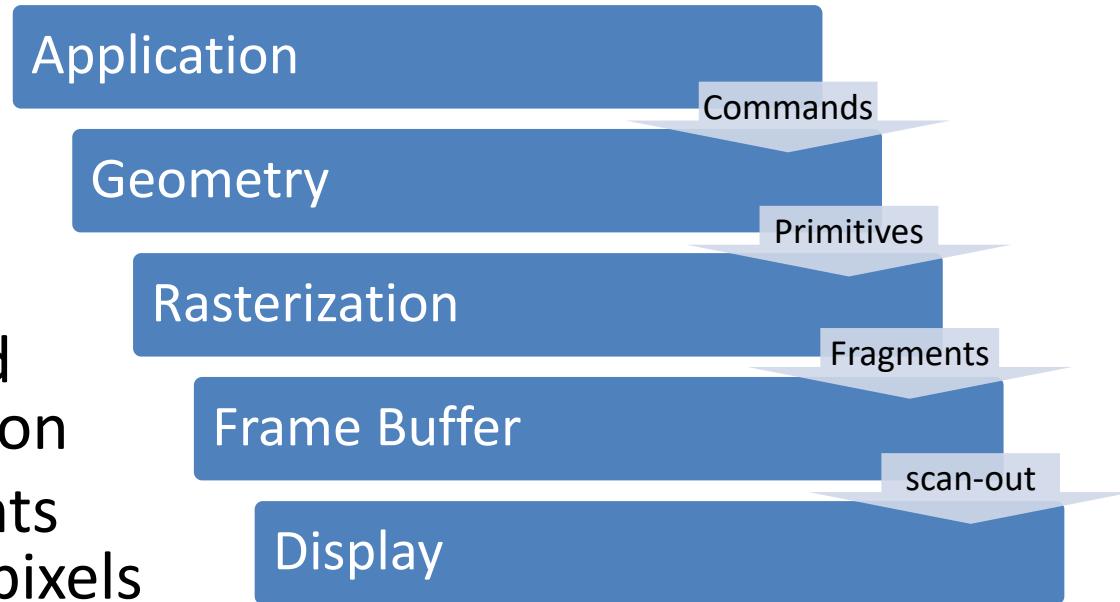


The Graphics Pipeline

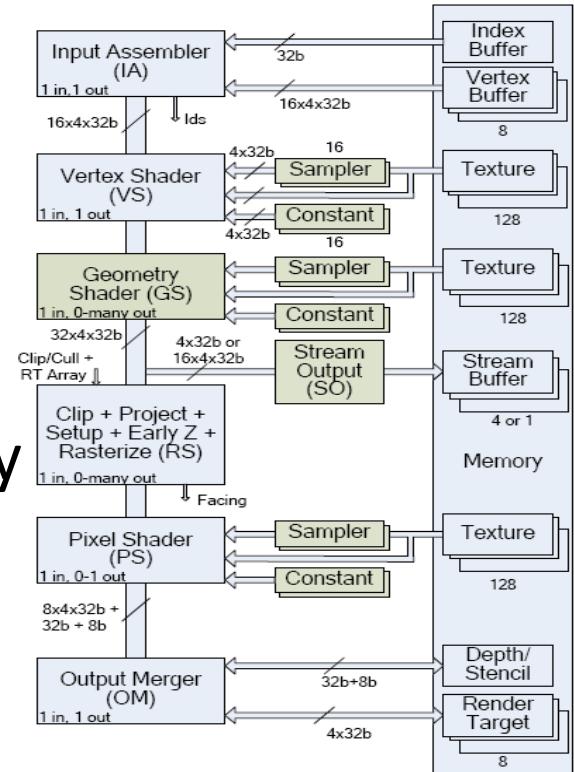
- High-level view:

- “Fragment”:
 - Sample produced during rasterization
 - Multiple fragments are *merged* into pixels



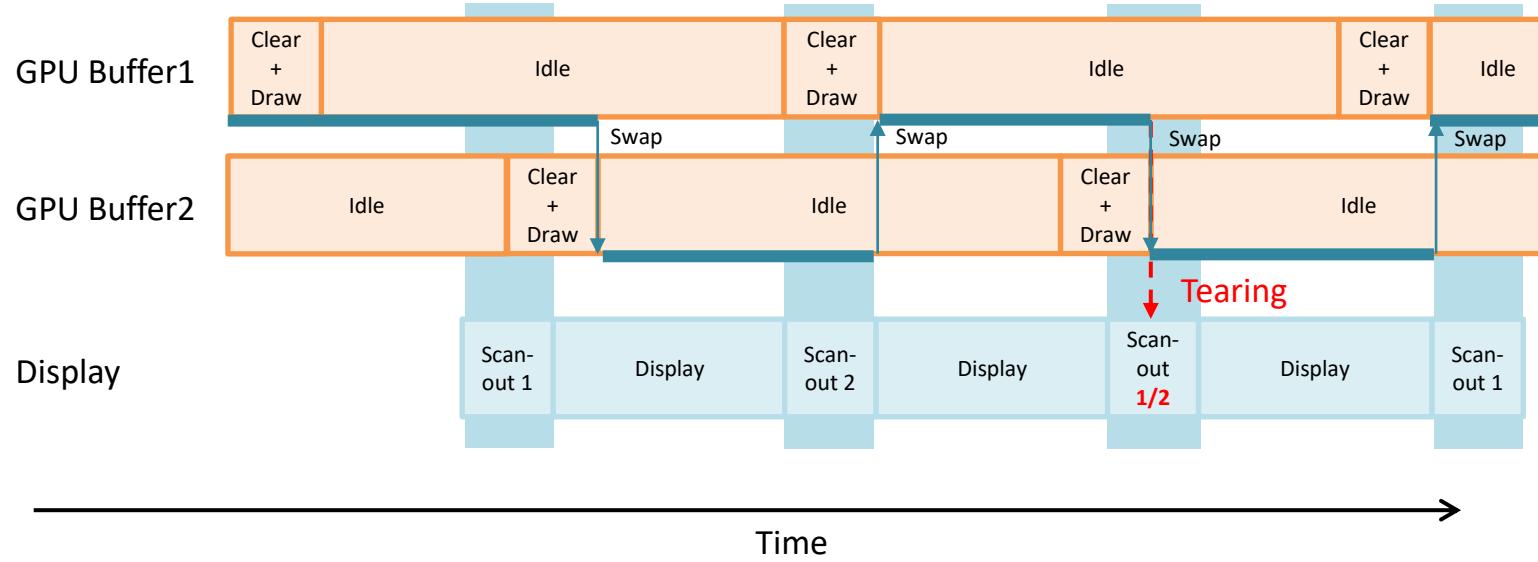
Geometry Shader

- Between vertex and pixel shader
- Can generate primitives dynamically
- Procedural geometry
 - E.g., growing plants
- Geometry shader can write to memory
 - Called „stream output“
 - Enables multi-pass for geometry



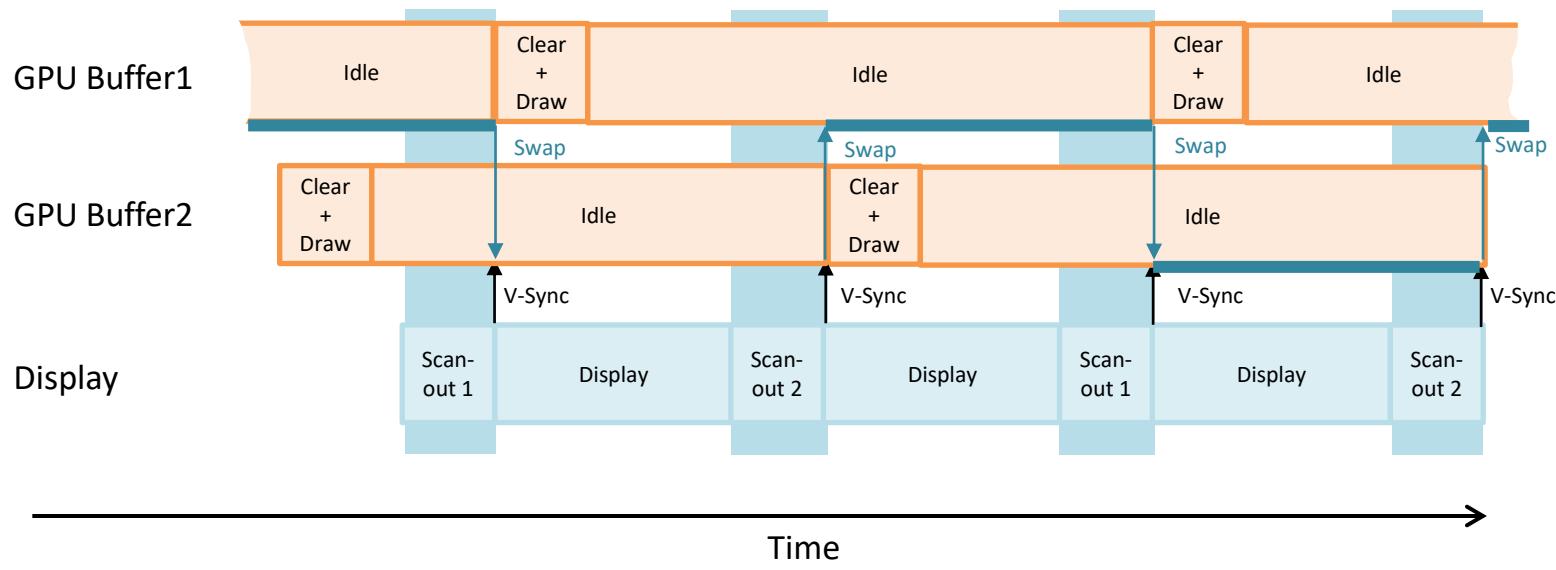
Double Buffering without V-Sync

- Front buffer used for scan-out GPU → display
- SwapBuffers() changes front and back buffer
- No flickering, but **tearing** artefacts



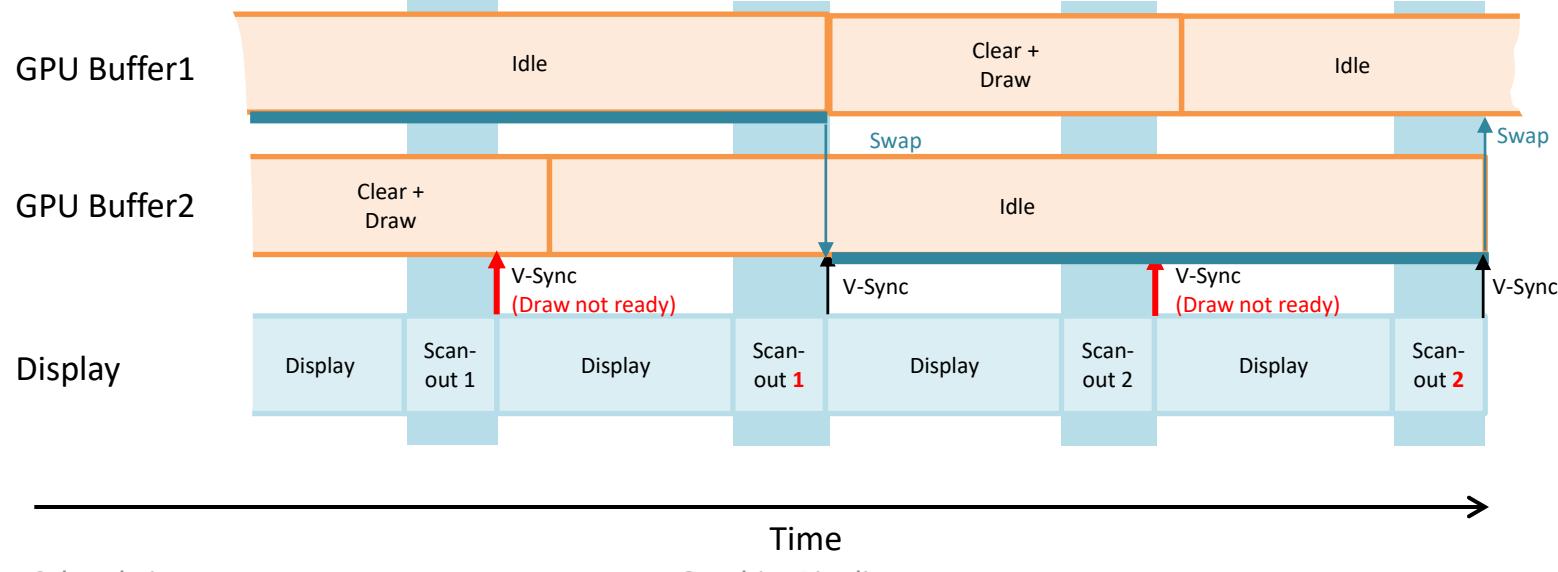
Double Buffering with V-Sync, Fast

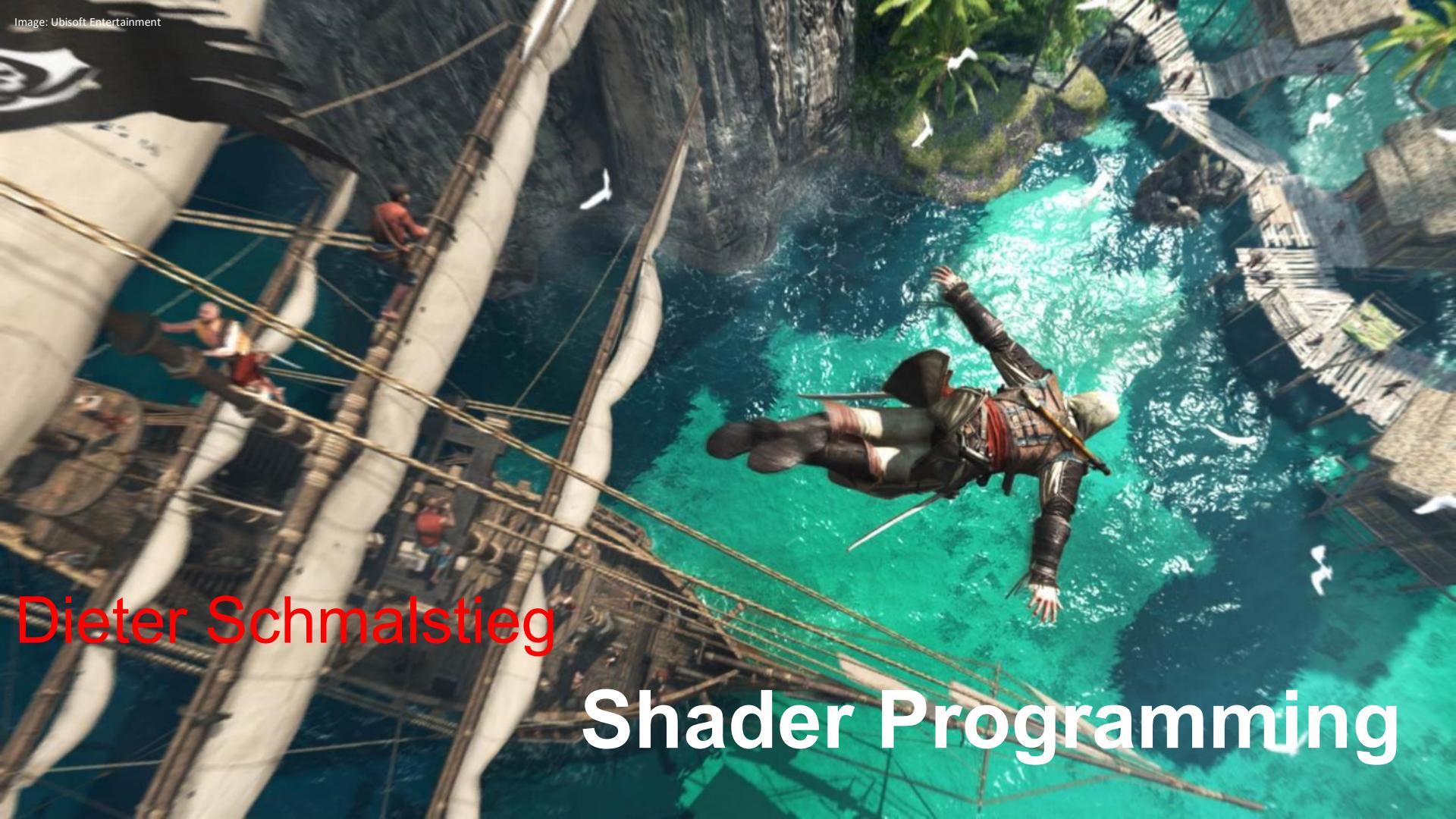
- V-Sync means the display is done with frame scan-out from GPU
- When rendering is **fast**, the frame rate is limited by display rate
- **Additional latency** of max. one frame time



Double Buffering with V-Sync, Slow

- When rendering is **slow**, frame rates can **only be integer fractions of display rate**
- Display rate 60Hz → frame rates 60, 30, 20, 15, ... (but not 59) possible
- Adaptive V-Sync (NVIDIA) turns off V-Sync automatically if rendering is slow





Dieter Schmalstieg

Shader Programming

Traditional API

- **Graphics context:** the system object in a traditional API
 - OpenGL, DirectX11 and below
 - Represents a virtual GPU
 - One process can have multiple contexts
 - Multiple contexts can share resources
- **Current context** for a given process
 - One to one mapping
 - Maximum of one current context per thread
 - Current context only assigned to one thread at the same time
 - All OpenGL operations work on current context

Modern API

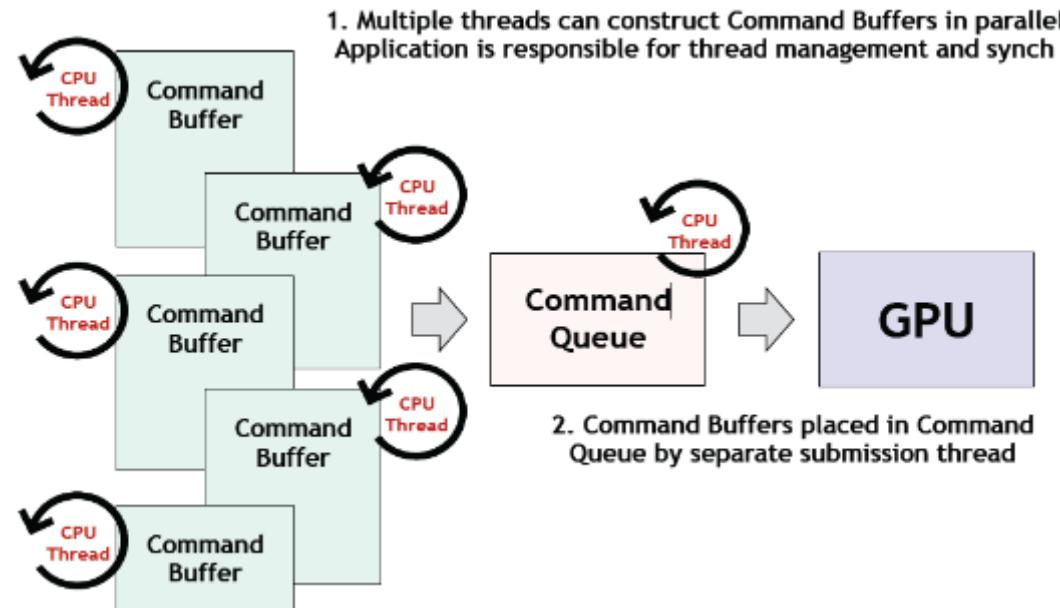
- DirectX 12, Vulkan (=OpenGL successor)
- Designed for modern GPU types (including mobile)
- Make (CPU driver side of) pipeline more programmable
- CPU multi-threading possible (at your own risk)
- Split rendering context into command buffers and queues
- Make pipeline state (and render passes) explicit
- Low overhead
- Fine-grained control (minimal program 1K lines of code)

Command Buffers

- Commands collected in command buffers
 - Optimize and validate command buffers during building, not during submission
 - Yields *immutable*, re-useable pipeline state configurations
 - Selected pipeline state variables can be declared *mutable*
- Command buffer submitted to *queue* for execution
- Build *many* command buffers from *many* threads
 - When the buffers are ready, one can submit them all at once
- Each command buffer just *switches* to its favorite pipeline
- Can use *synchronization* primitives across command buffers
 - Event, barrier, semaphore, fence

Queues

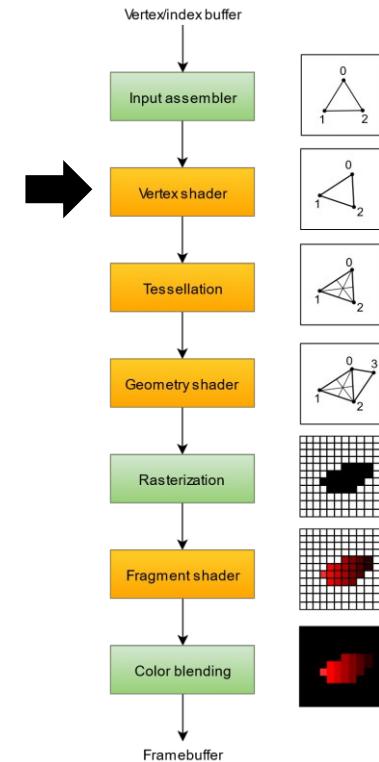
- Queues replace traditional contexts
- Insert command buffer into queue to schedule it



Vertex Shader

- Processes each vertex
- Input: vertex attributes
- Output: vertex attributes
- Mandatory output: `gl_Position`

Blinn-Phong Vertex Shader



Fragment Shader

- Processes each fragment
- Input: interpolated vertex attributes
- Output: fragment color

Phong, Blinn-Phong Fragment Shader

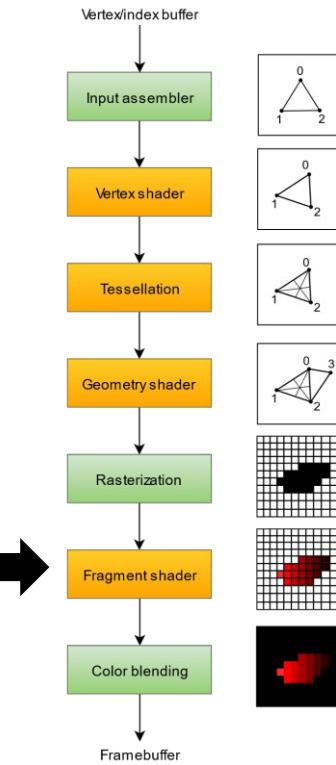


Image source: Epic Games



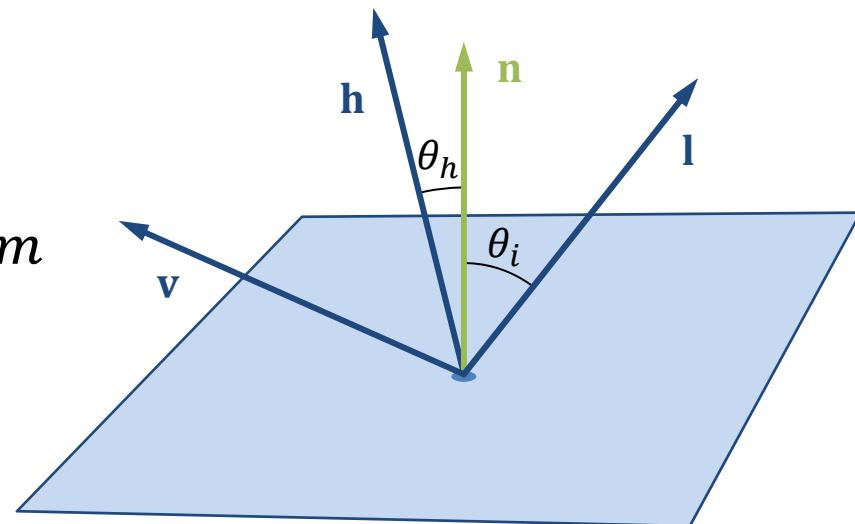
Shading Models

Es berechnet die Beleuchtung auf einer Oberfläche anhand der Intensität des einfallenden Lichts, der Materialeigenschaften (z.B. Farbe und Oberflächenbeschaffenheit) und der Richtung des beobachtenden Auges.

Blinn-Phong Model

- Approximates energy conservation
- $f_r = \frac{c_d}{\pi} + \frac{m+8}{8\pi} c_s (\cos \theta_h)^m$
 - c_d Diffuse color
 - c_s Specular color
 - m Specular power

All vectors assumed to be normalized!



$$\mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}$$

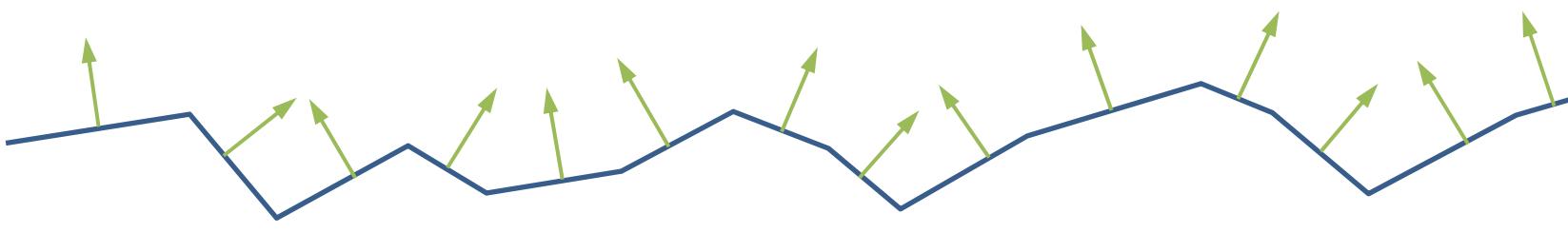
Der Half-Vector ist ein vereinfachter Schätzwert für die Richtung des spiegelnden Lichts, der berechnet wird, indem man die Richtung des einfallenden Lichts und die Richtung des beobachtenden Auges miteinander verbindet. Dieser Half-Vector wird verwendet, um die Reflexionen auf der Oberfläche zu berechnen, anstatt den exakten Glanzwinkel zu berechnen, wie es beim Phong-Modell der Fall ist.

Fresnel Laws

- What happens at boundary between two media
 - Phase speed of light different in each medium
- Part of the light wave is directly *specularly* reflected
- Part of the light wave is transmitted
 - We assume the material is not transparent
 - The transmitted part becomes the *diffuse* reflection
- What is the exact ratio?
 - *Augustin-Jean Fresnel knows.*

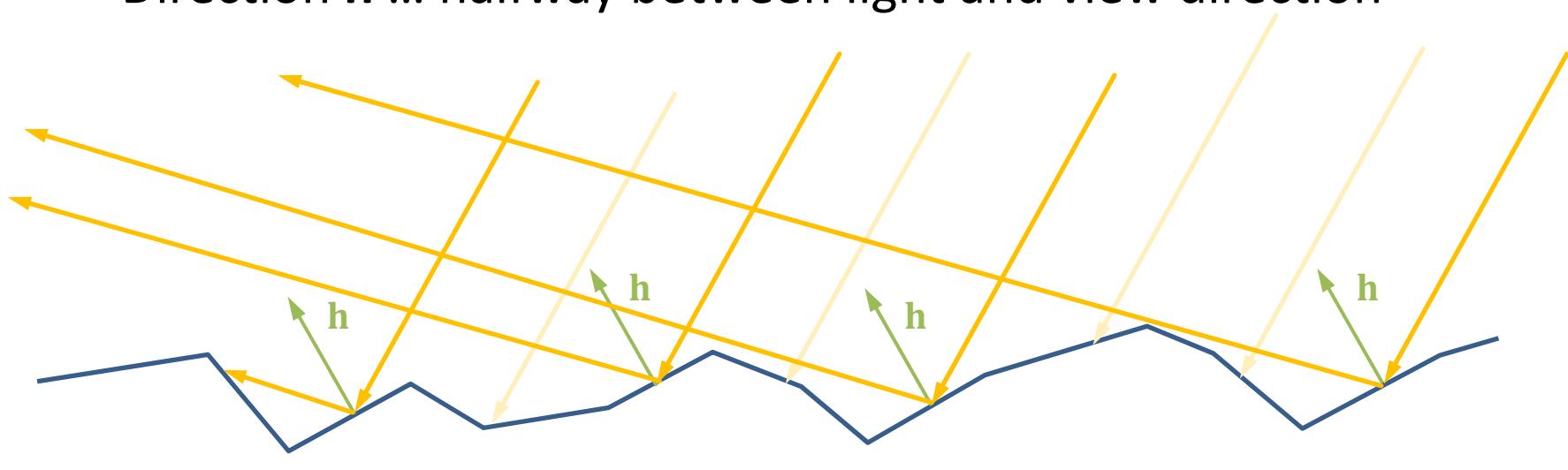
Microfacets

- Surface assumed to be made up of tiny mirrors
 - Lots of tiny mirrors
- Need to model the distribution of mirrors



Half-Vector

- Only mirrors oriented halfway between light and view direction reflect light into camera
- Direction \mathbf{h} ... halfway between light and view direction



Microfacet Distribution

- Fraction of microfacets oriented in direction \mathbf{h}
- Example: Beckmann Distribution

$$D(\mathbf{n}, \mathbf{h}, m) = \frac{1}{4m^2(\mathbf{n} \cdot \mathbf{h})^4} \exp\left(\frac{(\mathbf{n} \cdot \mathbf{h})^2 - 1}{m^2(\mathbf{n} \cdot \mathbf{h})^2}\right)$$

m Root mean squared slope of microfacets
(corresponds to roughness)

[Beckmann, Spizzichino 1963]

Geometric Attenuation

- Accounts for shadowing
 - Light blocked from reaching microfacet by other microfacets
- Accounts for masking
 - Reflected light blocked from reaching camera by other microfacets
 - Example: Torrance-Sparrow model

$$G(\mathbf{l}, \mathbf{v}) = \min\left(1, \frac{2 \cos \theta_h \cos \theta_o}{\cos \alpha_h}, \frac{2 \cos \theta_h \cos \theta_i}{\cos \alpha_h}\right)$$

$$\begin{aligned}\cos \theta_h &= \mathbf{h} \cdot \mathbf{n} \\ \cos \theta_o &= \mathbf{v} \cdot \mathbf{n} \\ \cos \alpha_h &= \mathbf{v} \cdot \mathbf{h}\end{aligned}$$

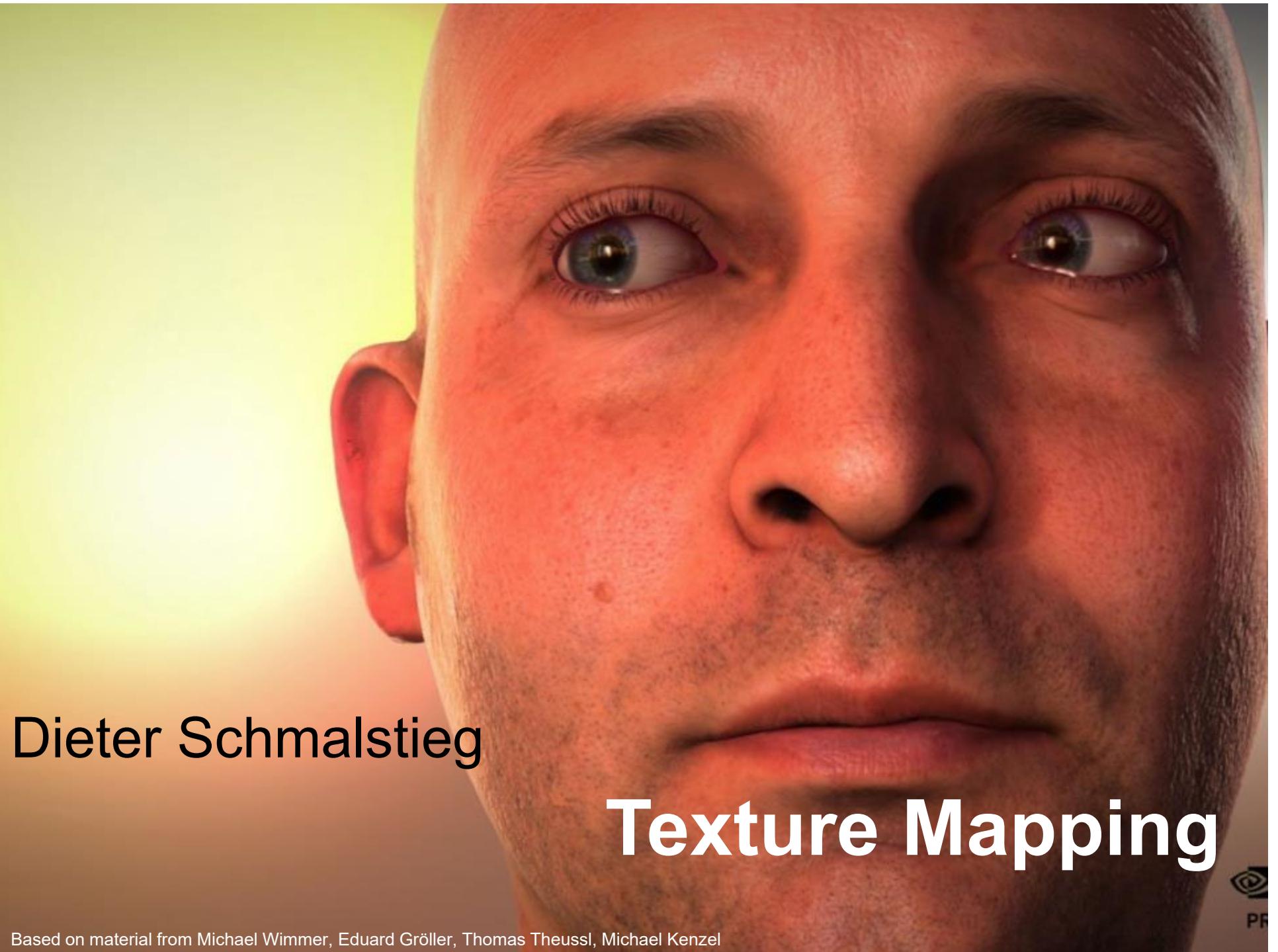
[Torrance, Sparrow 1967]

Cook-Torrance Model

Popular shading model which is putting all these components together

$$f_r(\mathbf{l}, \mathbf{v}) = \frac{D(\mathbf{h})F(\mathbf{l}, \mathbf{v})G(\mathbf{l}, \mathbf{v})}{4(\mathbf{n} \cdot \mathbf{v})(\mathbf{n} \cdot \mathbf{l})}$$

Microfacet distribution
 (accounts for surface roughness) Fresnel
 reflectance Geometric attenuation
 (amount of self-shadowing)



Dieter Schmalstieg

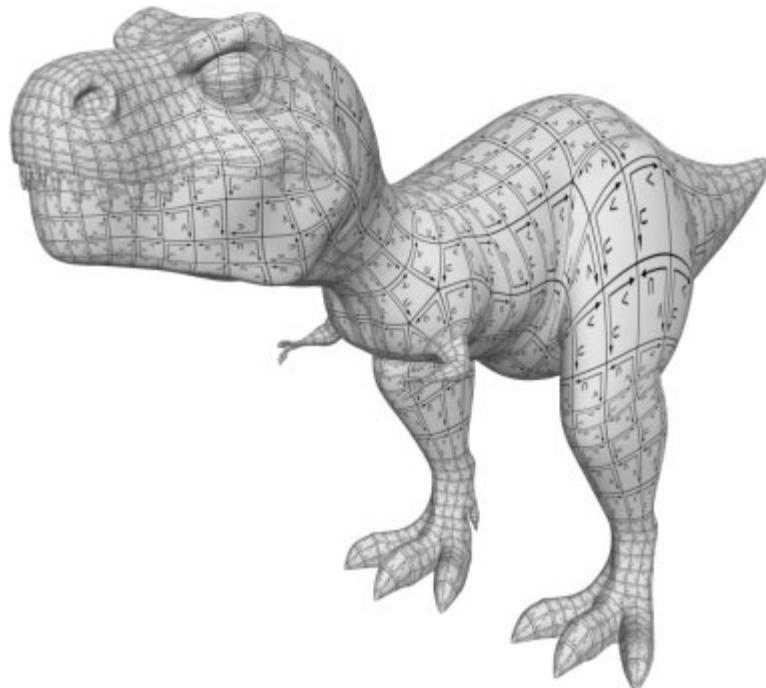
Texture Mapping

Based on material from Michael Wimmer, Eduard Gröller, Thomas Theussl, Michael Kenzel

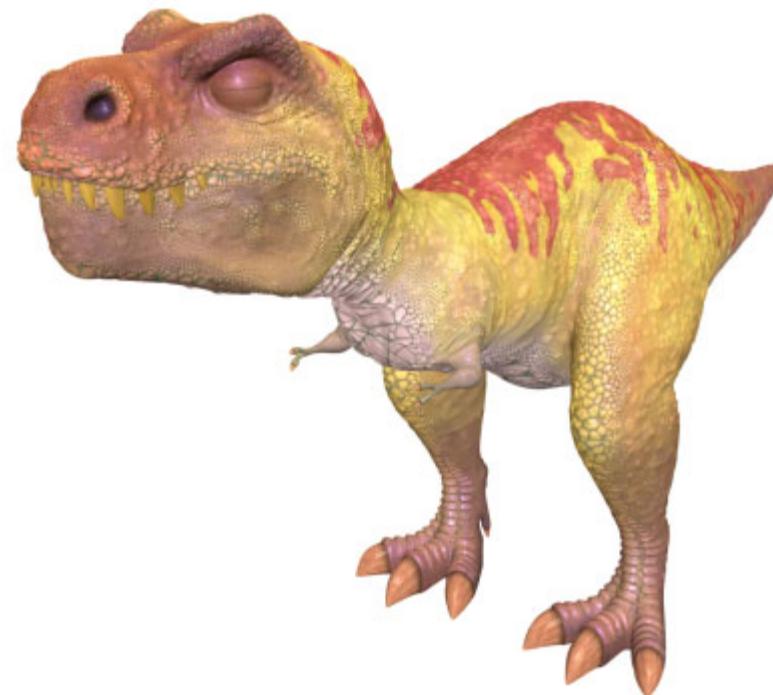


Why Texturing?

