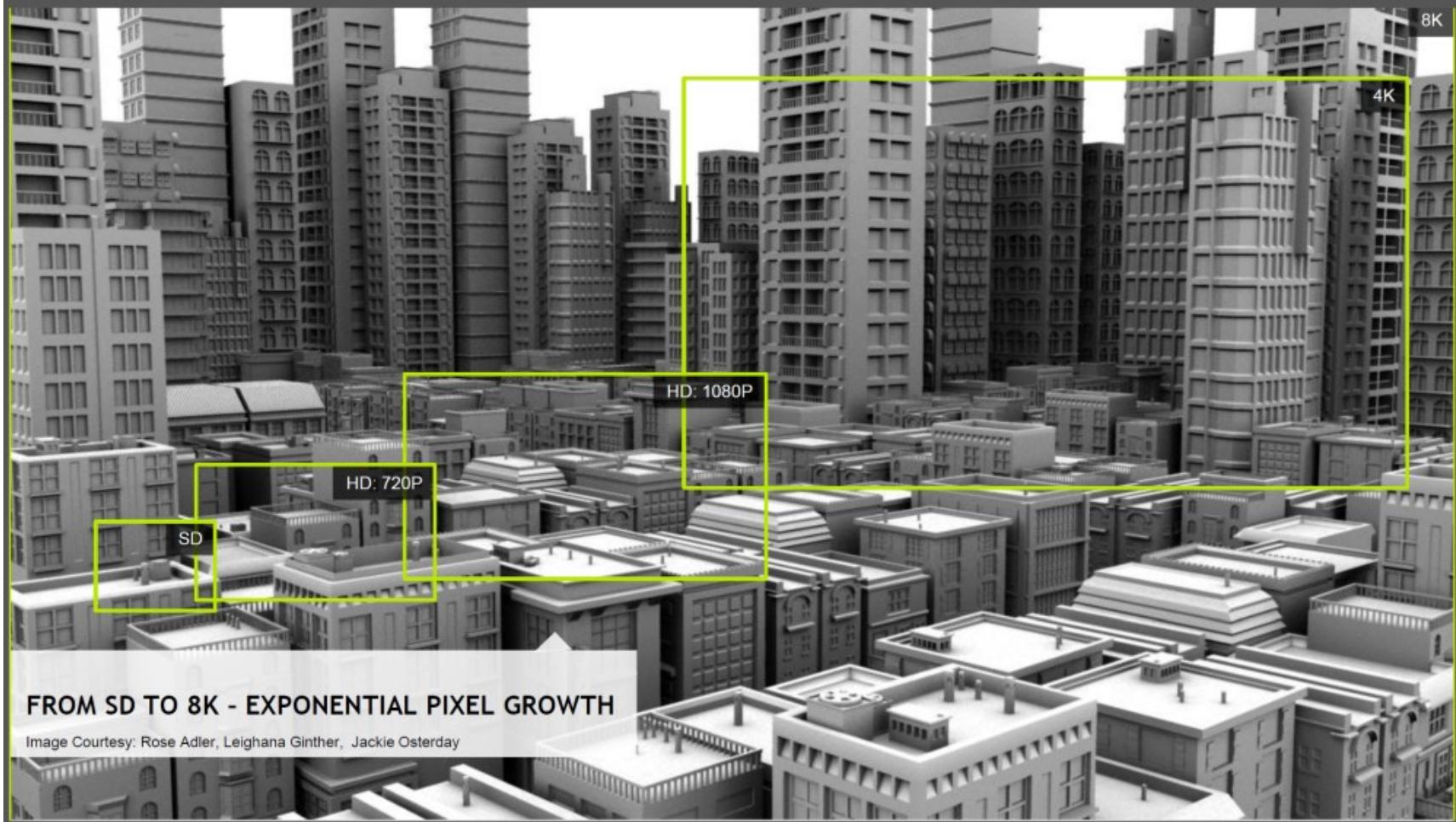


# Dieter Schmalstieg

# The Graphics Pipeline

# What do we want?

- Computer-generated imagery (CGI) of complex 3D scenes in real-time
- Computationally extremely demanding
  - Full HD at 60 Hz:  
 $1920 \times 1080 \times 60 \text{ Hz} = 124 \text{ Mpx/s}$
  - And that's just output data!
- Requires specialized hardware

