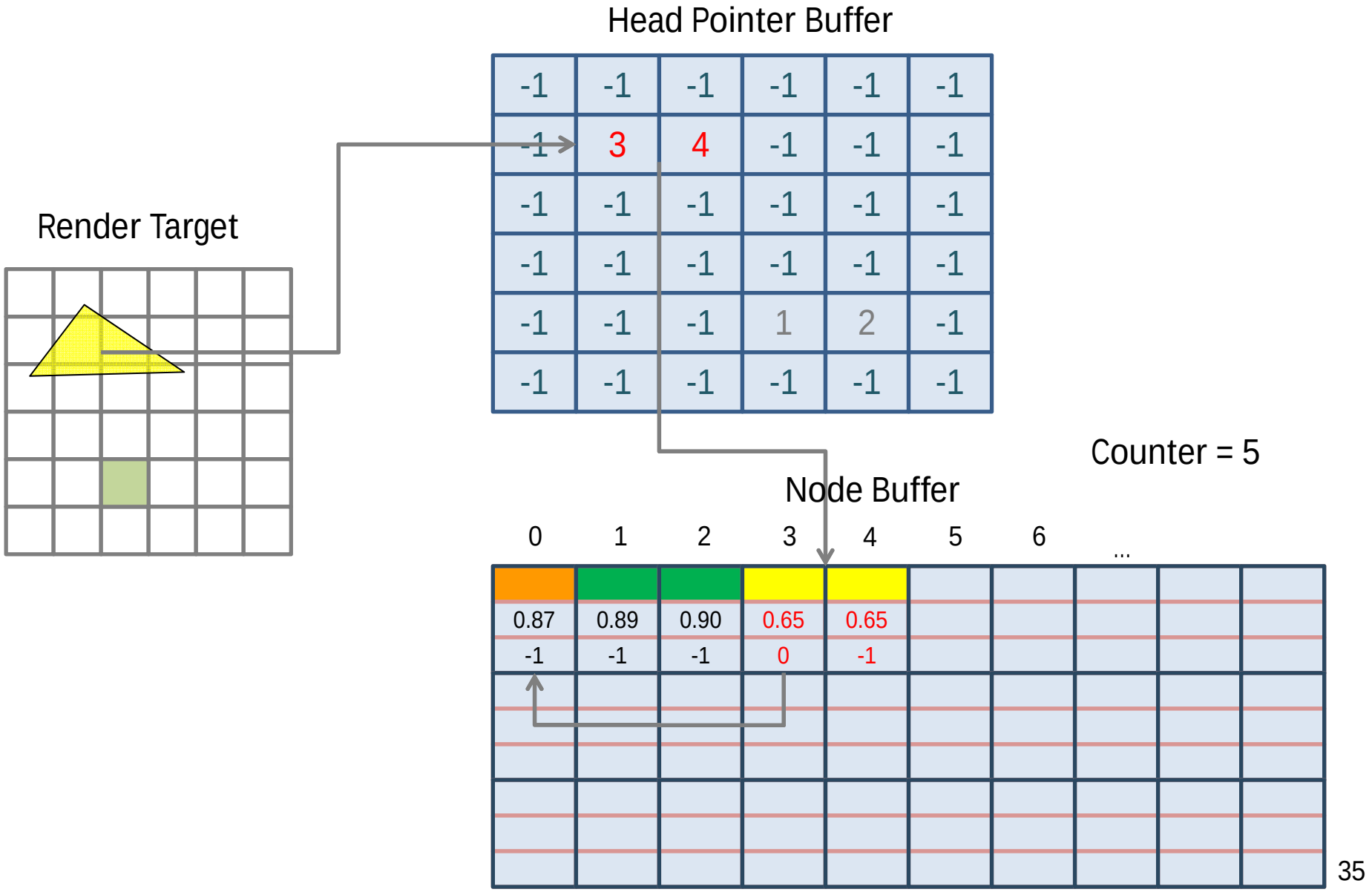
Exkurs: Per-Pixel Linked Lists

Erzeugung der verketteten Listen



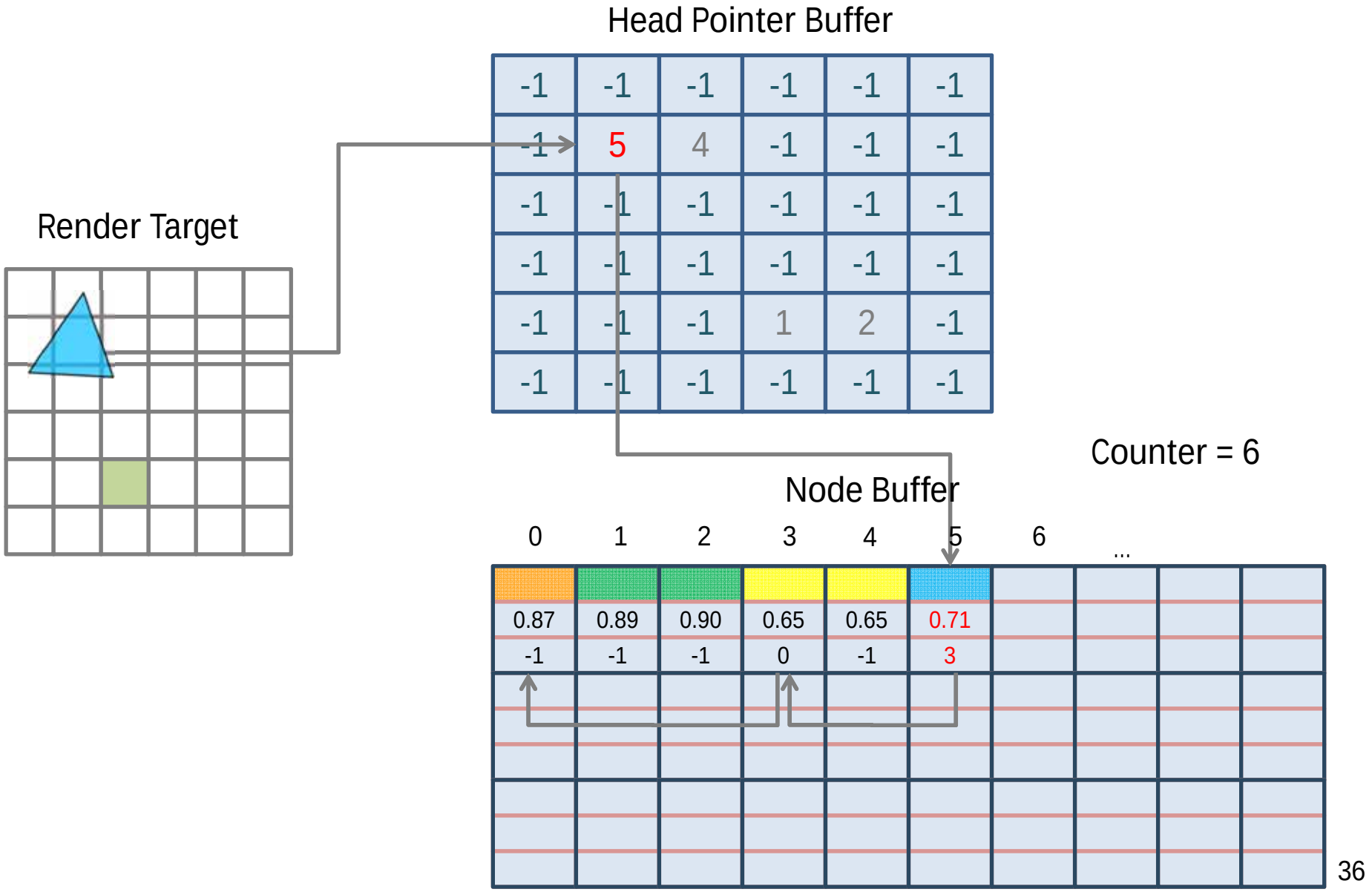
Exkurs: Per-Pixel Linked Lists

Erzeugung der verketteten Listen



Exkurs: Per-Pixel Linked Lists

Erzeugung der verketteten Listen



Exkurs: Per-Pixel Linked Lists

- Welche OpenGL-Funktionalität wird dazu benötigt?