- Enhance visual appearance of plain surfaces and objects by applying fine structured details

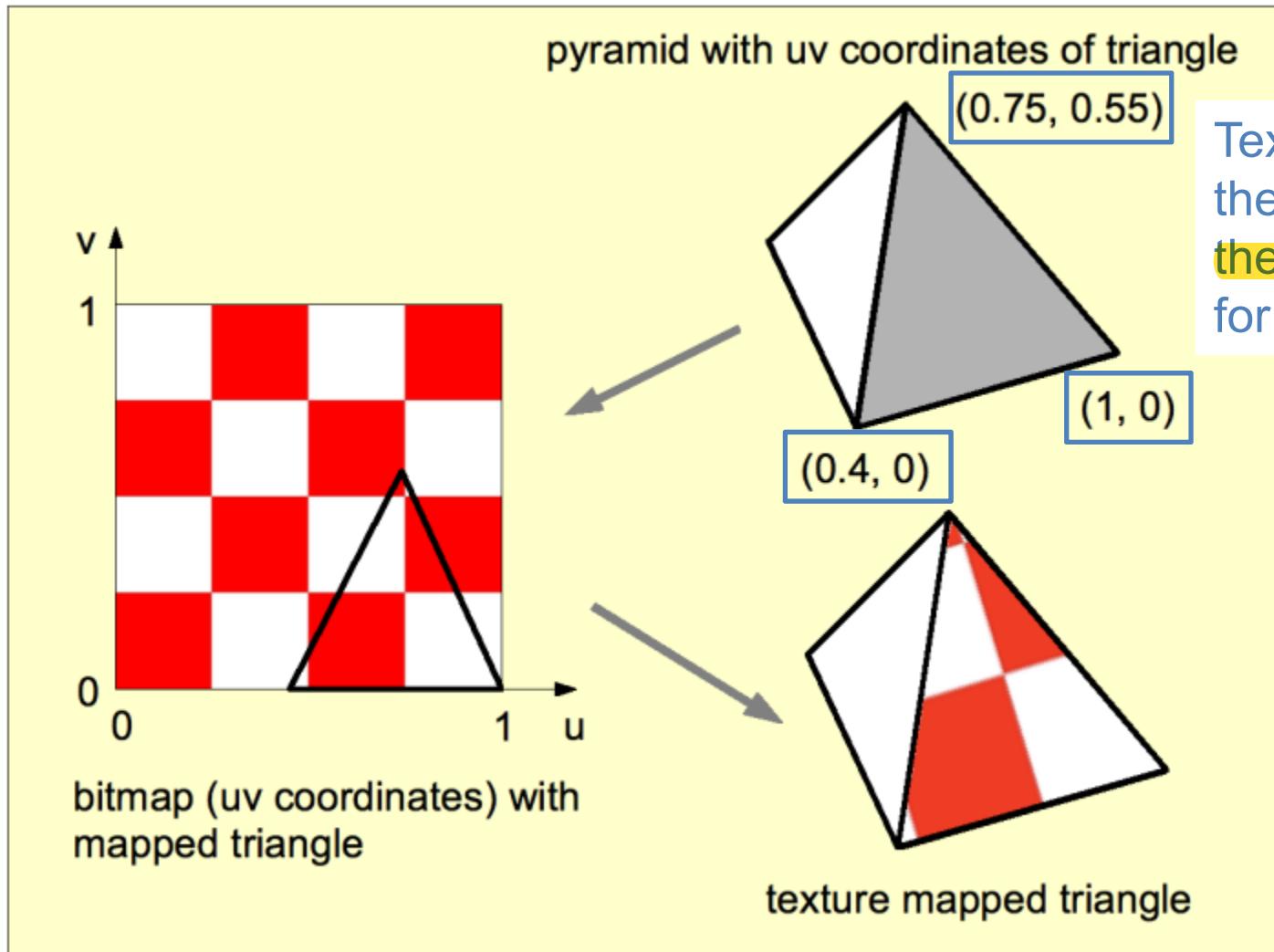


Dieter Schmalstieg



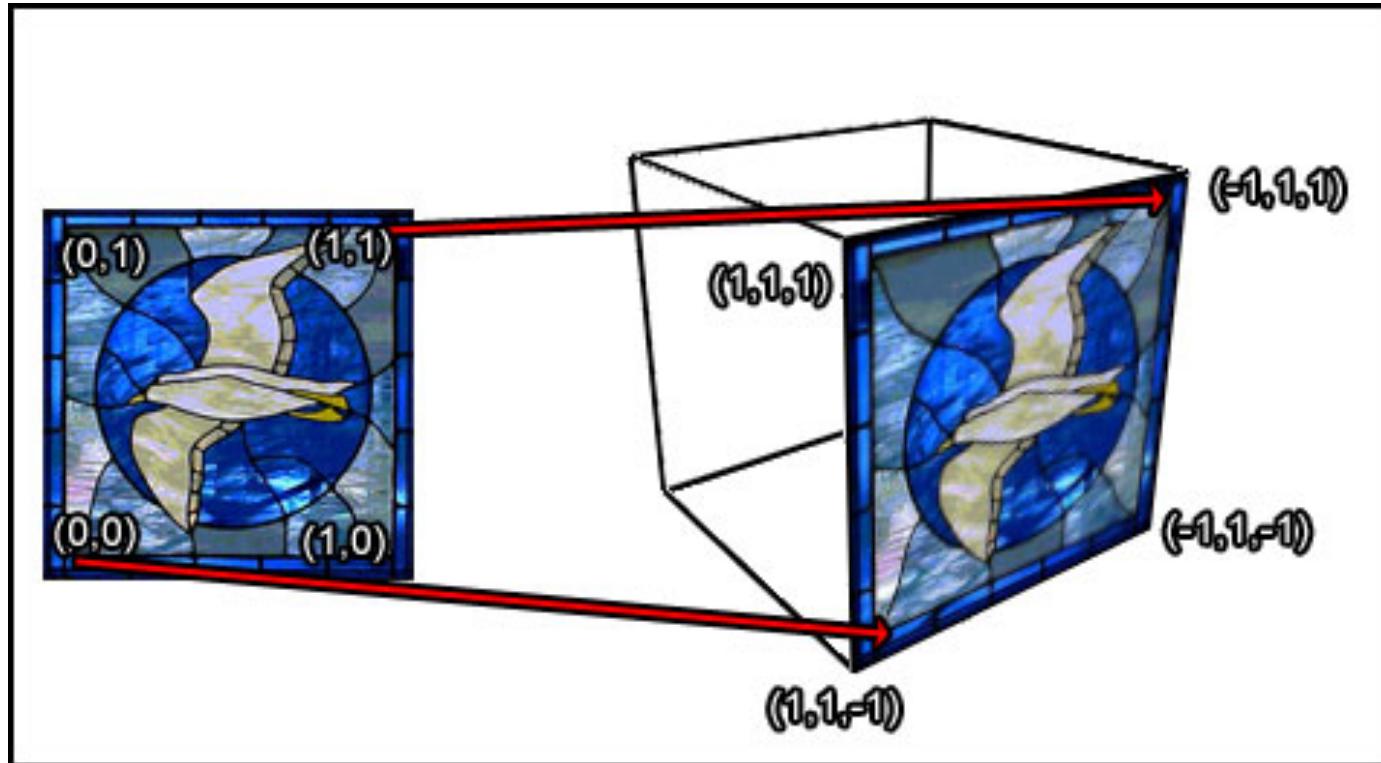
Texture mapping

Texture Coordinates



Parametrization

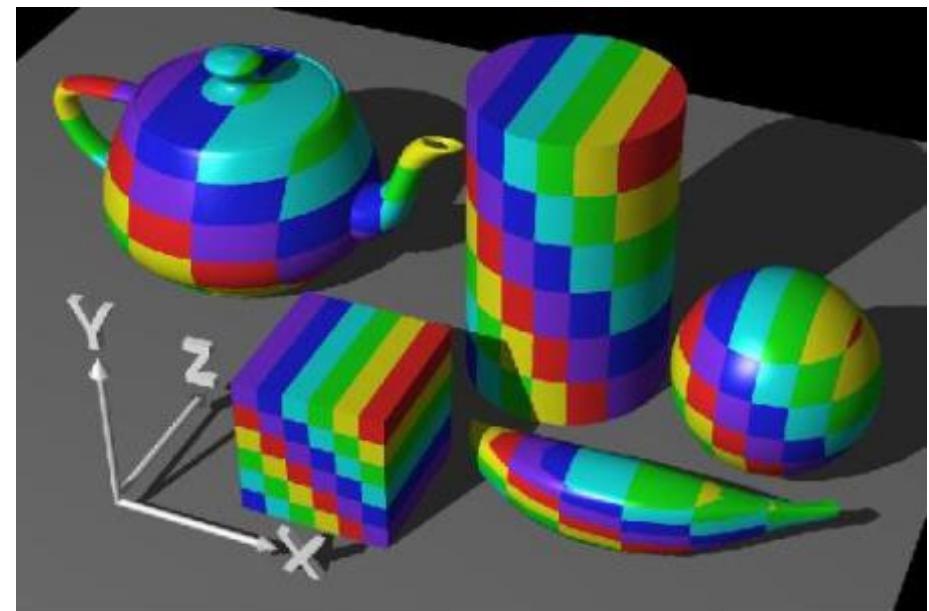
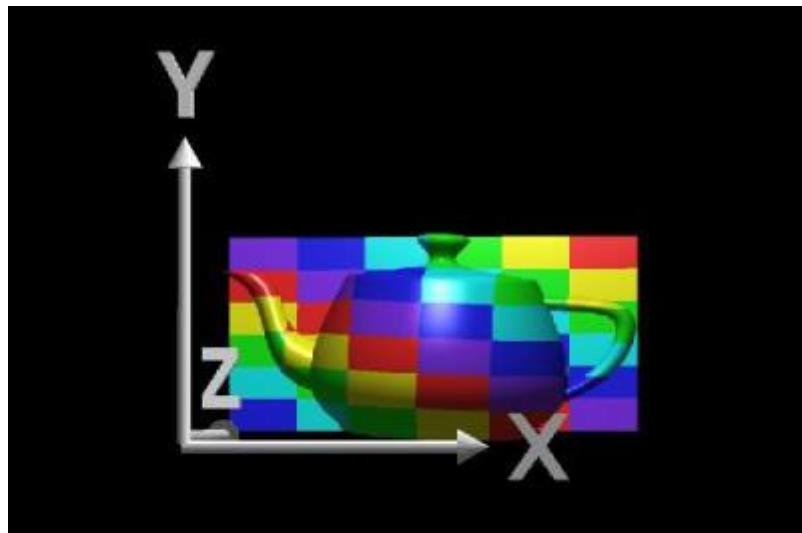
- How to do the mapping?



- Usually objects are not that simple

Parametrization: Planar

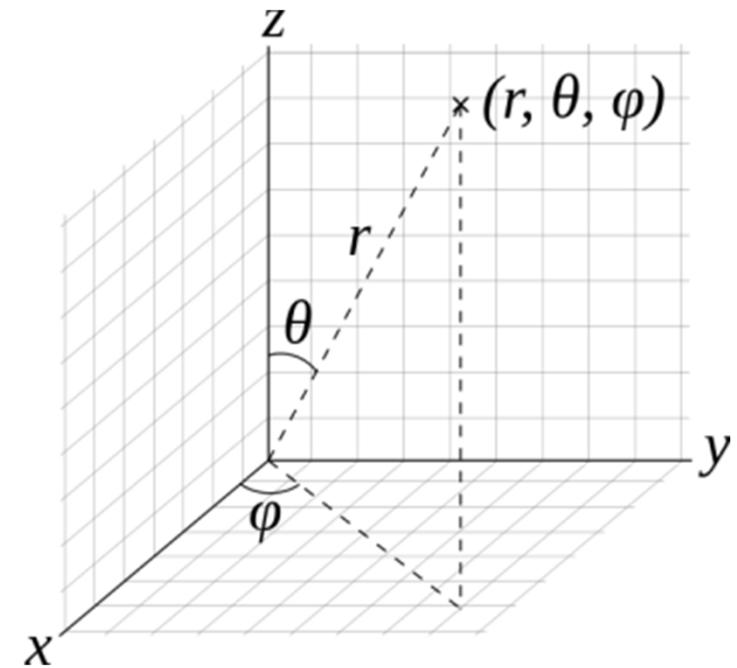
- Planar mapping: dump one of the coordinates



Only looks good from the front!

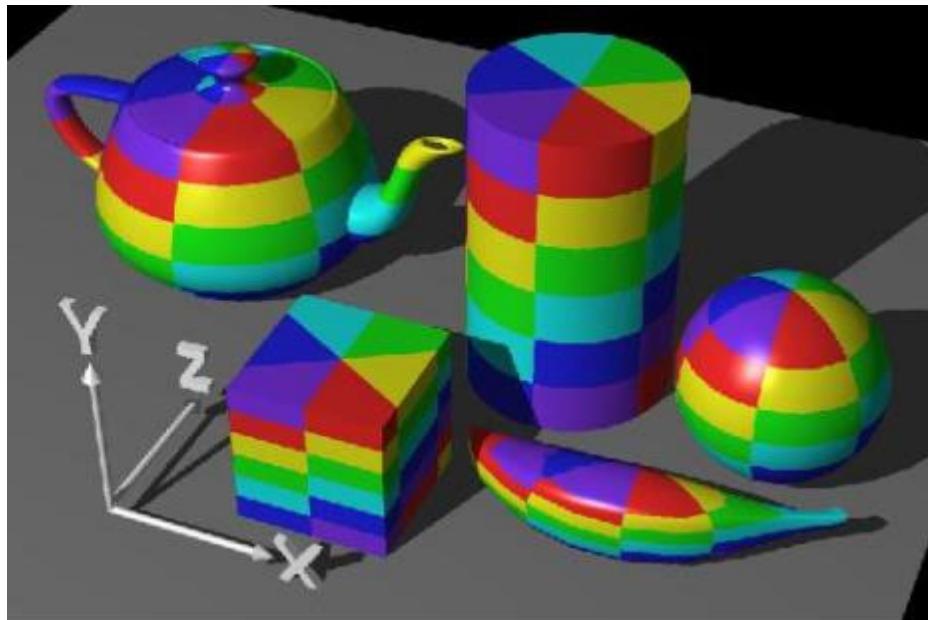
Parametrization: Cylindrical

- Cylindrical and spherical mapping: compute angles between vertex and object center
- Compare to polar/spherical coordinate systems

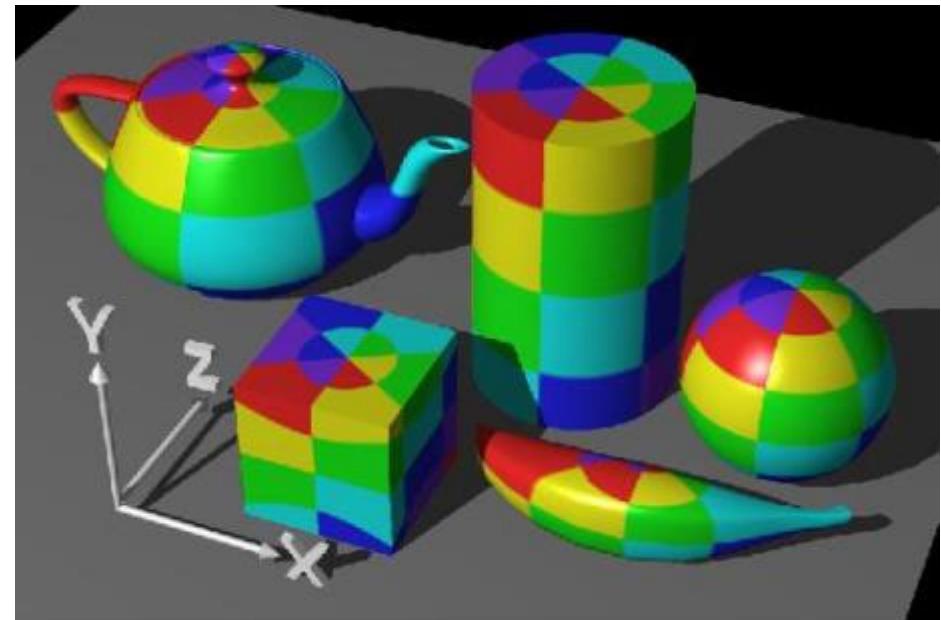


Parametrization: Polar

- Cylindrical and spherical mapping



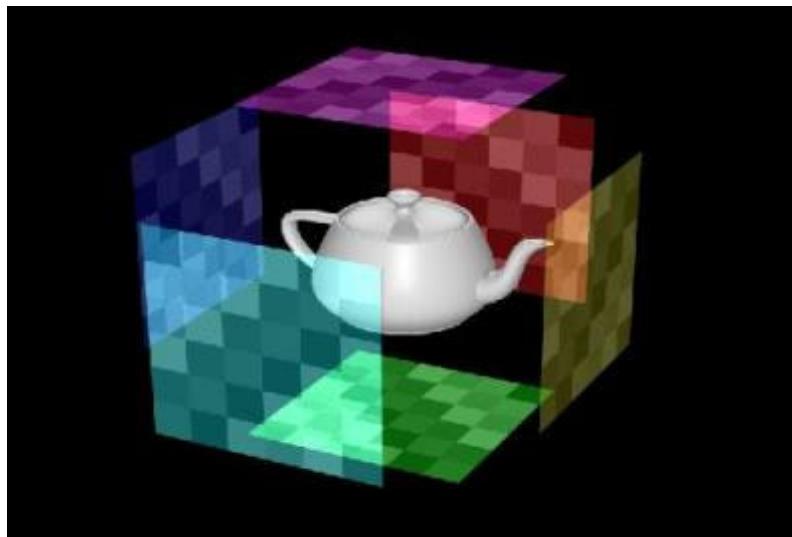
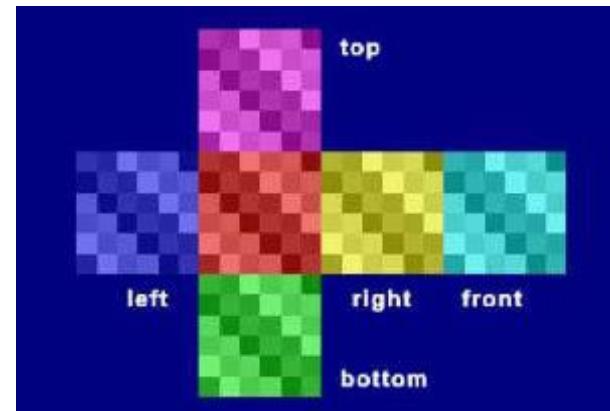
Cylindrical



Spherical

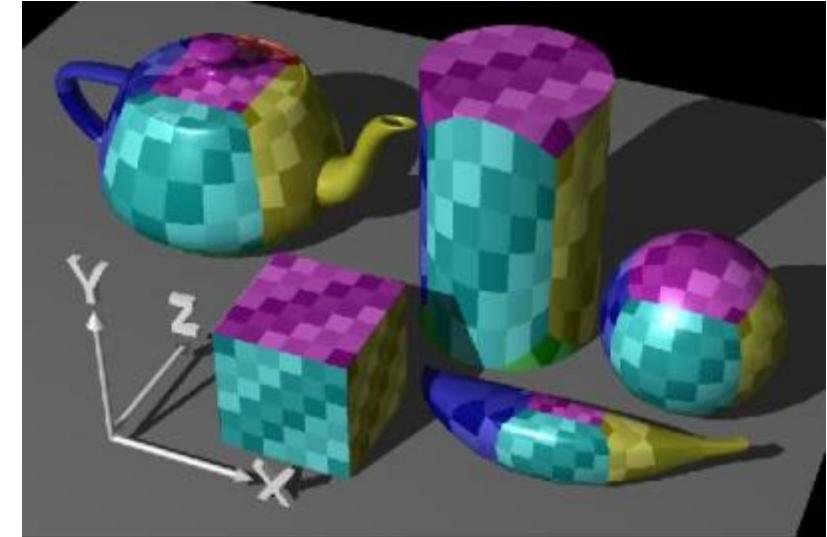
Parametrization: Box

- Box mapping: used mainly for environment mapping
(see later)



Dieter Schmalstieg

Texture mapping

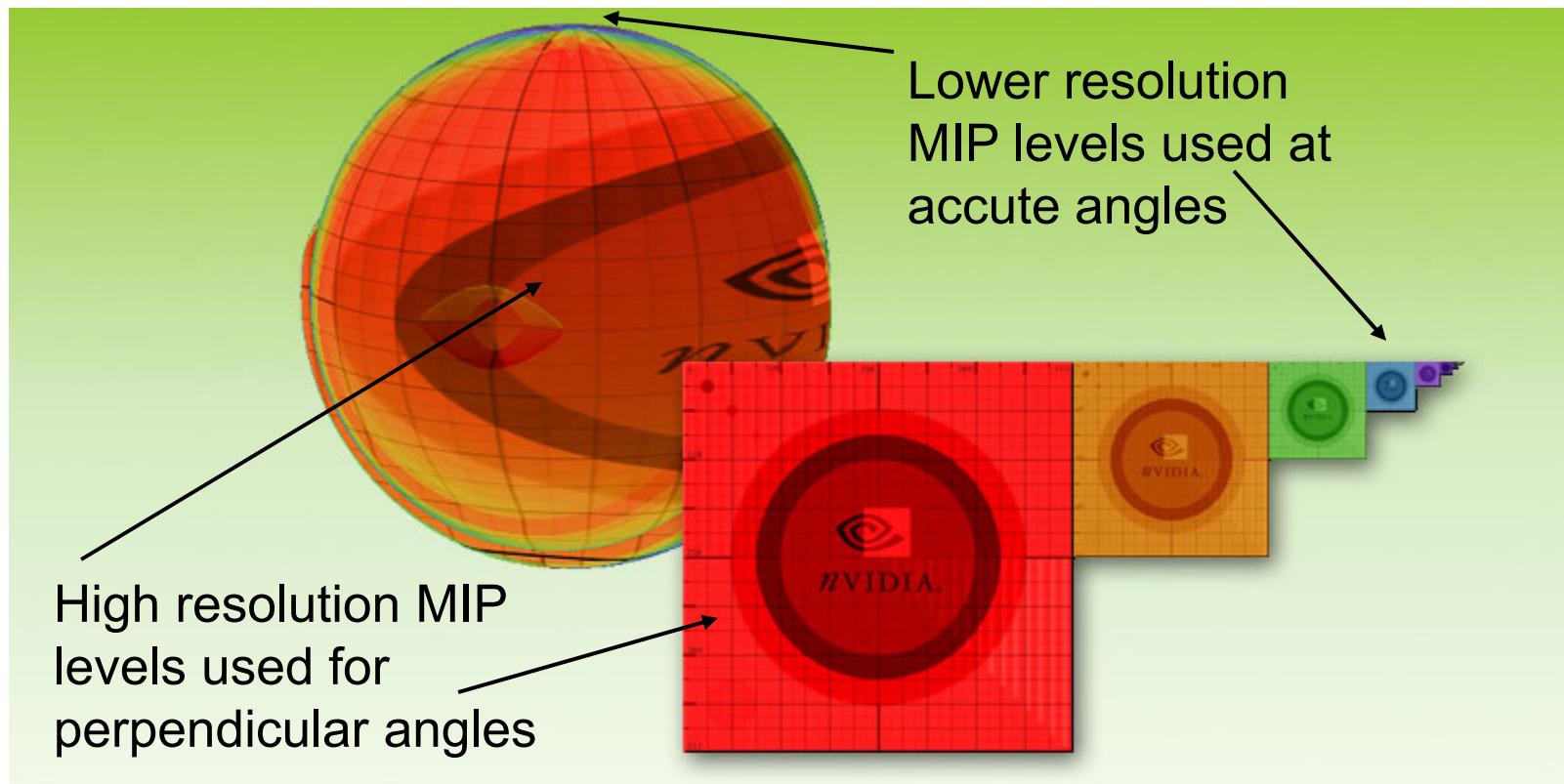


Mip Mapping

downsampling ⇒ much in a small area
= “multum in parvo” (mip)

- Texture is **precalculated for different sizes**
- Heavily used in real-time graphics
- Texture is **reduced by factors of two**
 - Simple
 - Memory efficient
- The **last image is only one texel**

Mip Mapping Example

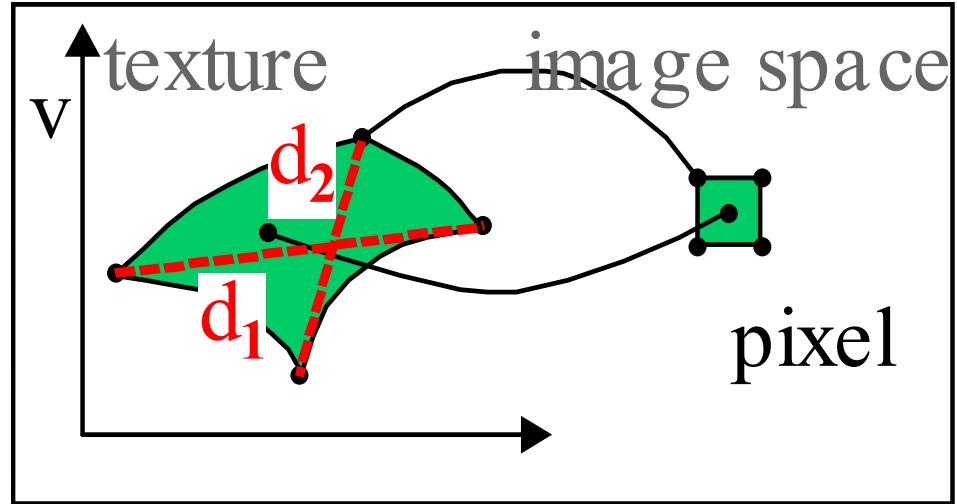


Mip Maps are used to reduce bandwidth requirements of distant textures and minimize scintillating pixels

Mip Mapping Algorithm

$$D := \text{Id}(\max(|d_1|, |d_2|))$$

D ... Level of Detail (Welche Texture auf der Pyramide)
 d ... Distanz zwischen Pixeleckpunkten im Texture Space
 Berechnet die Größe des Pixels im Verhältnis zur Textur



Passende Miplevel herausfinden durch Abrunden des Ergebnisses

$$T_0 := \text{value from table } D_0 = \text{trunc}(D)$$

$$T_1 := \text{value from table } D_1 = D_0 + 1$$

D0 und D1 sind Indizes für die verschiedenen Mip-Map-Stufen und .

- T0, T1 Texel auf den verschiedenen level?
- T_0, T_1 obtained with bilinear interpolation
 - How to combine T_0, T_1 ?

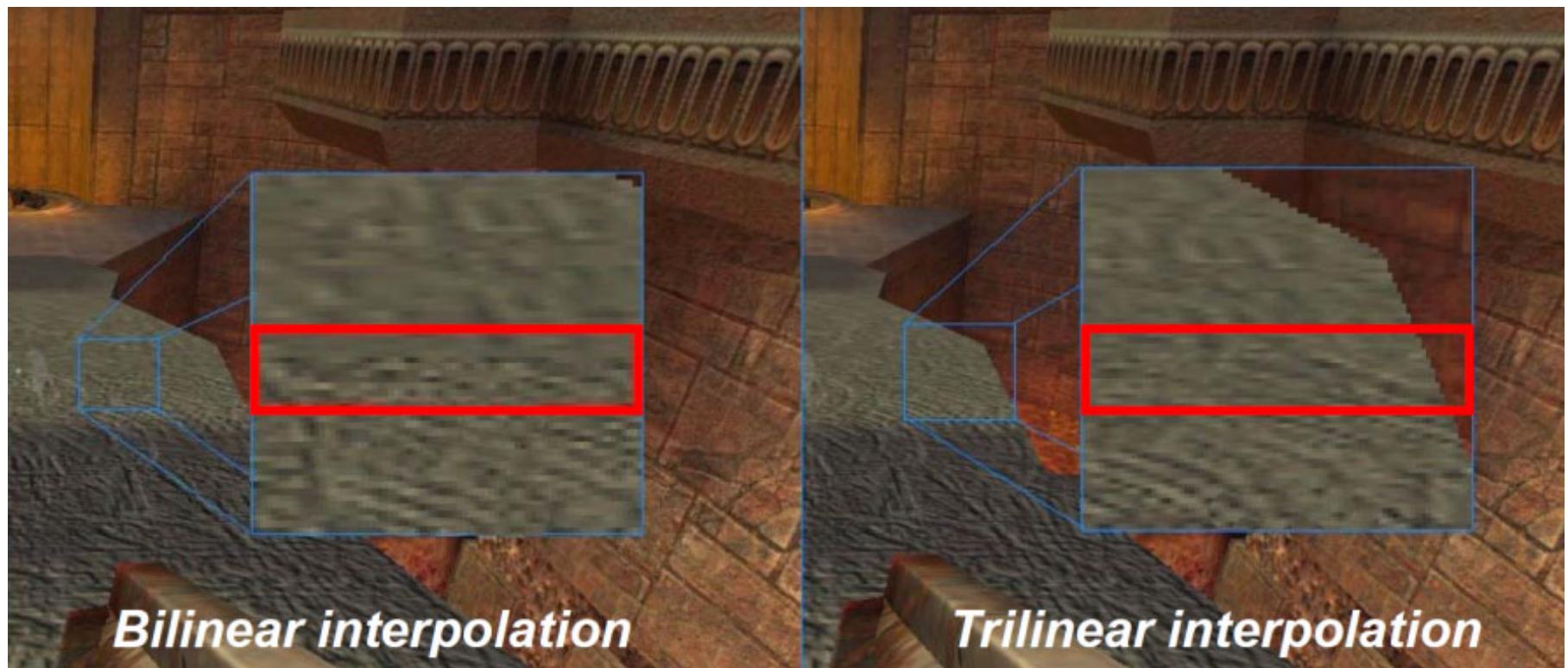
Bilinear Interpolation wird verwendet, um die Textur des Pixels auf der Grundlage dieser Mip-Map-Stufen zu berechnen

Bilinear vs Trilinear

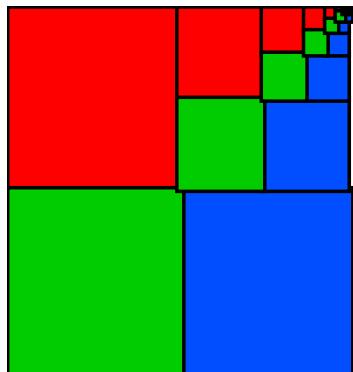
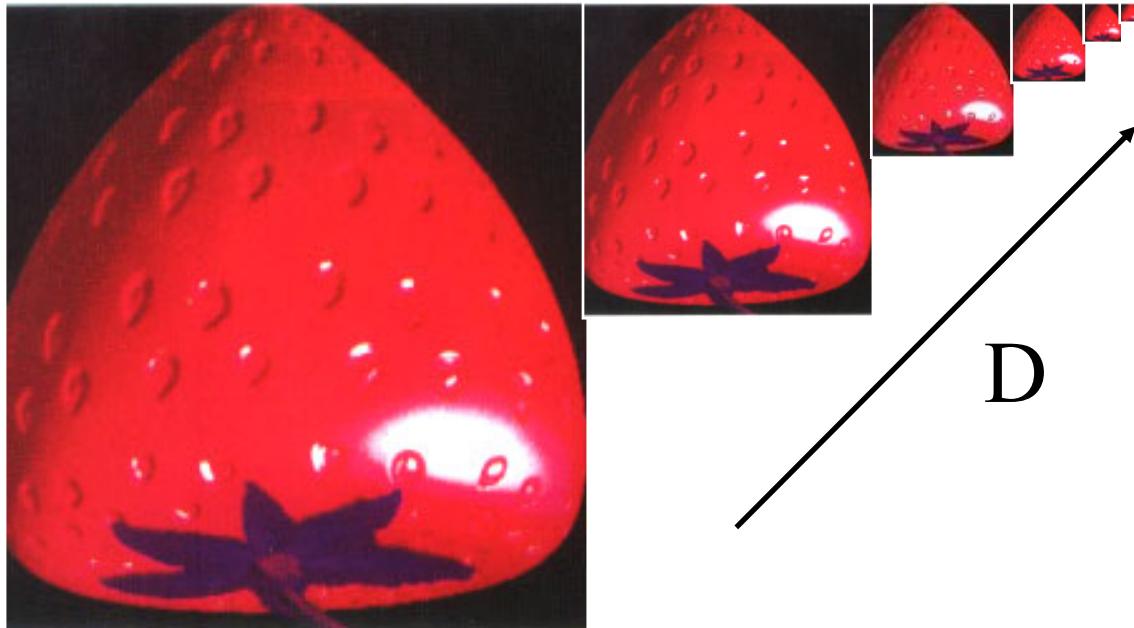
- **Bilinear** := $(D_1 - D > 0.5) ? T_1 : T_0$
- **Trilinear** := $(D_1 - D) \cdot T_0 + (D - D_0) \cdot T_1 =$
 $= (D_1 - D) \cdot (T_0 - T_1) + T_1$

Bilinear:
4 benachbarte Pixel auf gleicher Ebene Interpoliert

Trilinear:
8 benachbarte Pixel auf 2 Ebenen interpoliert



Mip Mapping Memory Layout



Memory overhead is just 33%
when channels are packed this way

Mip Mapping Upsampling

- When the texture resolution is too small:
Upsampling of the mip level with highest res.
- „**Magnification**“



Dieter Schmalstieg



Texture mapping

Multipass Rendering

- Basic GPU lighting model is
 - Local
 - Limited in complexity
- Many effects possible with multiple passes
 - Environment maps
 - Shadow maps
 - Reflections, mirrors
 - Transparency

Multipass rendering ist eine Technik in der Computergrafik, bei der ein 3D-Szenenbild in mehreren Schritten oder "Passes" berechnet wird. Jeder Pass fokussiert sich auf einen bestimmten Aspekt des Bildes und die Ergebnisse werden zusammengeführt, um das endgültige Bild zu erstellen.

Multipass Rendering: How?

Two main methods

- **Render to auxiliary buffers**, use result as texture
 - E.g.: environment maps, shadow maps
 - Requires FBO support
- **Redraw scene** using fragment operations
 - E.g.: reflections, mirrors, light mapping
 - Uses framebuffer blending
 - Uses depth, stencil, alpha, ... tests
- Can **mix both techniques**

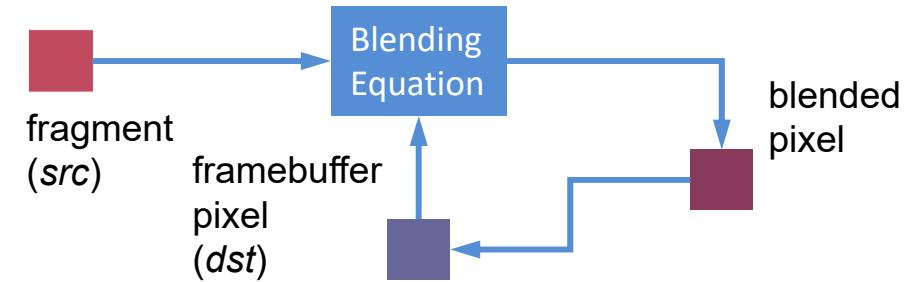
Multipass via Render to Texture

Auxiliary buffer method = *render to texture* (RTT)

1. Frame buffer with color attachment is bound
1. Framebuffer erstellen der einen Farbanhang als Renderziel hat
2. Scene (or subset) is rendered,
possibly using a custom shader program
2. Scene wird in Texture gerendert
3. Frame buffer is unbound
Color attachment is bound as texture
3. Nach dem Rendern wird der Framebuffer entbunden und der Farbanhang wird als Textur gebunden, um sie in weiteren Schritten verwenden zu können.
4. Scene is rendered to the back buffer,
Rendered texture can be used either by
fixed function pipeline or a custom shader
4. Die Szene wird schließlich auf dem Back-Buffer gerendert, wobei die gerenderte Textur entweder vom festen Funktionspfad oder einem benutzerdefinierten Shader verwendet wird, um das endgültige Render-Ergebnis zu erzeugen.

Blending

- So far, every new pixel simply overwrote existing framebuffer
- Blending **combines new pixels with what is already in the framebuffer**
- Blending does not always require alpha buffer

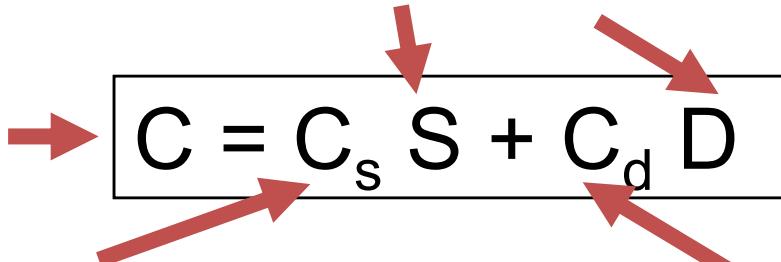


Multipass – Framebuffer Blending

- Select a blend equation
- Hardware supports a **finite number of equations**
- Most common is **linear blending**

$$C = C_s S + C_d D$$

weighting factors



The diagram shows the linear blending equation $C = C_s S + C_d D$ enclosed in a rectangular box. Four red arrows point to the components: one from the left labeled "result color", one from the top labeled "weighting factors", one from the bottom-left labeled "incoming fragment color", and one from the bottom-right labeled "framebuffer color".

Prüfungsfrage!

Blending Source and Destination

- Weights can be defined arbitrarily
- Alpha/color weights can be defined separately
- Choices of weight
 - *One, zero, dst color, src color, alpha, 1 - src color...*
- Example: transparency blending (window)

$$C = C_s \cdot \alpha + C_d \cdot (1 - \alpha)$$

Alpha Buffer

Transparency Buffer

- Measure of **opacity** to
 - Simulate translucent objects (glass, water, etc.)
 - Composite images
 - Antialiasing
- Do not forget to enable blending first!
- Beware
 - Usage of alpha requires depth-sorting of objects
 - Z-Buffer not helpful for partially transparent pixels

Light Mapping

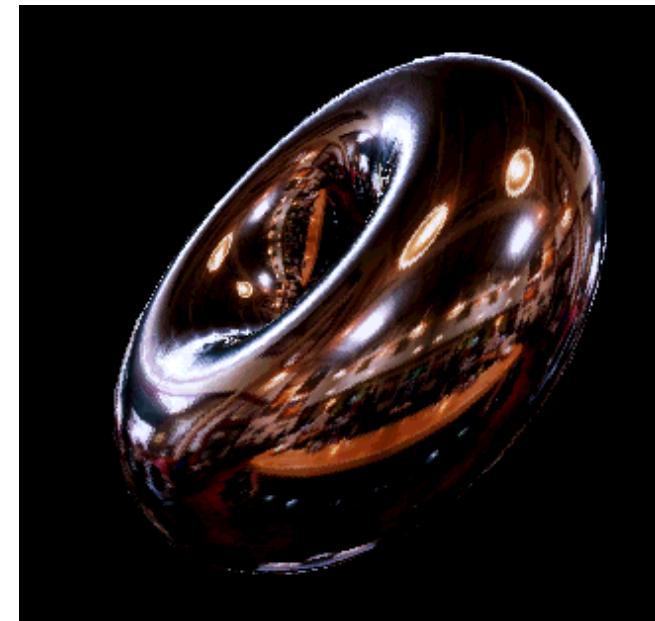
- Used in most first-person shooters
 - Precalculate (*bake*) diffuse lighting on static objects
 - Only low resolution necessary
 - Diffuse lighting is view independent!
 - Advantages
 - No runtime lighting necessary
(good for older cards)
 - More accurate than vertex lighting
(no need to tessellate for highlights)
 - Can take global effects (shadows, radiosity) into account
- 
- The diagram illustrates the light mapping process. It shows three images arranged horizontally: a brick wall texture on the left, a grayscale diffuse lighting map in the middle, and the resulting light-mapped image of the brick wall on the right. To the left of the first image is a multiplication symbol (×), and to the left of the second image is an equals sign (=).