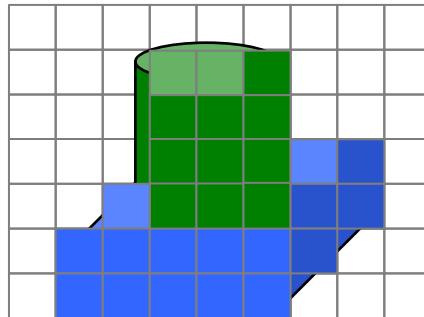


# Image Synthesis

**Object Order**

Go through all objects

„What do I see where“

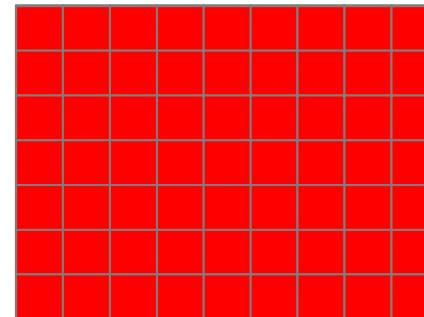


→Rasterization

**Image Order**

Go through all pixels

„Where do I see what“



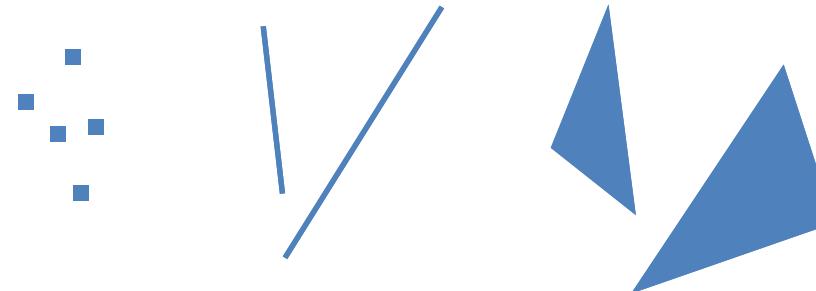
→Raytracing

# Rasterization Hardware

Most of real-time graphics is based on

- Rasterization of graphic *primitives*

- Points
- Lines
- Triangles
- ...

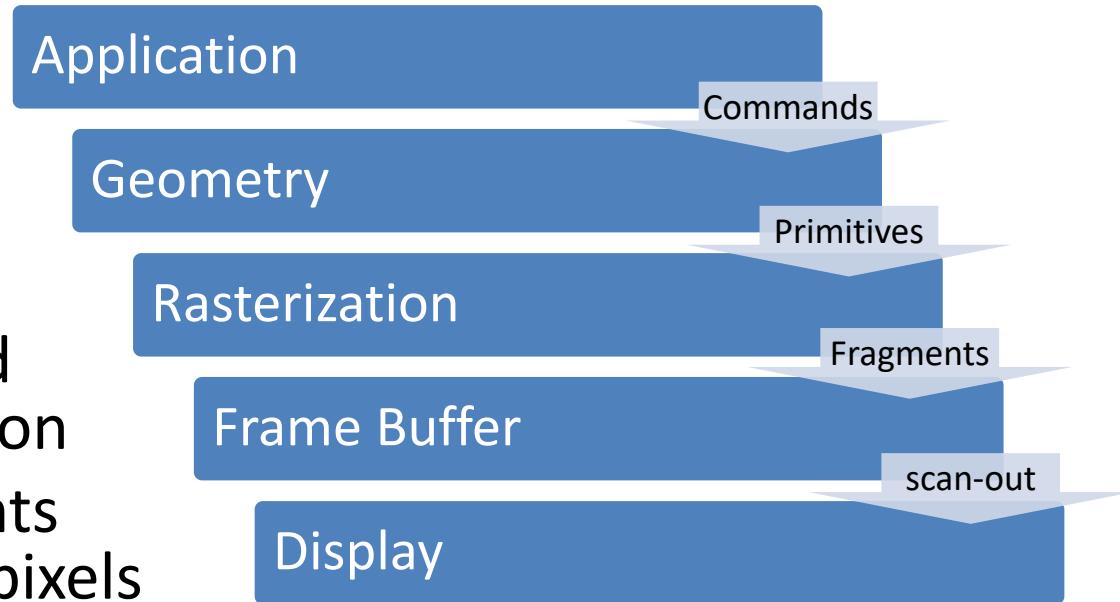


- Implemented in hardware
  - *Graphics processing unit (GPU)*

# The Graphics Pipeline

- High-level view:

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



# Application Stage

- Generate database
  - Usually only once
  - Load from disk or generate algorithmically
  - Build acceleration structures (hierarchy, ...)
- Repeat main loop
  - Input event handlers
  - Simulation → modify data structures
  - Database traversal → issue **graphics commands**
- Until exit