- Atomare Zähler
  - unsigned int, Inkrementieren/Dekrementieren
  - keine explizite Synchronisation notwendig
  - http://www.opengl.org/wiki/Atomic_Counter

- Shader Storage Buffer Object
  - (große) Speicherblöcke mit wahlfreien und atomaren Zugriffen
  - z.B. **int atomicExchange(inout int mem, int data);**

- Größe des Node Buffers
  - = Anzahl transparenter Fragmente, die den Tiefentest bestehen
  - Bestimmung z.B. mit sogenannten Occlusion Queries (siehe späteres Kapitel)

Exkurs: Per-Pixel Linked Lists

Beispielanwendung: Transparenz ohne Sortierung der Geometrie

- Schritt 1: zeichne opaque Geometrie
- Schritt 2: Rendering transparenter Objekte
- Schritt 3: Pro-Pixel Compositing
  - Zeichnen eines bildschirmfüllenden Rechtecks zum Anstoßen eines Fragment-Programms
  - Sortierung der verketteten Listen pro Pixel (z.B. Insertion Sort)
  - Compositing der sortierten Fragmente mit Hintergrundfarbe

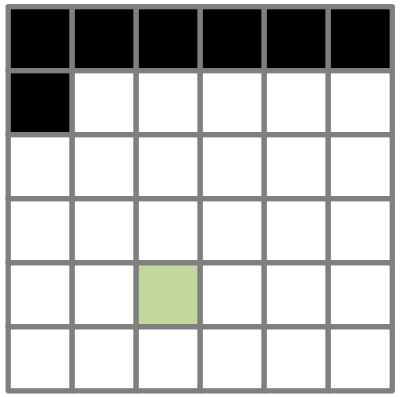
Exkurs: Per-Pixel Linked Lists

Compositing, Rendering der Fragmente

Head Pointer Buffer

-1	-1	-1	-1	-1	-1
-1	5	4	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	1	2	-1
-1	-1	-1	-1	-1	-1

Render Target



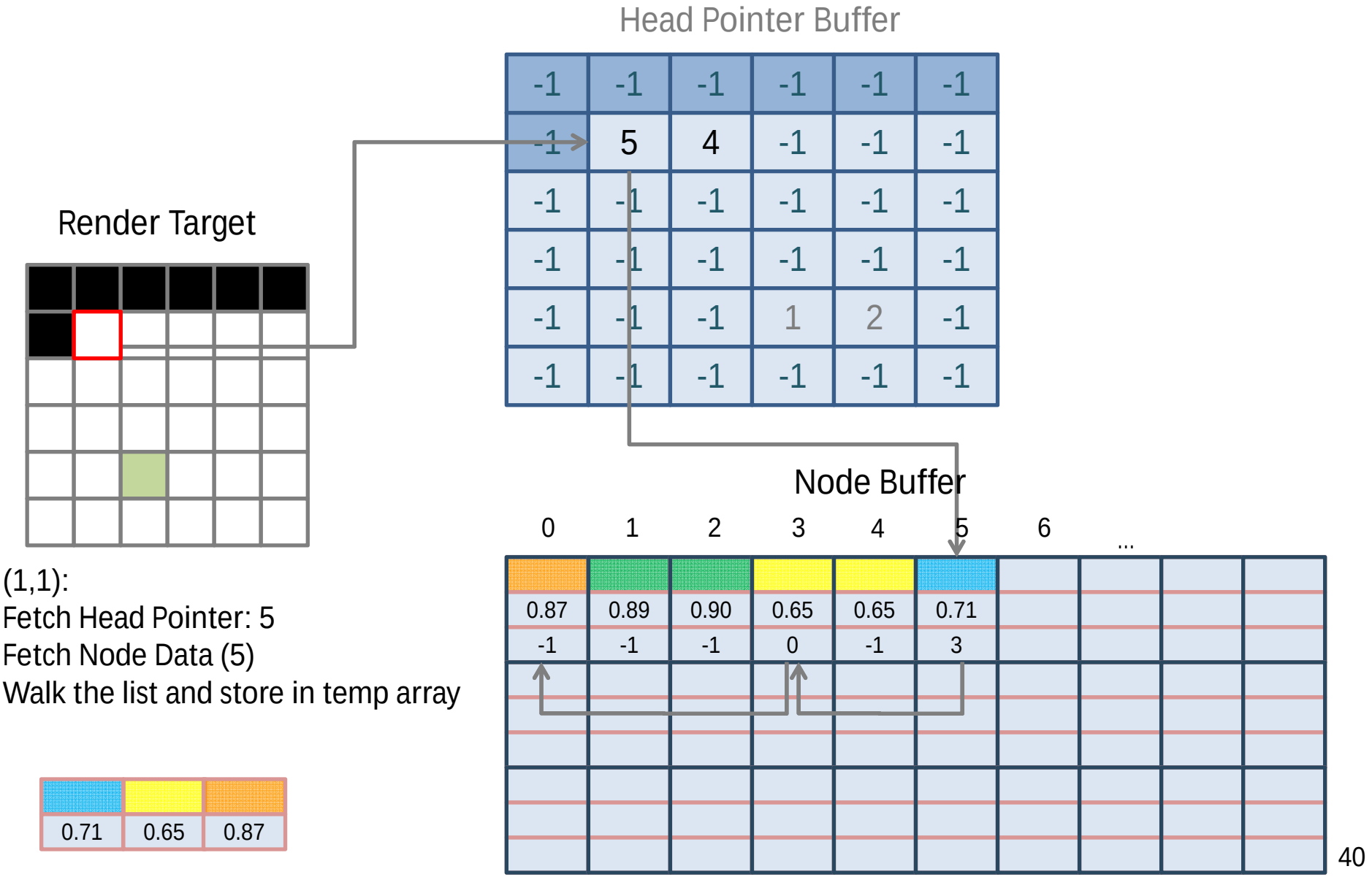
(0,0)->(1,1):
Fetch Head Pointer: -1
-1 indicates no fragment to render

Node Buffer

0	1	2	3	4	5	6	...
0.87	0.89	0.90	0.65	0.65	0.71		
-1	-1	-1	0	-1	3		

Exkurs: Per-Pixel Linked Lists

Compositing, Rendering der Fragmente

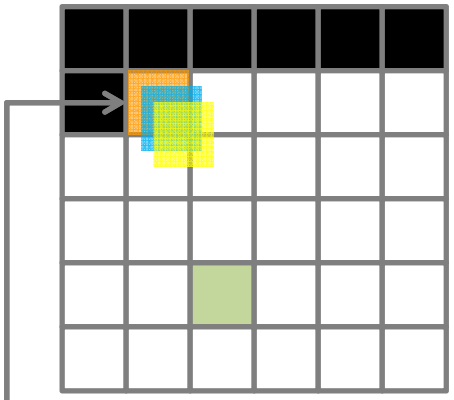


Exkurs: Per-Pixel Linked Lists

Compositing, Rendering der Fragmente

-1	-1	-1	-1	-1	-1
-1	5	4	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	1	2	-1
-1	-1	-1	-1	-1	-1

Render Target



(1,1):
Sort temp array
Blend colors and write out

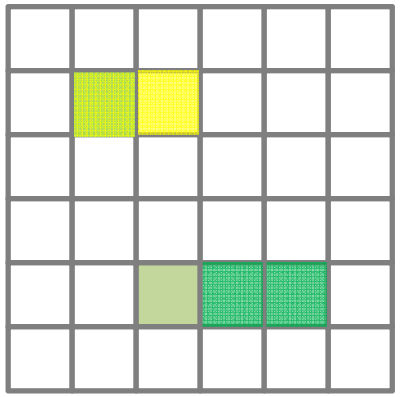
Node Buffer

0	1	2	3	4	5	6	...
0.87	0.89	0.90	0.65	0.65	0.71		
-1	-1	-1	0	-1	3		

Exkurs: Per-Pixel Linked Lists

Compositing, Rendering der Fragmente

Render Target



Head Pointer Buffer

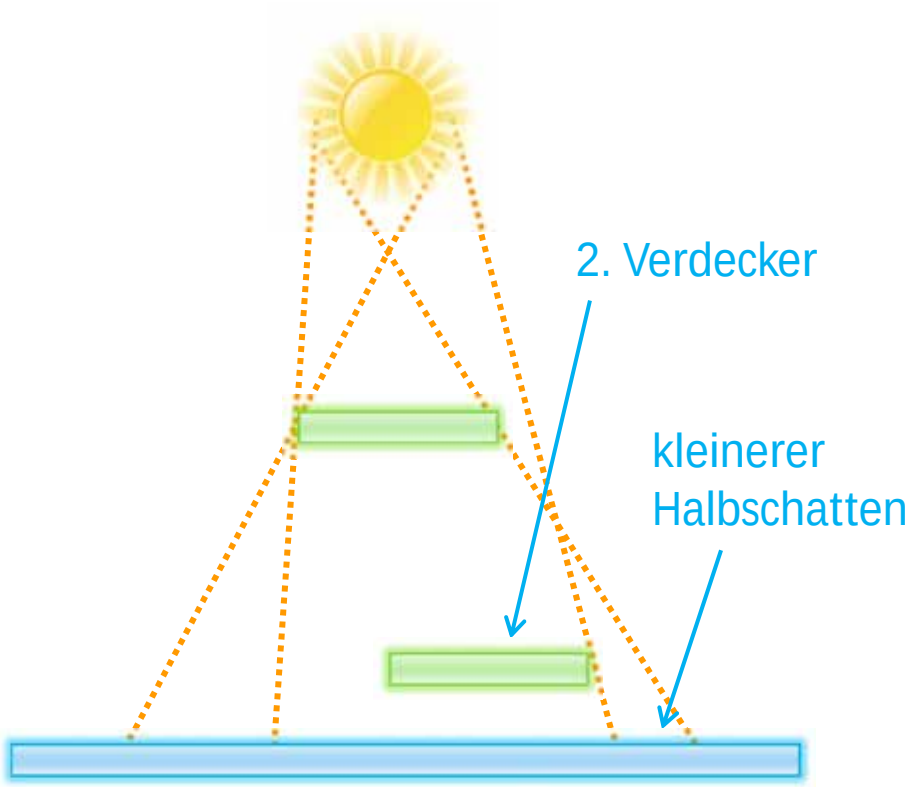
-1	-1	-1	-1	-1	-1
-1	5	4	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	1	2	-1
-1	-1	-1	-1	-1	-1