Light Mapping Procedure

- Map generation
 - Use single map for group of coplanar polys
 - Map back to worldspace to calculate lighting
- Map application
 - Premultiply textures by light maps
 - Large textures, no dynamics
 - Multitexturing at runtime
 - Fast, flexible

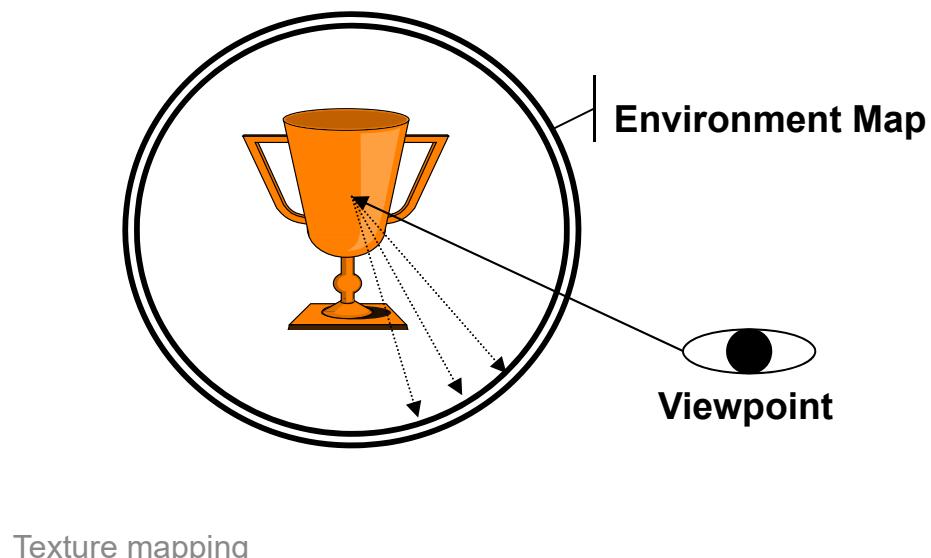
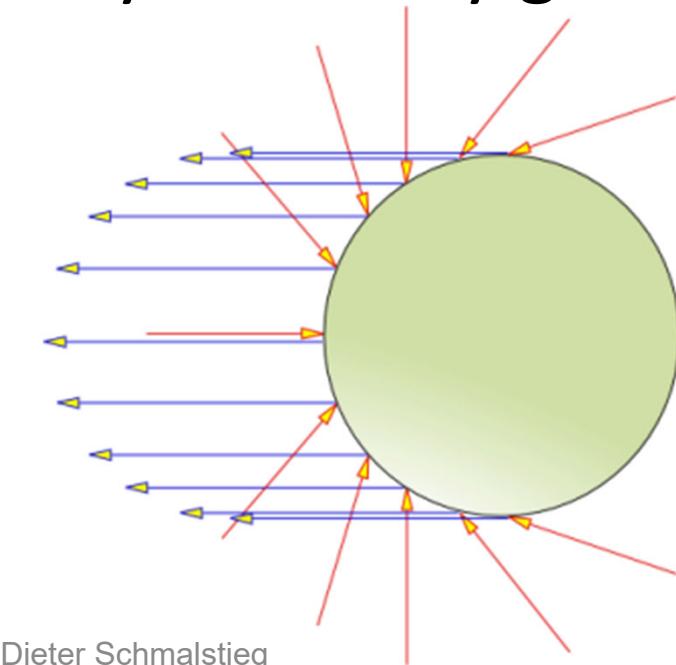
Environment Mapping

- Idea: **use texture to create reflections**
- Uses texture coordinate generation, multitexturing, RTT, ...



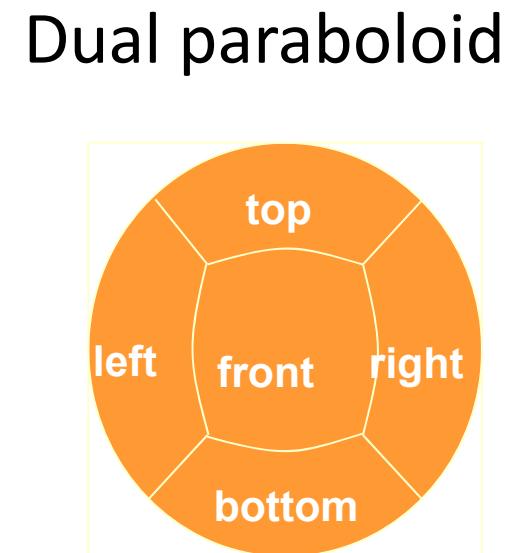
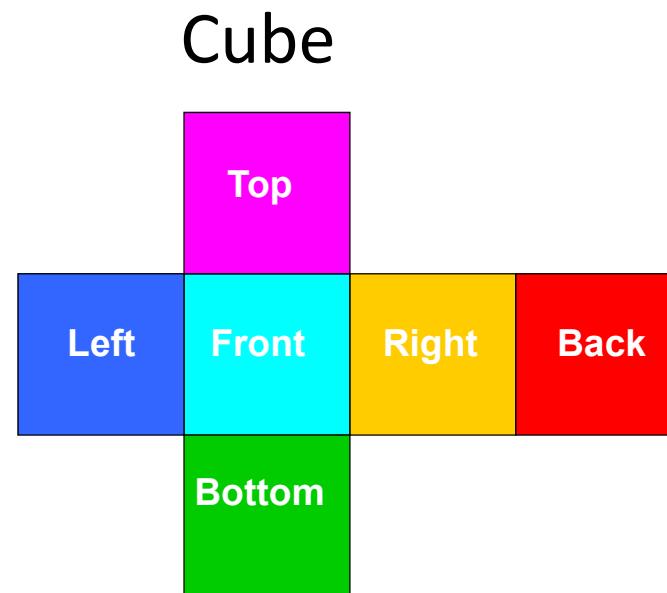
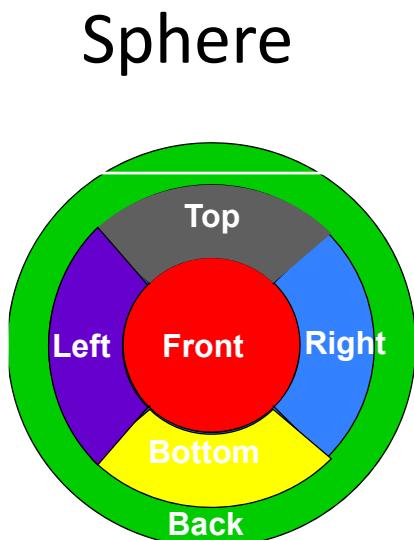
Environment Mapping Indexing

- Assumption: index envmap via orientation →
 - Reflecting object shrunk to a single point
 - Or: environment infinitely far away
- Eye not very good at discovering the fake



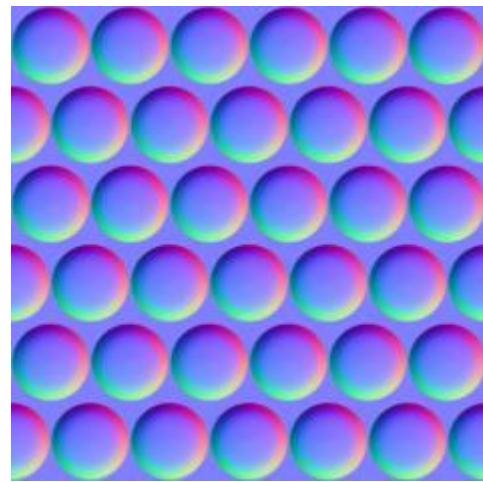
Environment Mapping Types

- Typical mappings:
Each maps all directions to a 2D texture

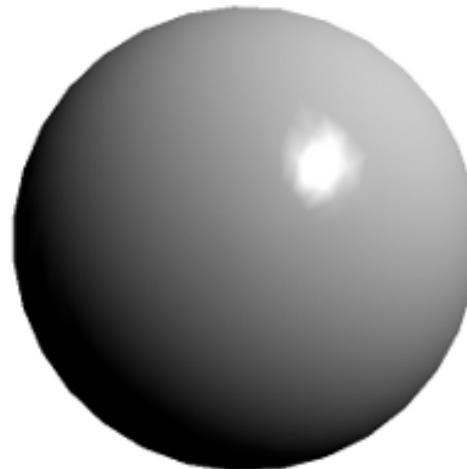


Bump Mapping

- Per-pixel normals are a **prerequisite** for BM
- Simulating **rough surfaces** by **calculating per-pixel illumination**
- Replace **geometry** with (faster) texture



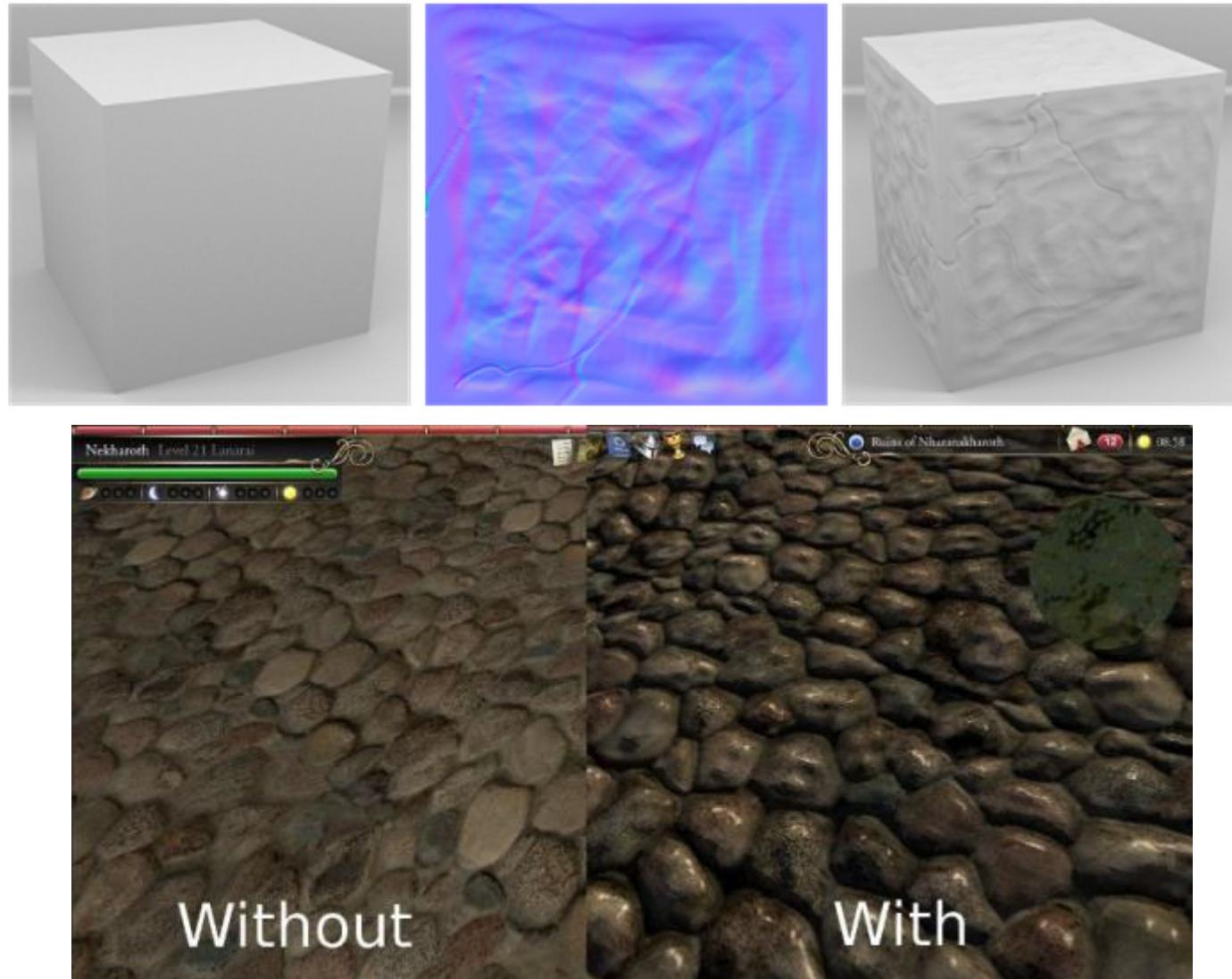
+



=

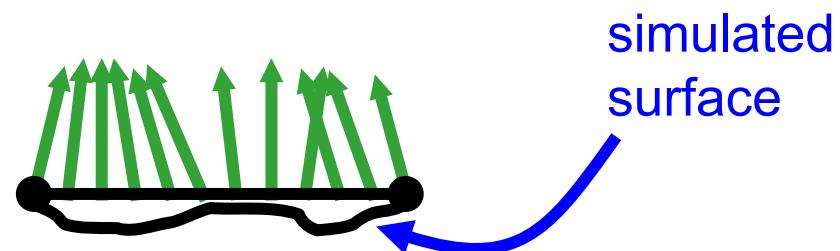


Bump Mapping Examples



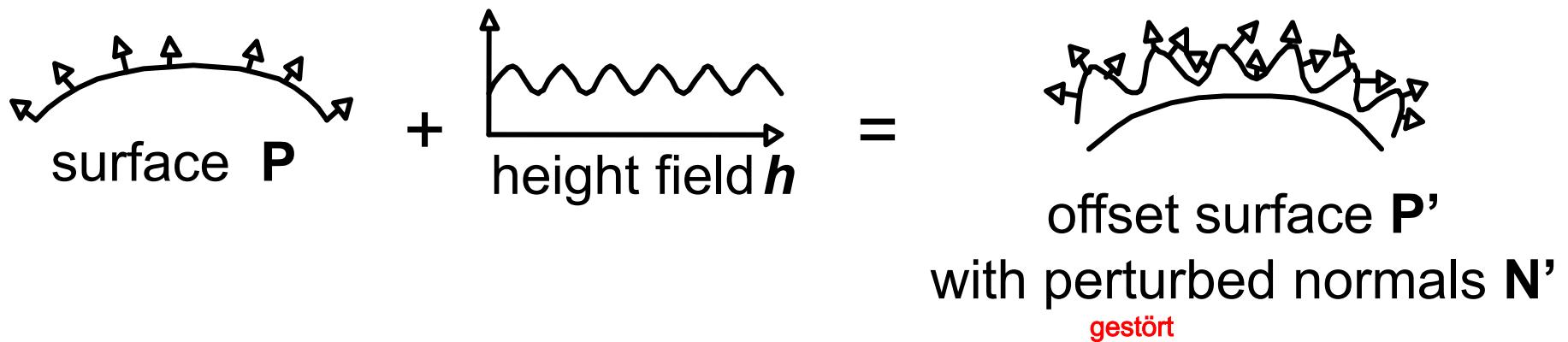
Bump Mapping Basics

- Original idea from ray tracing [Blinn 1978]
- Simulate surface features with illumination only
- Instead of interpolating normals, take them from a map



Bump Mapping Basics

- Assume a (u,v) -parameterization
 - Points \mathbf{P} on the surface given as $\mathbf{P}(u,v)$
- Surface \mathbf{P} is modified by 2D height field h



Mathematics of Bump Mapping

- Displaced surface:

$$\mathbf{p}'(u, v) = \mathbf{p}(u, v) + h(u, v)\mathbf{n}(u, v)$$

- Partial derivatives:

$$\frac{\partial \mathbf{p}'}{\partial u} = \frac{\partial \mathbf{p}}{\partial u} + \frac{\partial h}{\partial u} \cdot \mathbf{n} + h \cdot \underbrace{\frac{\partial \mathbf{n}}{\partial u}}_{\text{blue bracket}}$$

$$\frac{\partial \mathbf{p}'}{\partial v} = \frac{\partial \mathbf{p}}{\partial v} + \frac{\partial h}{\partial v} \cdot \mathbf{n} + h \cdot \underbrace{\frac{\partial \mathbf{n}}{\partial v}}_{\text{blue bracket}}$$

- Perturbed normal:

$$\mathbf{n}'(u, v) = \frac{\partial \mathbf{p}'}{\partial u} \times \frac{\partial \mathbf{p}'}{\partial v}$$

= 0 for locally flat base surface

after some computation...

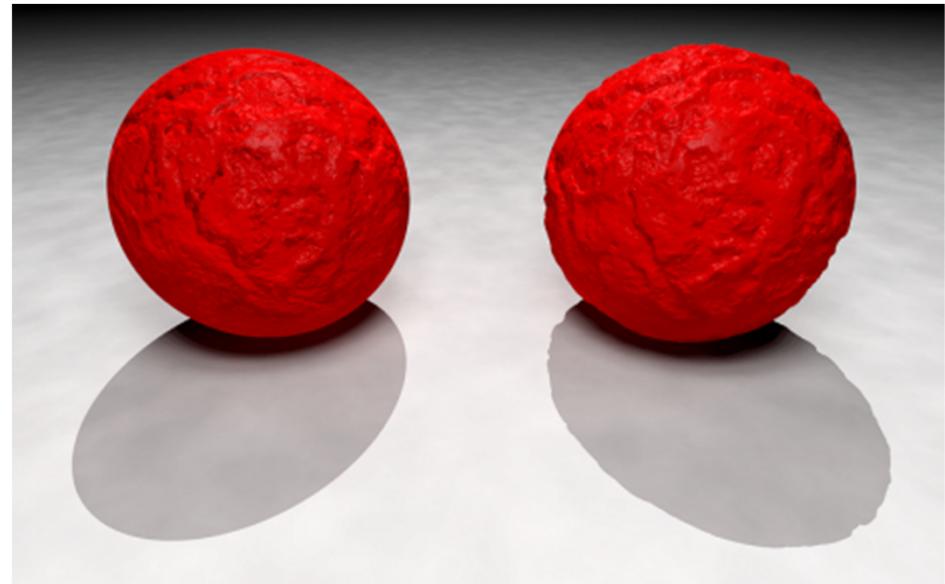
$$\Rightarrow \mathbf{n}' = \mathbf{n} + \frac{\partial h}{\partial u} \cdot \left(\mathbf{n} \times \frac{\partial \mathbf{p}}{\partial v} \right) - \frac{\partial h}{\partial v} \cdot \left(\mathbf{n} \times \frac{\partial \mathbf{p}}{\partial u} \right)$$

Bump Mapping Representations

- Height fields
 - Must approximate derivatives during rendering
 - Simple case: emboss bump mapping
- Offset maps
 - Encode orthogonal offsets from unperturbed normal
 - e.g.: $\left(\frac{\partial h}{\partial u}, \frac{\partial h}{\partial v}\right)$, or whole vector, ...
 - Require renormalization
 - Enable bump environment mapping
 - Calculate using finite differencing on height field
- Normal (perturbation/rotation) maps
 - Encode direction vectors (in whatever space)
 - No normalization required
 - Standard method

Bump Mapping Issues

- Artifacts
 - Shadows
 - Silhouettes edgy
 - No parallax
- Effectiveness
 - No effect if neither light nor object moves
 - In this case, use light maps
 - Exception: specular highlights



Deferred Shading



Overview

- Motivation: why not conventional shading?
 - Shader permutation problem
 - Combinatorial explosion
- Deferred shading method
- G-Buffers
- Other post-processing effects
- Advantages/disadvantages

Conventional Forward Rendering

- After rasterization
- Shading calculations in fixed function pipeline or fragment shader
- Complexity = $O(\text{Light sources} * \text{Objects})$



Dieter Schmalstieg



Deferred Shading

Overdraw

- Complex scenes, large virtual environments → overdraw
- Overdraw → redundant calculations
- Why shade a pixel, if it gets overdrawn in final image?
- Idea: perform shading at the end of rendering
→ we need only one shading operation per pixel

Intermediate:



Dieter Schmalstieg

Final:

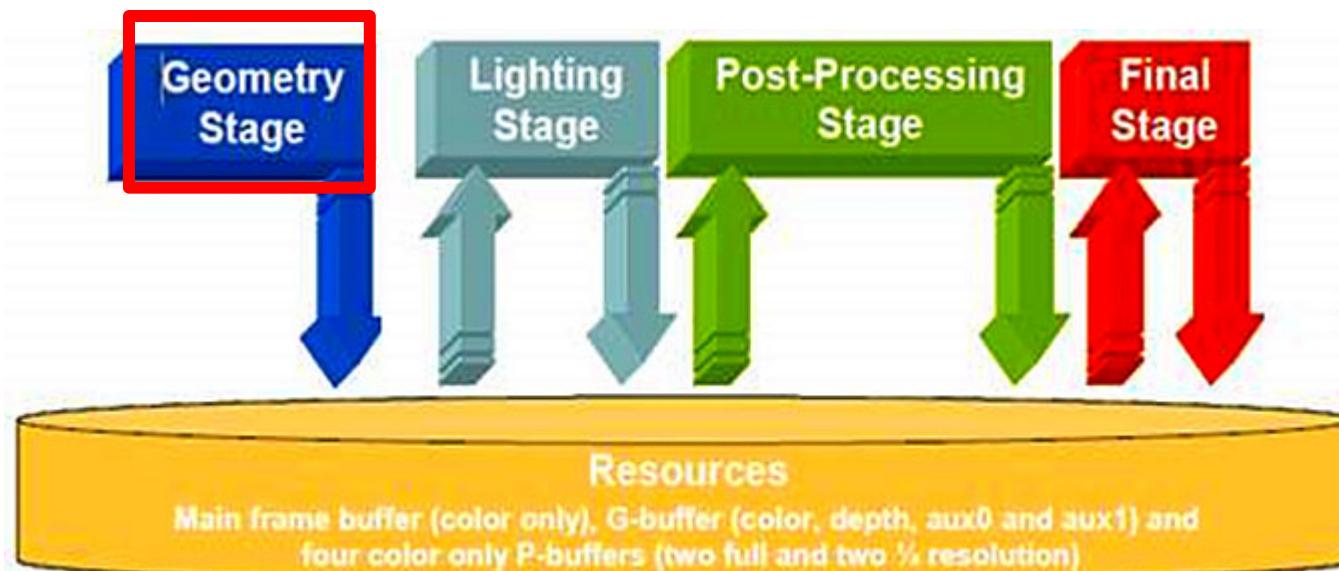


Deferred Shading

Basic Deferred Rendering

Split rendering pipeline in two separate stages

1. Geometry transformation and rasterization



Types of G-Buffers

G-Buffers store per pixel:

- Color
- Depth
- Normal vector
- Position
- Object identity
- etc.

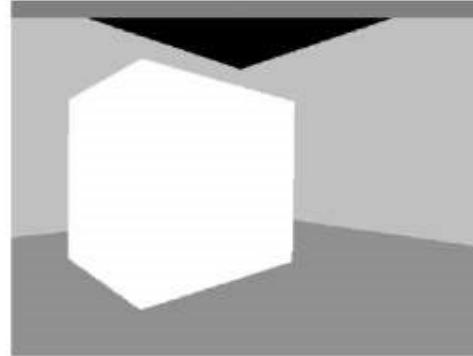
Normal



Color



Identity



Result

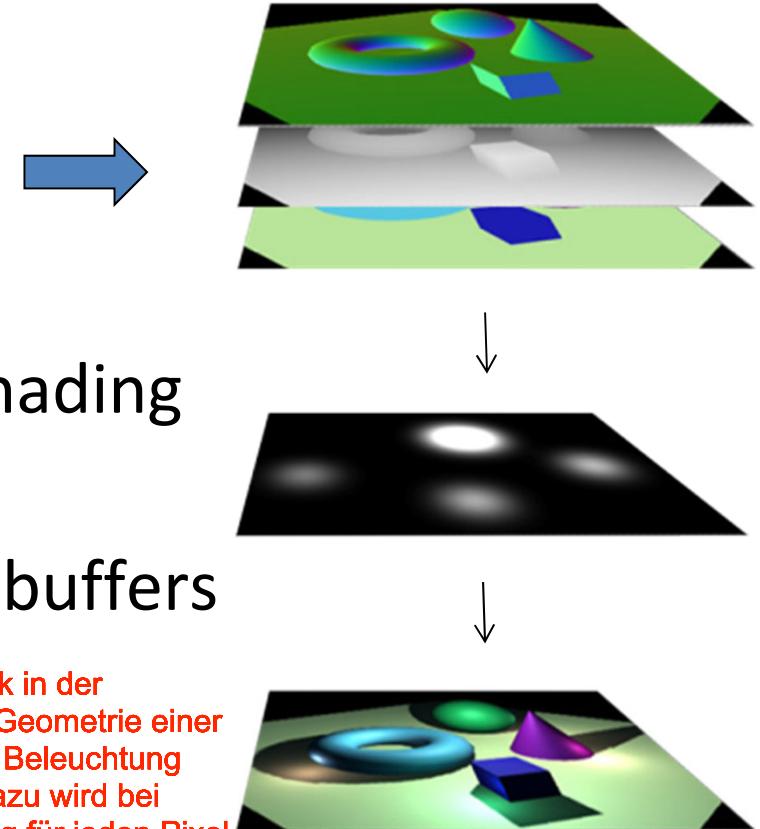


Deferred Rendering Overview

- DR is a framework supporting various effects
 - Screen-space ambient occlusion
 - Non photo-realistic rendering
 - High-dynamic range rendering
 - Deferred shading
 - ...
- Implemented as a post-processing effect
- Most important sub-type: deferred shading

Deferred Shading

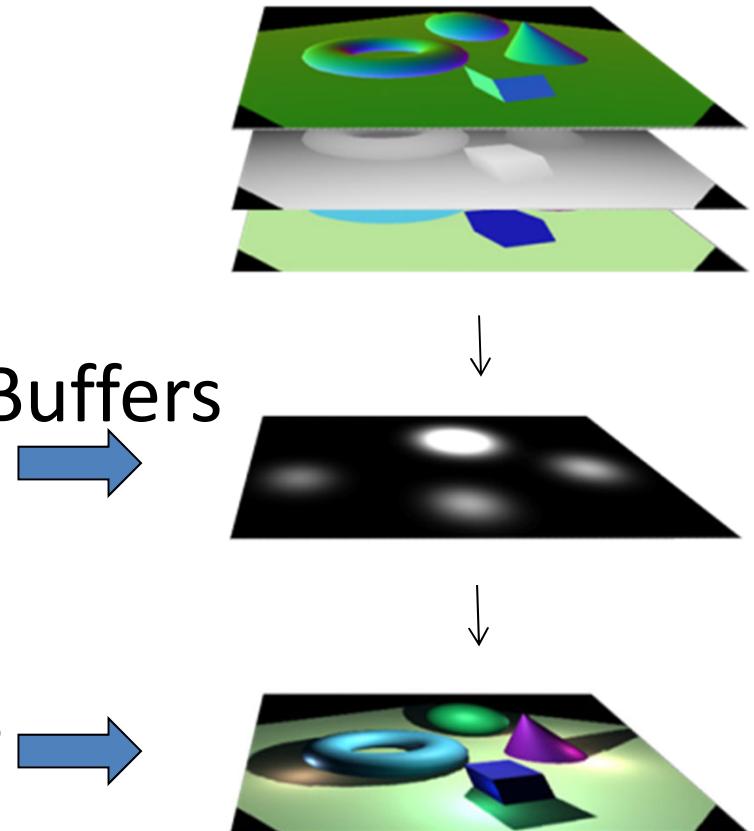
- Geometry stage as usual
 1. Geometry transformations
 2. Rasterization
 3. Material, texturing, but no shading
- Instead of shading,
store intermediate results in G-buffers
 - Diffuse color
 - Depth
 - Position
 - Normal vectors



Deferred Shading ist eine Technik in der Computergrafik, bei der erst die Geometrie einer Szene berechnet wird, bevor die Beleuchtung berechnet wird. Im Gegensatz dazu wird bei Forward Shading die Beleuchtung für jeden Pixel einzeln berechnet, während die Geometrie berechnet wird. Deferred Shading ermöglicht es, dass man mehrere Lichtquellen und deren Wechselwirkungen mit Oberflächen gleichzeitig berechnen kann, was zu realistischeren Ergebnissen führt. Es ist jedoch auch rechenintensiver als Forward Shading.

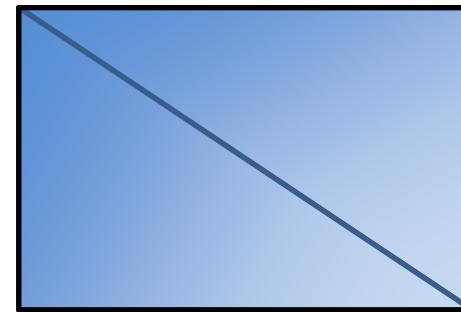
Actual Shading Stage

- Render screen-sized quad
- Fragment shader reads G-Buffers
- Perform shading and postprocessing
- Store result in framebuffer

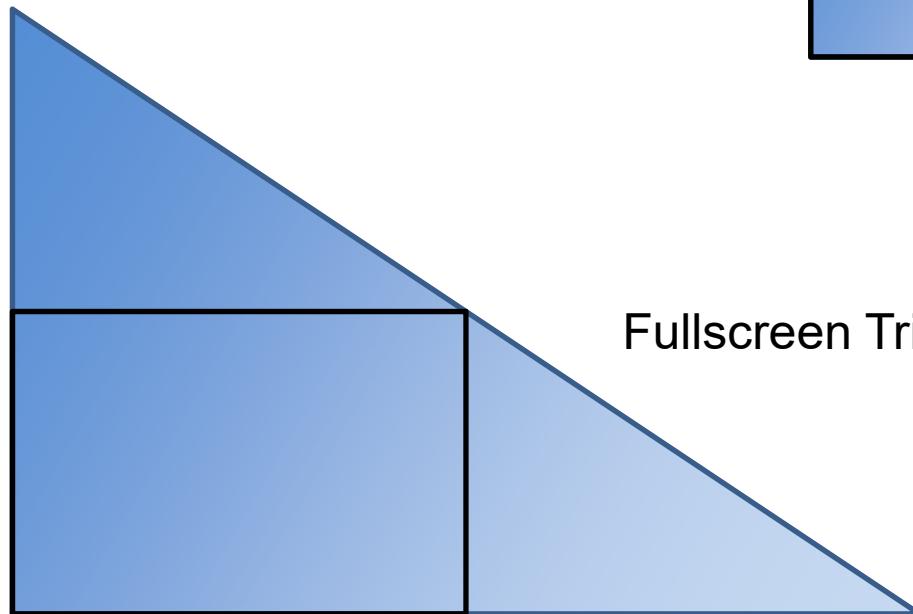


Screen-filling Primitives

Fullscreen Quad



Fullscreen Triangle



Deferred Rendering Advantages

- Render geometry only once
- Perform complex shading and post-processing per pixel
- Complexity $O(\text{Light sources} + \text{Objects})$ instead of $O(\text{Lights} * \text{Objects})$
- Independent of geometry and depth complexity
- Time for shading can be predicted well
→ good for games

Deferred Rendering Disadvantages

- Requires more memory and frequent read/write operations
 - Advanced effects (transparency, ghostings) require per-pixel sorting
 - Cannot use hardware anti-aliasing
-
- Forward shading may be faster, if
 - Low number of light sources
 - Low depth complexity
 - No need for post-processing effects

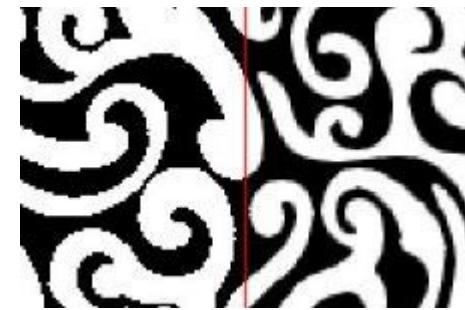
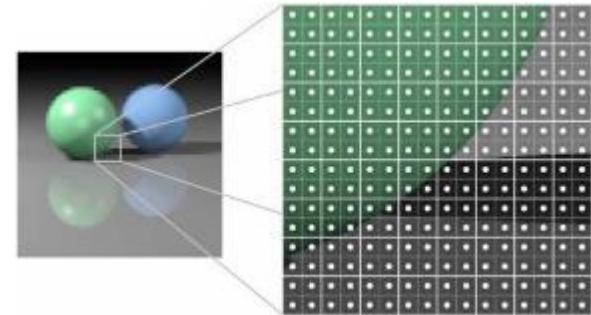


Dieter Schmalstieg
Image-Space Special Effects

Anti-aliasing in Real-Time Graphics

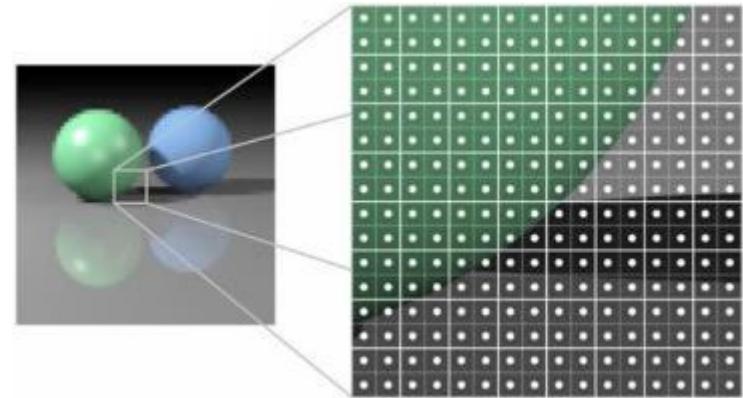
Anti-aliasing can mainly be categorized in

- Better sampling (SSAA, MSAA, TAA)
 - Take more samples during rasterization
- Better filtering (MLAA, TAA)
 - After rasterization as a post process
 - Often incorporates temporal and/or spatial information for filtering



Supersampled Anti-Aliasing (SSAA)

- Multiple samples per pixel
- Accumulated via reconstruction filter
- Equivalent to rendering in higher resolution, then downsampling
- Most accurate antialiasing method
- Helps with all forms of spatial aliasing
- But very expensive, since load increases across whole pipeline



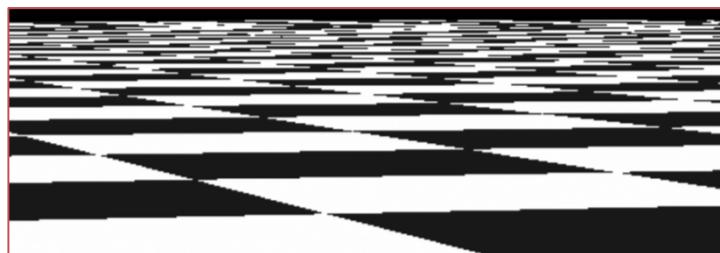
SSAA bekämpft Kantenbildung, indem es das Bild mit einer höheren Auflösung als die tatsächliche Anzeigeauflösung berechnet. Beispielsweise kann ein Bild mit einer Auflösung von 1920x1080 mit SSAA mit einer Auflösung von 3840x2160 berechnet werden. Dies erzeugt ein hochauflößtes Bild mit glatteren Kanten und einer höheren Detailgenauigkeit.

Nach der Berechnung wird das hochauflößte Bild dann auf die tatsächliche Anzeigeauflösung herunter skaliert. Dies erzeugt ein endgültiges Bild mit glatteren Kanten und einer höheren Qualität.

SSAA Algorithm

- Allocate all screen-sized textures and buffers with $N \times$ resolution
- Render as usual (at higher resolution)
- For display, downsample final color texture to target resolution using fullscreen shader
- Box filter, Lanczos filter, Gaussian...
- Display final image in target resolution

Dieser Filter nutzt eine Art von gewichteter Mittelwertberechnung, um die Farben und Pixelwerte der hochauflösten Bilder auf die tatsächliche Anzeigeauflösung herunterzuskalieren.



1 sample per pixel



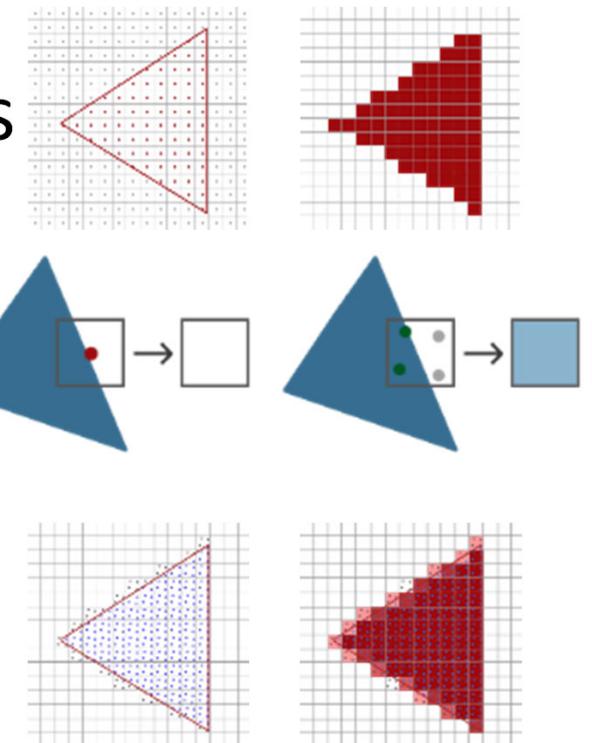
256 samples per pixel

Multisampling (MSAA)

Jaggies sind sichtbare Stufen, Ecken oder Kanten in einem gerenderten Bild, die aufgrund der begrenzten Auflösung des Bildschirms und der Unfähigkeit des Computers, glatte Linien zu zeichnen, entstehen.