# Graphics Commands

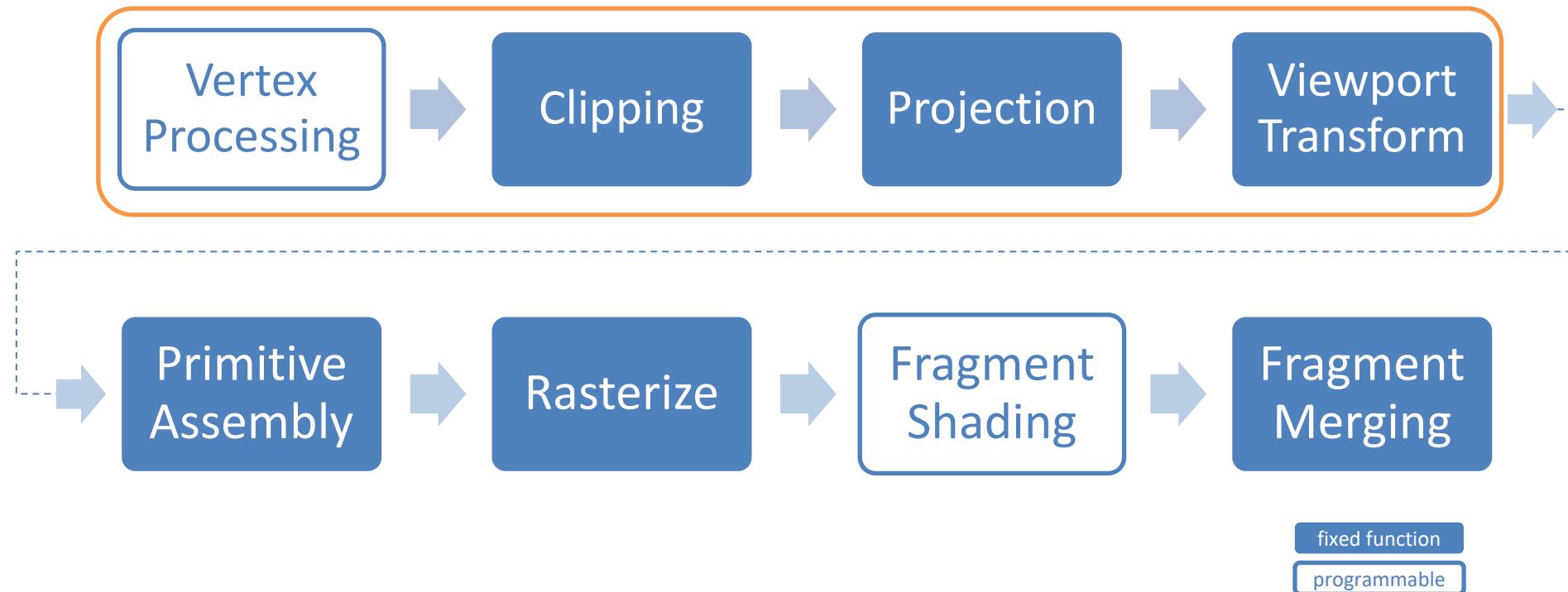
- Graphics command stream from CPU to GPU
  - Specify primitives
  - Manage resources
  - Modify GPU state
- Commands are abstracted as Graphics API
  - OpenGL, DirectX, Vulkan, Metal

# Graphics Driver

- Graphics hardware is shared resource
- Large user mode graphics driver
  - Prepares command buffers
- Graphics kernel subsystem
  - Schedule access to hardware
- Small kernel mode graphics driver
  - Submit command buffers to hardware

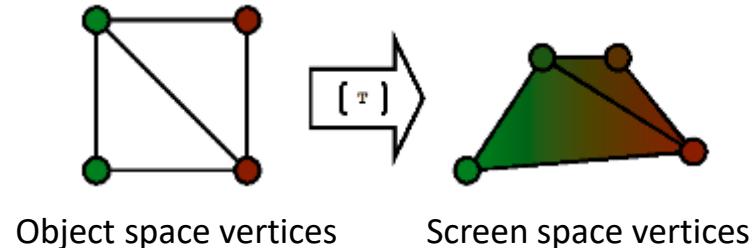
This is where new APIs  
(Apple Metal, DirectX 12, Vulkan)  
try to be more efficient

# Geometry Stage



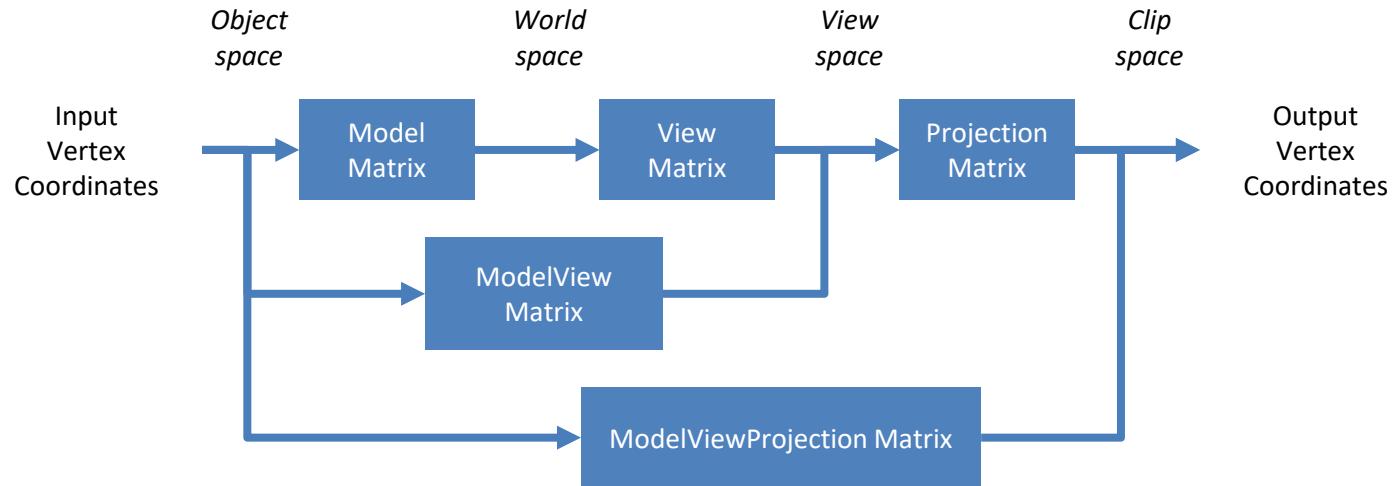
# Vertex Processing

- Input vertex stream
  - Composed of arbitrary *vertex attributes* (position, color, ...)
- Is transformed into stream of vertices mapped onto the screen
  - Composed of their *clip space* coordinates and additional user-defined attributes (color, texture coordinates, ...)
  - Clip space: homogeneous coordinates
- By the *vertex shader*
  - GPU program that implements this mapping
  - Historically, “shaders” were small programs performing lighting calculations