Node Buffer

0	1	2	3	4	5	6	...		
0.87	0.89	0.90	0.65	0.65	0.71				
-1	-1	-1	0	-1	3				

Filtering Multilayer Shadow Maps for Accurate Soft Shadows

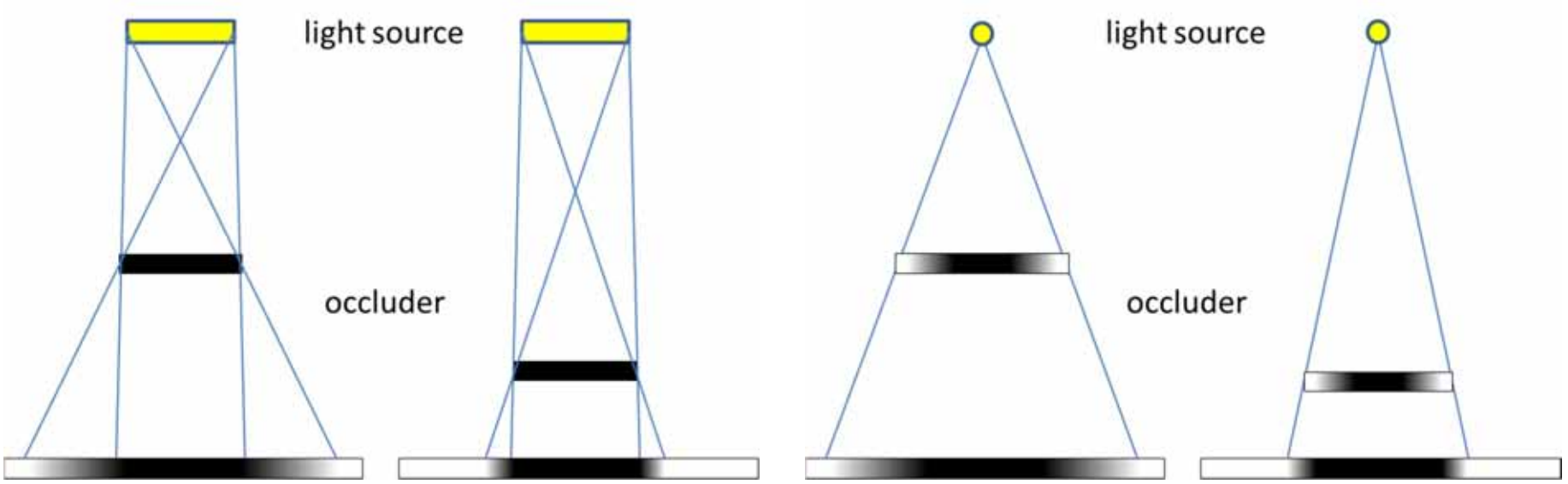
- Anwendung der Pro-Pixel Fragment Lists (oder vergleichbarer Technik) zum Sammeln der Information über verdeckte Flächen



Filtering Multilayer Shadow Maps for Accurate Soft Shadows

Grundidee

- Schatten eines opaken Objekts von einer Flächenlichtquelle…
- … lässt sich durch „weichgezeichnete“ Geometrie und eine Punktlichtquelle nachbilden



Filtering Multilayer Shadow Maps for Accurate Soft Shadows

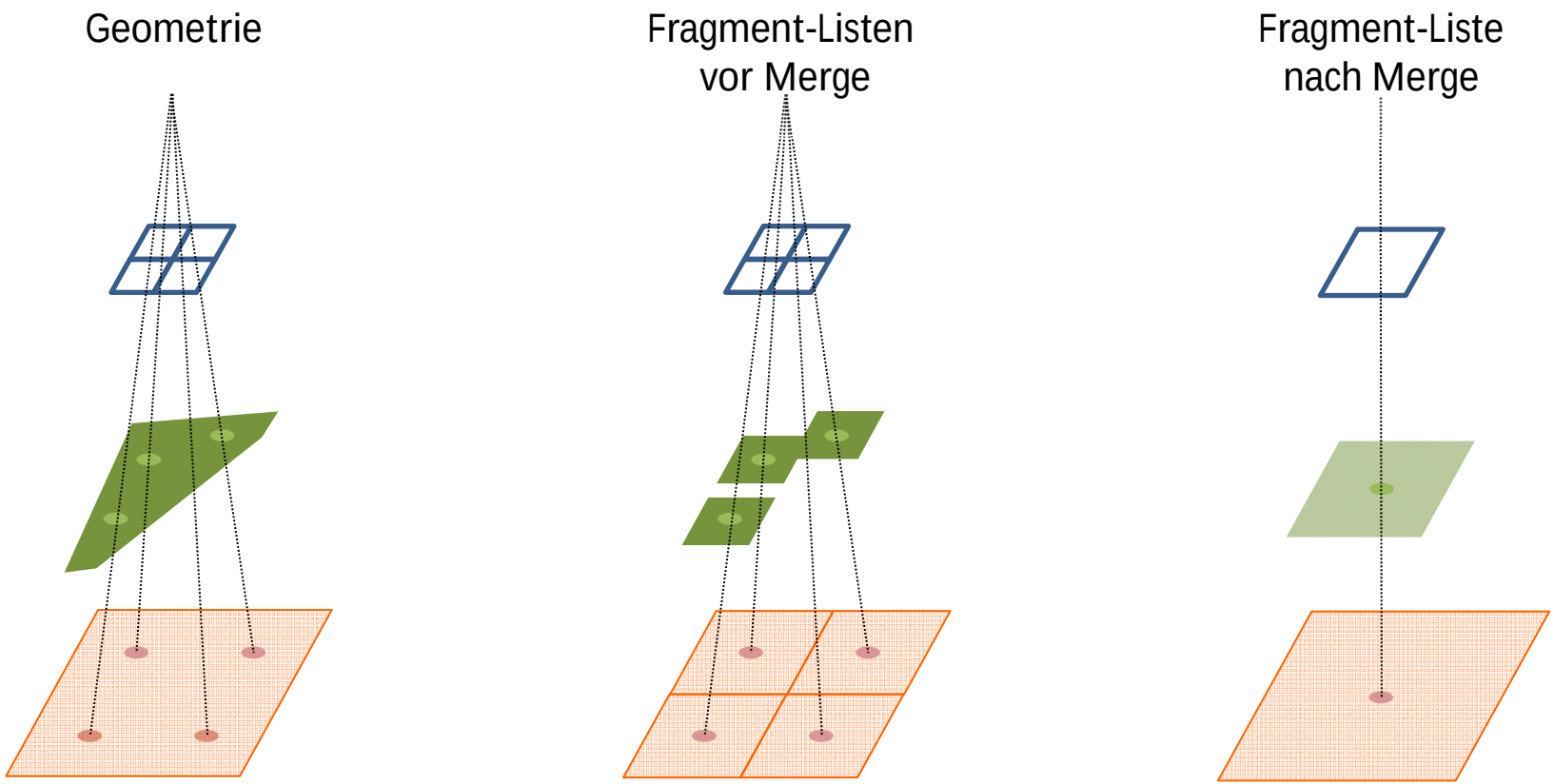
Grundidee

- Schatten eines opaken Objekts von einer Flächenlichtquelle…
- … lässt sich durch „weichgezeichnete“ Geometrie und eine Punktlichtquelle nachbilden