- Most aliasing effects related to “jaggies”
 - Depth and coverage aliasing
- MSAA places additional subsamples
 - Fragment shader is only invoked once per pixel
 - Output color is blended based on number of covered samples
- Hardware support in graphics API

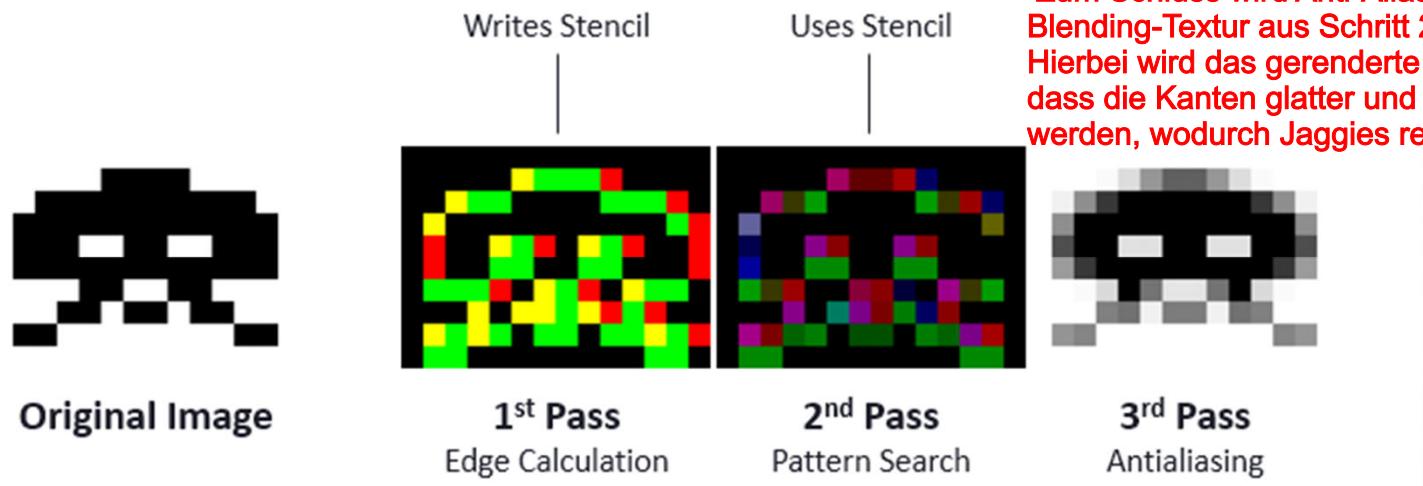
Pixel wird in mehrere Subpixel unterteilt und Farb und Tiefeninformation wird für jedes Farbpixel gespeichert. Danach werden diese „Samples“ kombiniert (blend) um den Endgültigen Wert für den Pixel zu bestimmen.



MLAA (Morphological Anti-Aliasing) ist eine Technik zur Reduzierung von Jaggies in Computergrafiken.
MLAA verwendet morphologische Verfahren, um Kanten in einem Bild zu erkennen und dann eine glattere Darstellung dieser Kanten zu erreichen.

Morphological Anti-Aliasing (MLAA)

1. Edge detection in input image e.g. Sobel, La Place
2. Compute blending texture based on distance to line end/crossing and line shape "Blending-Textur" berechnet, die auf Basis der Entfernung zu den Endpunkten oder Übergängen der Kanten und der Form der Kanten berechnet wird.
3. Anti-Aliasing using blending map from step 2 Zum Schluss wird Anti-Aliasing auf Basis der Blending-Textur aus Schritt 2 durchgeführt. Hierbei wird das gerenderte Bild so angepasst, dass die Kanten glatter und feiner dargestellt werden, wodurch Jaggies reduziert werden.



SSAA vs. MSAA vs. MLAA

- SSAA
 - Spatial anti-aliasing for everything (textures, edges, shading)
 - Very expensive, therefore hardly used for real-time rendering
- MSAA
 - Resolves aliasing for edges only (not for textures and shading)
 - Too expensive with deferred rendering, alpha aliasing causes issues
 - Current gold standard for VR
 - Not applicable to deferred rendering engines
- MLAA
 - Blurs mostly along edges, resolving edge-aliasing only
 - Very cheap post-process even for weaker systems
 - Can look overly blurred
 - Can cause issues with fine details such as text, requires fine-tuning
 - Not temporally stable → flickering
 - Preferred solution for deferred rendering engines

Temporal Accumulation

Prüfungsfrage



- Idea: combine multiple samples taken at different times
- Past samples are stored in a buffer
- Examples
 - Motion blur
 - Temporal anti-aliasing

Temporal Anti-Aliasing (TAA)



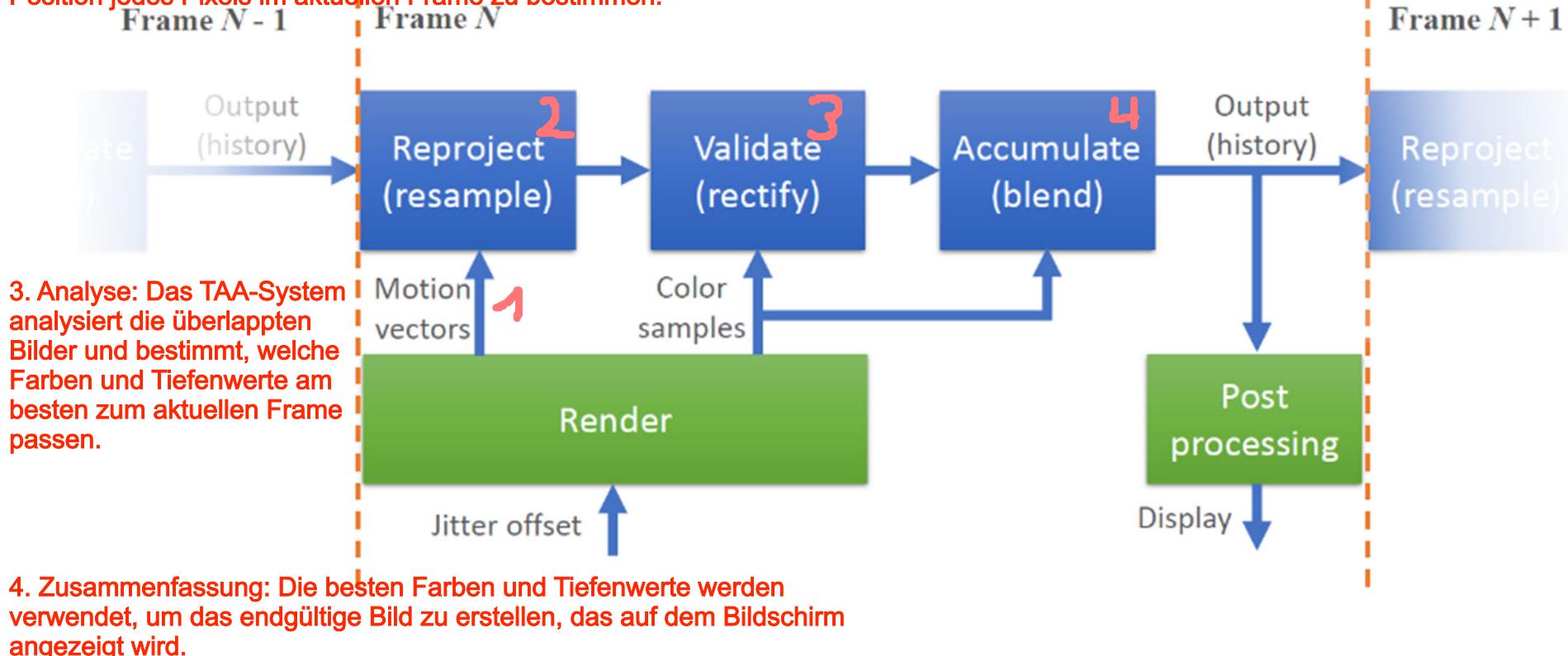
- Idea: Blend previous frames' **jittered shading result** with successive frames
 - Stochastic/subpixel supersampling
 - Works for **geometric and shading aliasing**
- Current gold standard for game engines
 - Works with any rendering architecture
- Requires more involved modifications to the renderer compared to other AA solutions

Die Schritte von TAA sind folgendermaßen:

1. Motion-Vektor-Erkennung: Das TAA-System erkennt die Bewegungen von Objekten im Bild, indem es Motion-Vektoren berechnet. Diese Vektoren beschreiben die Bewegung jedes Pixels im Vergleich zum vorherigen Frame. (TAA: Jittering wird hier verwendet)

Temporal Anti-Aliasing (TAA)

2. Überlagerung: Das TAA-System überlappt das aktuelle Frame mit dem vorherigen Frame, indem es die Motion-Vektoren verwendet, um die Position jedes Pixels im aktuellen Frame zu bestimmen.



Yang, L., Liu, S. and Salvi, M. (2020), A Survey of Temporal Antialiasing Techniques. Computer Graphics Forum, 39: 607-621. <https://doi.org/10.1111/cgf.14018>

Jittering is an AA method typically used for static scenes which works by very subtly moving (or shaking) world geometry in (viewspace?), by using an array of evenly distributed noise values (Halton [2, 3] sequence).

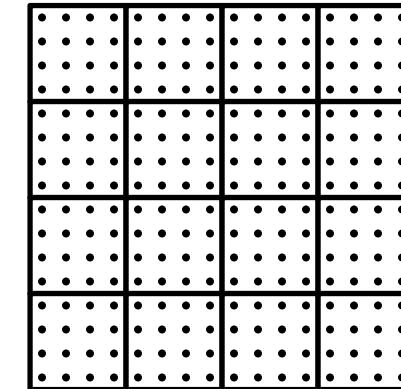
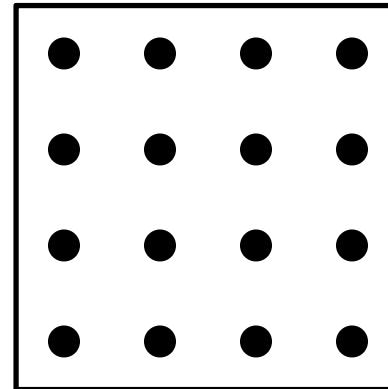
TAA: Jittering

- To generate sampling locations for temporal supersampling, we jitter the projection matrix

```
ProjMatrix[2][0] += ( SampleX * 2.0f - 1.0f ) / ViewRect.Width();  
ProjMatrix[2][1] += ( SampleY * 2.0f - 1.0f ) / ViewRect.Height();
```

Jittering is done via a vertex shader by manipulating the projection matrix.

Regular grid

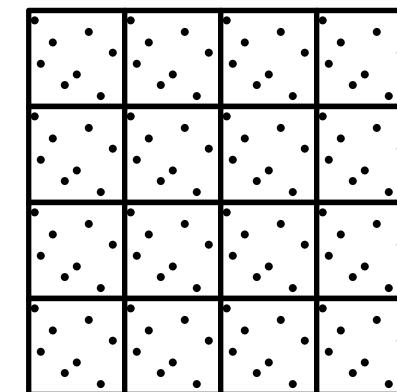
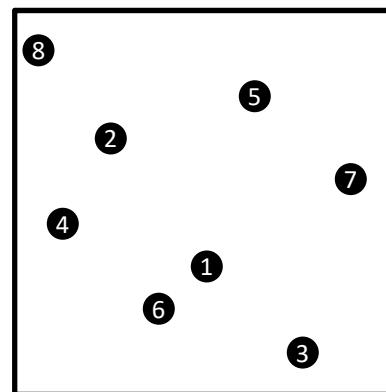


TAA: Jittering Sequence

- For fast convergence, each subsequence should be evenly distributed in the pixel domain
- **Halton(2, 3)** is a good candidate (used in UE4)

Halton(2,3)

Durch die Verwendung einer geeigneten Jittering-Sequenz, wie der Halton(2, 3)-Sequenz, kann sichergestellt werden, dass die Projektionsmatrix gleichmäßig verändert wird und so eine schnelle Konvergenz erreicht wird.



Der History Buffer ist ein wichtiger Teil bei der Implementierung von Temporal Anti-Aliasing (TAA). Es dient als Speicher für vergangene Bilder und wird verwendet, um das aktuelle und vorhergehende Bild miteinander zu verbinden. Dazu wird eine aktuelle Bildmarke (in der Regel eine Bool-Variable) verwendet, die bestimmt, welches History Buffer aktiv gezeichnet und welches geleert wird. Diese Marke wird jedes Frame zwischen zwei Zuständen umgeschaltet, um die Rolle der Puffer jedes Frame zu wechseln.



TAA: History Buffering

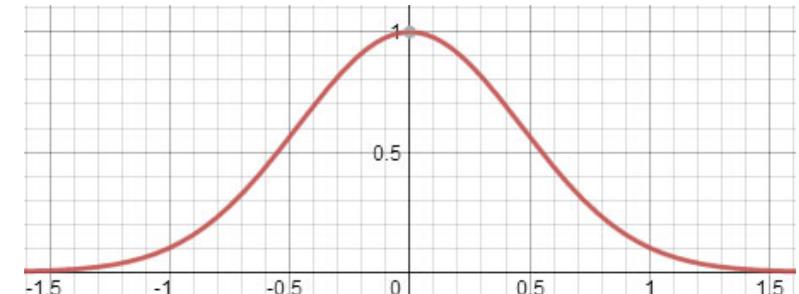
- Simple moving average is not feasible
 - Would require N times storage for N samples
- Exponential moving average
 - Nearly “infinite” number of samples with fixed storage
 - When α is small, approaches a simple moving average
 - $\alpha = 0.1$ is common
- This can simply be stored in a screen-sized buffer!

$$s_t = \alpha x_t + (1 - \alpha)s_{t-1}$$

TAA Reconstruction bezieht sich auf den Prozess, bei dem bei der Implementierung von Temporal Anti-Aliasing (TAA) das resultierende Bild nach der Überlagerung der aktuellen und vergangenen Bilder wiederhergestellt wird.

TAA: Reconstruction

- Simple approach
 - Average all subpixels inside pixel rectangle (Box filter)
 - However, box filter not stable under motion
- Alternative approach: Gaussian
 - UE4 uses a Gaussian fit to Blackman-Harris 3.3
 - Since every pixel shares the same jitter, filter weights are precomputed and passed as shader uniforms



$$W(x) = e^{-2.29x^2}$$

Ein einfacher Ansatz ist, alle Subpixel innerhalb eines Pixelrechtecks (Box Filter) zu berechnen. Diese Methode ist jedoch bei Bewegungen nicht stabil.

Hierbei wird ein Gauss-Fit auf Blackman-Harris 3.3 verwendet, da jedes Pixel die gleichen Verzerrungen teilt. Die Filtergewichte werden vorab berechnet und als Shader-Uniformen übergeben.

TAA: Dynamic Scenes

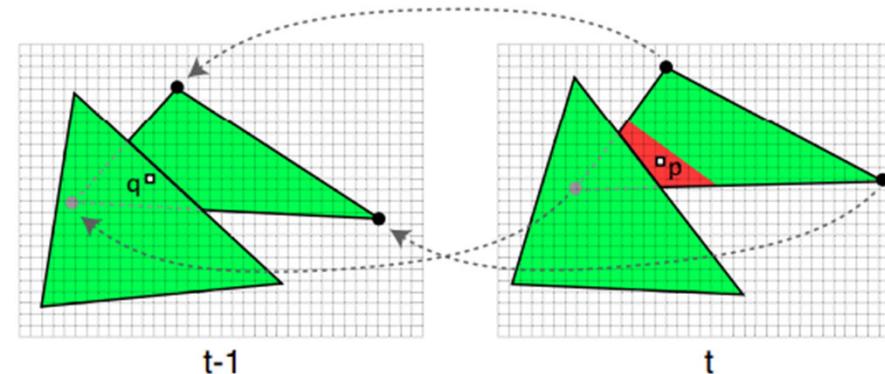
- So far, we discussed jittering, averaging and reconstruction
- What happens if the sample location changes between frames?
 - Animated objects, moving camera...
- For a given pixel, **we need to locate its previous location in the history buffer**
 - Reprojection

TAA Reprojection bezieht sich auf den Prozess, bei dem die Bewegung eines Pixel zwischen zwei aufeinanderfolgenden Frames berechnet wird, um seinen Standort im Vorherigen Frame zu bestimmen.

TAA: Reprojection

- The history for the current pixel may be at a different location, or might not exist at all
- The geometry is transformed twice, using
 - 1.) the current frame's transformation matrices
 - 2.) the previous frame's transformation matrices
- The resulting offsets are stored into a **motion vector texture**
- Using this texture, we can look up the previous location in the history buffer

Dies geschieht, indem die Geometrie des Objekts mit den Transformationen aus dem aktuellen Frame und dem vorherigen Frame transformiert wird. Die Resultierenden Verschiebungen werden in einem Motion-Vector-Textur gespeichert, aus denen man den vorherigen Standort im History-Buffer abrufen kann.



Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. 2007. Accelerating real-time shading with reverse reprojection caching. <https://dl.acm.org/doi/10.5555/1280094.1280098>

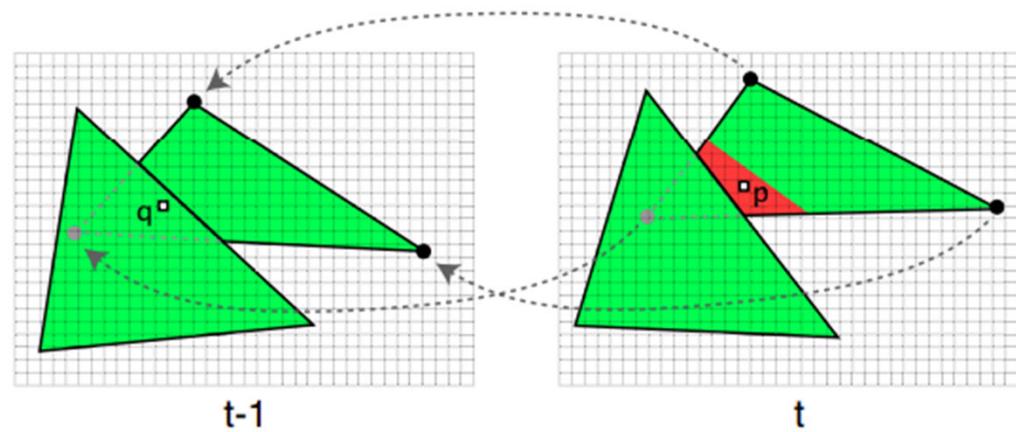
TAA History Rejection ist ein Teil des Temporal Anti-Aliasing (TAA), bei dem es darum geht, die Geschichte des letzten Frames für den aktuellen Pixel zu verwerfen, wenn sie ungültig ist.

TAA: History Rejection

- When reprojecting, the history might be invalid
 - Occlusion/Disocclusion events
 - Shading changes (lights turning on/off...)
- Use **geometry data** (depth, normal, object ID, motion vector) or **color variance** to detect history confidence
- When history is invalid, simply clear the history buffer (set α to 1)

Dies kann aufgrund von Veränderungen im Sichtfeld (z.B. Verdeckungen, Verdeckungen), Änderungen im Beleuchtungszustand oder Variationen im Farbwert festgestellt werden.

Hierfür werden Geometriedaten (Tiefe, Normalen, Objekt-ID, Bewegungsvektor) oder Farbvariationen verwendet. Wenn die Geschichte ungültig ist, wird einfach der History-Buffer gelöscht (auf 1 gesetzt).

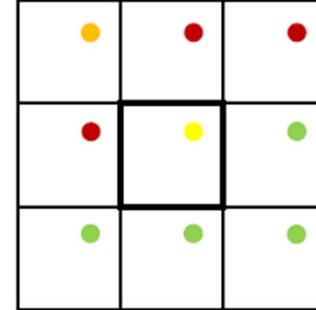


TAA History Rectification ist ein Ansatz, um das Temporal Anti-Aliasing (TAA) zu verbessern, indem die bestehende Geschichte an die neuen Samples angepasst wird.

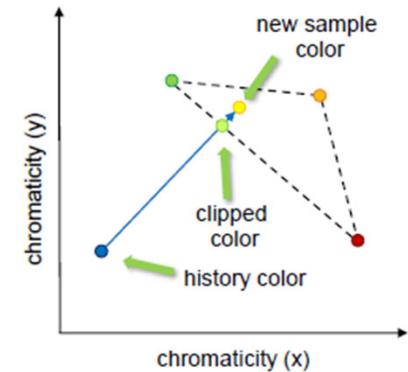
TAA: History Rectification

- Instead of rejecting history, we can make data more consistent with the new samples
 - Clip existing history to neighborhood of new sample
 - Further reduces temporal artifacts

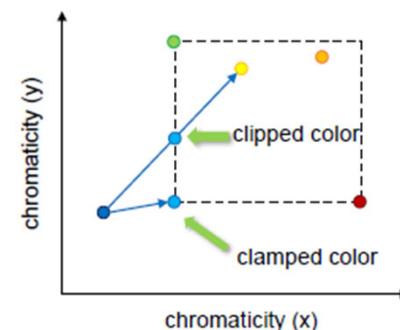
Dies geschieht, indem bestehende Geschichte auf das Nachbarschaftsgebiet des neuen Samples beschränkt wird. Dadurch werden temporale Artefakte weiter reduziert. Die Rektifikation wird durchgeführt, indem die bestehenden Geschichten mit den neuen Samples verglichen und angepasst werden.



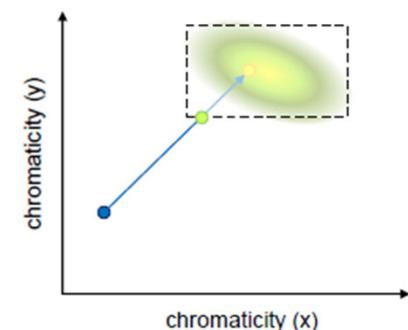
(a) New samples in the 3×3 neighborhood of a pixel



(b) Convex hull clipping



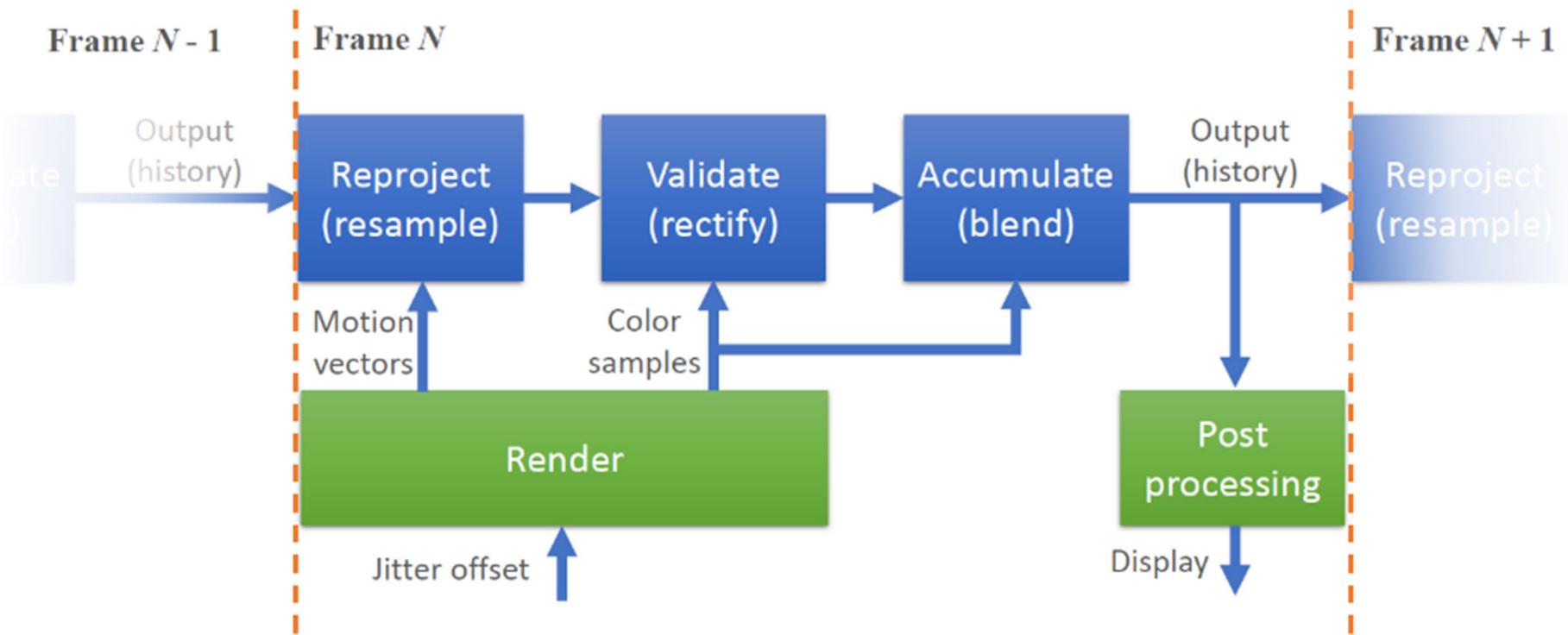
(c) AABB clipping and clamping



(d) Variance clipping

Yang, L., Liu, S. and Salvi, M. (2020), *A Survey of Temporal Antialiasing Techniques*. Computer Graphics Forum, 39: 607-621. <https://doi.org/10.1111/cgf.14018>

TAA: Review



Yang, L., Liu, S. and Salvi, M. (2020), *A Survey of Temporal Antialiasing Techniques*. Computer Graphics Forum, 39: 607-621. <https://doi.org/10.1111/cgf.14018>

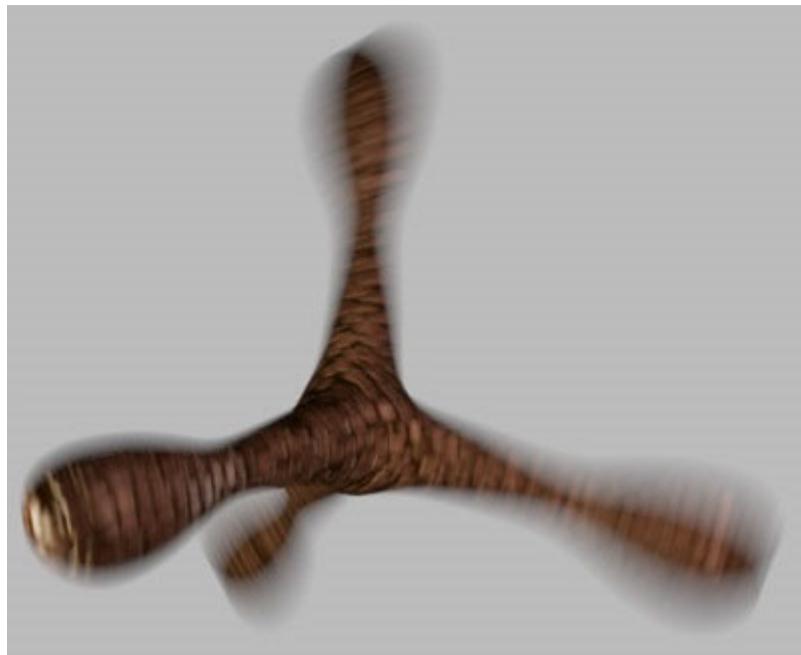
Motion Blur

- Fast moving objects appear blurry
- Property of the human eye and cameras
- Cameras: too long exposure
- Humans: moving the eye causes blur
- Advantages:
 - Looks good/realistic
 - Can cover performance problems

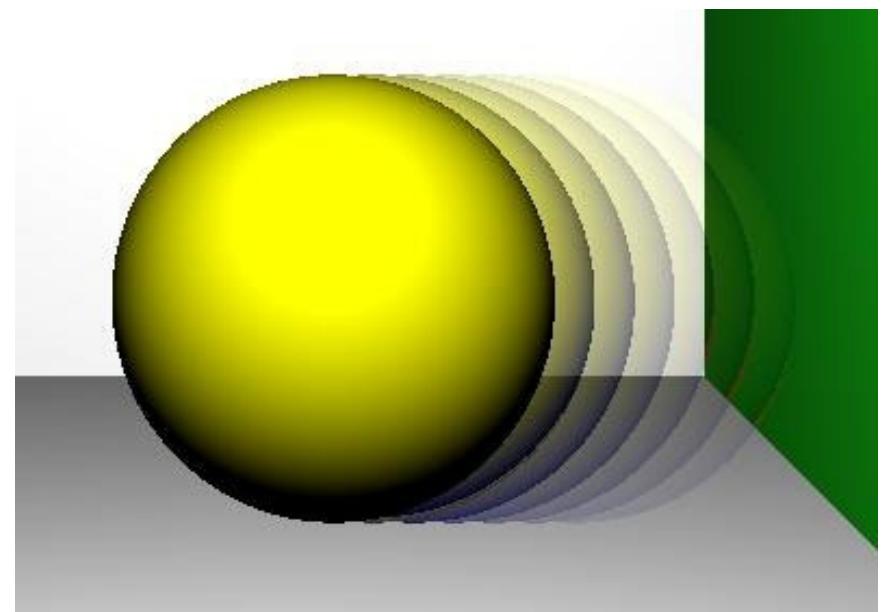


Continuous vs Discrete

Is a continuous effect



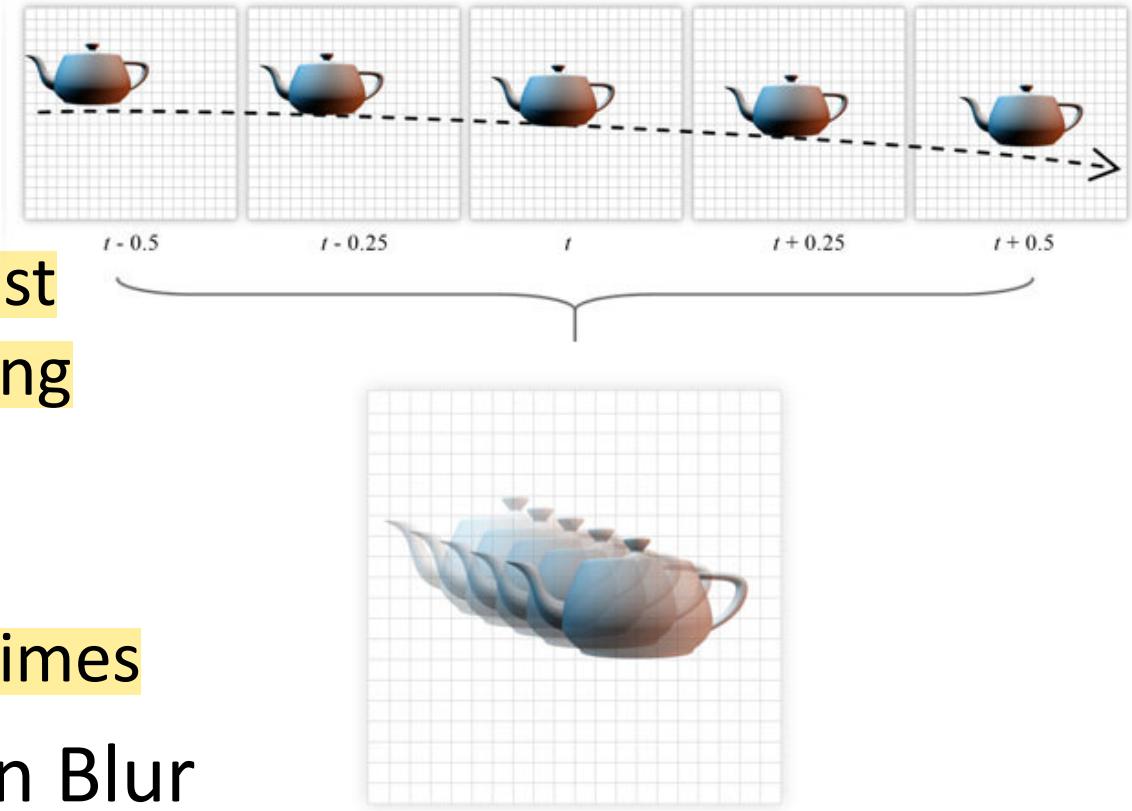
Correct, continuous MB



Approximated, discrete MB

Discrete Methods

- Simplest method
 - Render object at past positions with varying transparency
 - Object needs to be rendered multiple times

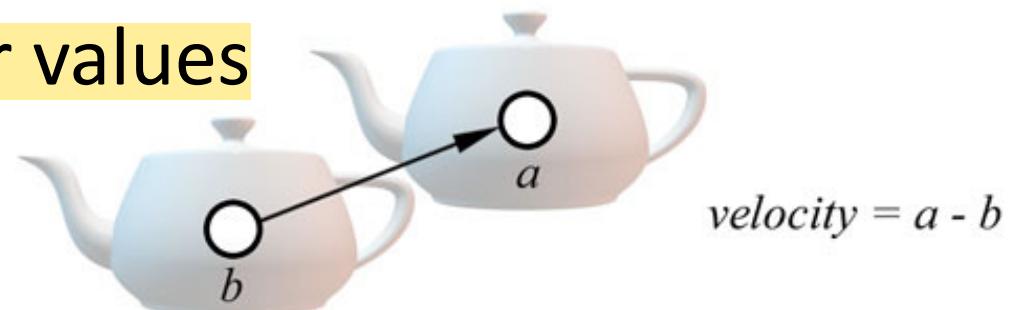


- Image Space Motion Blur
 - Render object to buffer
 - Copy buffer with varying transparency
 - More efficient

Continuous Motion Blur

For each pixel:

- Compute how pixel moves over time
- Current and previous model-view projection matrix form *velocity buffer*
- Sample line along that direction
- Accumulate color values



Particle Systems: Introduction

- Modeling of objects changing over time
 - Flowing
 - Billowing
 - Spattering
 - Expanding
- Typically many small objects
 - Rendering with billboards
- State-less vs. state-full particles



Applications

- Modeling of **natural phenomena**:
 - Rain, snow, clouds
 - Explosions, fireworks, smoke, fire
 - Sprays, waterfalls



https://youtu.be/H_Ico7TSUIY



<https://youtu.be/UnZMp17lqy4>



<https://youtu.be/RpIDGqjhOX8>

Procedure

- All particles of a system use the same update method (share the same properties)
- The particle system handles
 - Initializing
 - Updating
 - Randomness
 - Rendering



Particle System Parameters

- Particle parameters change over time:
 - Location, Speed, lifetime
- Particles “die” after some time
- Particle shapes
 - points, spheres, boxes, arbitrary models
 - Size and shape may vary over time

```
struct particle
{
    float t;           // life time
    float v;           // speed
    float x, y, z;    // coordinates
    float xd, yd, zd; // direction
    float alpha;       // fade alpha
};
```



Particle Physics

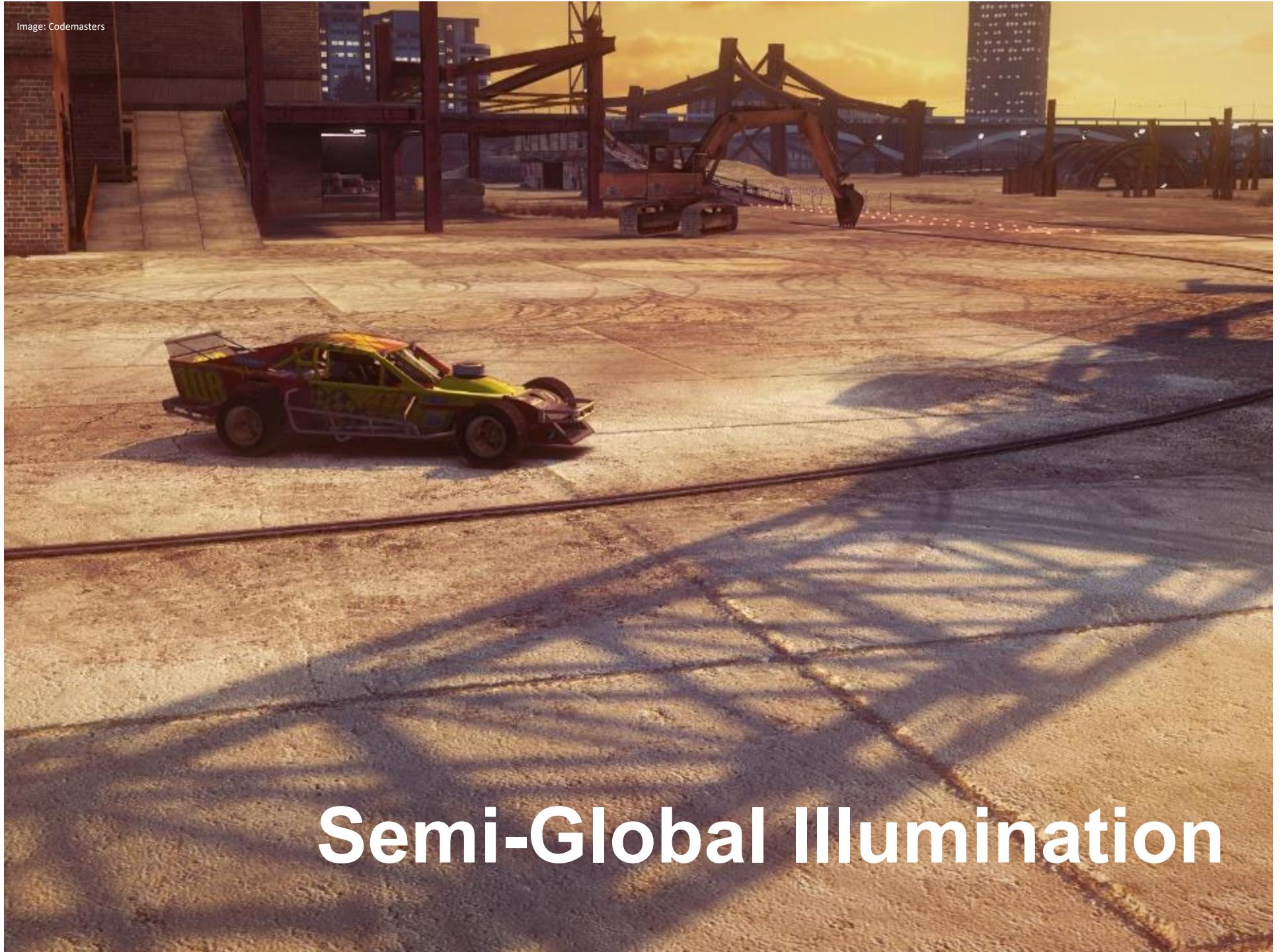
- Motion may be controlled by external forces
 - E.g., gravity, collision
- Particles can interfere with other particles
- Causes a more entropic movement, e.g., sprays of liquids



Things to Consider

- Requires fast physics and collision detection
- Correct modeling not important
- Memory consumption important
- Rendering speed important

Image: Codemasters



Semi-Global Illumination

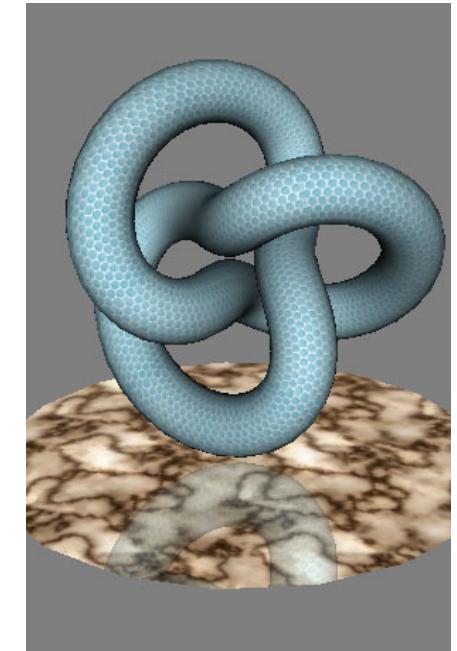
Stencil Buffer

- Allows to mask parts of the framebuffer
- 1 Bit Stencil Buffer ($\in \{0,1\}$) e.g. for masks on or off
- 8 Bit Stencil Buffer (256 states)
- Applications
 - Reflections
 - CSG (constructive solid geometry)
 - Shadow volumes
- Today:
 - Most applications do stenciling in shader code
 - Hardware stencil still very fast – example: Portal



Stencil Buffer Reflections

- Reset stencil buffer
 - Clear stencil buffer to zero
- Render mirror into stencil buffer
 - Enable stencil test
 - Set stencil operation to always write
(on stencil-fail, depth-fail, depth-pass)
- Render mirrored object only in mirror
 - Set stencil operation to write if stencil bit set
- Render object normally

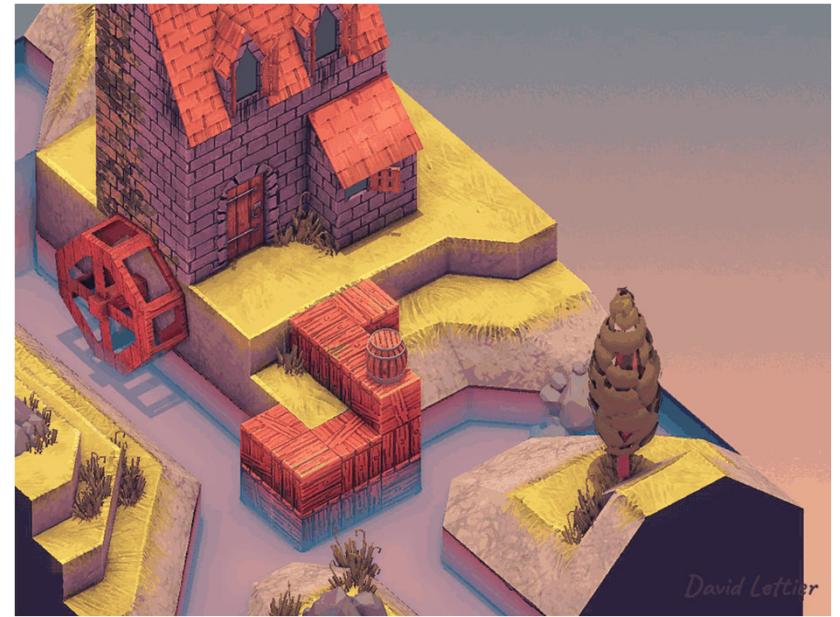


Screen Space Reflections (SSR)

mimic ray tracing in the depth buffer

- Fully dynamic raytraced reflections are very expensive (even with RTX)
- Idea:
 - Reflect view ray across normal
 - Find depth buffer intersection in screen space
- Result: cheap dynamic approximation of reflections

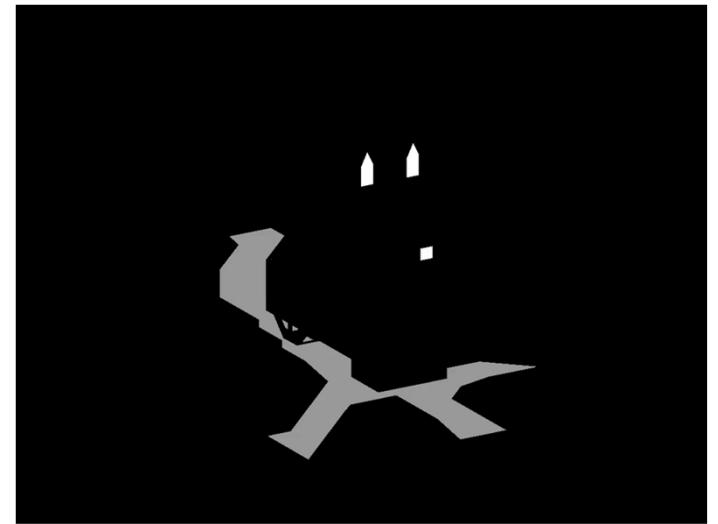
Screen Space Reflections (SSR) sind eine Technik in der Computergrafik, die es ermöglicht, Spiegelungen auf Oberflächen in Echtzeit darzustellen. SSR berechnet die Reflexionen auf der Grundlage der Informationen, die auf dem Bildschirm sichtbar sind, anstatt die Reflexionen in der 3D-Szene selbst zu berechnen. Dies ermöglicht es, die Reflexionen schnell und effizient zu berechnen, aber es hat auch einige Einschränkungen, wie z.B. begrenzte Genauigkeit und die Unfähigkeit, Reflexionen von Objekten darzustellen, die nicht im Sichtfeld des Kameras sind.