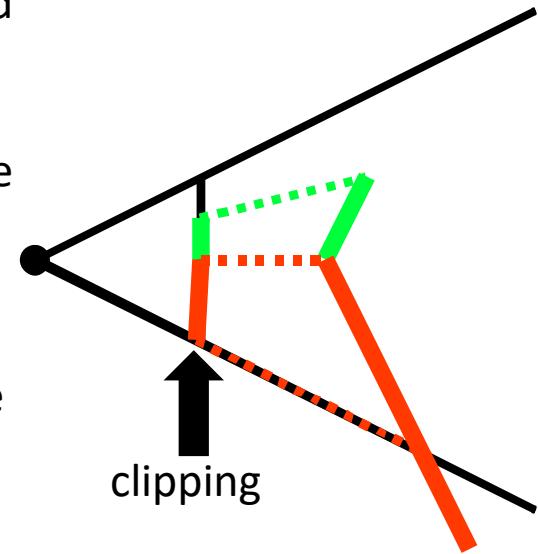
# Vertex Coordinate Transformation

Common model in rasterization-based 3D graphics

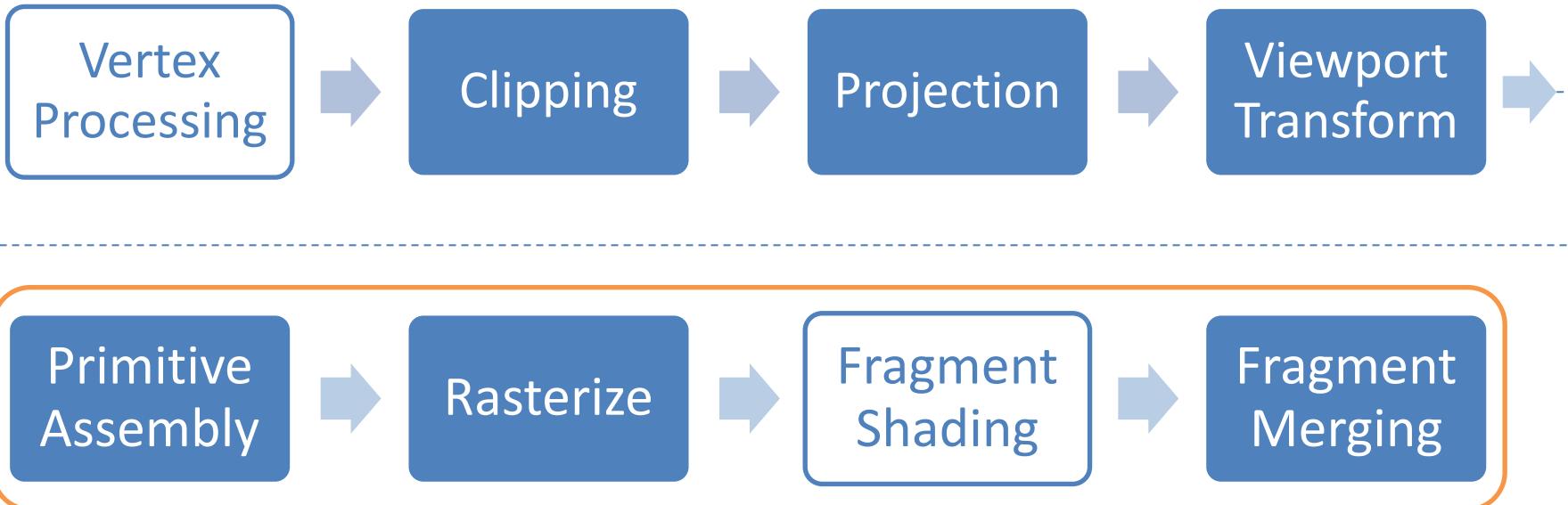


# Geometry Stage Tasks

- Clipping
  - Primitives not entirely in view are clipped to avoid projection errors
- Projection
  - Projects clip space coordinates to the image plane
  - Primitives in *normalized device coordinates*
- Viewport Transform
  - Maps resolution-independent normalized device coordinates to a rectangular window in the frame buffer, the *viewport*
  - Primitives in window (pixel) coordinates