Filtering Multilayer Shadow Maps for Accurate Soft Shadows

Grundidee

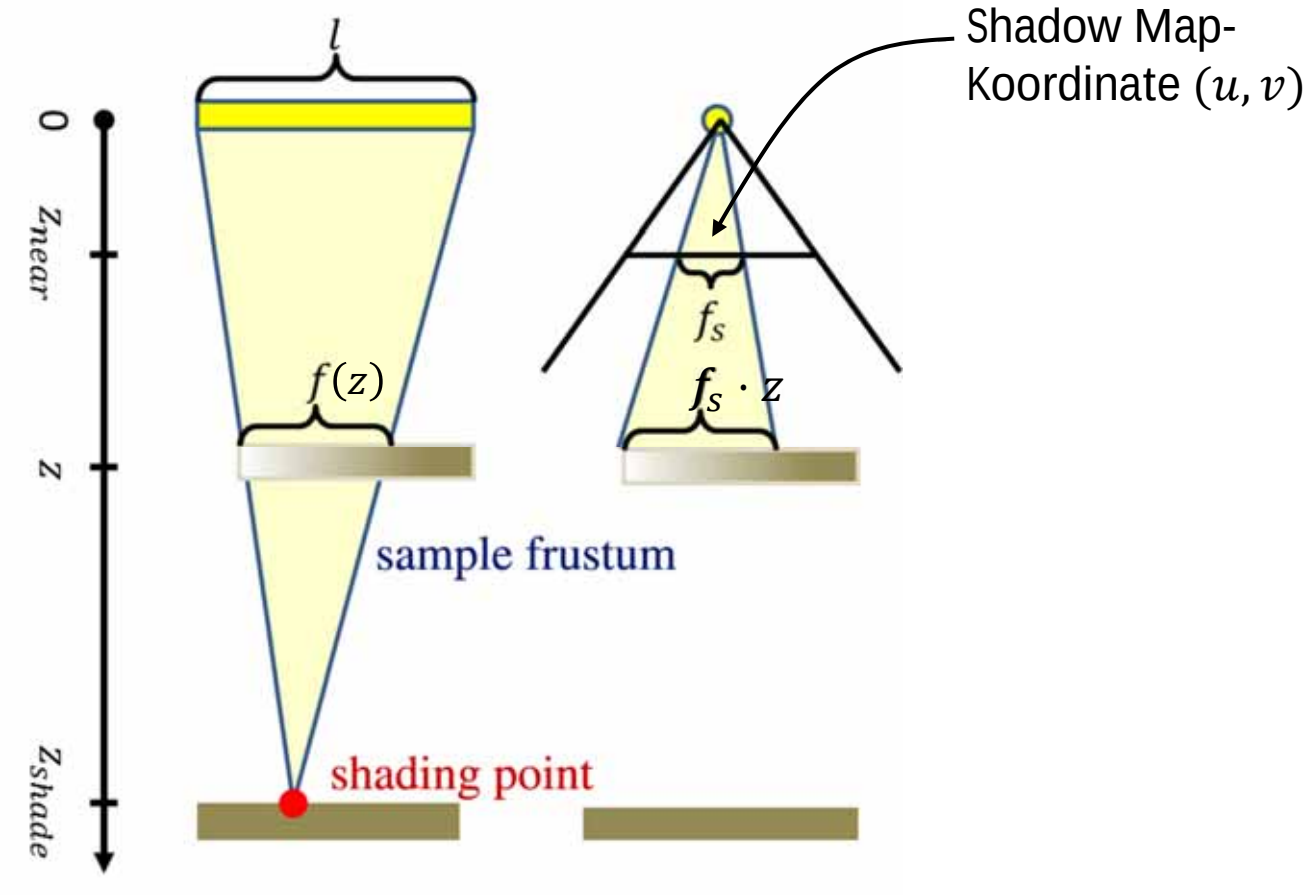
- „Multilayer Shadow Map“ speichert alle Flächen pro Pixel
- Filterung der Shadow Map durch Zusammenfassen von Fragmenten mit ähnlicher Tiefe (ergibt u.U. semitransparente Fragmente)
- berechne damit eine Mip-Map-Pyramide



Filtering Multilayer Shadow Maps for Accurate Soft Shadows

Grundidee: Lookup

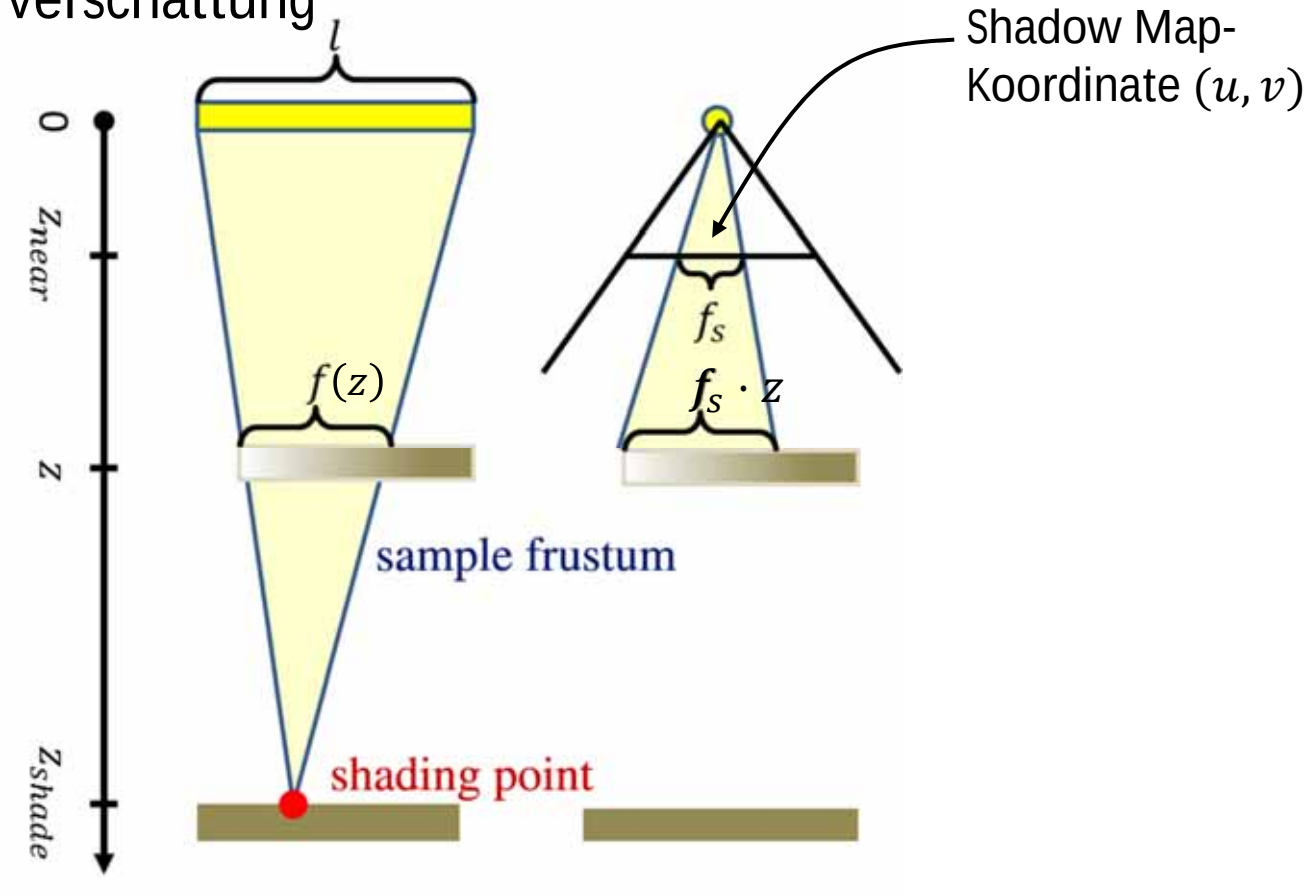
- wie groß ist der Bereich (abhängig vom Abstand zu einem Oberflächenpunkt) in dem Geometrie verschatten könnte? → $f(z)$
- $f(z)$ projiziert in die Shadow Map → Filtergröße $f_s(z)$



Filtering Multilayer Shadow Maps for Accurate Soft Shadows

Grundidee: Lookup

- für alle Auflösungsstufen
  - für alle Fragmente in der Liste für (u, v)
  - wenn $f_s \cdot z$ (Größe des Fragments in Weltkoord.) == $f(z)$
    - akkumuliere Verschattung



Voxelisierung

- bisher hatten wir den Fall betrachtet, dass es eine oder wenige Lichtquellen gibt, mit der wir beleuchten und Schatten berechnen
  - im Prinzip bestimmen wir die Sichtbarkeit von einem/wenigen Punkten zu vielen Punkten
  - benötigen wir die Sichtbarkeit von vielen zu vielen Punkten, dann sind andere „globalere“ Datenstrukturen u.U. besser geeignet
  - eine Möglichkeit: „Voxelisierung“ der Szene, d.h. Repräsentation durch ein reguläres Gitter → effiziente Verfahren dazu existieren
  - Sichtbarkeitsberechnung durch Ray Marching

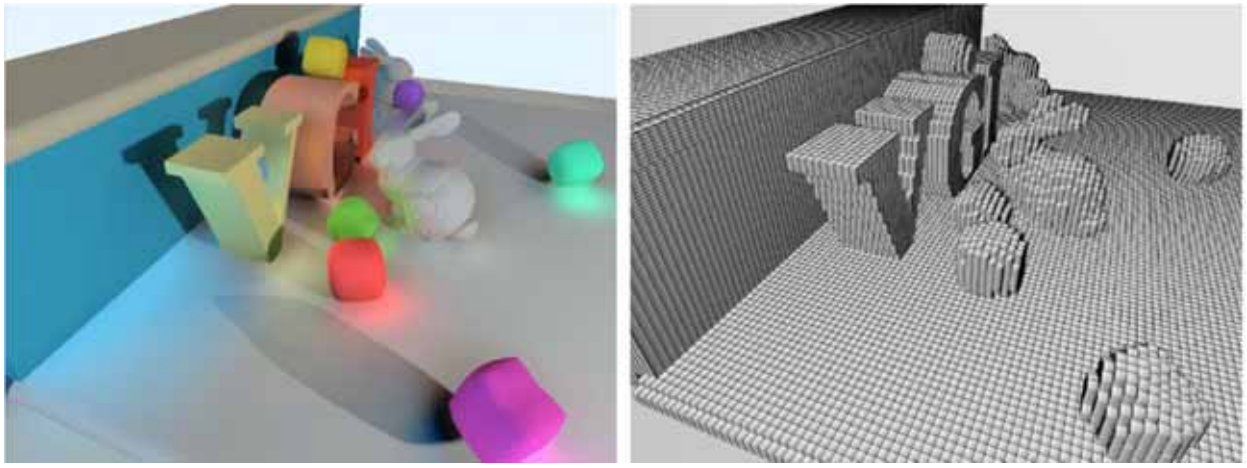


Bild: Sinje Thiedemann et al. „Voxel Global Illumination“

Voxelisierung

- Voxelisierung als Beschleunigungsstruktur ermöglicht auch globale Beleuchtung mittels Ray Marching