<https://lettier.github.io/3d-game-shaders-for-beginners/screen-space-reflection.html>

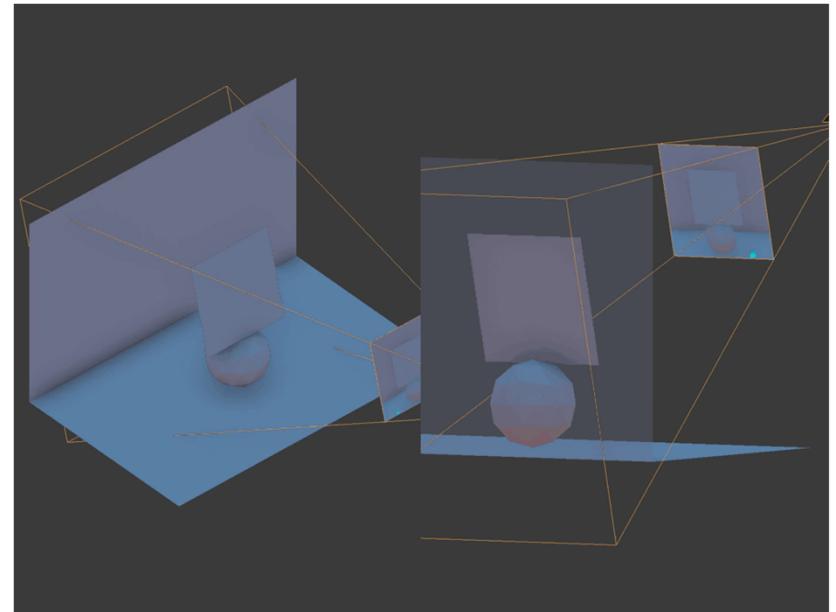
SSR: Reflection Map

- Compute a reflection map
 - Controls amount of reflection per fragment in screen space, depending on material
- Black pixels do not show SSR
- White pixels show full SSR



SSR: View Ray

- For each non-zero pixel in reflection map, reflect view ray across surface normal
- Along reflection vector
 - Choose a point
 - Project it
 - Compute reflection vector in screen space



<https://lettier.github.io/3d-game-shaders-for-beginners/screen-space-reflection.html>

Ray werden verwendet um das
nächste Objekt im Depth Buffer zu
finden wo rauf reflektiert wird

SSR: Ray Marching

- From initial screen space position, **ray march** towards reflected screen space position
- If ray **intersects depth buffer**, we found our reflection point

<https://sakibsaikia.github.io/graphics/2016/12/26/Screen-Space-Reflection-in-Killing-Floor-2.html>



SSR: Limitations 1

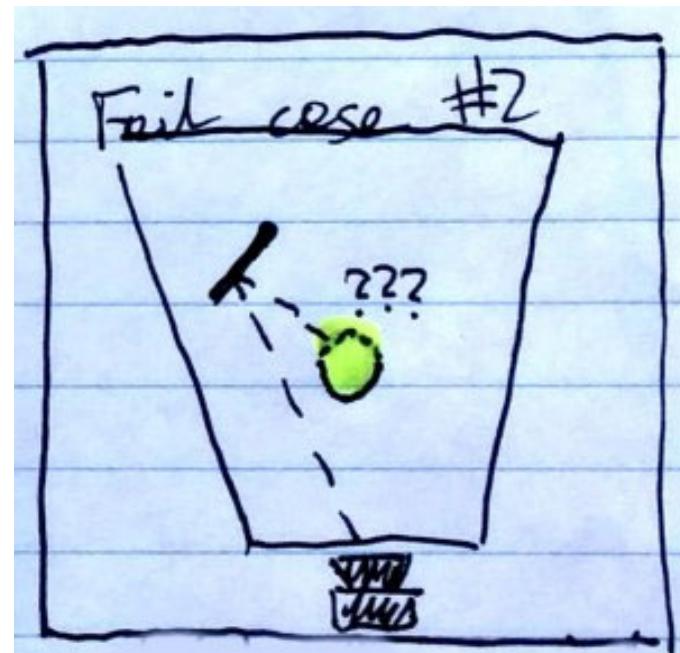
- Viewer facing reflections
- Reflections outside viewport



<https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/>

SSR: Limitations 2

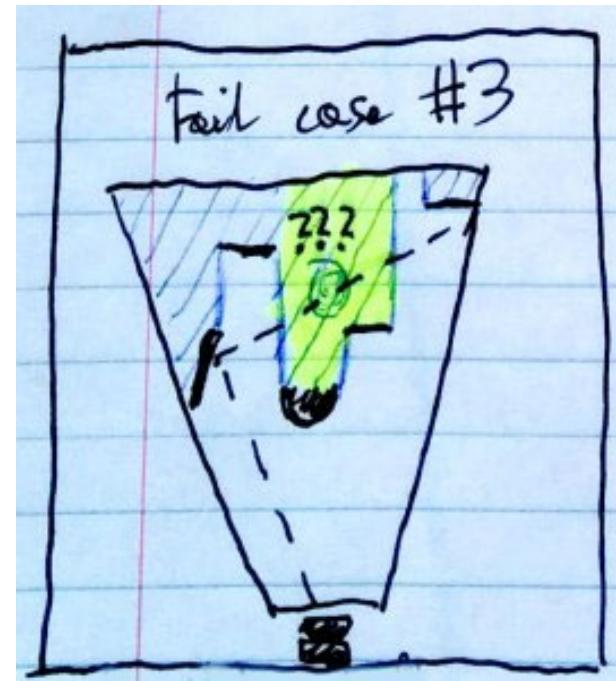
- Reflections of back faces



<https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/>

SSR: Limitations 3

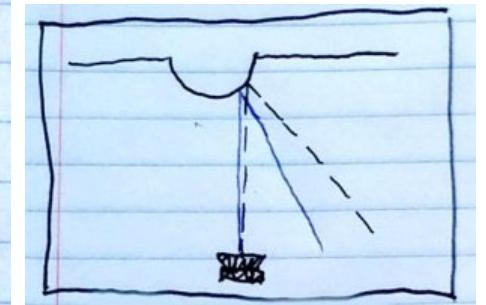
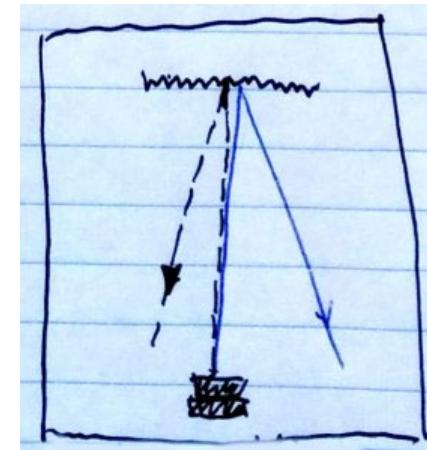
- Depth complexity



<https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/>

SSR: Limitations 4

- Flickering
- Holes
- Temporal artifacts



<https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/>

SSR: Further Improvements

- Blend with cubemaps or baked reflections
- Use temporal filtering to smoothen results
 - TAA is often exploited for this
- Use spatial filtering or blurring for rough surfaces
- Limit ray range in world space
 - For efficiency
 - To prevent far-away objects from flickering

Transparency

- Transparency blending is order dependent
- Blending must be back to front
- Need to sort scene elements by depth



Methods for Sorting Fragments

- **Explicit depth sorting**
 - Write sorted order into draw buffer
- **Depth peeling**
 - Multi-pass rendering, once per *depth layer*
- **Deep frame buffer**
 - Write per-pixel linked lists
- **Weighted blended order-independent rendering**
 - Use depth as implicit weight (heuristic)

Depth Peeling ist eine Technik in der Computer-Grafik, die verwendet wird, um transparente Objekte in einer 3D-Szene korrekt darzustellen. Normalerweise kann die herkömmliche Methode zur Darstellung von Transparenz, das alpha blending, Probleme bei der Darstellung von Überlappungen von transparenten Objekten verursachen.

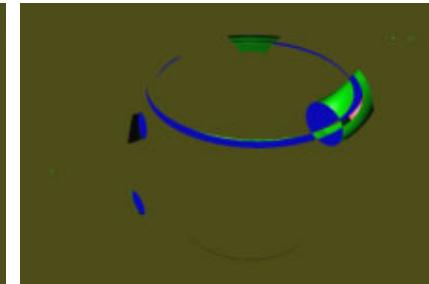
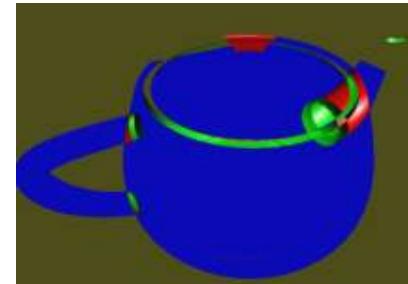
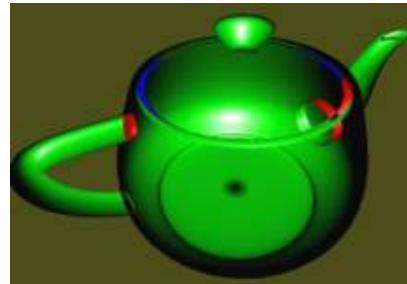
Depth Peeling löst dieses Problem, indem es mehrere Schichten von Pixeln separat renderet und dann zusammenfügt. Jede Schicht enthält nur die Pixel, die an der aktuellen Front (also an der Stelle mit dem geringsten Tiefenwert) sichtbar sind. Die Front wird durch den Vergleich des Tiefenwertes jedes Pixels mit den bereits gerenderten Schichten bestimmt.

Depth Peeling

Dieser Prozess wird mehrere Male wiederholt, bis alle transparenten Pixel in der Szene gerendert wurden. Am Ende werden alle Schichten zusammengefügt, um ein korrekt dargestelltes Bild der transparenten Objekte in der Szene zu erhalten.

Interactive order independent transparency [Everitt'01]

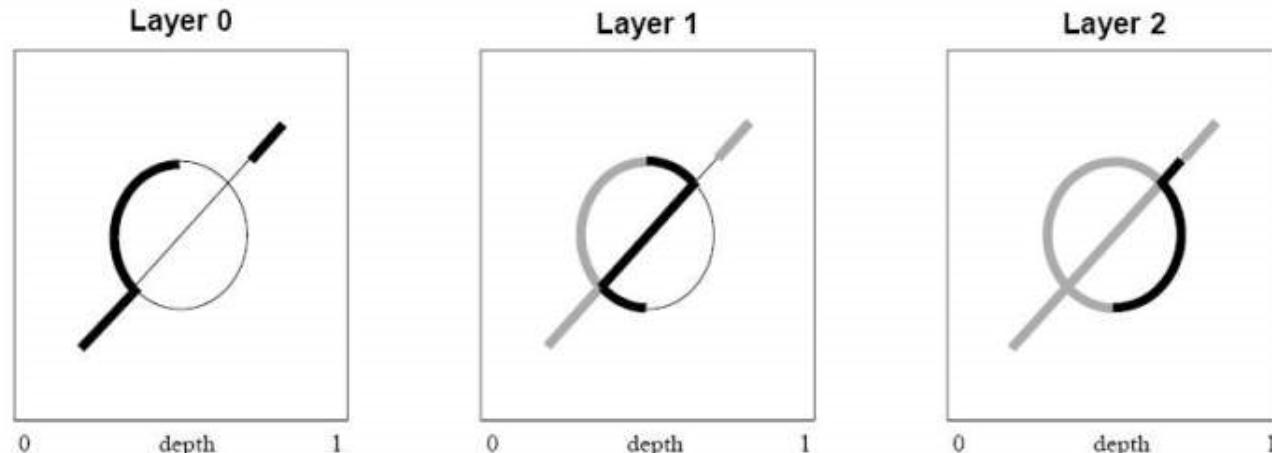
- Multiple passes
 - First pass - find the front-most depth/color
 - Each successive pass - finds the depth/color for the next nearest fragment on a per-pixel basis
- Compare previous layer and current layer



Rendering Depth Peeling

- Render the scene n times Die Szene wird n-mal gerendert, wobei jeder Durchlauf eine eigene Schicht bildet.
- Save the results in texture using render-to-texture/FBO Jede Schicht wird in einer Textur gespeichert
- Each pass forms a layer Jeder Durchlauf bildet eine eigene Schicht, die später zusammengefügt wird, um das endgültige Bild zu erhalten.
- Starting from the second pass, use a shader for comparison with previous layer Ab dem zweiten Durchlauf wird ein Shader verwendet, um jedes Pixel mit der vorherigen Schicht zu vergleichen. Hierbei wird bestimmt, ob das Pixel sichtbar ist oder nicht, basierend auf seinem Tiefenwert.
- Compose n layers

Am Ende werden alle n Schichten zusammengefügt, um das endgültige Bild zu erhalten.



Dual Depth Peeling ist eine Optimierung des Depth Peeling Rendering-Verfahrens, bei dem die Anzahl der benötigten Render-Durchläufe reduziert wird. Hierbei werden gleichzeitig die Vorder- und Rückseiten transparenten Objekte in einem einzigen Durchlauf gerendert.

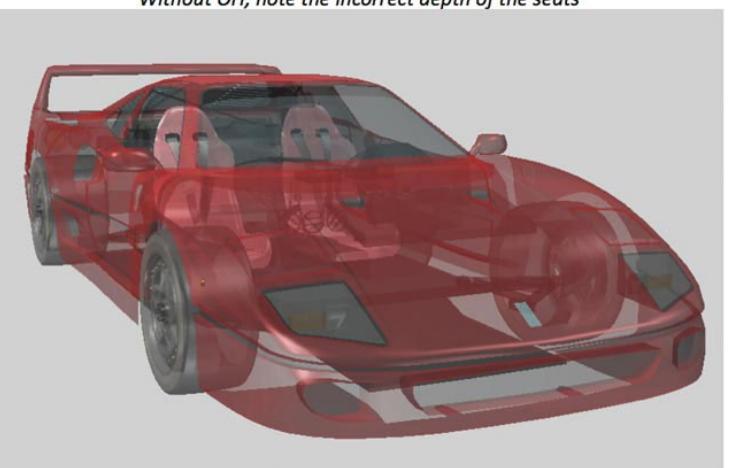
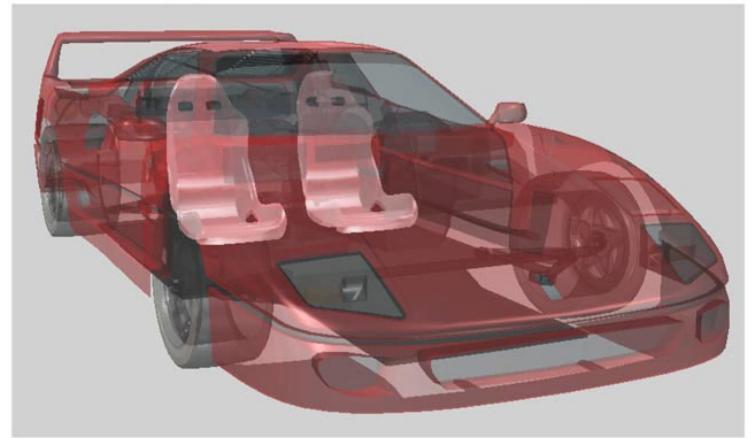


Dual Depth Peeling

- Peel front and back in one pass
Vorder- und Rückseiten transparenten Objekte werden gleichzeitig in einem einzigen Durchlauf gerendert.
- Keep track of min/max depth in two textures
Es werden gleichzeitig die minimale und maximale Tiefe in zwei Texturen verfolgt.
- Compare current depth with min/max values
 - If < min write to front layer
Der aktuelle Tiefenwert wird mit den Minimal- und Maximalwerten verglichen.
 - If > max write to back layer
Wenn der Tiefenwert kleiner als der Minimalwert ist, wird das Pixel in die Vordergrundschicht geschrieben sonst in das Hintergrundbild.
- Use previous front/back layer depth values in current pass
Die Tiefenwerte der Vorder- und Rückseitenschichten des vorherigen Durchlaufs werden im aktuellen Durchlauf verwendet.
- Decreases number of passes from n to $n/2+1$
Die Anzahl der benötigten Durchläufe wird von n auf $n/2+1$ reduziert, was eine bessere Leistung und eine höhere Effizienz bei der Darstellung von transparenten Objekten ermöglicht.

Deep Framebuffer

- Depth peeling creates too much geometry overhead
 - $O(\text{triangles} * \text{overdraw})$
- Use pixel operations instead
 - Write every fragment to a per-pixel linked list
- Sort and blend the lists
- Historically called *accumulation buffer (A-buffer)*



Sie speichert die Informationen über die Transparenz für jeden Pixel auf dem Bildschirm in einer verknüpften Liste. Jeder Eintrag in der Liste enthält Informationen über das transparente Polygon, das für diesen Pixel sichtbar ist, sowie die Farbinformationen für diesen Pixel.



Per-Pixel Linked Lists

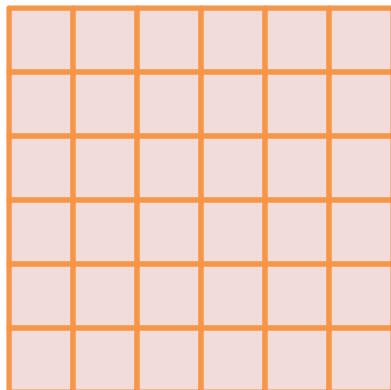
- Algorithm
 - Render scene and generate linked list per pixel
 - Sort list and compose fragments
- Requires read/write buffer
- Need two additional buffers
 - Fragment and link buffer
 - Start offset buffer

Beim Rendern von transparenten Polygone wird zunächst die Tiefeninformationen der Sichtbare Polygone berechnet. Dann werden die Einträge in der PPLL aktualisiert, indem die Einträge für die unsichtbaren Polygone entfernt und neue Einträge für die sichtbaren Polygone hinzugefügt werden. Schließlich werden die Farbinformationen für jeden Pixel aus der PPLL berechnet.

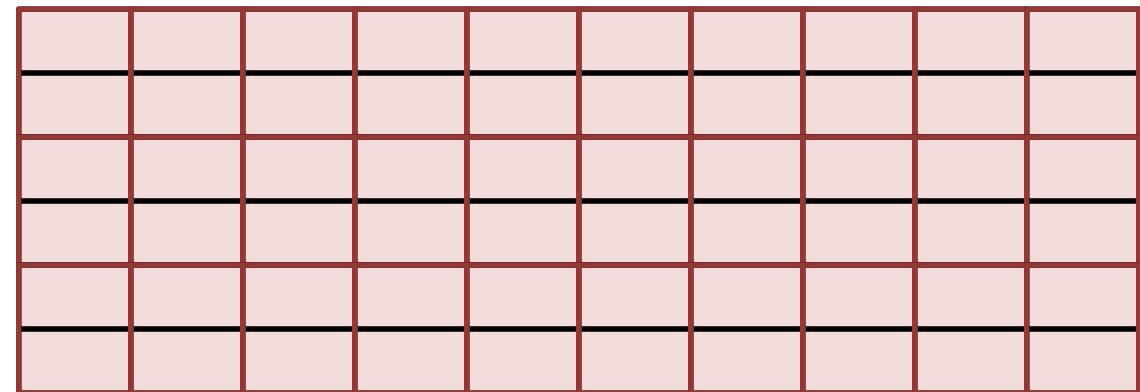
Fragment and Link Buffer

- Contains all fragment data produced during rasterization
- Must be large enough to store all fragments

Viewport



Fragment and Link Buffer



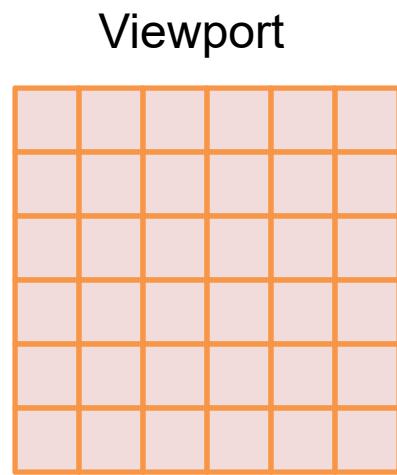
Start Offset Buffer

- Contains the offset of the last fragment written at every pixel location
- Screen-sized: `(width * height * sizeof(UINT32))`
- Initialized to magic value
 - Magic value indicates end of list (no more fragments stored)