# Rasterization Stage

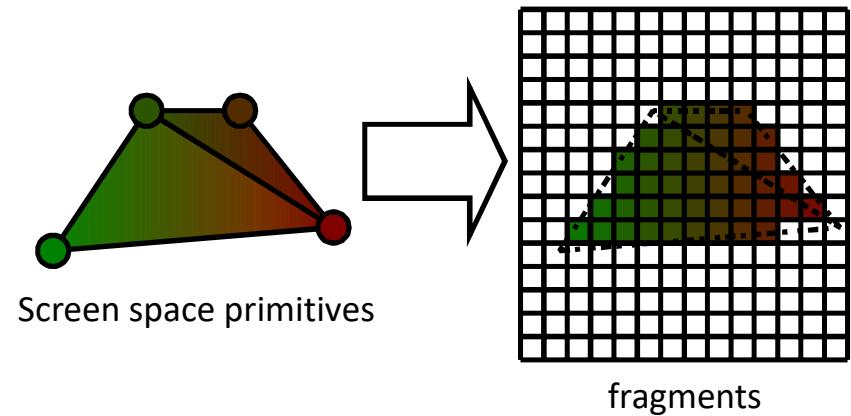


fixed function

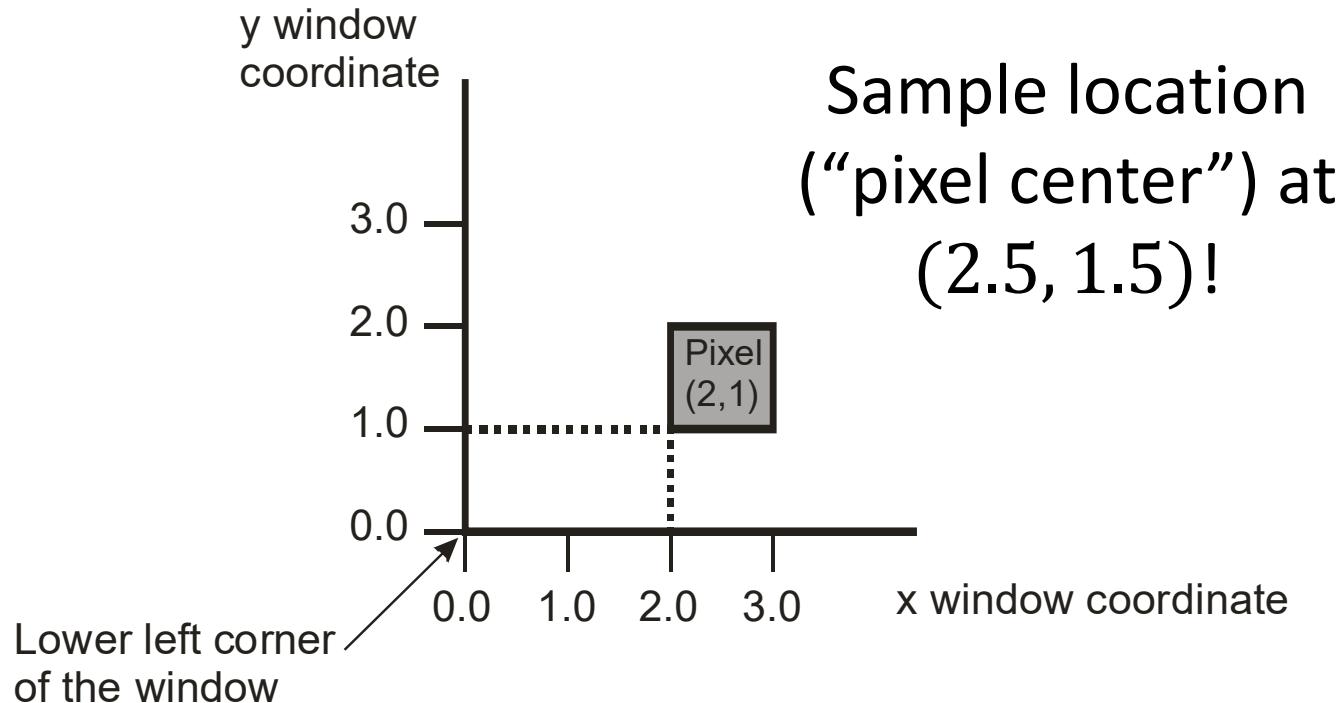
programmable

# Rasterization Stage Tasks

- Primitive assembly
  - Backface culling
  - Setup primitive for traversal
- Rasterization (“primitive traversal”, “scan conversion”)
  - Sampling (triangle → fragments)
  - Interpolation of vertex attributes (depth, color, ...)
- Fragment shading
  - Compute fragment colors
- Fragment merging
  - Compute pixel colors from fragments
  - Depth test, blending, ...

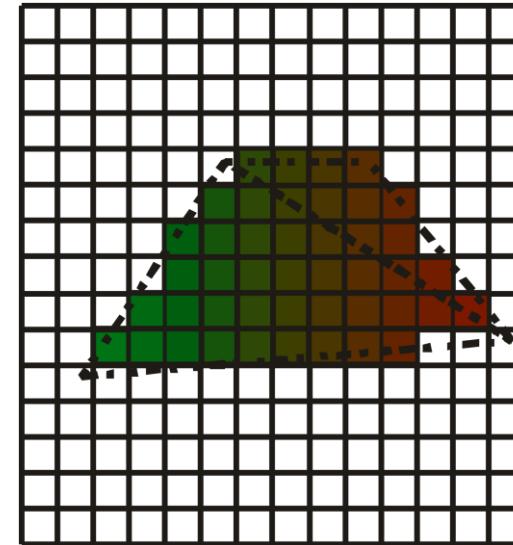


# Rasterization – Coordinates



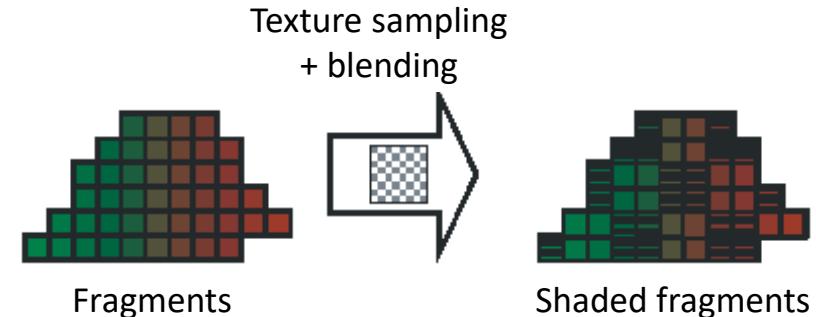
# Rasterization – Rules

- Different rules apply for each primitive type
  - “Fill convention”
- Non-ambiguous!
  - Avoids artifacts
- Polygons
  - Pixel center contained in polygon
  - Pixels on edge: only one rasterized