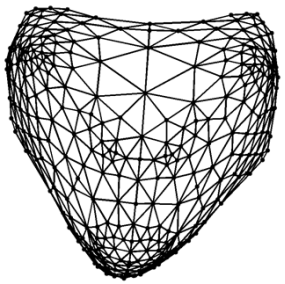
Beispiel: **Real-time global illumination with voxel-based cone tracing**
Crassin et. al, Pacific Graphics 2011

Voxelisierung

Beispiel: **Real-time voxel cone tracing in Unreal Engine 4**

Motivation

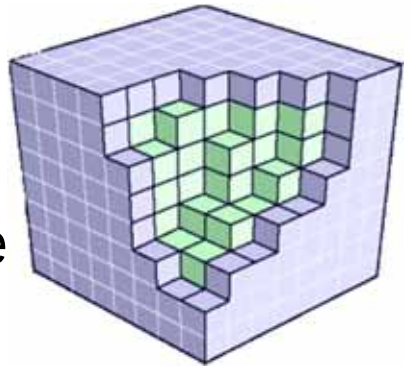
- Objekte in CG meist Oberflächenmodelle (boundary representations)
  - Primitive: Dreiecke, parametrische Flächen
  - bietet sich an: die meiste Interaktion mit dem Licht findet an den Flächen statt

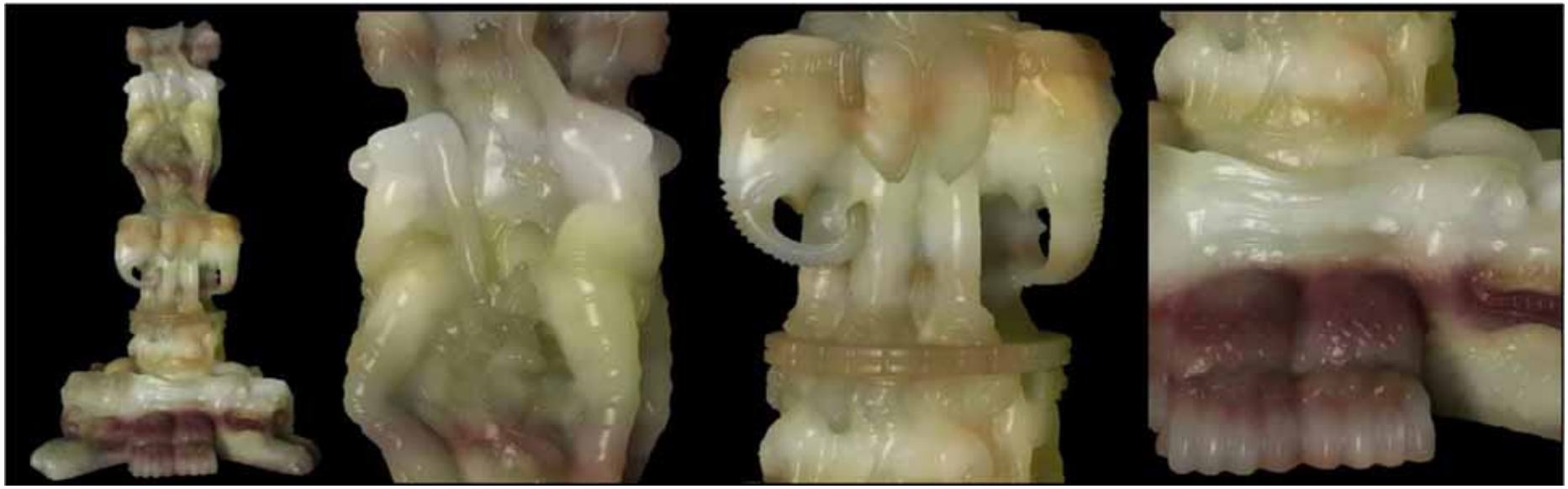


Origami Teapot, Tomoro Tachi, Japan 2006
gefaltet aus einem quadratischen Papierbogen

Motivation

- Volumen-Repräsentationen
  - Primitive: Voxel (volumetric pixels), Zellen
  - Gitterauflösung wird festgelegt, der Aufwand ist dann unabhängig von der Komplexität der Eingabe
  - volumetrische Darstellungen sind u.a. für Transluzenz, physikalische Simulation etc. wichtig

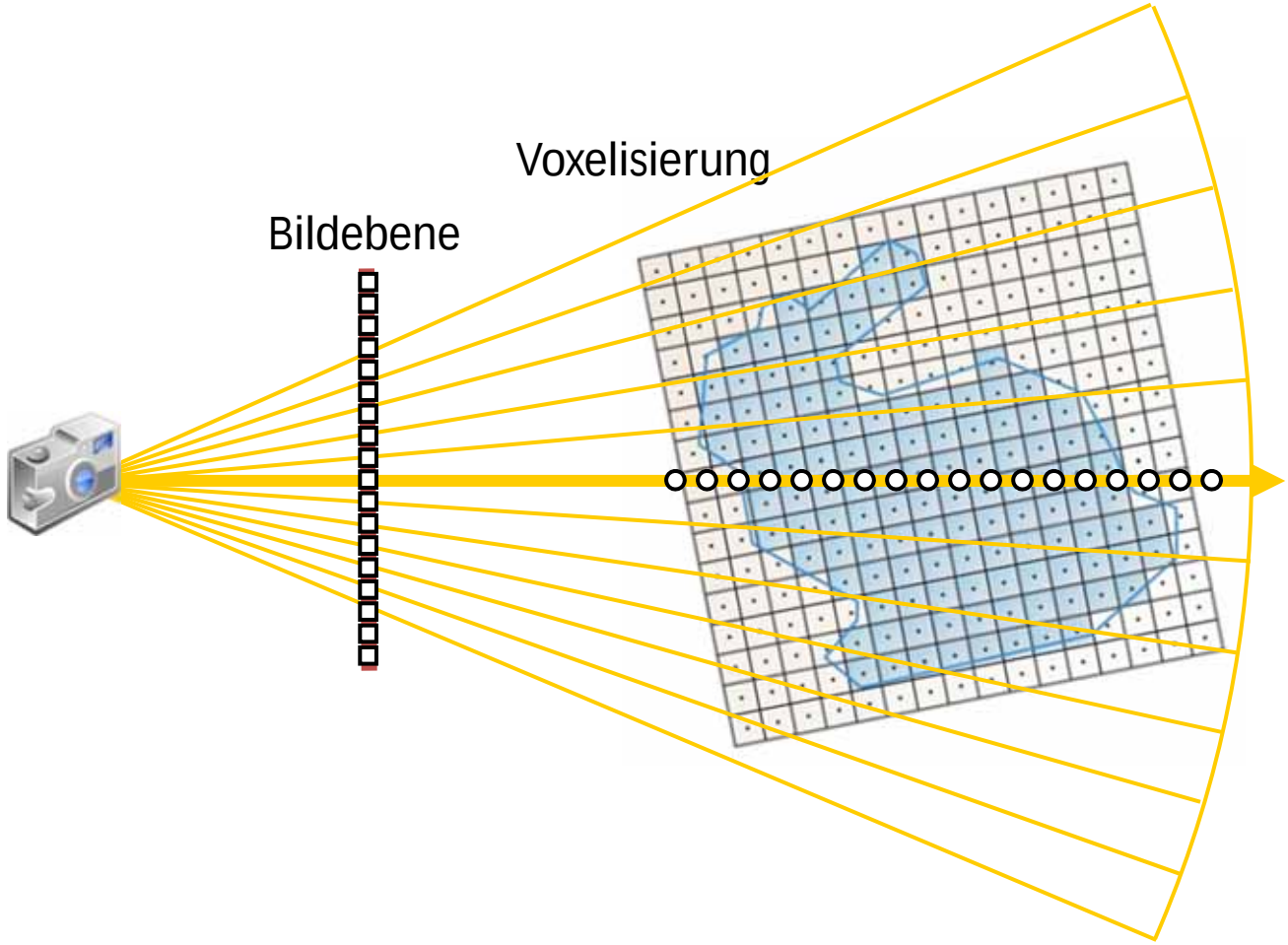




Voxelisierung: Konzepte

Rendering (und Schatten- bzw. Sekundärstrahlen) mit Raymarching

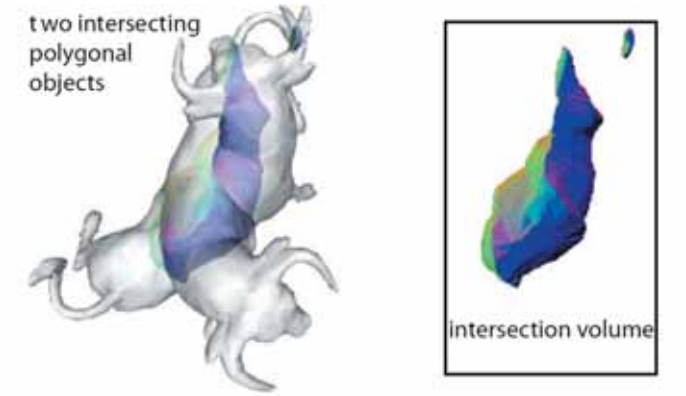
- Filterung durch Mipmapping möglich („Voxel Cone Tracing“)



Noch mehr Beispiele...