-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

Start Offset Buffer

Linked List Creation 1



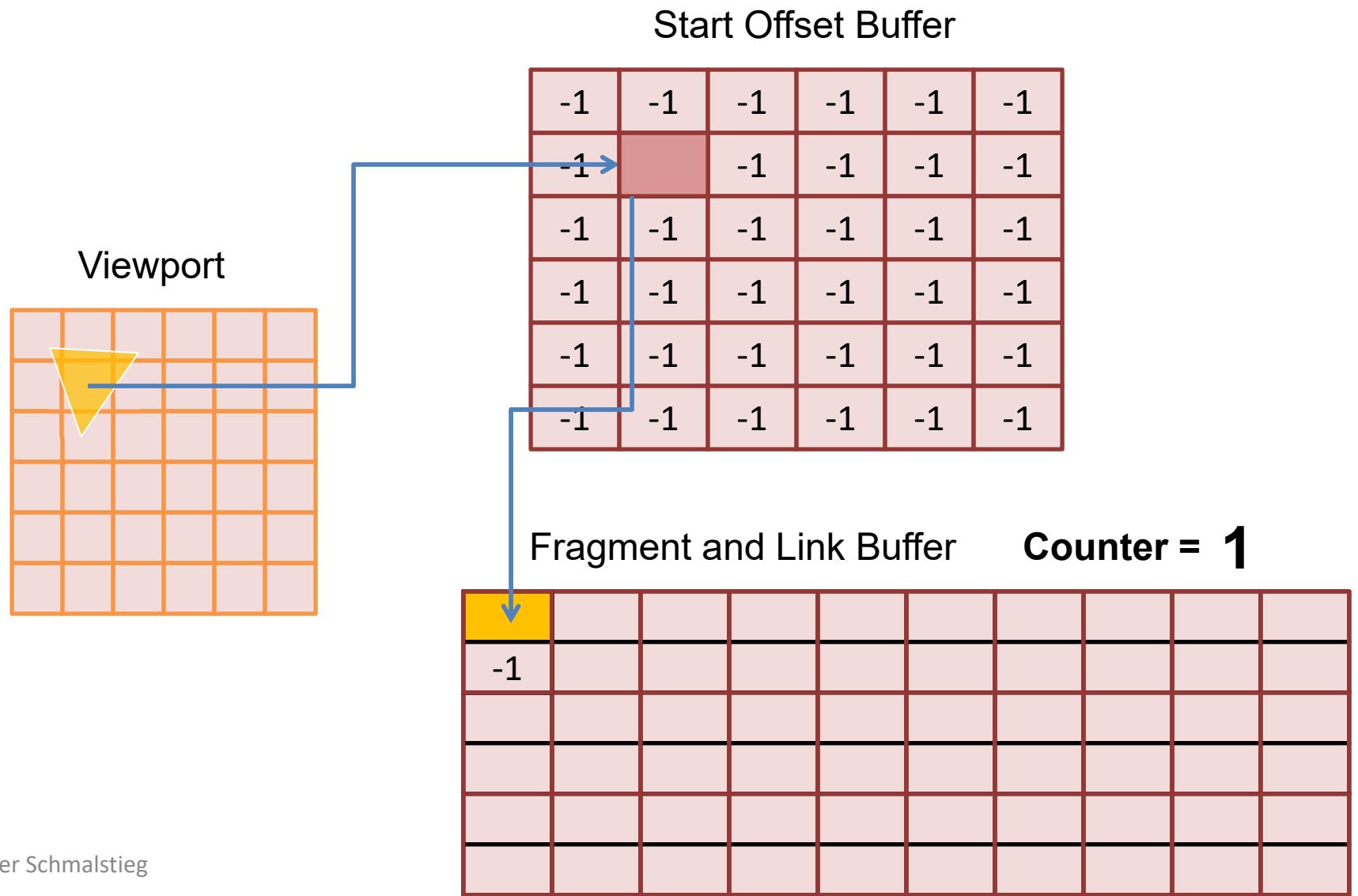
Start Offset Buffer

-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

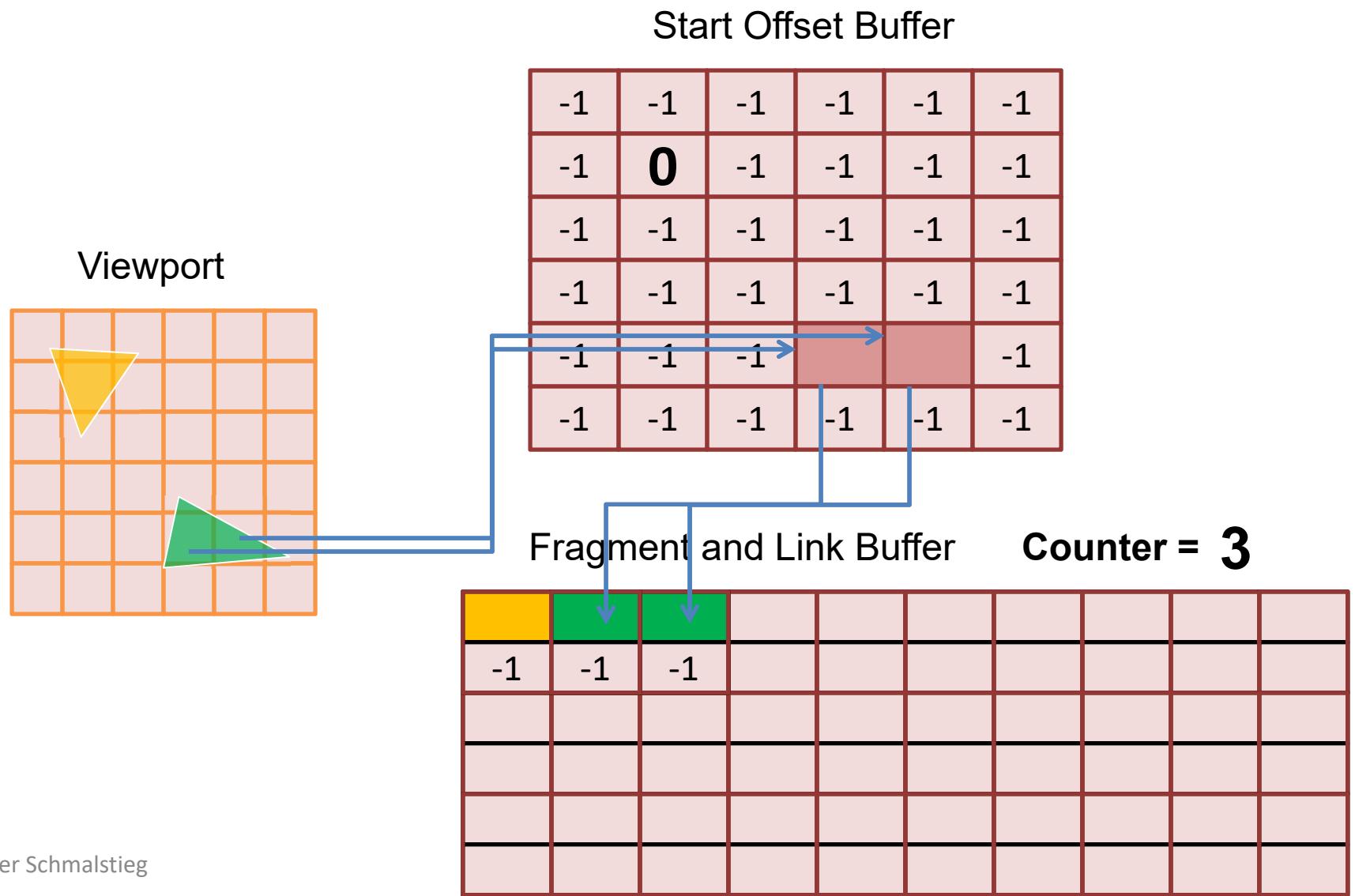
Fragment and Link Buffer

Counter = 1

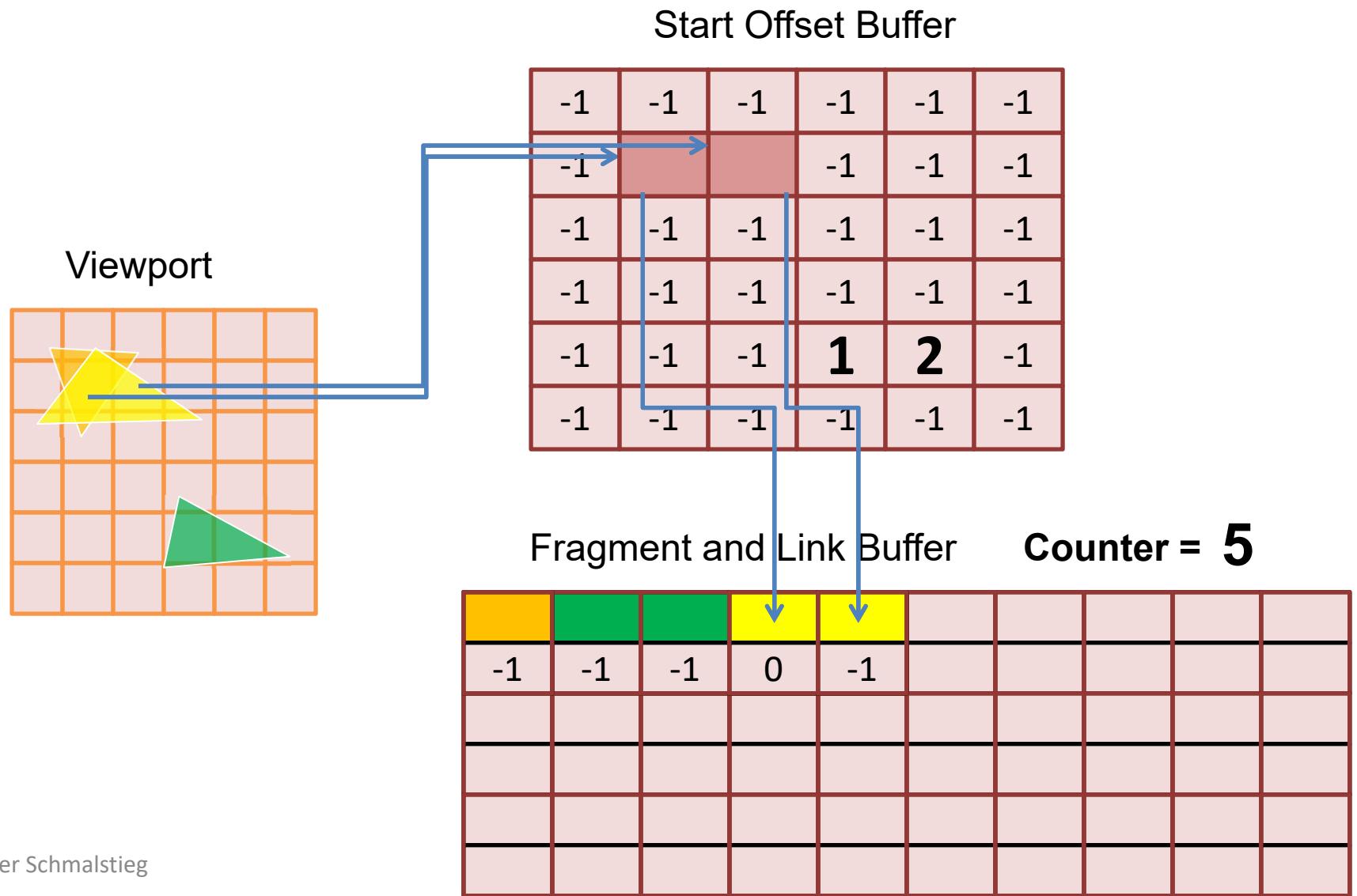
Linked List Creation 2



Linked List Creation 3



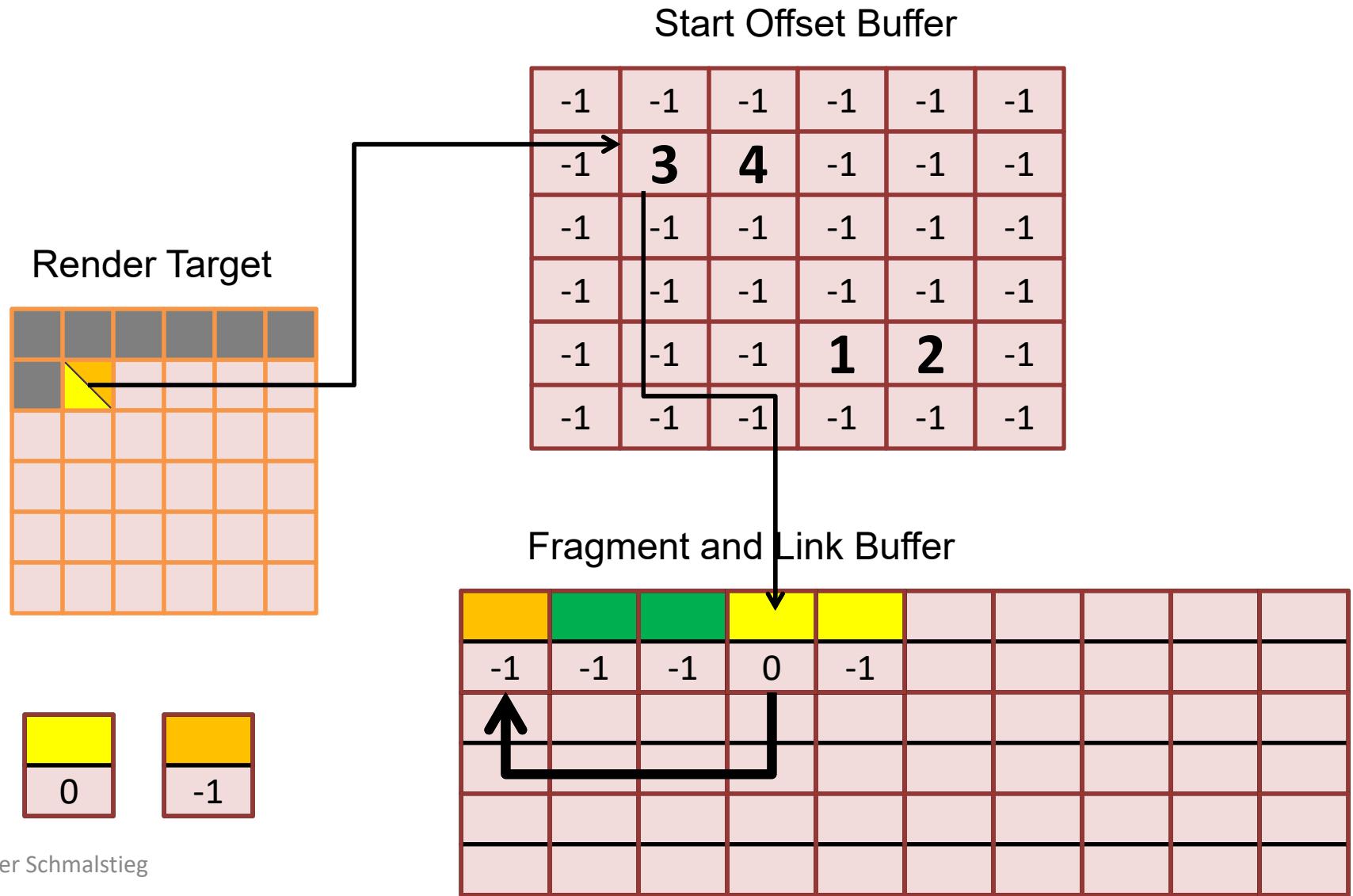
Linked List Creation 4



Linked List Traversal

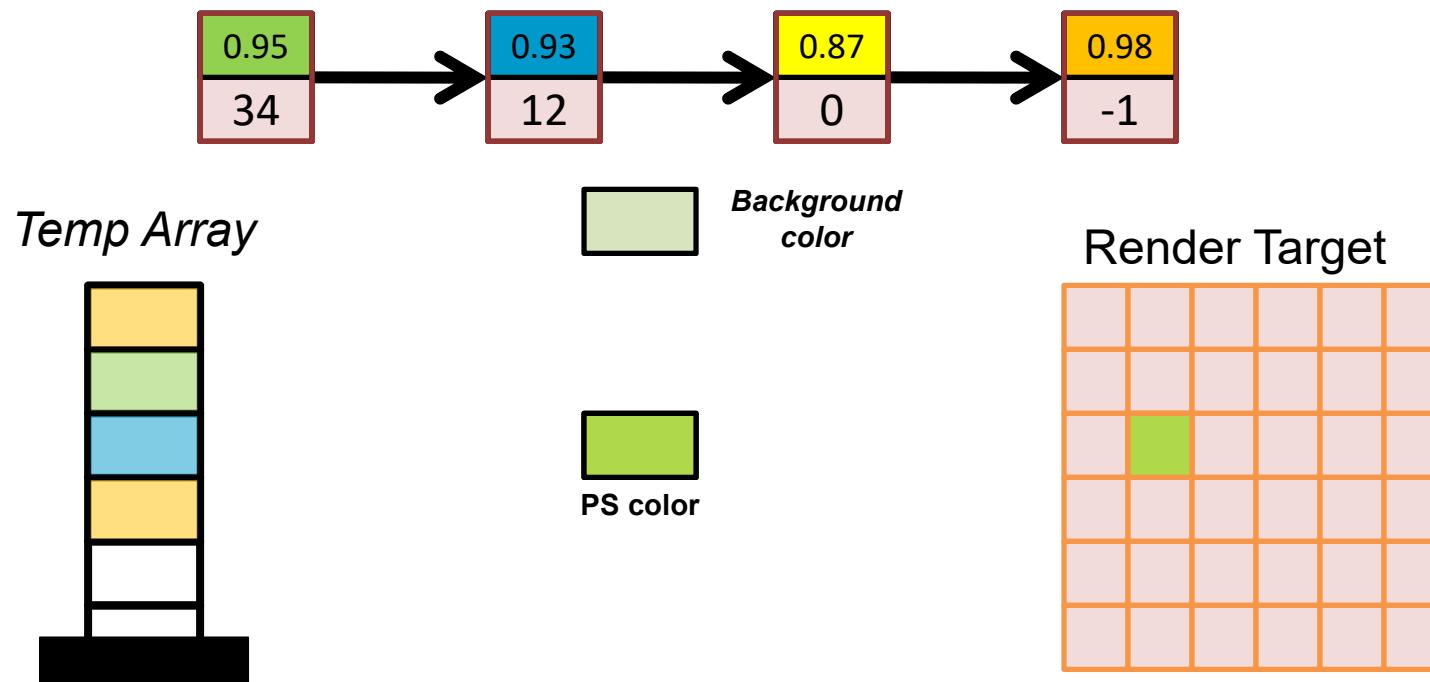
- Render a fullscreen quad (or compute shader)
- For each pixel
 - Parse the linked list
 - Retrieve fragments for this screen position
- Process list of fragments (Sort and Blend)

Rendering from Linked List



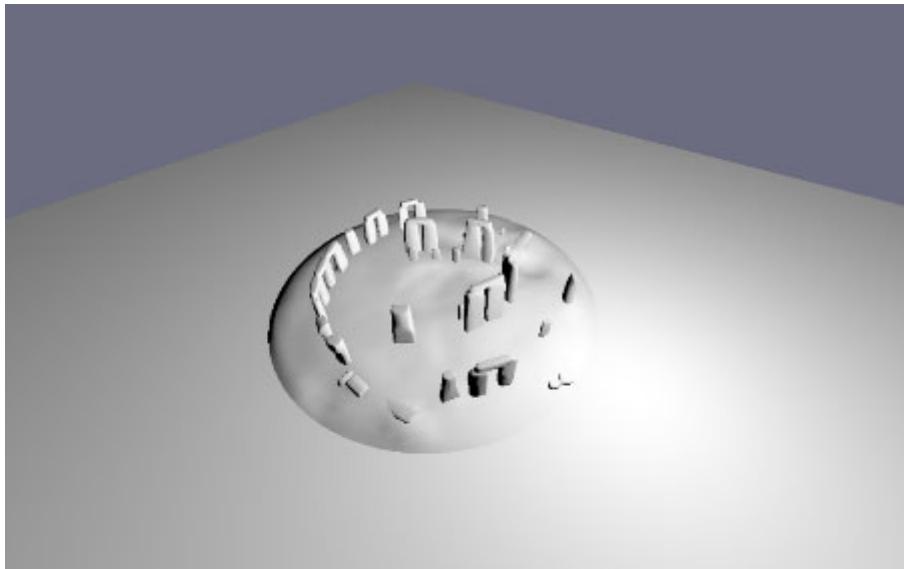
Sorting and Blending

- Blend fragments back to front in PS
- Must sort by depth value in temp.array



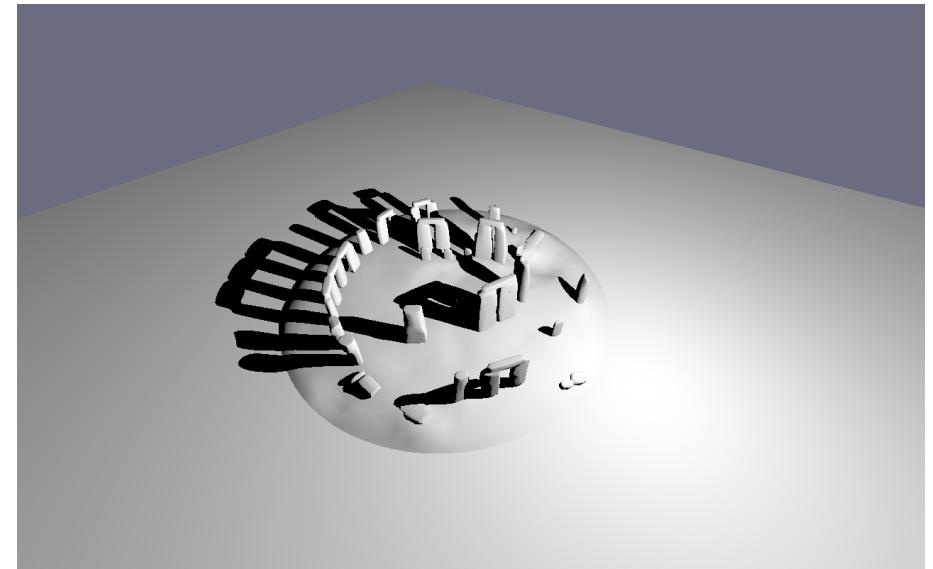
Shadows

- Light blocked by other objects
- Important visual cue
 - Relative location
 - Position of light

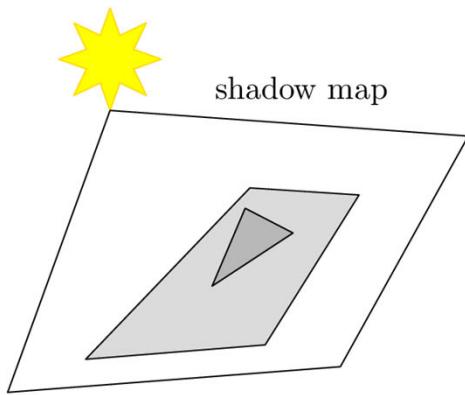


Dieter Schmalstieg

Semi-Global Illumination

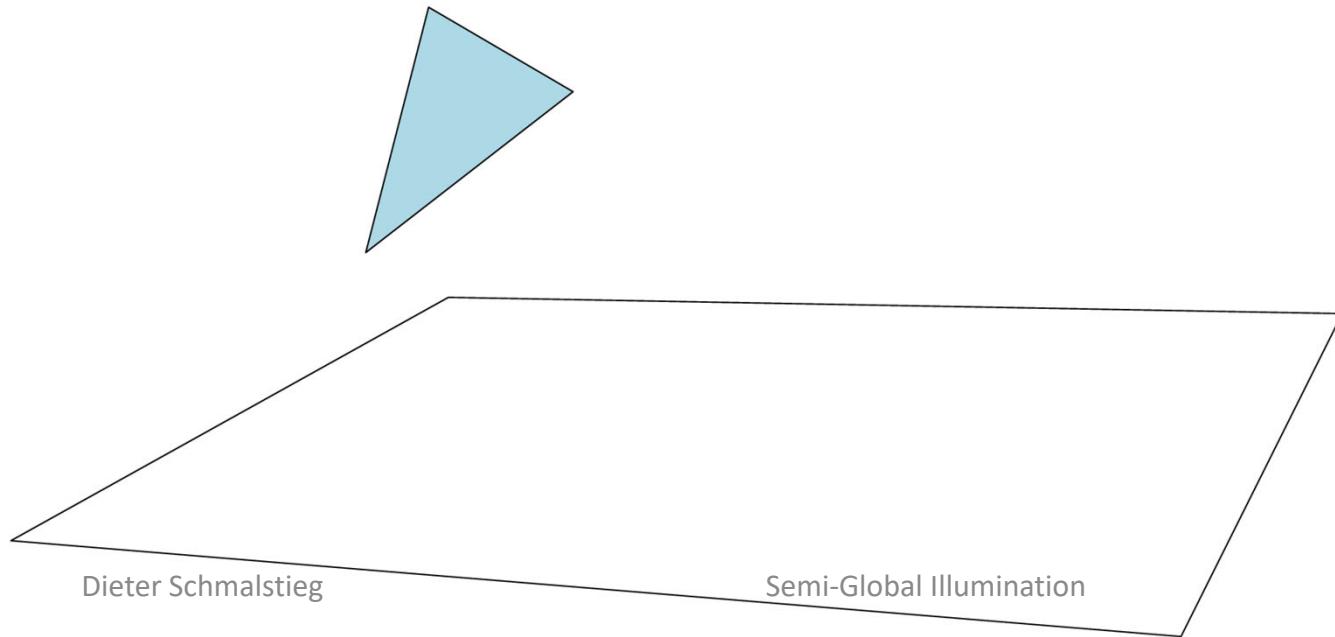


Shadow Mapping Algorithm 1

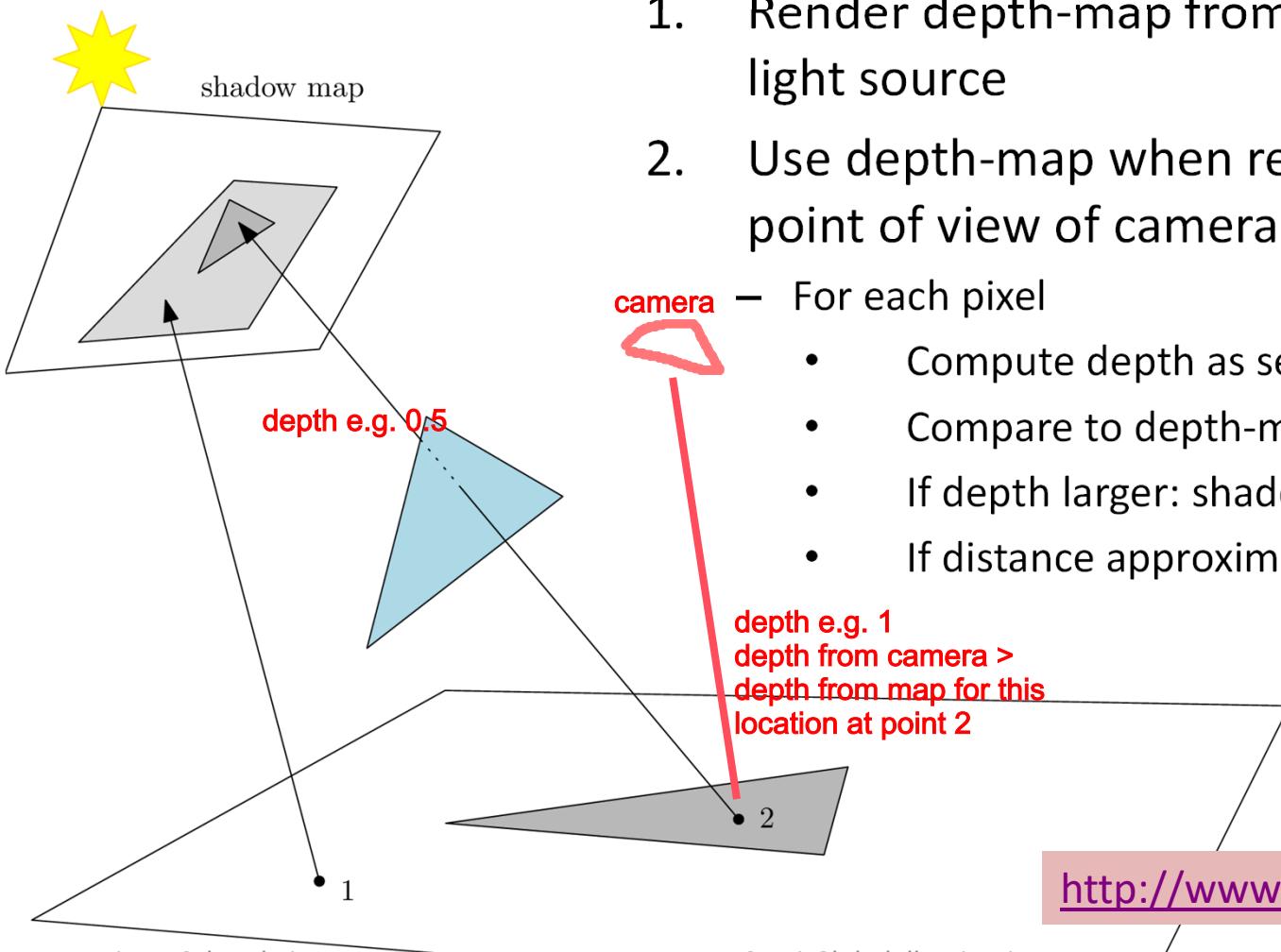


Two-step algorithm:

1. Render depth-map from point light source of view
2. Use depth-map when rendering scene from point of view of camera



Shadow Mapping Algorithm 2

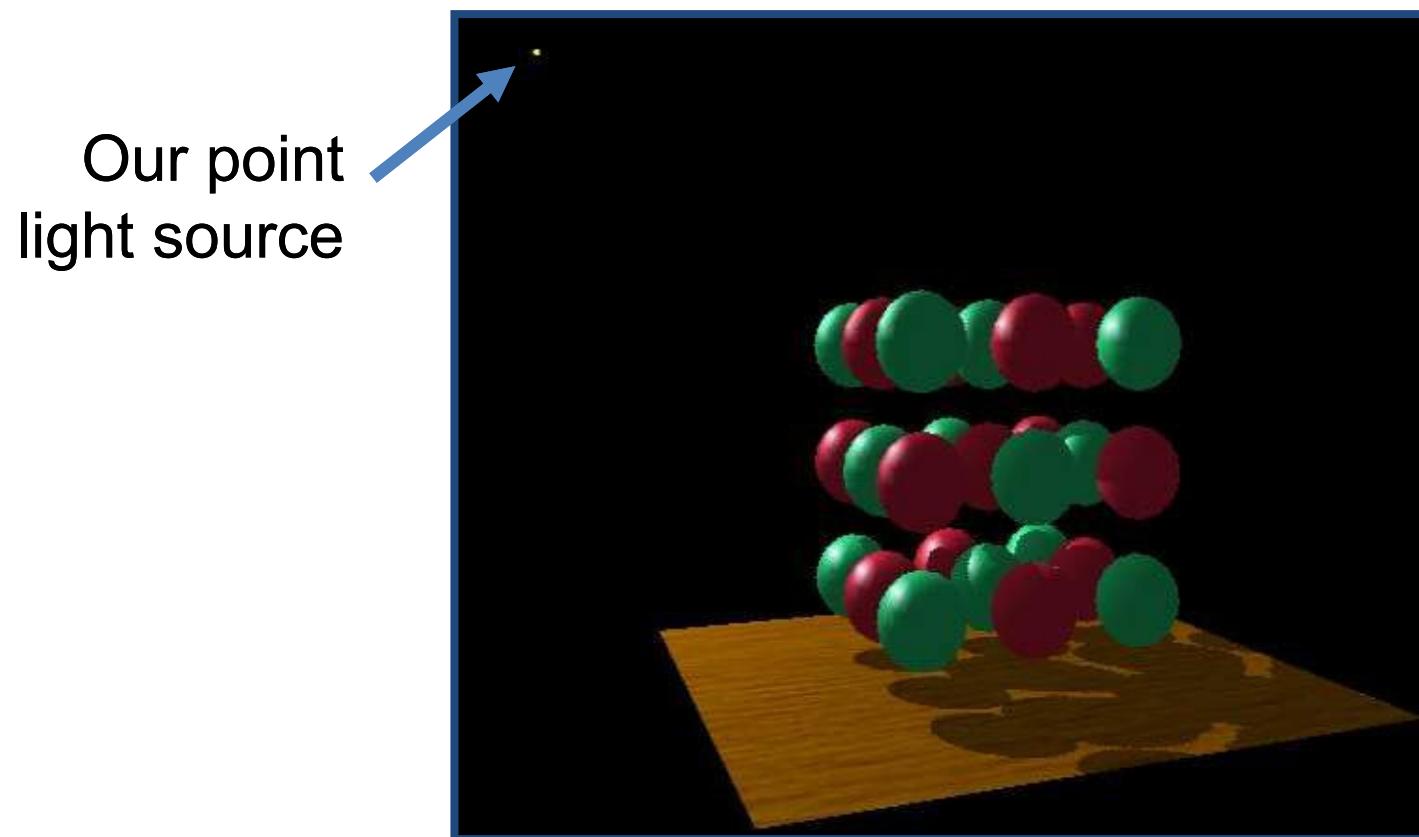


1. Render depth-map from point of view of the light source
2. Use depth-map when rendering scene from point of view of camera
 - For each pixel
 - Compute depth as seen from light source
 - Compare to depth-map
 - If depth larger: shadow
 - If distance approximately equal: light

<http://www.nutty.ca/webgl/shadows/>

Shadow Mapping - Point Light

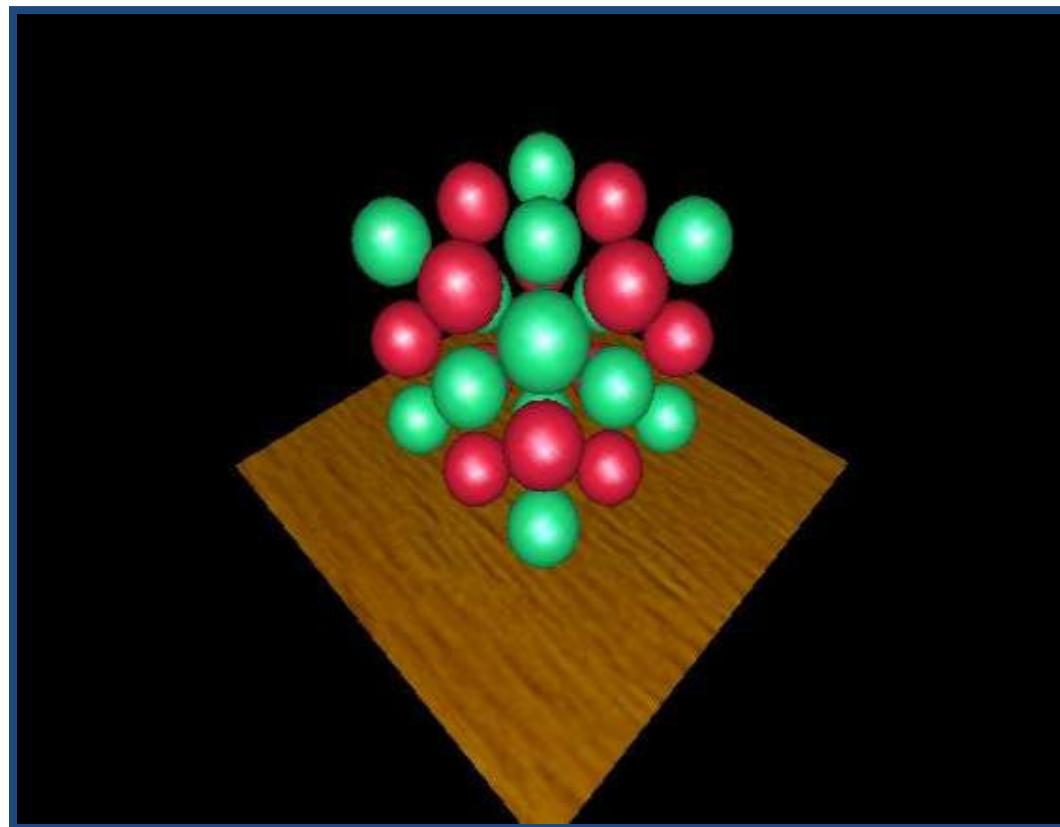
Beispiel des oben beschriebenen Hurensalgorithmus



Our point
light source

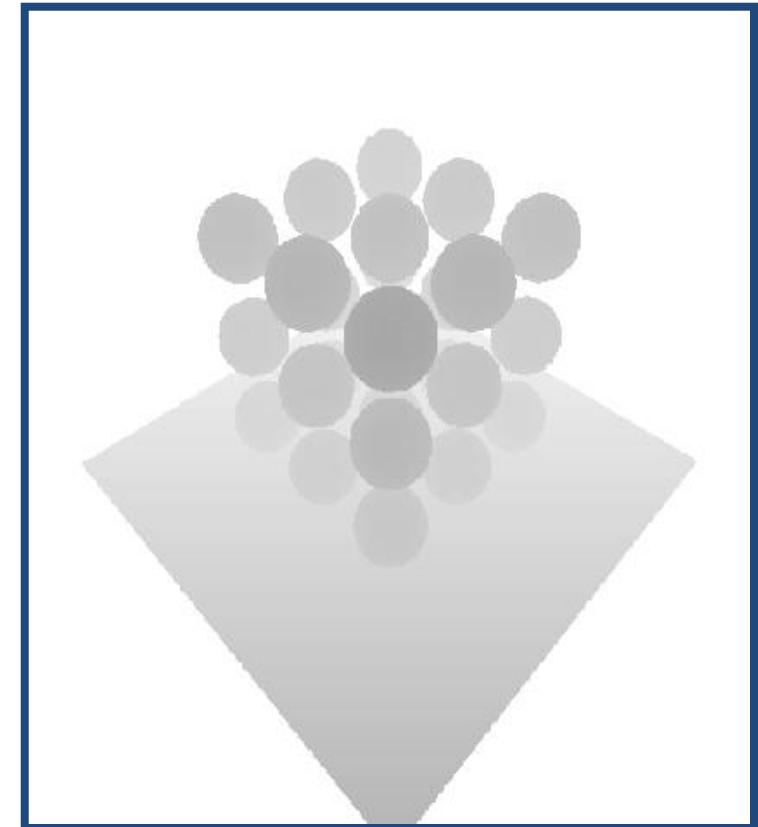
Shadow Map as Depth Map

- Rendering the Depth-Map



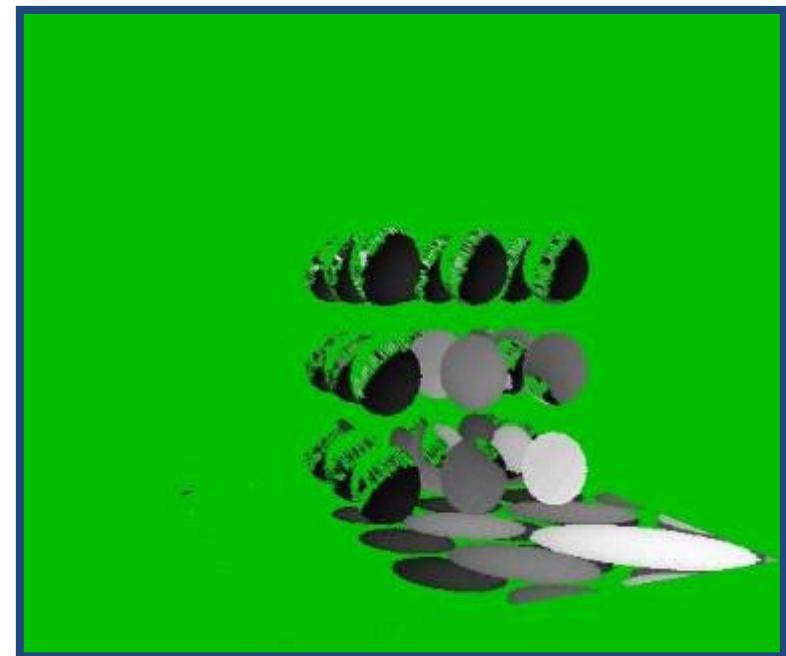
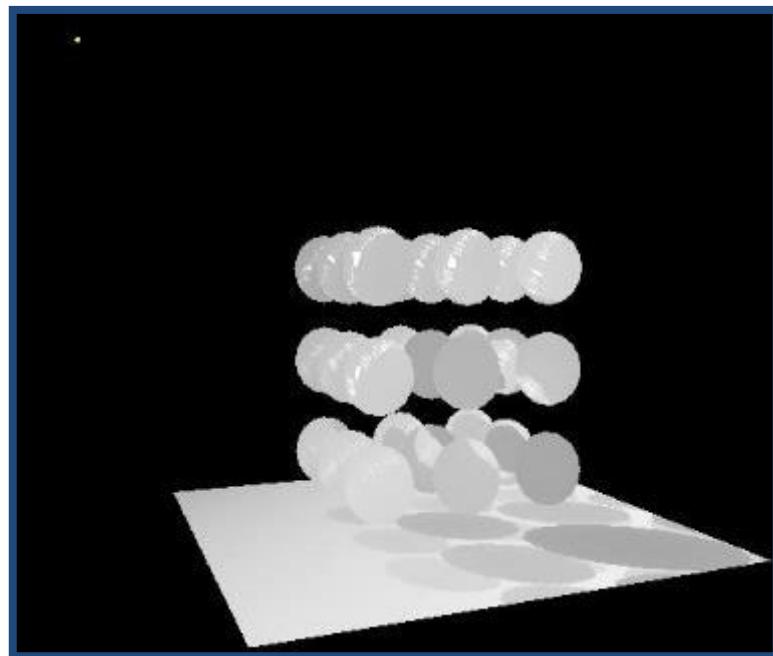
Dieter Schmalstieg

Semi-Global Illumination



Shadow Mapping - Depth Test

- Green: depth approximately equal
- Non-green: shadow



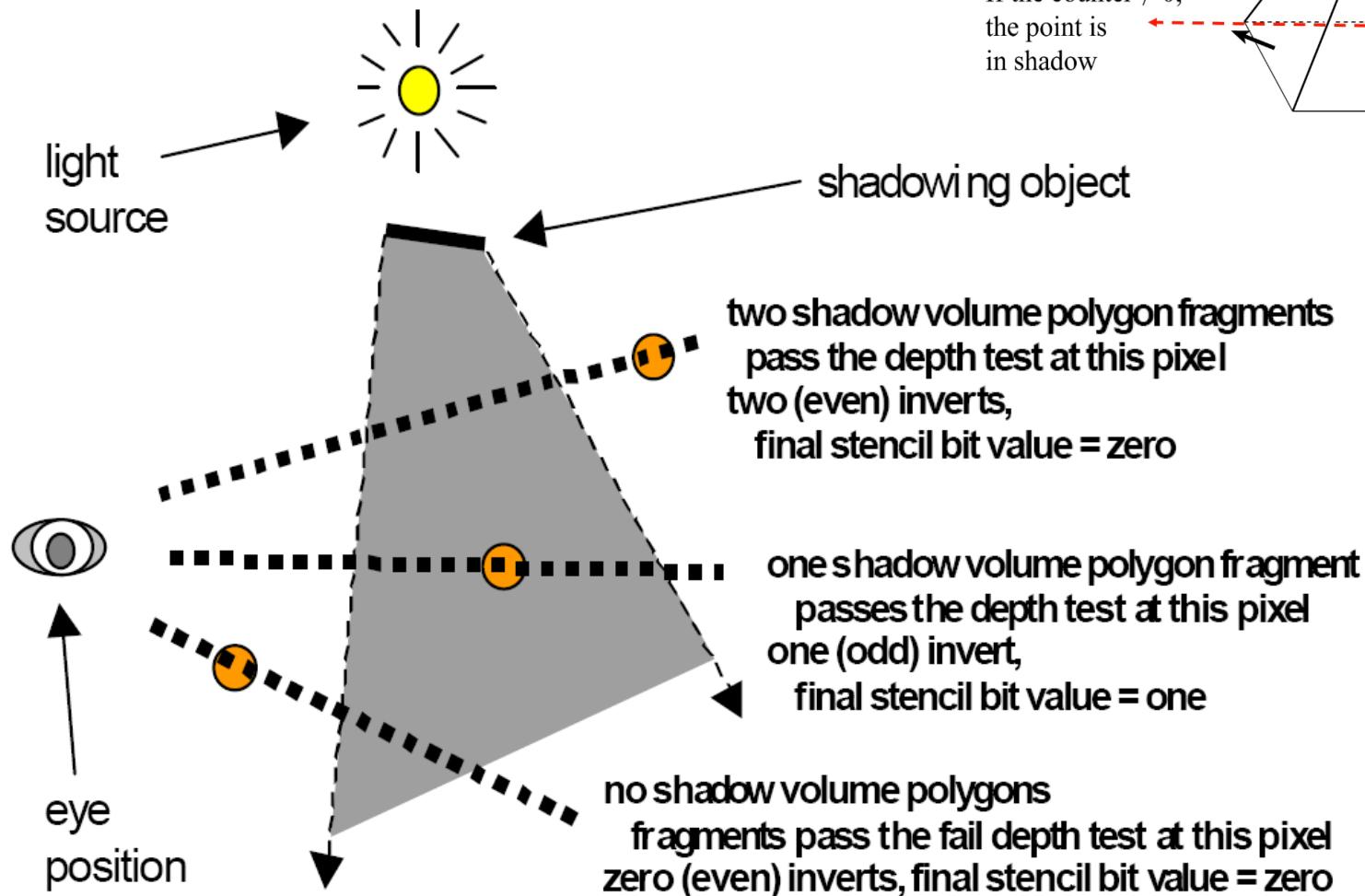
Shadow volumes sind 3D-Formen, die die Schattenbereiche einer Szene definieren, wenn eine Lichtquelle vorhanden ist. Sie werden verwendet, um Schatten von flachen und sich gegenseitig überlappenden Objekten zu berechnen.

Shadow Volumes

[Frank Crow, 1977]

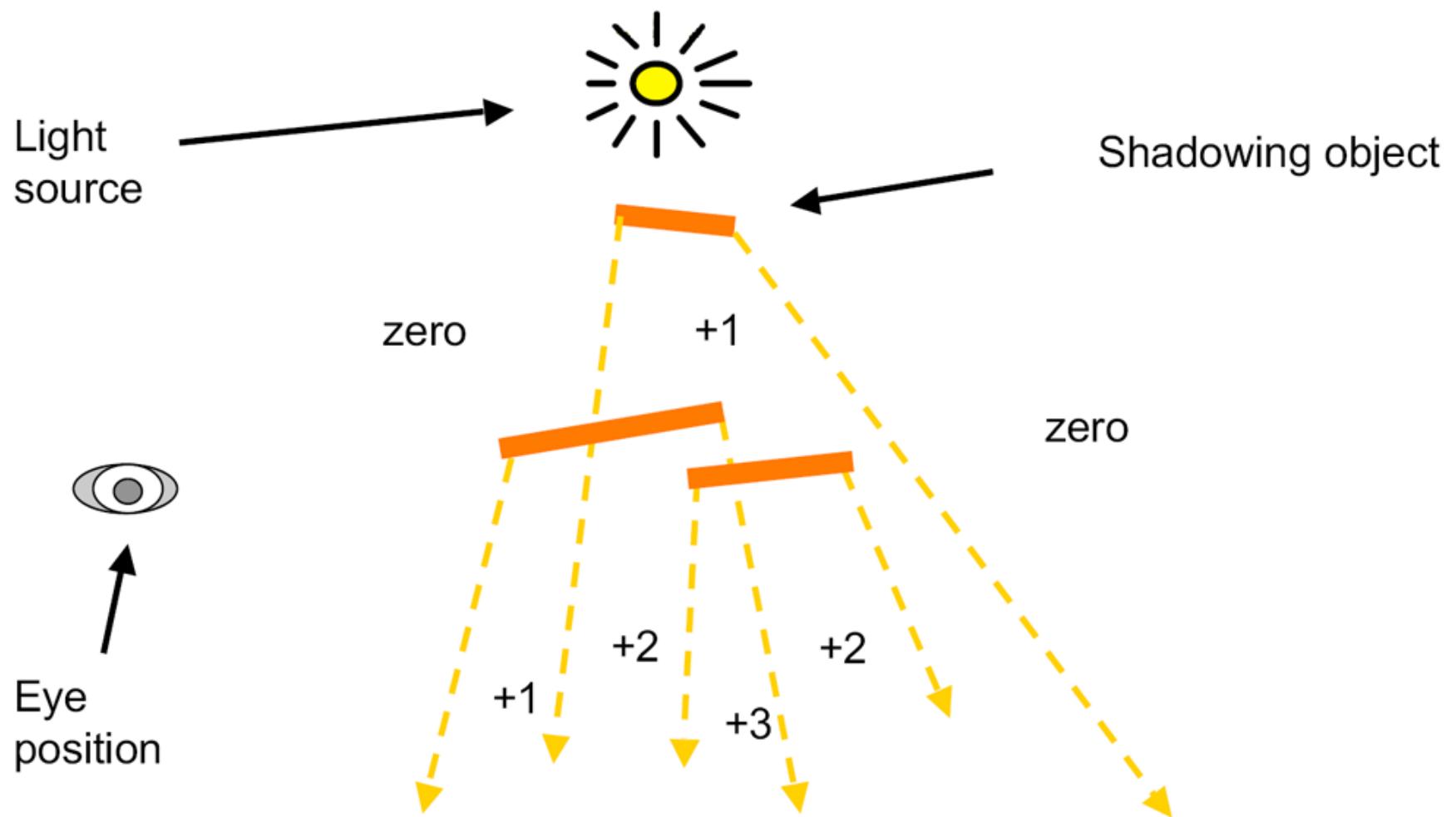
- Complex, but can be implemented efficiently using stencil buffer
- No aliasing
- Method
 - Intersect view rays with shadow volume
 - Count number of intersections, until receiver is hit





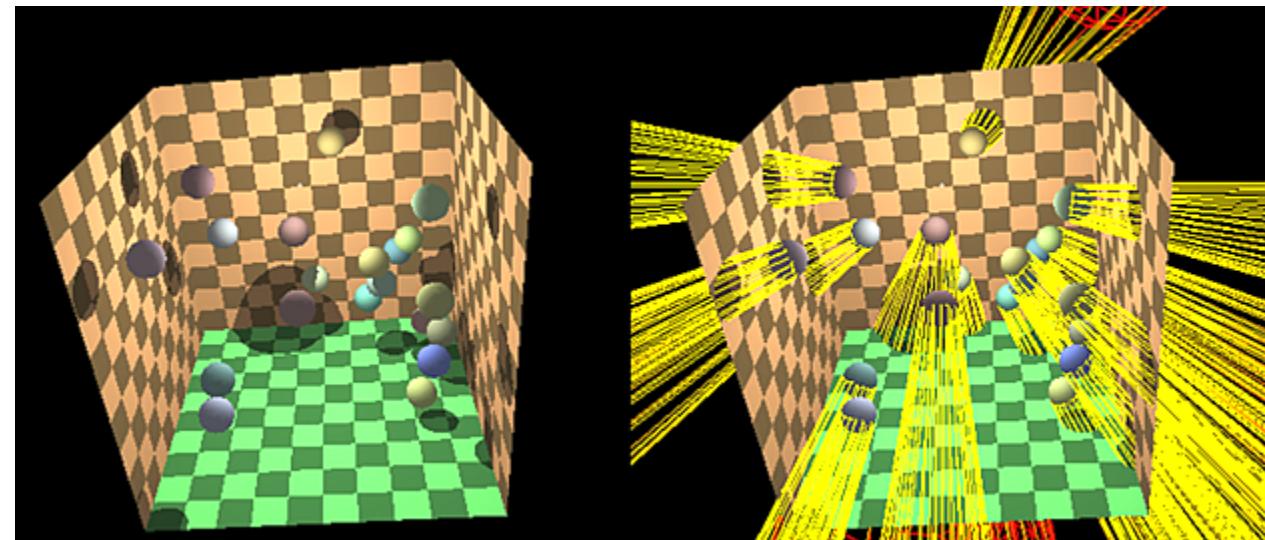
- Shoot a ray from the eye to the visible point
 - Increment/decrement a counter each time we intersect a shadow volume polygon
 - If the counter $\neq 0$, the point is in shadow
-
- A ray is cast from the eye (blue eye icon) through a point on the shadowed plane. The ray intersects several shadow volume polygons. As it enters a polygon, the counter is incremented (+1). As it exits, the counter is decremented (-1). If the counter reaches zero, the point is determined to be in shadow.