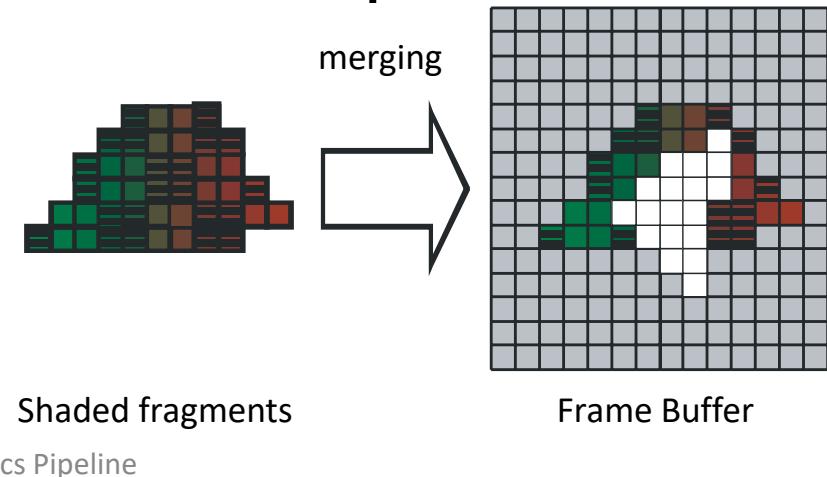
# Fragment Shading

- Aka *pixel shader*
- Given the interpolated vertex attributes
  - Output by the vertex shader
- The *fragment shader* computes color values for each fragment
  - Apply textures
  - Lighting calculations
  - ...

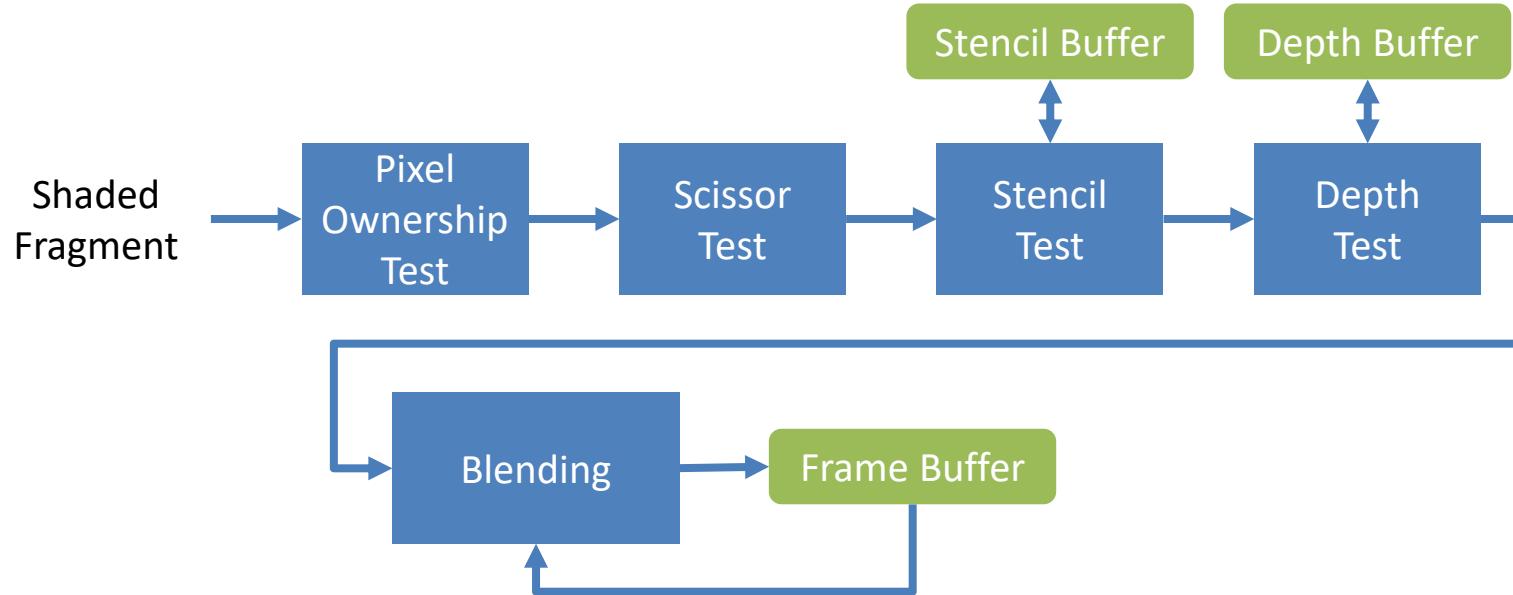


# Fragment Merging

- Also known as “raster operations” (ROP)
- Multiple primitives can cover the same pixel
- Their fragments need to be composed to form the final pixel values
  - Blending
  - Resolve visibility via depth buffering