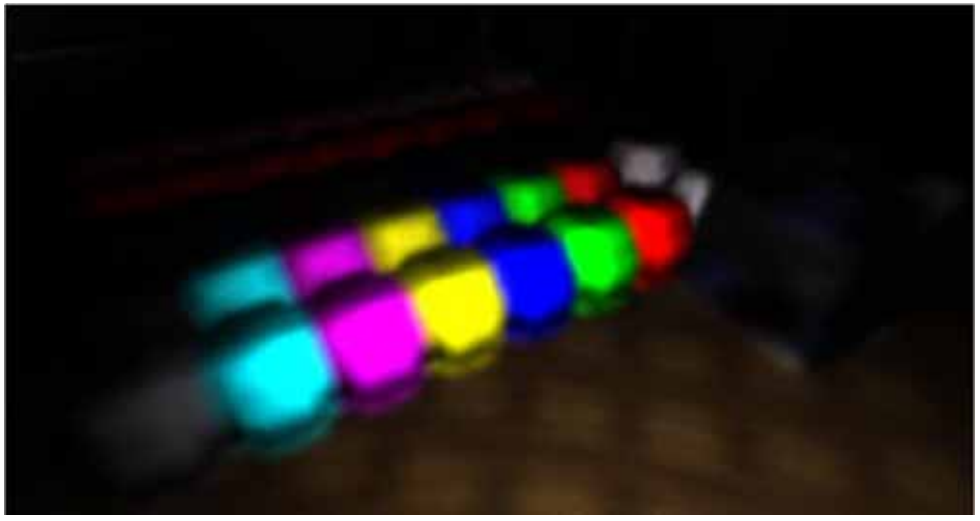
[Crane et al. 2007]

[Crassin et al. 2011]



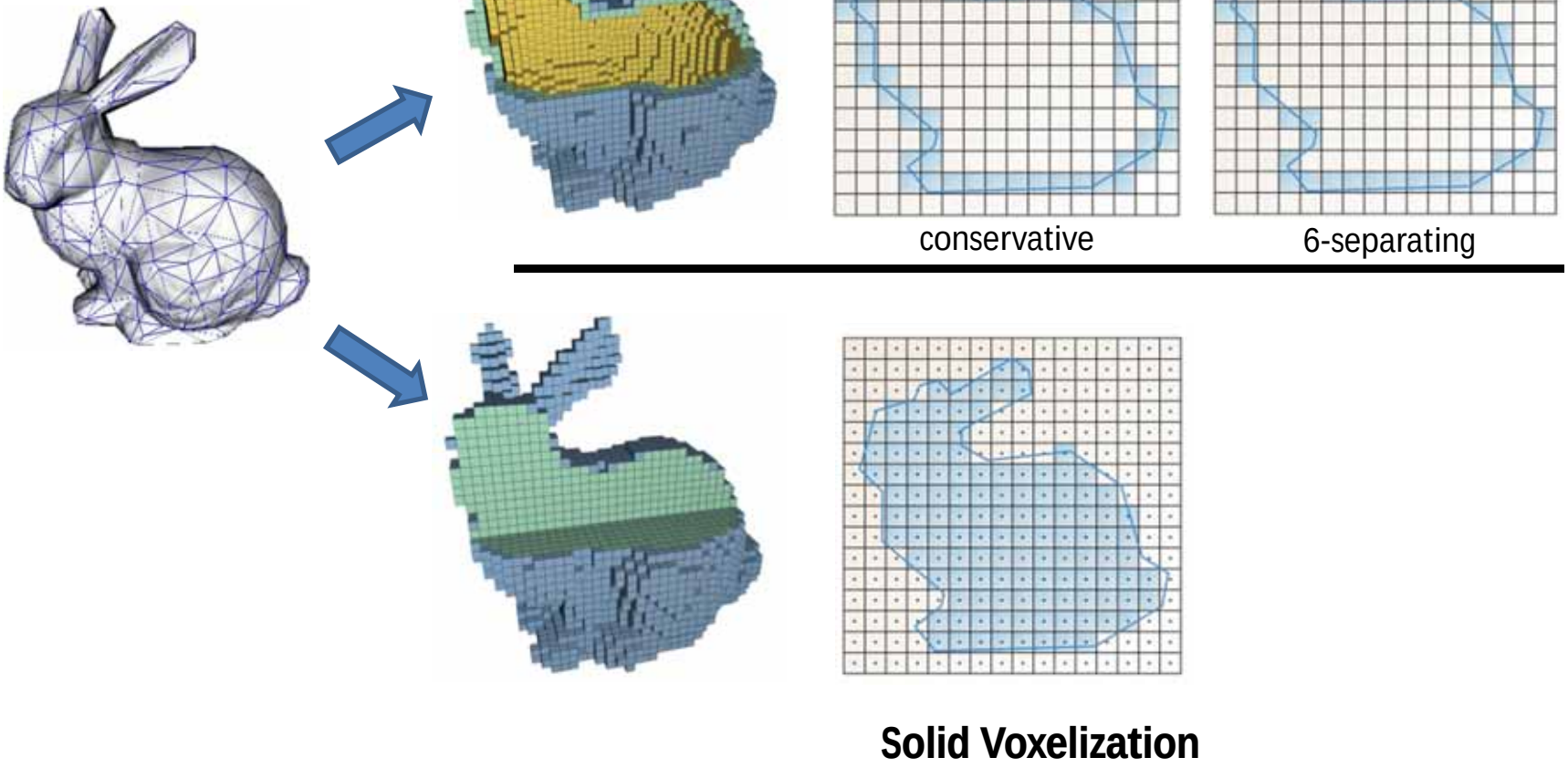
[Eisemann and Decoret, 2008]

Voxelisierung in der Unreal Engine



Bilder: The Technology Behind the
“Unreal Engine 4 Elemental demo”, SIGGRAPH‘12

Voxelisierung: Konzepte

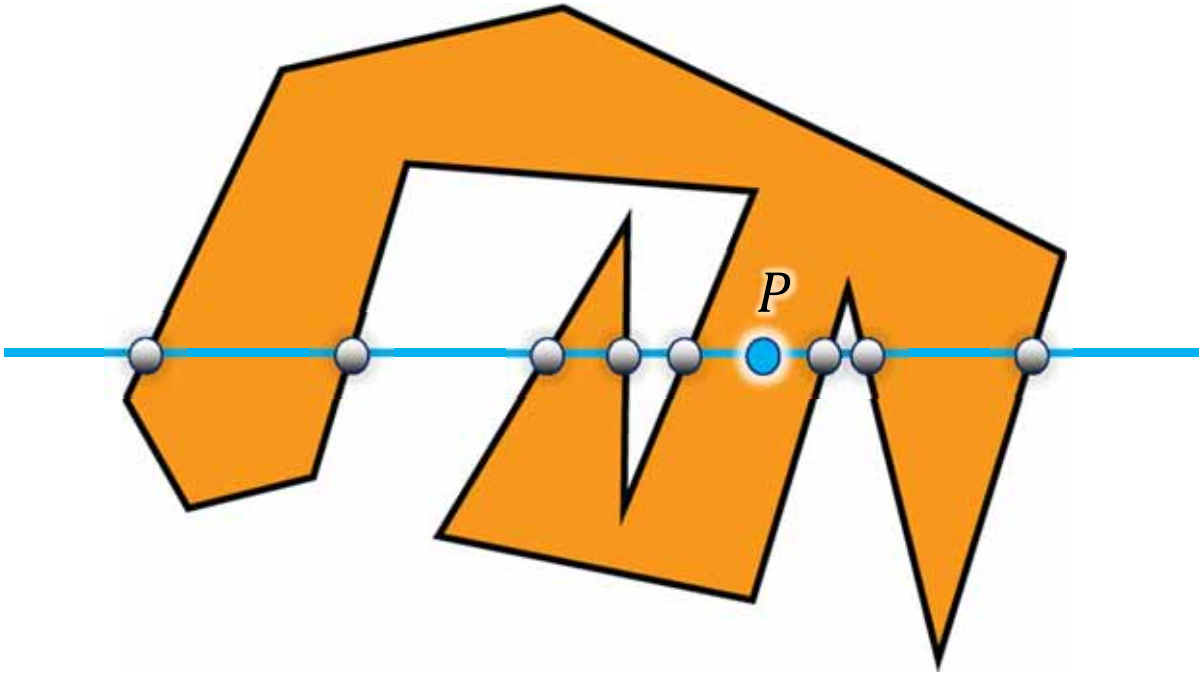


Bilder: Schwarz and Seidel, 2010

Stencil-Based Solid Voxelization

- solide Voxelisierung bedeutet: feststellen, ob ein Voxel (bzw. sein Mittelpunkt) im Inneren eines geschlossenen Dreiecksnetzes liegt