Counting Scheme



Shadow Volume Problems

- Camera inside shadow volume
 - Treat as special case or use “depth-fail” method
- Shadow volume intersects near-plane
 - Solution: render “front-caps”
- Objects must be manifold
- High amount of overdraw
- Fill-rate bound

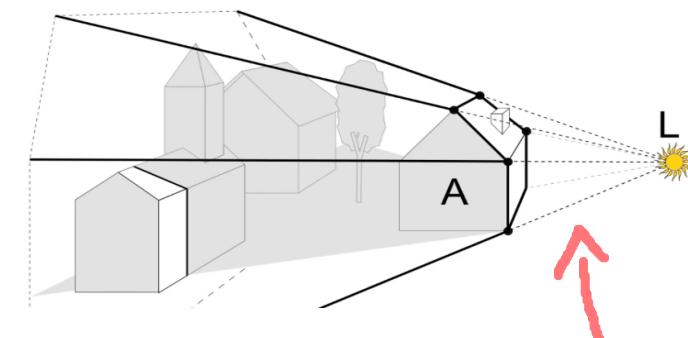


Shadow Volume Geometry

- Closed polyhedron with 3 sets of polygons
- Light cap
 - Object polygons facing the light
- Dark cap
 - Object polygons facing away from the light
 - Projected to infinity (with $w=0$)
- Sides
 - Actual extruded object edges
 - Which edges?

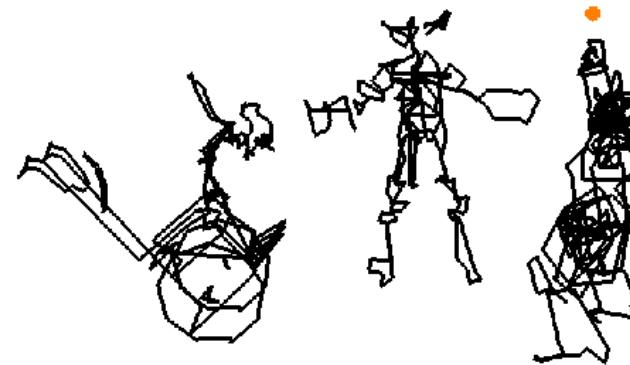
Optimizing Shadow Volumes

- Use silhouette edges only (edge where a back-facing & front-facing polygon meet)



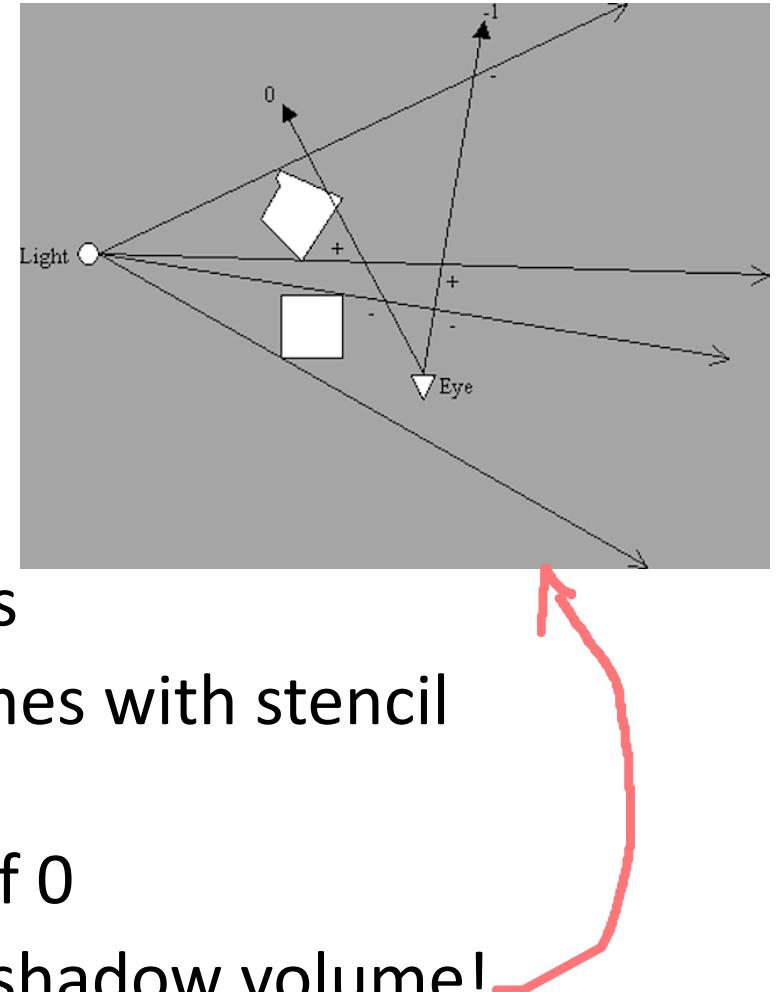
Silhouette Detection

- Classify polygons into frontfaces and backfaces
- Edge shared by front + back face is on silhouette
- Extrude silhouette edges away from light source
 - In vertex shader
- If necessary: add front-cap and back-cap



Stencil Shadow Volumes Algorithm

- Fill depth-buffer with scene
- Disable depth writes
- Render front-faces of stencil volumes with stencil increment on depth test pass
- Counts shadows in front of objects
- Render back-faces of stencil volumes with stencil decrement on depth test pass
- All lit surfaces have stencil value of 0
- Incorrect if the camera is inside a shadow volume!



Ambient Occlusion ist eine Technik, die in der Computergrafik verwendet wird, um Schatten und Tiefe in 3D-Modellen zu berechnen und zu simulieren. Die Berechnung von Ambient Occlusion findet pro Vertex (pro Eckpunkt) statt.

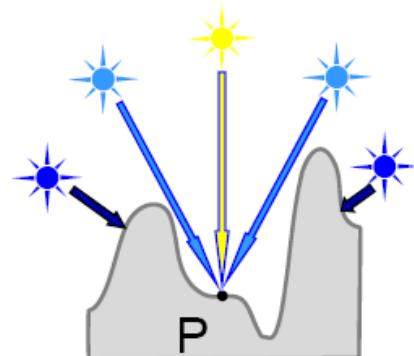
Ambient Occlusion

- Compute (per vertex)
 - Mean visibility $V_{AO} = 1 - AO$
 - Bent normal vector pointing into direction of average visibility
 - Used for advanced lighting instead of surface normal
- Weigh ordinary shading using mean visibility

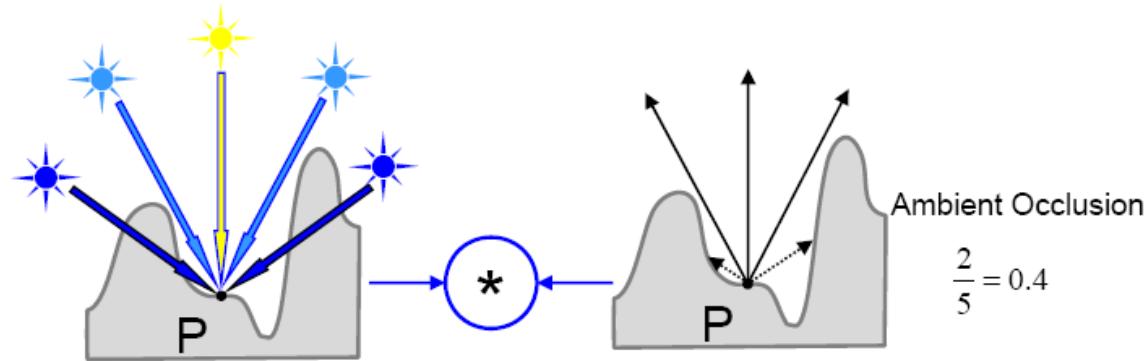
Die resultierende Information besteht aus einem Durchschnittswert für die Sichtbarkeit (VAO). Dieser Wert wird dann verwendet, um das übliche Beleuchtungsmodell zu gewichten und die Helligkeit des Modells zu beeinflussen.

Zusätzlich wird auch ein gebogener Normalvektor berechnet, der in Richtung der durchschnittlichen Sichtbarkeit zeigt. Dieser Vektor kann für eine fortgeschrittenere Beleuchtung verwendet werden, anstelle des Oberflächennormalvektors.

$$I = (1 - AO)(k_a I_a + k_d I_d \max(\mathbf{n} \cdot \mathbf{l}, 0))$$



Dieter Schmalstieg



Semi-Global Illumination

Mean Visibility (durchschnittliche Sichtbarkeit) ist ein Konzept, das in Verbindung mit Ambient Occlusion in der Computergrafik verwendet wird. Es bezieht sich auf den Prozentsatz an Licht, der an einem bestimmten Punkt in einer Szene ankommt.

Mean Visibility

Vorabberechnung durch Raycasting,

- Pre-computation = raycasting to find self-occlusions
- For each vertex
 - Cast n rays into the half sphere oriented by the normal
 - Count blocked rays m
 - Mean visibility $V_{AO} = 1 - m/n$
- Account for Lambert's law
 - Apply cosine distribution around normal vector
- To account for other nearby objects
 - Use ordinary raycasting with maximal distance
- Problem: raycasting from every point is expensive

Für jeden Vertex werden n Strahlen in eine halbkugelförmige Richtung geworfen, die von dem Normalvektor ausgerichtet ist. Die Anzahl der geblockten Strahlen (m) wird gezählt und die durchschnittliche Sichtbarkeit (VAO) wird berechnet, indem man 1 von (m/n) subtrahiert.

Zusätzlich wird auch Lamberts Gesetz berücksichtigt, indem eine Kosinus-Verteilung um den Normalvektor angewendet wird.

Um die Auswirkungen anderer nahegelegener Objekte zu berücksichtigen, wird ein gewöhnliches Raycasting mit einer maximalen Distanz verwendet.

Das Problem bei dieser Methode ist, dass Raycasting von jedem Punkt aus sehr rechenintensiv ist.

Ambient Occlusion



(a) Gouraud shading



(b) Ambient Occlusion

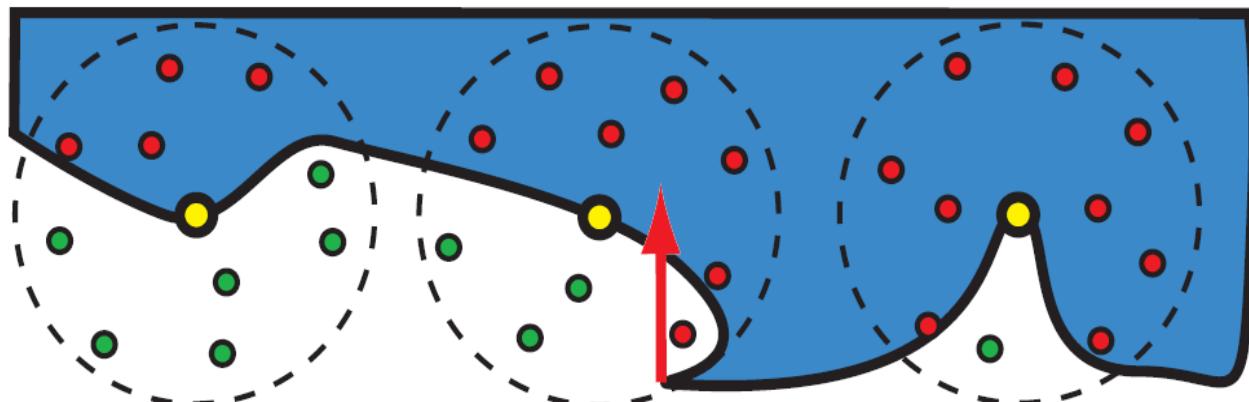


(c) Ground truth

<https://www.youtube.com/watch?v=9Fe1nYnvmiA>

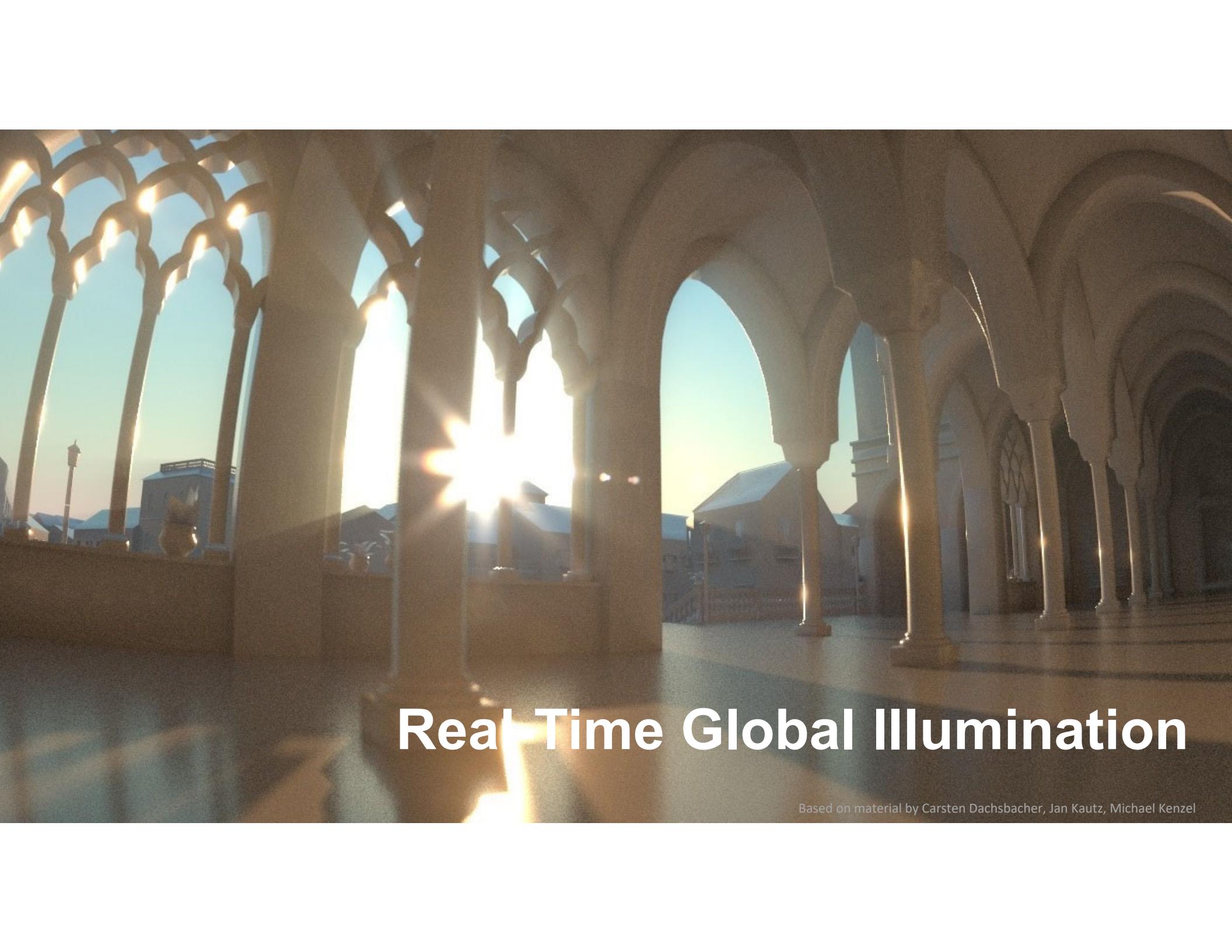
Screen Space Ambient Occlusion

- Z-buffer = approximation of surrounding
- Compute ambient occlusion from neighboring pixels
- Use random sampling of z-buffer
 - V_{AO} = Ratio occluded/unoccluded samples
 - L_{in} = incoming radiance
 - ω_i = incoming (=light source) direction
 - θ_i = angle to normal



Questions?





Real-Time Global Illumination

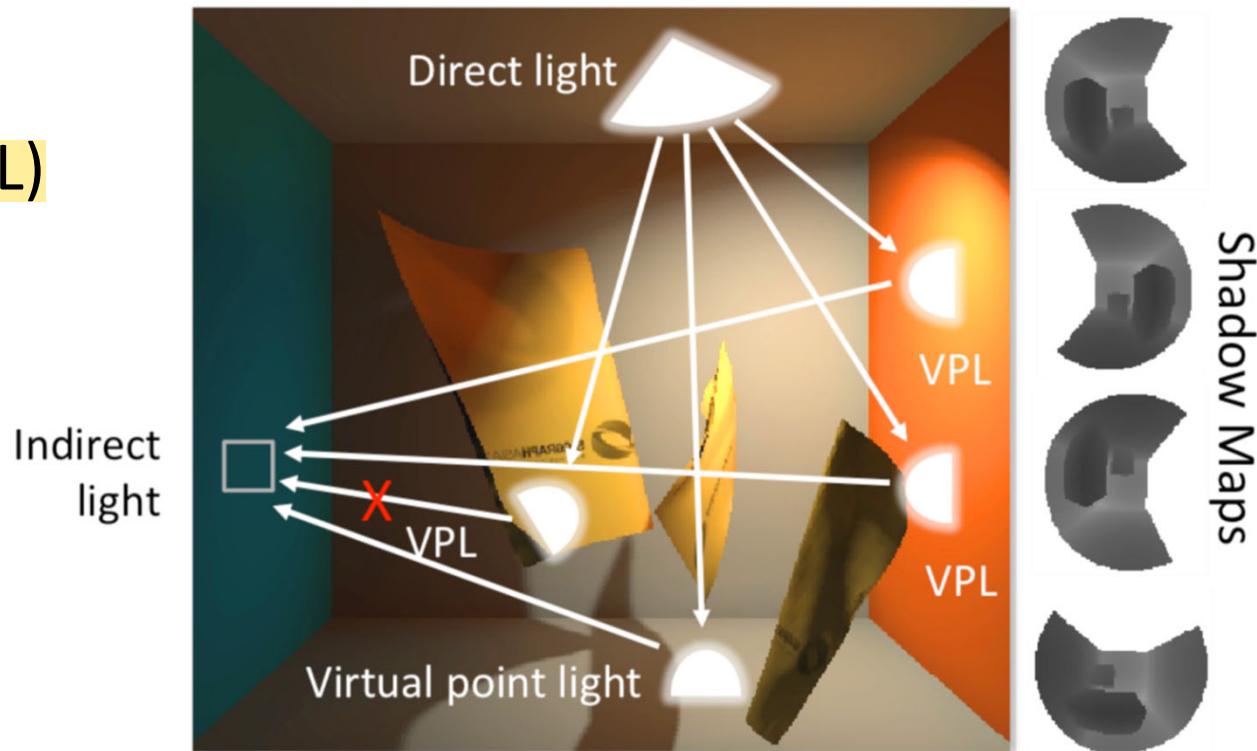
Based on material by Carsten Dachsbacher, Jan Kautz, Michael Kenzel

Instant Radiosity

- Idea: model the light bounces as light sources

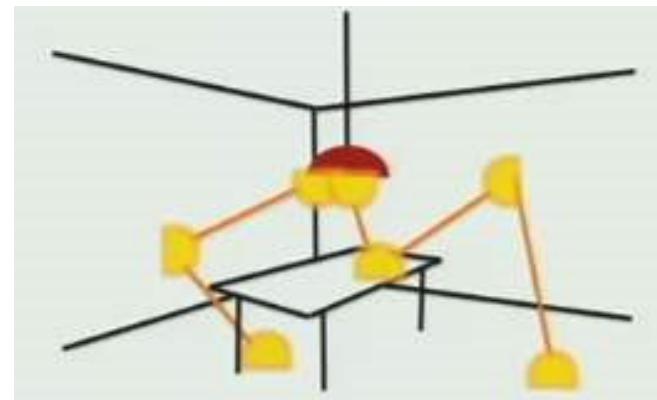
- Virtual point lights (VPL)

- Convert indirect into direct light transport



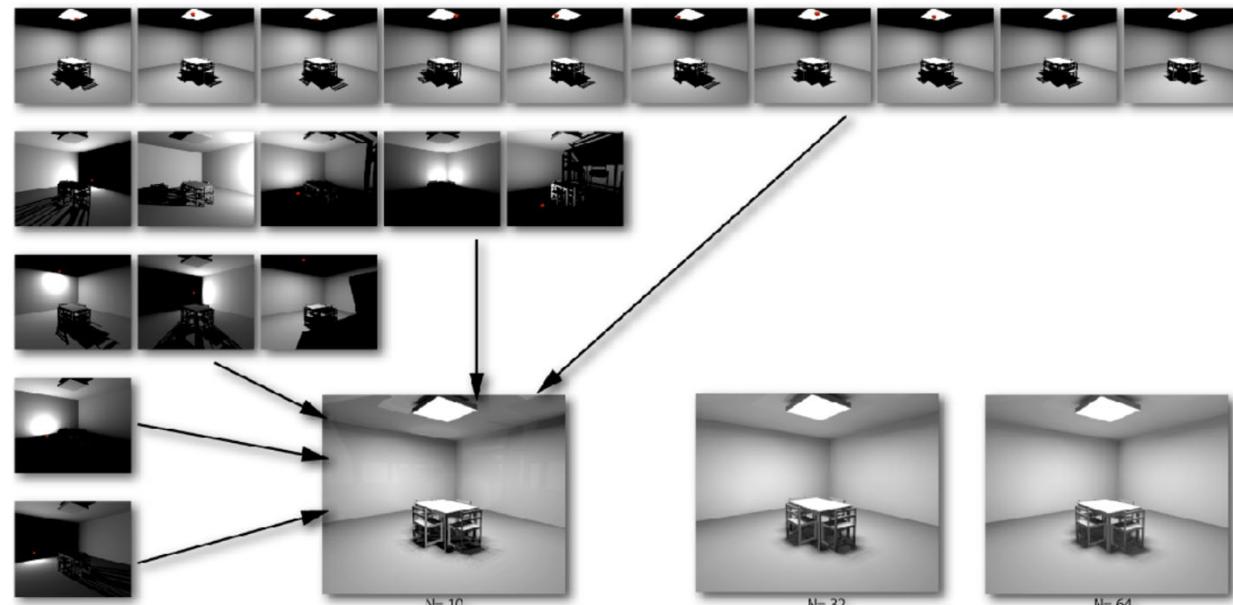
Virtual Point Light Sources

- Pass 1: VPL generation
 - Shoot photons from light source
 - Follow them through scene
 - At each hit point: create VPL
 - Russian roulette to end path
 - One shadow map for each VPL
- Pass 2: Deferred rendering
 - Render considering all the VPL and their shadow maps



VPL Distribution

- VPLs are distributed according to
 - $\rho^i = \text{av. Reflectance}, i = \text{bounce}$

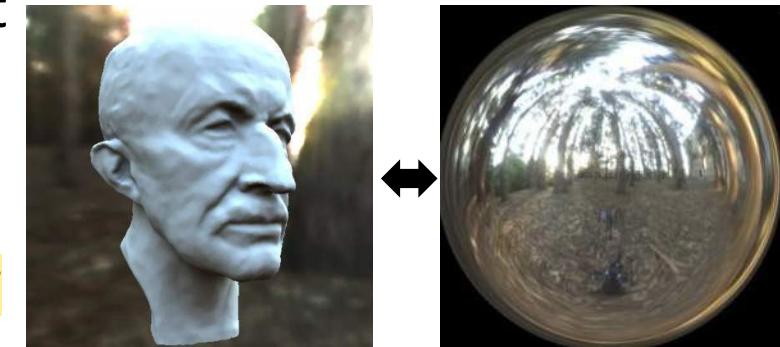


PRT berechnet und speichert Lichtenergie und Farbwert für jeden Punkt in der Szene und die Art und Weise, wie Licht zwischen den Punkten übertragen wird. Diese Informationen werden dann verwendet, um die Beleuchtung und Schatten in der Szene in Echtzeit zu berechnen.



Precomputed Radiance Transfer

- Realistic, interactive illumination of complex scenes
 - Complex materials
 - Dynamically changing lighting environment
 - General light sources
 - Shadows, interreflections, translucency
- Do not want to use real-time ray tracing



Method

Light sources and light transport are independent

1. Step: Precompute light transport

- E. g., with ray tracing
- Offline, slow and costly
- Store compressed representation
 - Light sources as an environment map
 - Light transport function for each surface point

2. Step: evaluate scene illumination

- During rendering, in real-time

1 Schritt: Vorabberechnung von Lichtübertragung:
Hier wird die Lichtübertragung in der Szene mit einer Technik wie Raytracing berechnet. Dies geschieht offline, ist jedoch langsam und kostspielig. Das Ergebnis dieser Berechnungen wird in einer komprimierten Darstellung gespeichert. Diese Darstellung beinhaltet die Lichtquellen als Umgebungskarte und eine Lichtübertragungsfunktion für jeden Punkt auf der Oberfläche.

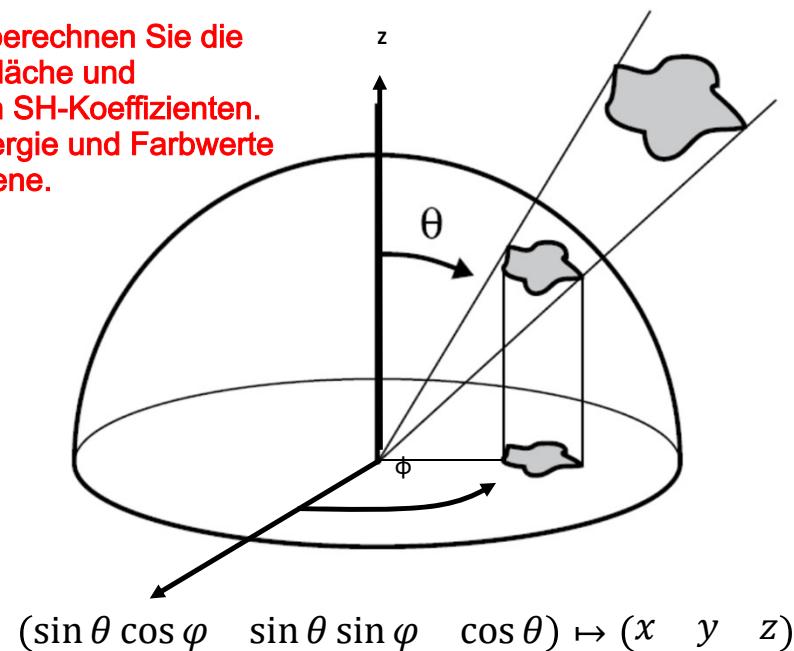
2 Schritt: Bewertung der Szenenbeleuchtung:
Während des Renderings werden die gespeicherten Informationen verwendet, um die Beleuchtung in Echtzeit zu berechnen. Da die Berechnungen bereits vorab durchgeführt wurden, ist die Bewertung der Szenenbeleuchtung schnell und effizient, und es müssen keine langsamen Berechnungen während des Renderings mehr durchgeführt werden.

Spherical Harmonics (SH) spielen eine wichtige Rolle bei der Implementierung von Precomputed Radiance Transfer (PRT). SH sind mathematische Funktionen, die verwendet werden, um die Umgebungsbeleuchtung in einer 3D-Szene zu modellieren.

Spherical Harmonics

Base functions for representing functions with a spherical domain

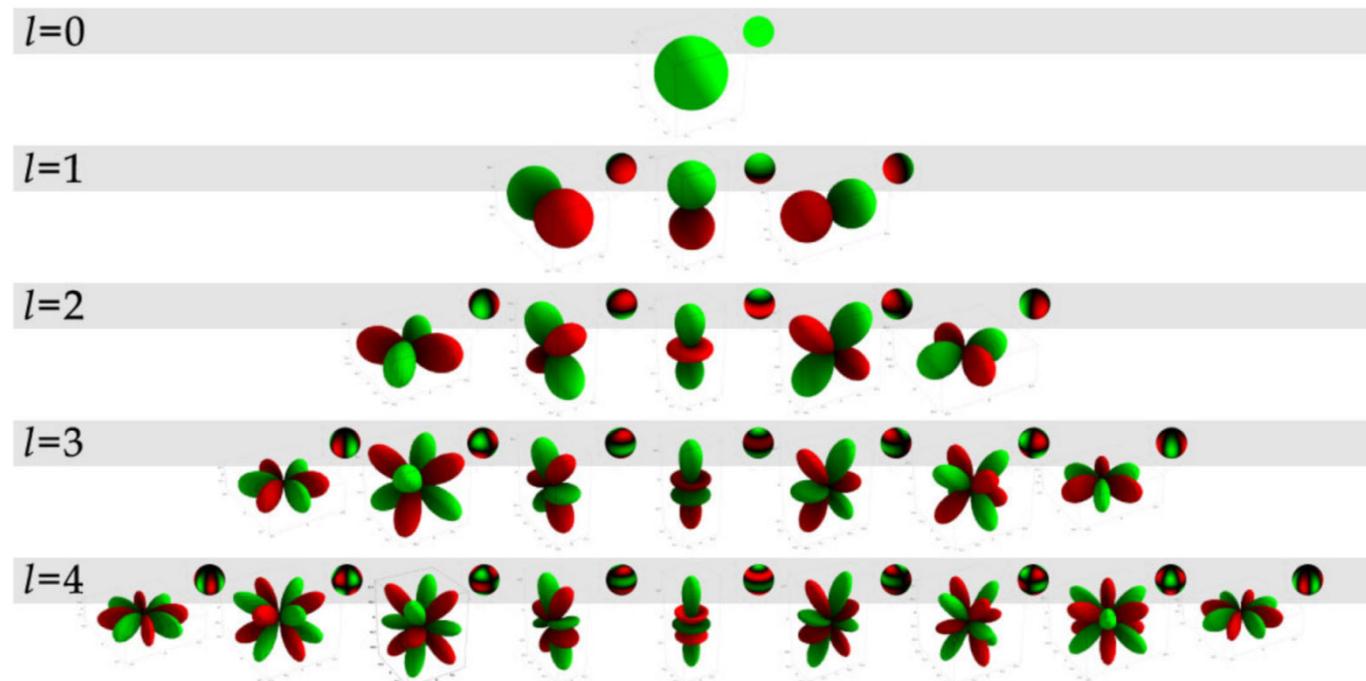
Während der Vorabberechnung von PRT berechnen Sie die Beleuchtung für jeden Punkt auf der Oberfläche und speichern diese Informationen in Form von SH-Koeffizienten. Diese Koeffizienten repräsentieren die Energie und Farbwerte der Beleuchtung für jeden Punkt in der Szene.



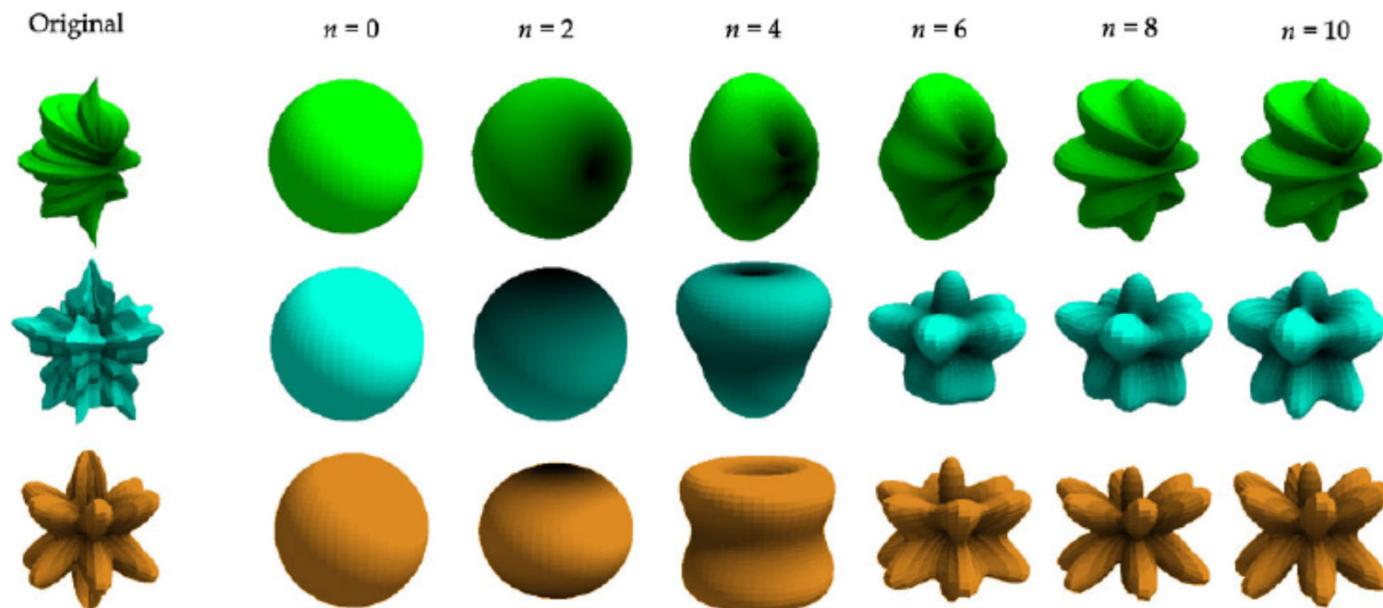
Während des Renderings können die berechneten SH-Koeffizienten verwendet werden, um die Beleuchtung in Echtzeit zu berechnen. Sie können auch verwendet werden, um Schatten und interne Reflexionen zu berechnen, indem Sie die Lichtübertragung zwischen den Punkten in der Szene modellieren.

Spherical Harmonics Bands

SH ... Basis functions for representing an arbitrary spherical function as a linear weighted sum



Examples for SH-Reconstruction



Spherical Harmonic (SH) Representation for Light Transfer ist eine Technik, bei der die visuelle Informationen, die bei der Berechnung von Ambient Occlusion (AO) gewonnen werden, in Form von SH-Koeffizienten gespeichert werden.

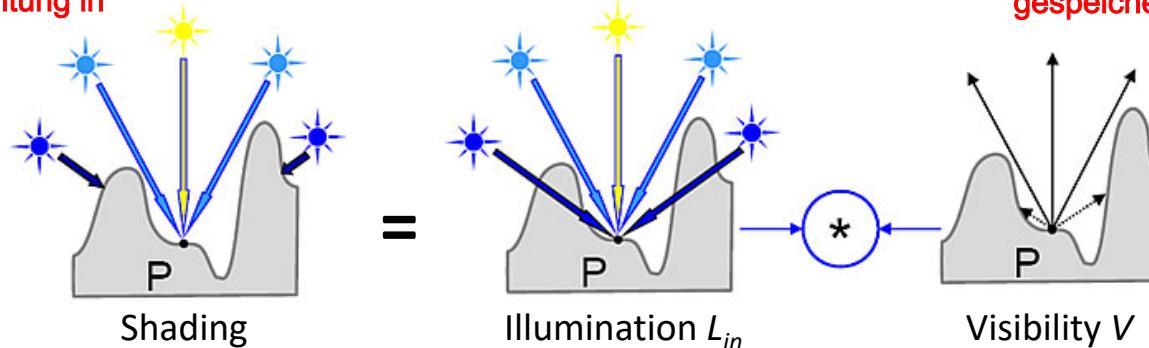
SH Representation for Light Transfer

- Directional visibility, computed as for AO
- But stored per-point as SH coefficients (vertex texture)

Diese SH-Koeffizienten repräsentieren die Informationen über die Beleuchtung und Schattierung, die auf einer Oberfläche eintreffen, und werden während des Renderings verwendet, um die Beleuchtung in Echtzeit zu berechnen.

$$I_{dir}(\mathbf{P}) = \sum_{i=1}^N \frac{\rho}{\pi} L_{in}(\omega_i) V(\omega_i) \cos \theta_i \Delta \omega$$

Während der Berechnung von Ambient Occlusion wird für jeden Punkt auf der Oberfläche ein Schätzwert für die Durchsichtbarkeit berechnet. Diese Informationen werden in Form von SH-Koeffizienten gespeichert und in einer Textur gespeichert, die pro Vertex verfügbar ist.