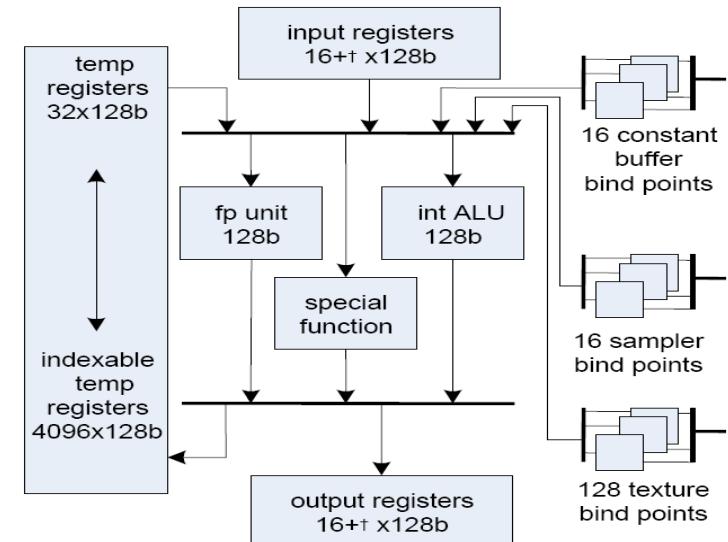


# Fragment Merging Workflow



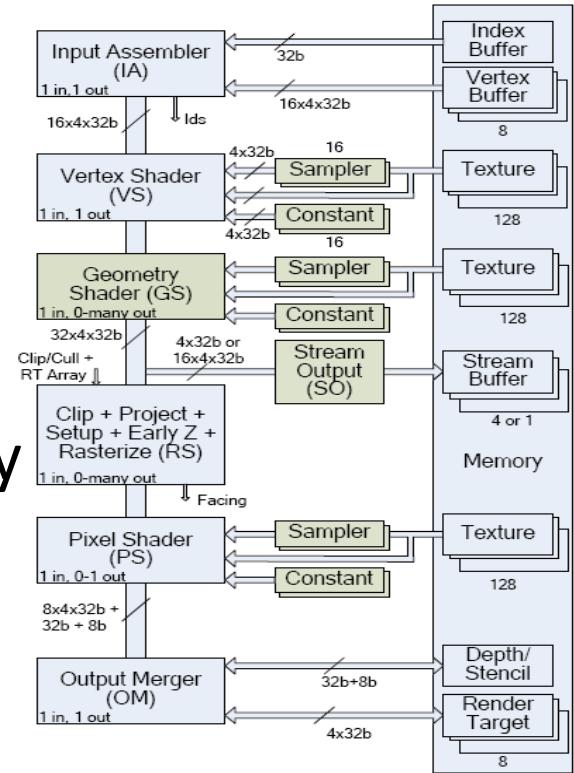
# Unified Shader Model

- Same instruction set and capabilities for all shaders
- Dynamic load balancing between vertex and fragment shaders
- IEEE-754 floating point
- Enables new GPGPU languages like CUDA



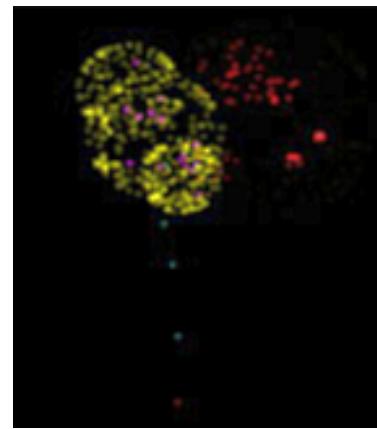
# Geometry Shader

- 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



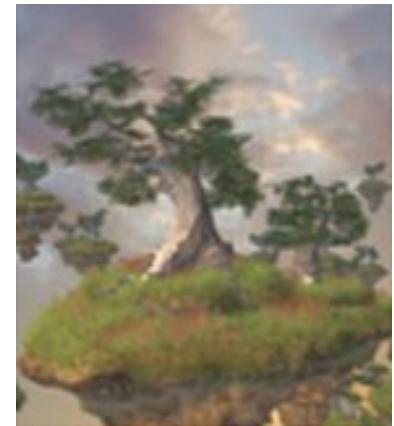
# Geometry Shader Examples 1

Cube Map: GS instances  
every triangle 6x



Particles produced by  
GS as stream-out

Plants created as  
parameterized instances

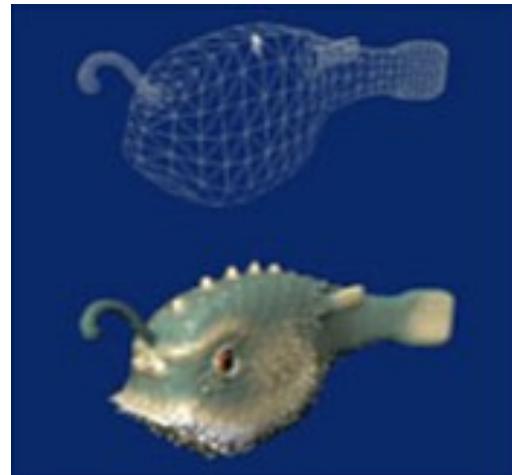


# Geometry Shader Examples 2



Shadow volume created  
by GS extrusion

Displacements created by GS



# Tesselation Shader

- Since DirectX11
- New shader stage: Tesselation
- Input: low-detail mesh
- Output: high-detail mesh