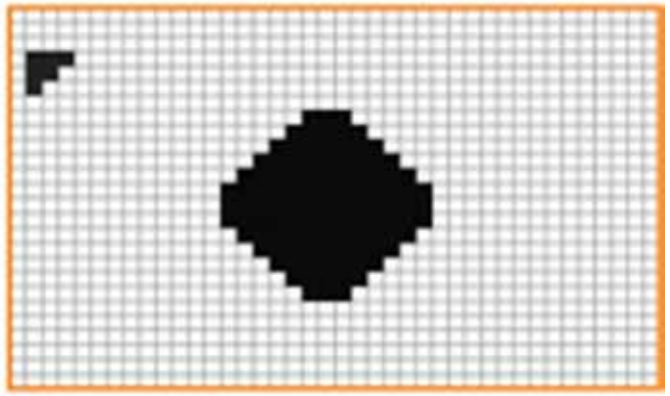
- Lösung: der Odd-Even-Test
  - wenn der Punkt im Inneren liegt, dann haben alle Strahlen von dort ausgehend eine ungerade Anzahl Schnitte mit der Oberfläche
  - ähnlich wie Schattenvolumen → zähle Schnitte mit dem Stencil Buffer

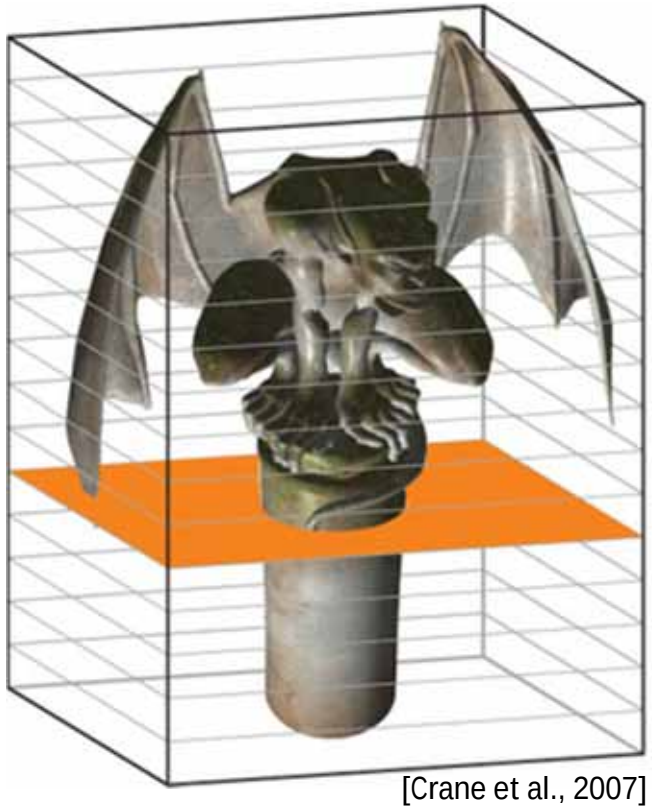


Stencil-Based Solid Voxelization

Voxelisieren einer 2D-Schicht

- zeichne mit einer orthographischen Kamera und clippe das Objekt an einem Halbraum (z.B. schneide alles oberhalb der orangen Ebene ab)
- setze alle Stencil Buffer Werte auf 0
- zeichne das Objekt
  - Vorderseiten erhöhen, Rückseiten erniedrigen den Stencil Buffer Wert
- jeder Voxel im Inneren hat einen Wert $\neq 0$





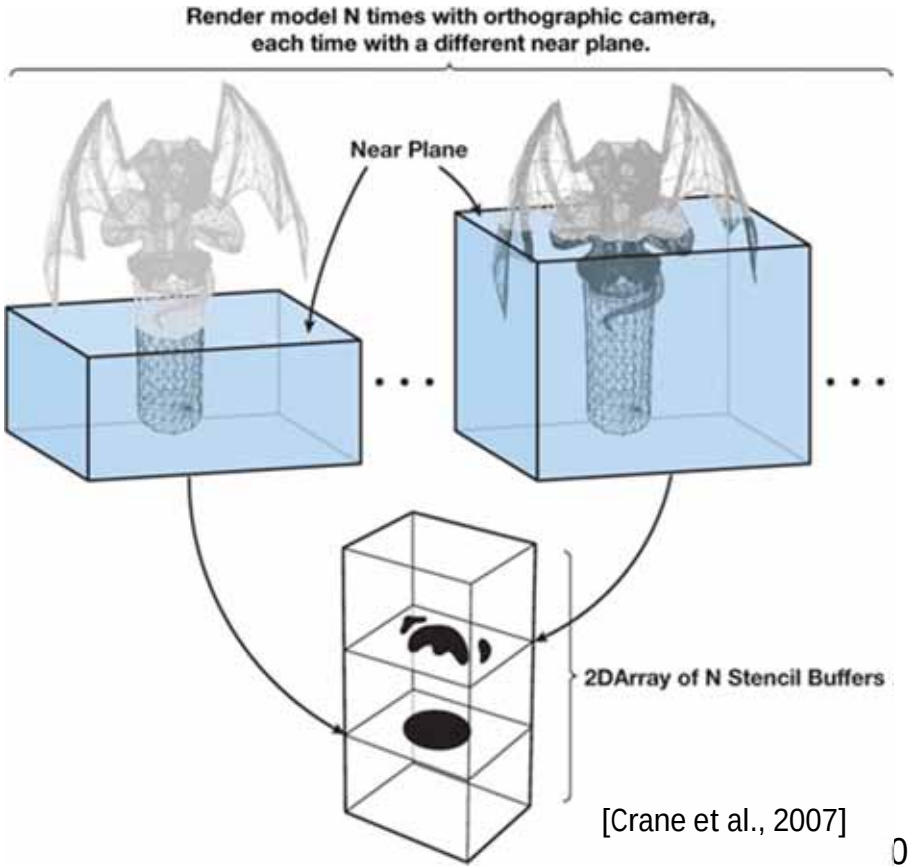
[Crane et al., 2007]

Stencil-Based Solid Voxelization

Voxelisieren des 3D-Volumens

- führe die Methode für jede Schicht (insg. N-mal) durch
- das Volumen kann gespeichert werden als
  - 3D Textur (eine Schicht in z-Richtung als Rendertarget)
  - Schichten nebeneinander in einer 2D-Textur

- einfache und robuste Methode
- aber nicht sehr effizient: N-mal Zeichnen für Volumen mit Tiefe N



[Crane et al., 2007]

OpenGL Implementation (Prinzip)

- **Rendering Setup:**
  aktiviere Stencil Test, deaktiviere Schreiben in Farb- und Tiefen-Puffer

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
glDisable(GL_DEPTH_TEST);

glEnable(GL_STENCIL_TEST);
glClearStencil(0x0);                    // Stencil Clear Value
glClear(GL_STENCIL_BUFFER_BIT);         // lösche Stencil Buffer
```

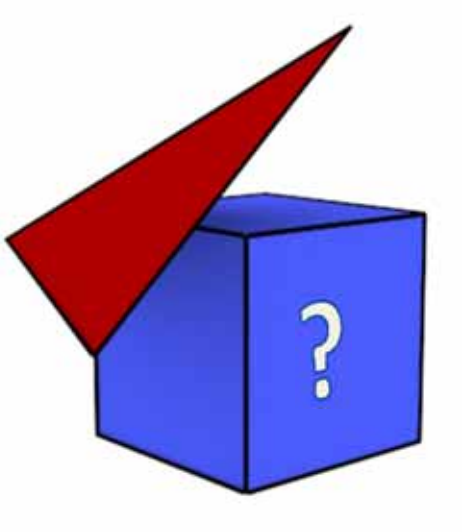
- **Inkrementiere/dekrementiere Stencil für Vorder- bzw. Rückseiten**

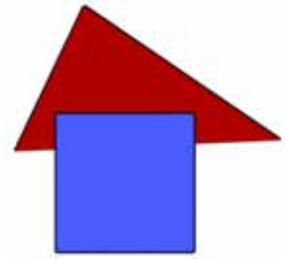
```
glStencilFunc(GL_ALWAYS, 1, 0xffffffff); // Stencil Operation immer ausführen

glFrontFace(GL_CCW);
glDisable(GL_CULL_FACE);