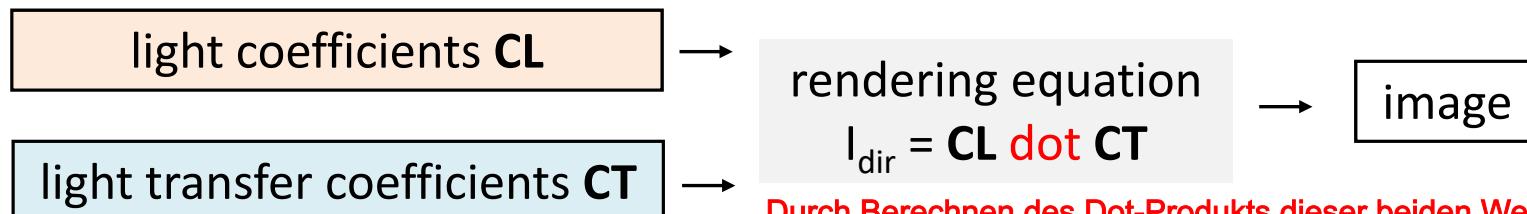
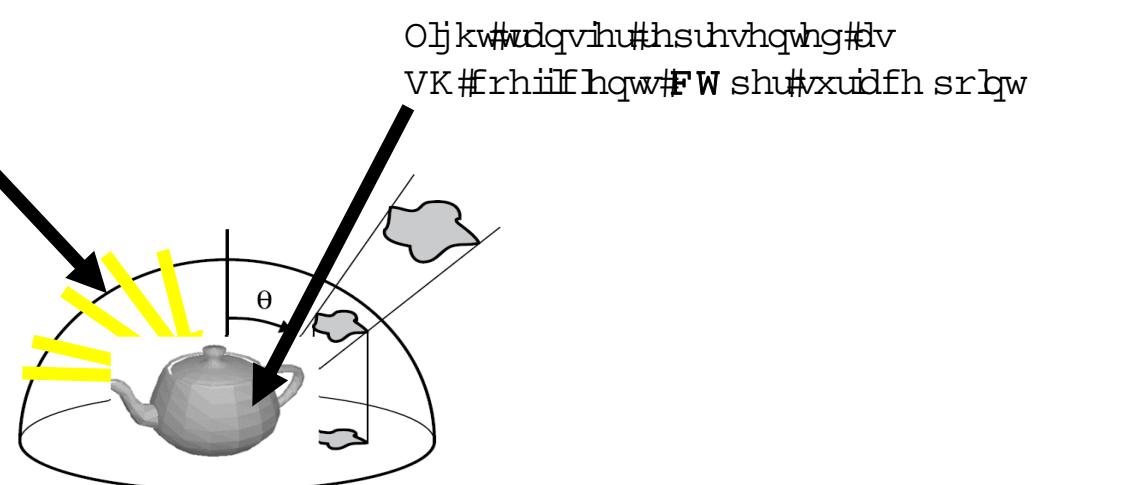


PRT Rendering

Light coefficients beschreiben, wie viel Licht von einer Lichtquelle auf ein bestimmtes Objekt trifft. Diese Informationen werden berechnet, indem eine bestimmte Anzahl von Strahlen aus der Lichtquelle in alle Richtungen gesendet werden und die Menge an Licht, die auf ein Objekt trifft, berechnet wird.

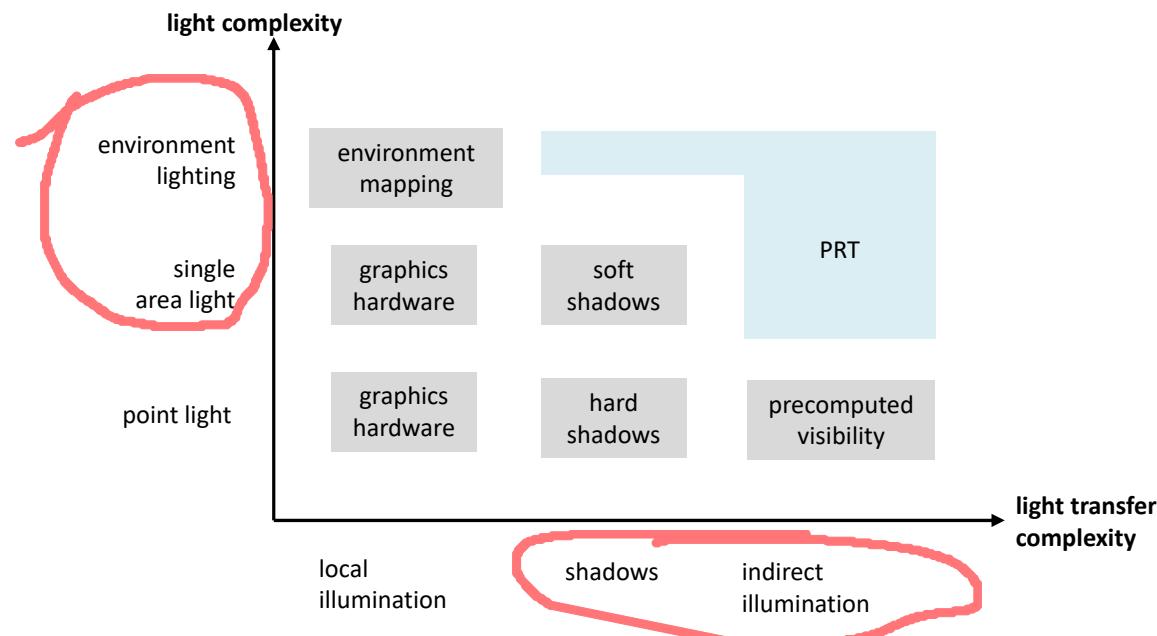
```
Obj#vrxufh#hsuhvhqwg#dv
VK#Erhiifhqw#FO
```

Light transfer coefficients beschreiben, wie das Licht von einem Punkt zu einem anderen Punkt auf einem Objekt übertragen wird. Diese Informationen werden berechnet, indem eine bestimmte Anzahl von Strahlen von einem Punkt zu einem anderen Punkt auf einem Objekt gesendet werden und die Menge an Licht, die übertragen wird, berechnet wird.



Durch Berechnen des Dot-Produkts dieser beiden Werte kann die Gesamtmenge an Licht berechnet werden, die von einer bestimmten Lichtquelle auf einen bestimmten Punkt auf einem Objekt trifft

PRT – Applications



PRT – Summary

- Advantages
 - Fast rendering (1 dot product)
 - Dynamic lighting environments
 - Supports complex light transfer (ambient occlusion and interreflections)
- Disadvantages
 - Only efficient for lighting environments of low-frequency
 - Typically 9 to 16 SH coefficients (3 to 4 bands)
 - Static scenes
 - Only diffuse surfaces (if using SH-based compression)



GPU Raytracing

Dieter Schmalstieg



DirectX Ray Tracing Pipeline

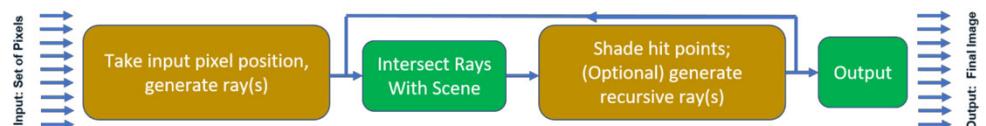
- Pipeline is split into five new shaders:
 - Ray generation shader** defines how to start ray tracing
 - Intersection shader** define how rays intersect geometry
 - Miss shader** define behavior when rays miss geometry
 - Closest-hit shader** run once per ray (e.g., to shade final hit) **when geometry hit**
 - Any-hit shader** run once per hit (e.g., to determine transparency)

Traversal Loop

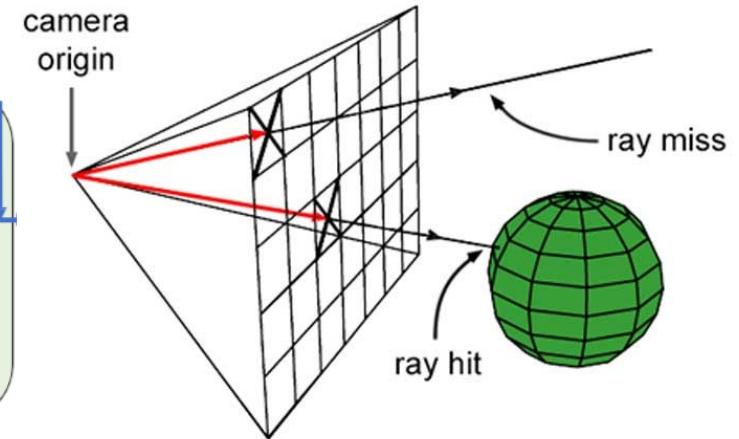
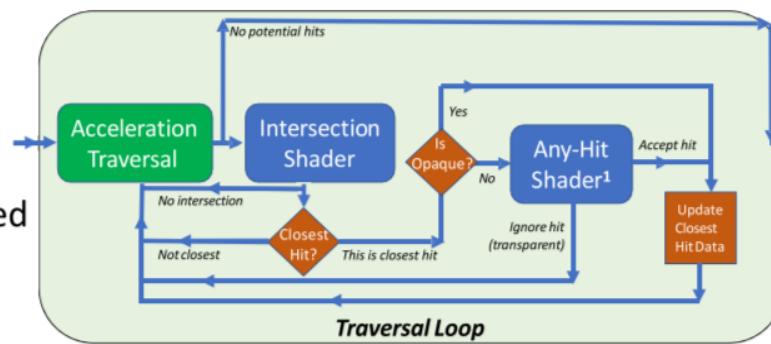
- If all geometry trivially ignored
 - Ray traversal ends
- For potential intersections
 - Call **intersection shader** is invoked
 - Triangles (special hardware), spheres, Bezier patches etc.
- If no intersection detected or not closest, continue traversal
- If hit
 - If transparent, run **any-hit shader** (call `IgnoreHit()` to continue)
 - Update closest hit data, continue to look for closer intersections

Tracing A Ray

- First, we traverse our scene to find what geometry ray hits
- When we find the closest hit, shade the point using **closest-hit shader**
- If ray misses all geometry (hits background), **miss shader** gets invoked



- } **Runs once per algorithm (or per pass)**
- } **Defines geometric shapes, widely reusable**
- } **Defines behavior of rays**
Different between shadow, primary, indirect rays



Output, camera data, scene geometry is

Image courtesy of Epic Games

A wide-angle photograph of a majestic, snow-covered mountain range under a clear blue sky. Light snow is falling from the sky, creating a sense of depth and atmosphere. The mountains have sharp, rugged peaks and deep, shadowed valleys. The lighting suggests either early morning or late afternoon, with warm sunlight hitting the peaks.

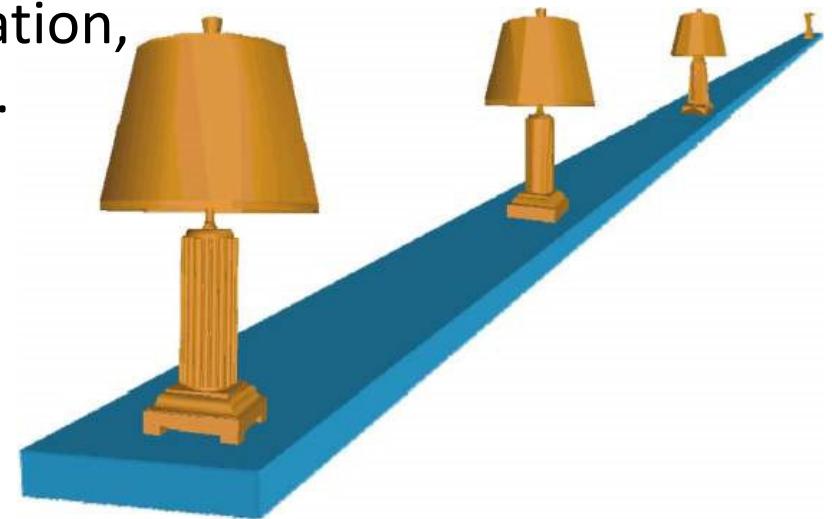
Dieter Schmalstieg

Levels of Detail

Based on material from Michael Wimmer and Markus Grabner

LOD - Basic Idea

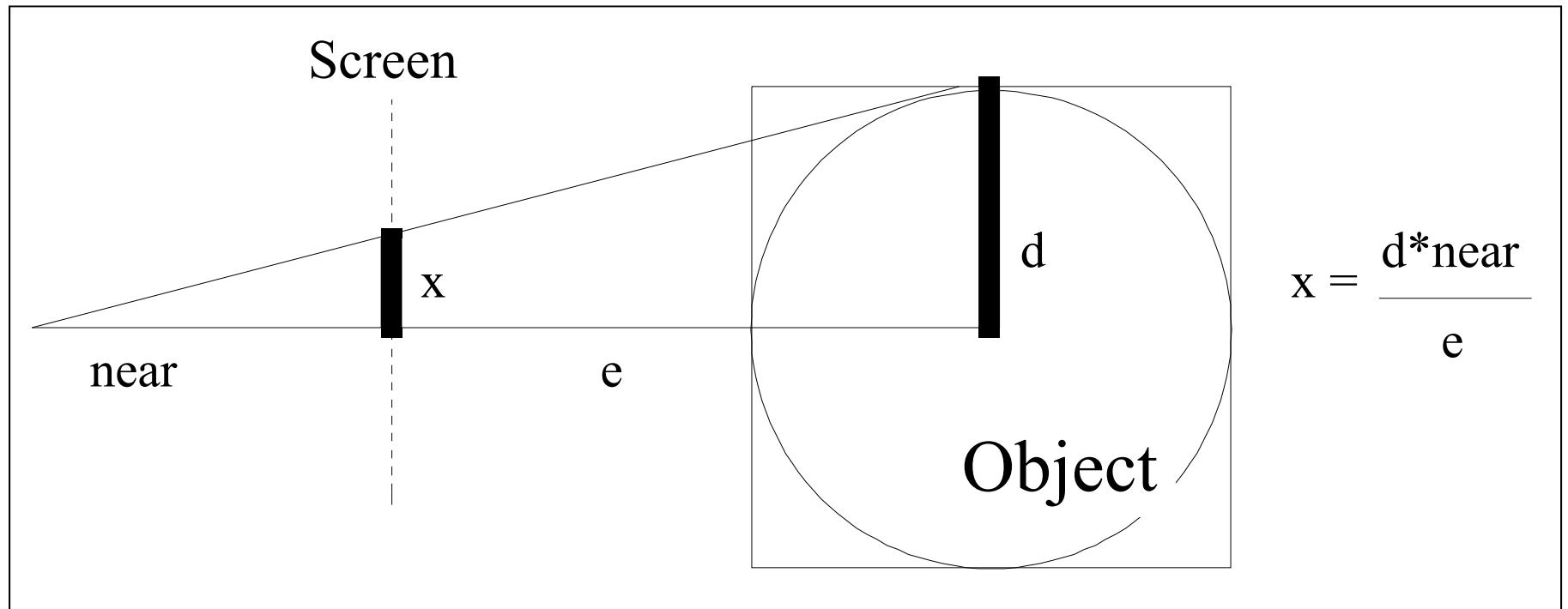
- Problem: even after visibility, model may contain too many polygons
- Idea: Simplify the amount of detail used to render small or distant objects
- Known as levels of detail (LOD)
A.k.a. multiresolution modeling,
polygonal/geometric simplification,
mesh reduction/decimation, ...



<https://www.youtube.com/watch?v=mlkIMgEVnX0>

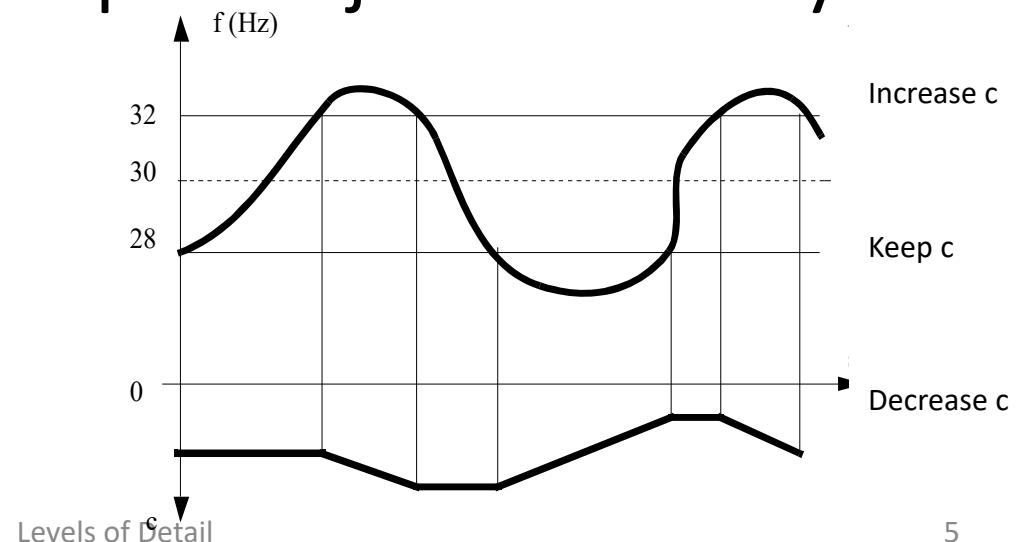
Static LOD Selection

- LOD Selection steered by size of object in image
- Cannot control resulting frame rate



Reactive LOD Selection

- Multiply object size with factor c
- If frame rate too low \rightarrow decrease c
- If frame rate too high \rightarrow increase c
- Results in roughly constant frame rate
- Problems occur, if complex objects suddenly become visible
- Requires hysteresis



Predictive LOD Selection

[Funkhouser & Sequin 1993]

- COST = time for drawing object with a given LOD
- Goal: best possible image quality
- BENEFIT = contribution of object to image quality
 - Most important: screen-size of object
- Optimization (*Rucksack*) problem
 - Sum(BENEFITS) → max, BUT
 - Sum(COSTS) ≤ FRAMETIME

LOD Switching

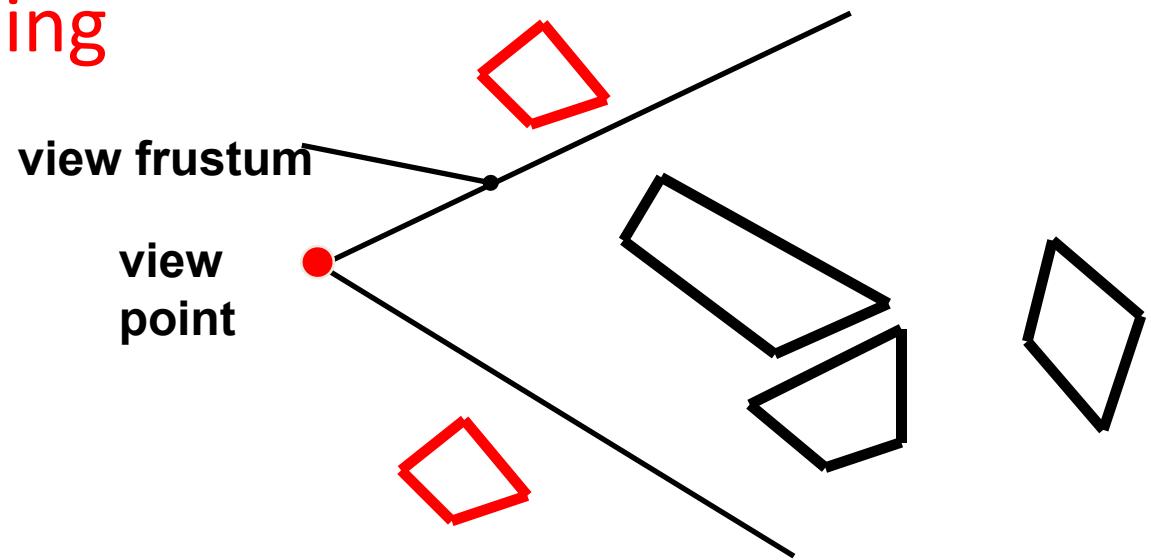
- Hard Switching
 - + Simple
 - “Popping” artefacts
- Blending
 - + For all types of LOD
 - Temporarily increased rendering load
 - Problems with transparencies, shadows, etc...
- Geomorphing https://www.youtube.com/watch?v=I20Zyr4U_Xk
 - Interpolate triangle shapes from 1st to 2nd LOD
 - + Best quality
 - Requires geometric correspondence between LODs



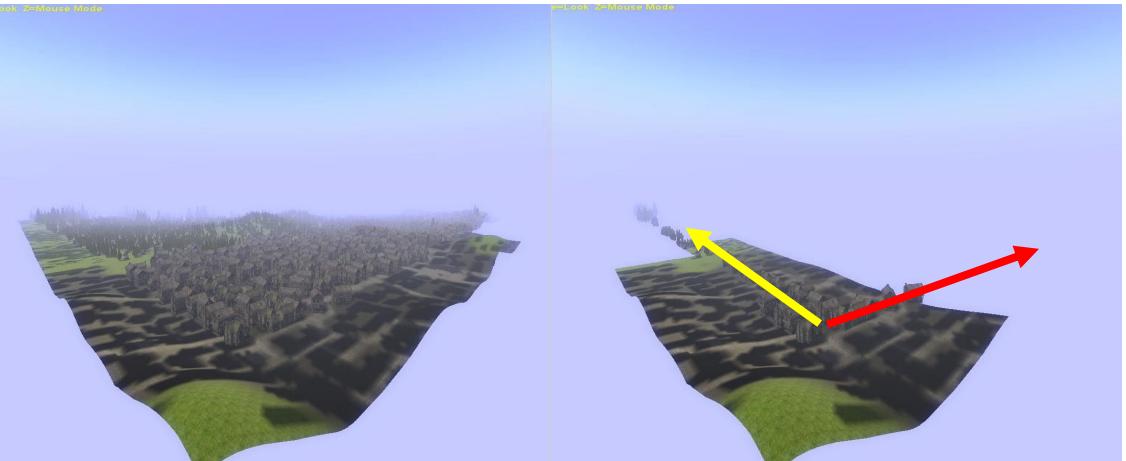
Visibility

View Frustum Culling

- View-frustum culling
- Occlusion culling
- Backface culling



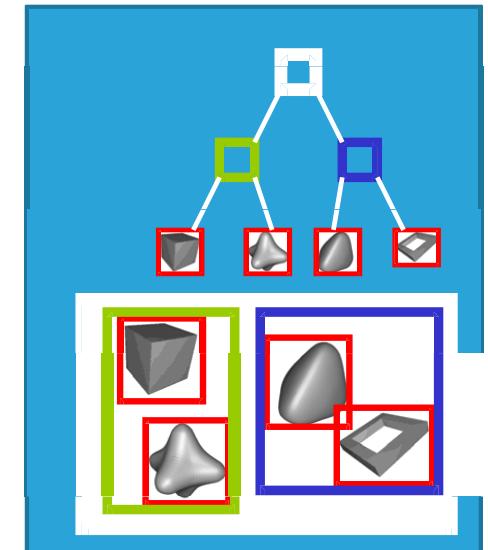
Dieter Schmalstieg



Visibility

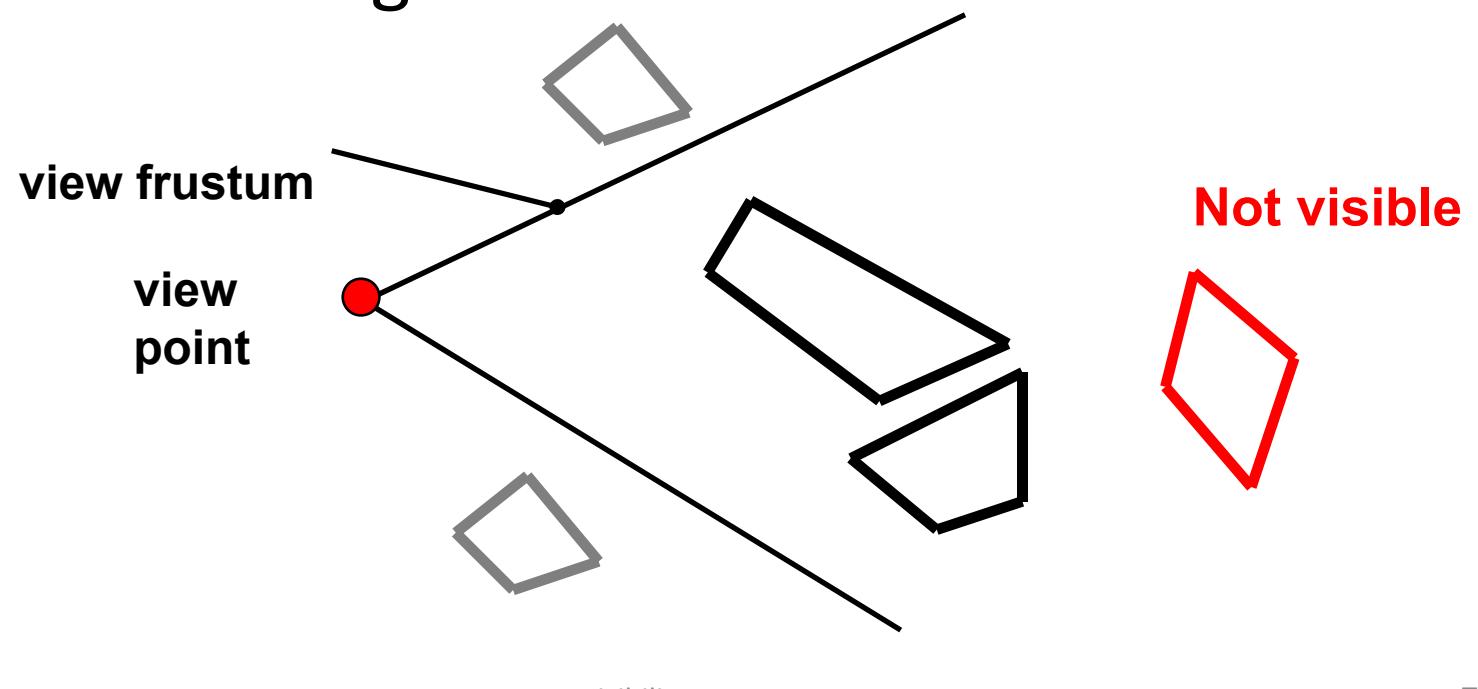
Bounding Volume Hierarchy

- Most important spatial data structure
- Common bounding volume types
 - Spheres
 - Axis-aligned bounding box (AABB)
 - Oriented bounding box (OBB)
- Encloses the bounded object
- Hierarchical data structure
 - Tree (binary or n -ary)
 - Leaves: store objects
 - Interior nodes: stores bounding volume enclosing all contained bounding volumes



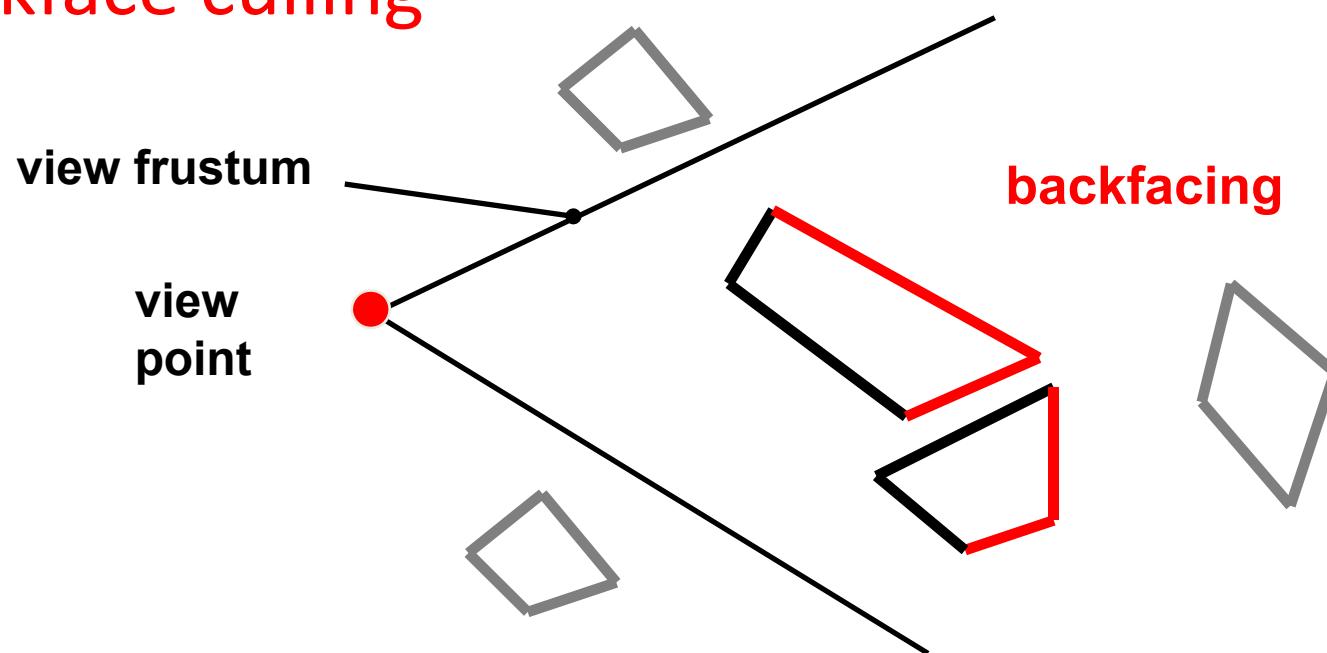
Occlusion Culling

- View-frustum culling
- Occlusion culling
- Backface culling



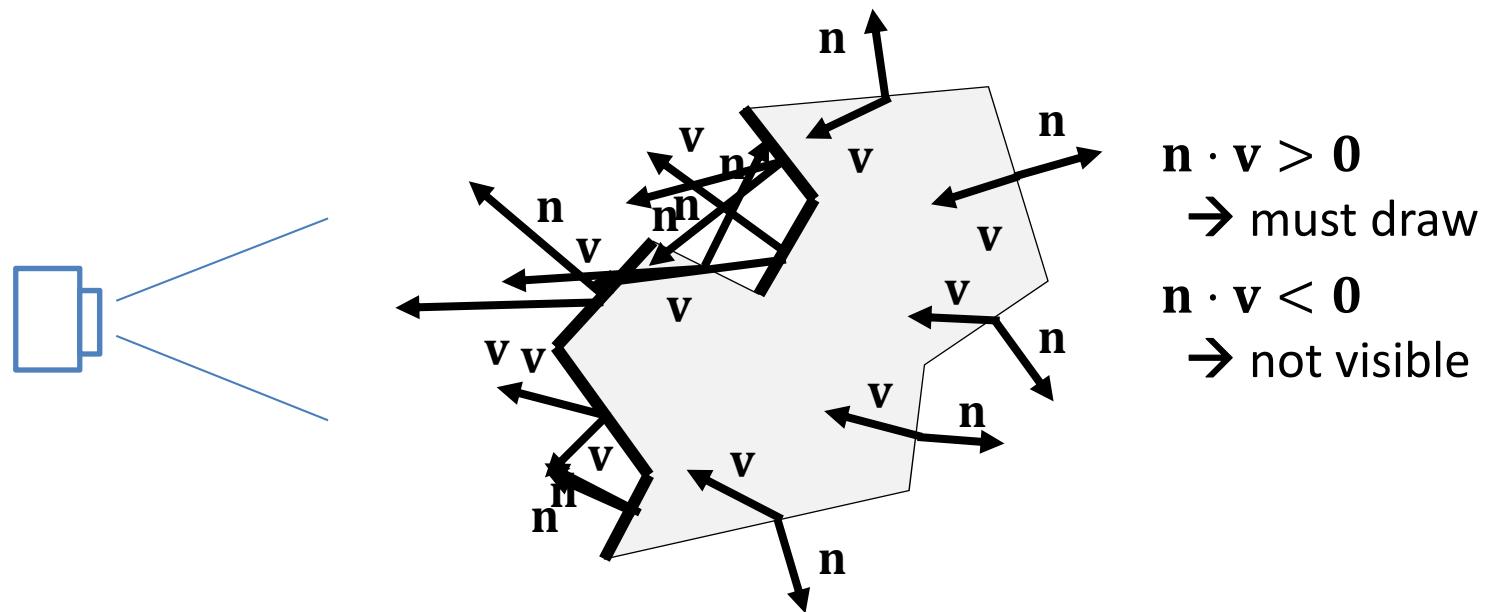
Backface Culling

- View-frustum culling
- Occlusion culling
- Backface culling



Backface Culling

- If an object is watertight, we cannot see the interior
- We only must draw those primitives facing the camera
- Can save up to 50% of primitives on average
- Simple to implement using dot product of normal \mathbf{n} + view vector \mathbf{v}



Exactly Visible Set

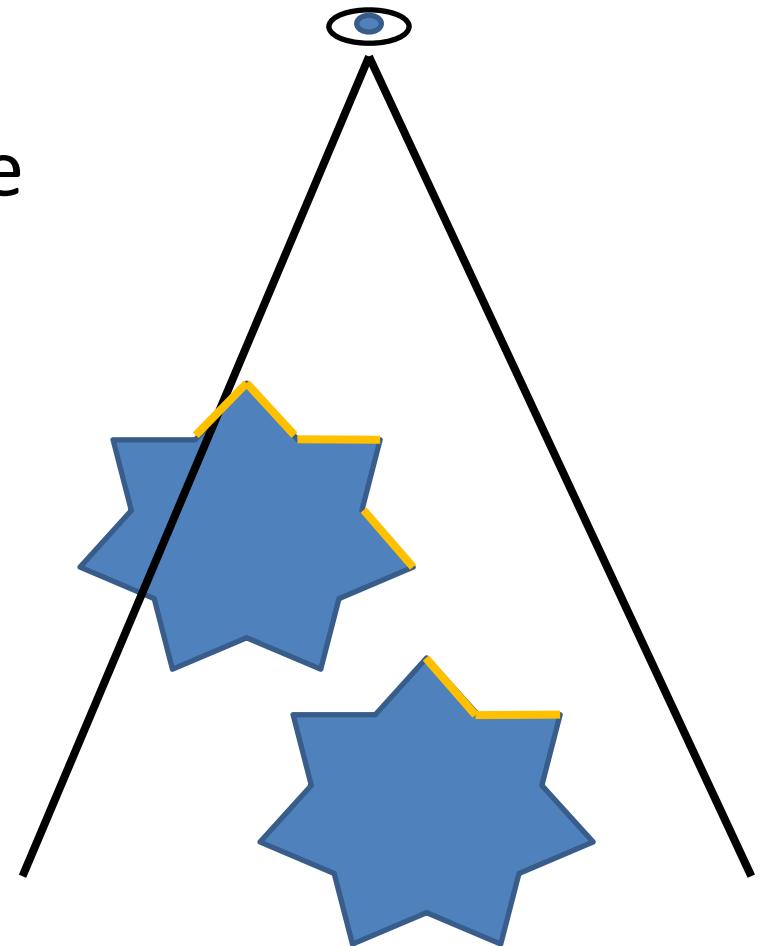
- All primitives that are visible
- No more, no less



(a)



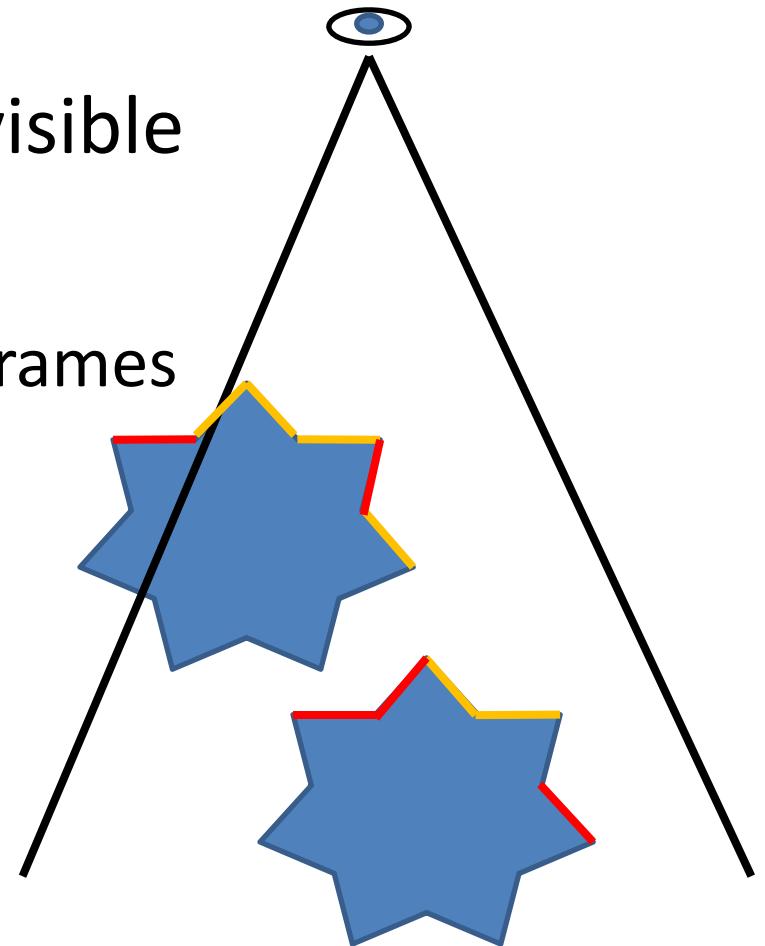
(b)



Visibility

Potentially Visible Set

- PVS = all primitives that are visible
- And a bit more
 - E.g., visible within the next n frames
 - Exact hidden surface removal done later by z-buffer
- How much more?
- What do we want to do with the PVS?



PVS Classification

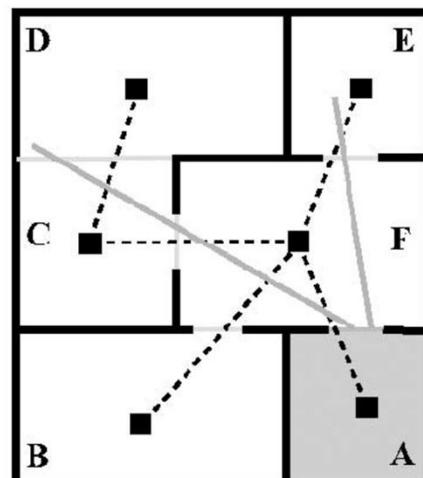
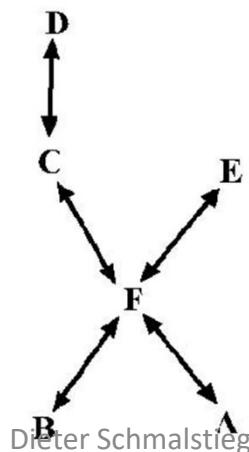
- The exact visible set (EVS) is unknown: $O(N^9)\dots$
- Potentially visible set (PVS) =
set of objects that *could* be visible
- PVS can be
 - Aggressive, $PVS \subseteq EVS$
 - Conservative, $PVS \supseteq EVS$ (preferred)
 - Approximate, $PVS \sim EVS$
- PVS can be precomputed
 - But we need discretize viewpoints into view *regions*

Cells and Portals

- Indoors
 - Most rooms (*cells*) occluded by walls
 - Store *portals* (windows, doors) instead of occluders (walls)
- Cells and portals form nodes and edges of a *portal graph*



Adjacency graph



Screen-Space Portals

- In runtime, find the portals of the current cell that are in the frustum
- Traverse through all found portals to the adjacent cells and find all portals that are visible to the camera through the original portal