// zwei-seitige Stencil-Operation
glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_KEEP, GL_INCR);
glStencilOpSeparate(GL_BACK, GL_KEEP, GL_KEEP, GL_DECR);
```

Konservative Oberflächen-Voxelisierung

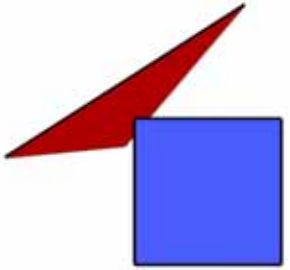
- **Grundidee:** teste, ob ein Voxel von der Oberfläche geschnitten wird

- Lösung: **Box-Triangle Schnitttest** (teste jeden Voxel gegen jedes Dreieck)
  - wir benötigen nicht die Schnittpunkte selbst (teuer zu berechnen)
  - ein Dreieck **T** und ein (axis-aligned) Quader **B** überlappen gdw.
    a) die Ebene in der **T** liegt sich mit **B** überlappt (liegt also mind. ein Eckpunkt vor und mind. einer hinter der Ebene) und
    b) wenn sich die 2D-Projektionen von T und B in jeder Koordinatenebene (XY, YZ, XZ) überlappen

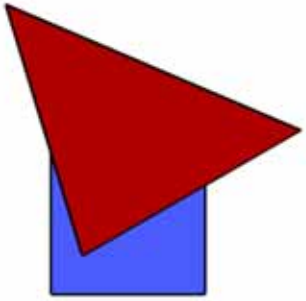




XY



YZ



XZ

Konservative Oberflächen-Voxelisierung

Schnitttest Ebene-Quader

- geg. Ebene mit Punkt $\boldsymbol{q}$ und Normale $\boldsymbol{n}$, und
- Quader durch
  - minimale und maximale Koordinaten $\boldsymbol{p}_{min}$ und $\boldsymbol{p}_{max} = \boldsymbol{p}_{min} + \boldsymbol{\Delta p}$
  - oder seine Eckpunkte $\boldsymbol{c_i}$, $i = 0..7$

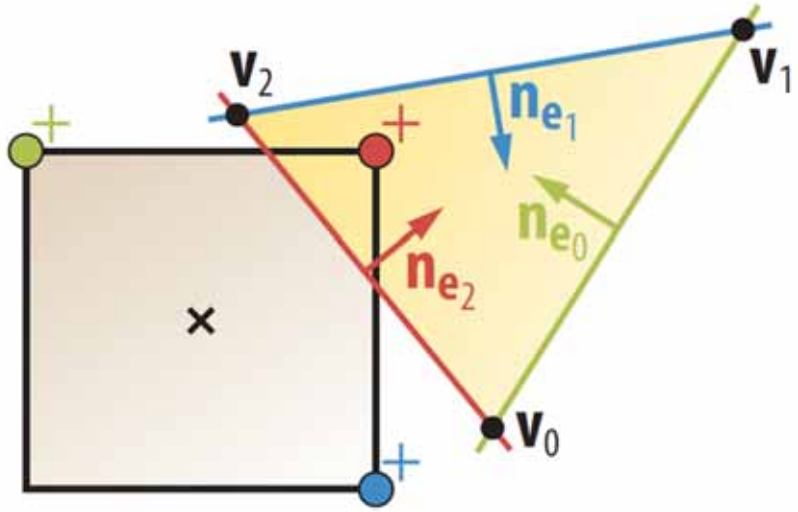
- Quader-Ebenen-Schnitt
  - es existiert ein Schnitt, wenn für zwei Ecken des Quaders $\boldsymbol{c_i}$ und $\boldsymbol{c_j}$ sich die Vorzeichen der impliziten Ebenengleichung unterscheiden:

$$\langle \boldsymbol{n}, \boldsymbol{q} - \boldsymbol{c_i} \rangle \langle \boldsymbol{n}, \boldsymbol{q} - \boldsymbol{c_j} \rangle \leq 0$$

Konservative Oberflächen-Voxelisierung

2D-Rechteck-Dreieck-Schnitt (für 2D-Projektionen von T und B)

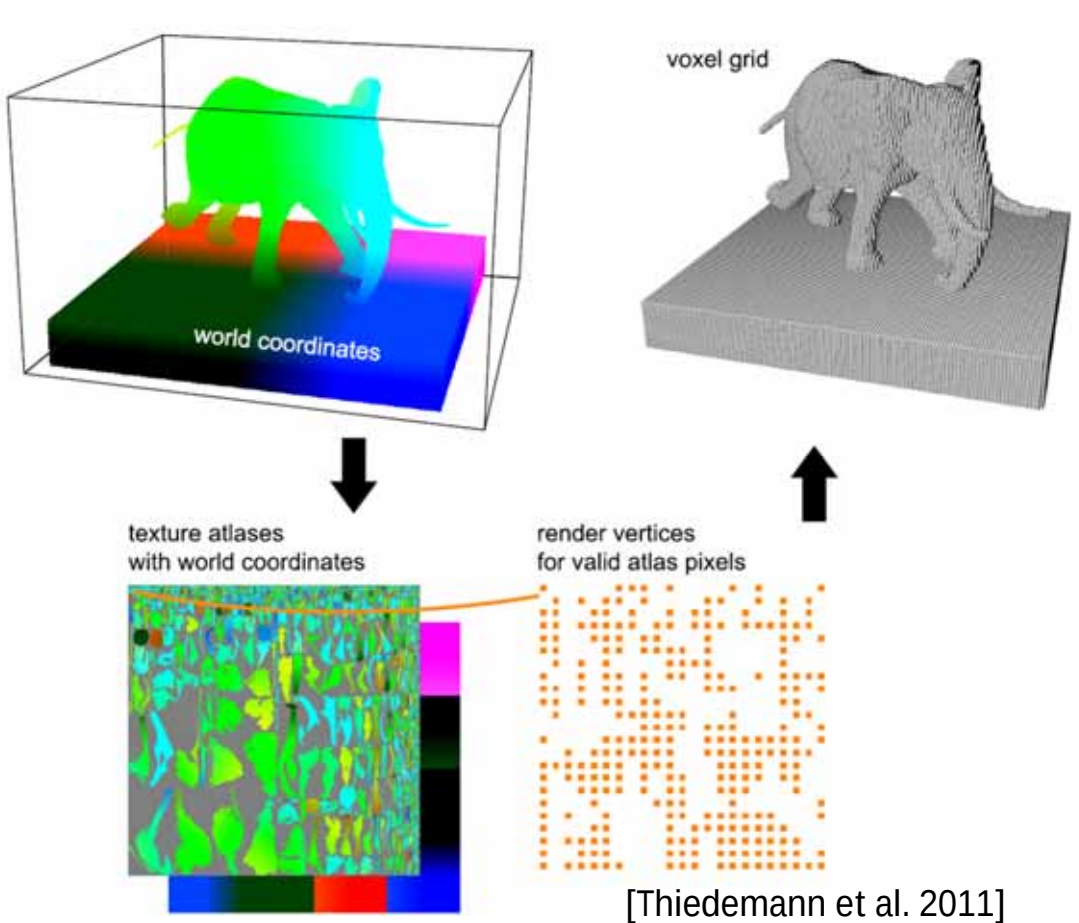
- verwende Kantengleichungen (vgl. Rasterisierung) um festzustellen, ob ein Punkt innerhalb des Dreiecks liegt
- wenn alle 3 Tests für einen (oder mehrere) Eckpunkt(e) des Rechtecks positiv sind, existiert ein Schnitt
- weiterer Test: liegen $\boldsymbol{v_0}$, $\boldsymbol{v_1}$ und $\boldsymbol{v_2}$ im Innern des Rechtecks



- Achtung: es müssen immer alle 2D-Projektionen betrachtet werden

Atlas-basierte Oberflächenvoxelisierung

- ein einfacher Ansatz mit herkömmlicher Rasterisierung
- Idee: wenn es für eine Szene einen Texturatlas gibt, können wir für jeden (belegten) Texel die Oberflächenposition bestimmen und ihn als Punktprimitiv in ein 3D-Volumen (bzw. die Schichten) zeichnen
- mögliche Probleme
  - Projektion von Texturraum in Weltkoordinaten ist i.A. nicht uniform, daher können Löcher in der voxelisierten Fläche auftreten
  - verstärkt bei animierten Objekten



[Thiedemann et al. 2011]

Binäre Oberflächenvoxelisierung

- bisher speichern wir für eine Voxelzelle nur ob sie belegt ist
  - diese „binäre Voxelisierung“ ist für Schattentests ausreichend
  - für globale Beleuchtung speichert man i.d.R. noch Farbe, Normale etc.
  - mit den bisherigen Techniken machbar

- direkte **binäre** Voxelisierung (Fast Scene Voxelization and Applications, Eisemann und Décoret, 2006)
  - Idee: ein Integer kann Belegung eines Voxels in jedem Bit speichern
  - zeichne Objekte in einen Integer-Framebuffer (z.B. ein 32-Bit Wert)
  - bei der Rasterisierung bildet ein Fragment Programm die Tiefe des Fragments auf einen Integer-Wert ab: gesetztes Bit entspricht diskretisierter Tiefe
  - verwende Bit-Operationen für Aktualisierung der Voxelisierung (unterstützt durch OpenGL und ab DirectX 11.1)

```
glEnable(GL_COLOR_LOGIC_OP);
glLogicOp(GL_OR);
```