Vertex Shader

Hull Shader

Tessellator

Domain Shader

Geometry Shader

Rasterizer

Pixel Shader



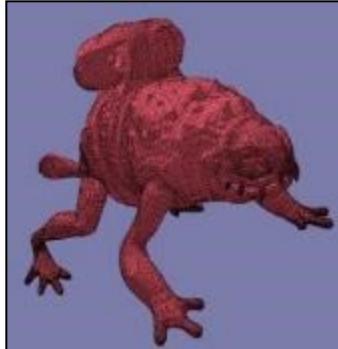
[https://www.youtube.com/watch?v=p\\_VpAMaxwpY](https://www.youtube.com/watch?v=p_VpAMaxwpY)

# Authoring without Tessellation

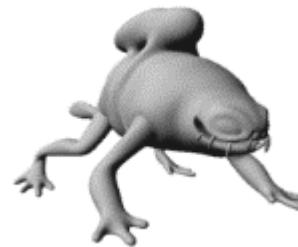
Sub-D modeling



Polygon mesh



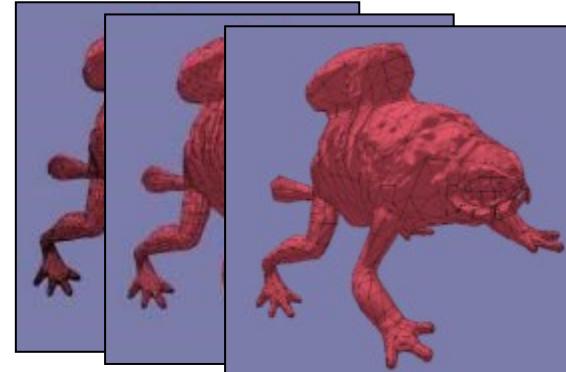
Animation



Displacement map



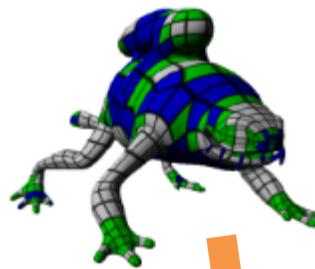
Generate LODs



Graphics Pipeline

# Authoring with Tessellation

Sub-D modeling



Animation

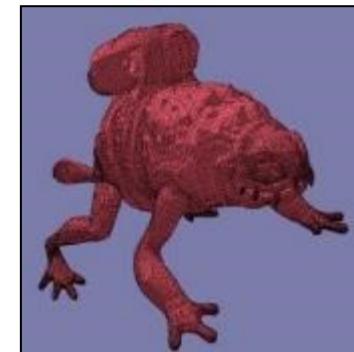


Displacement map



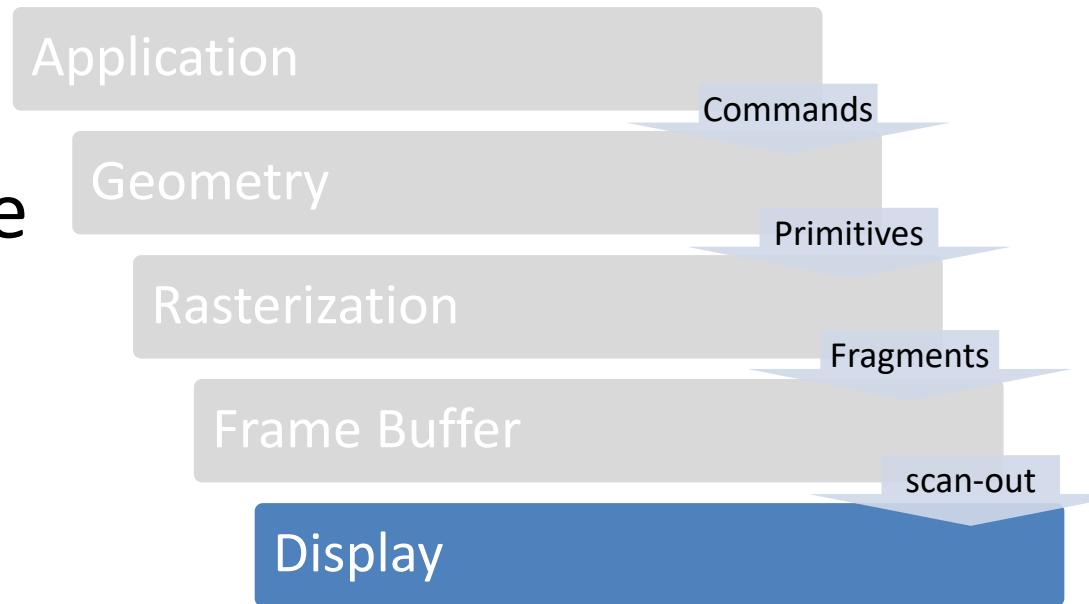
GPU

Optimally tessellated mesh



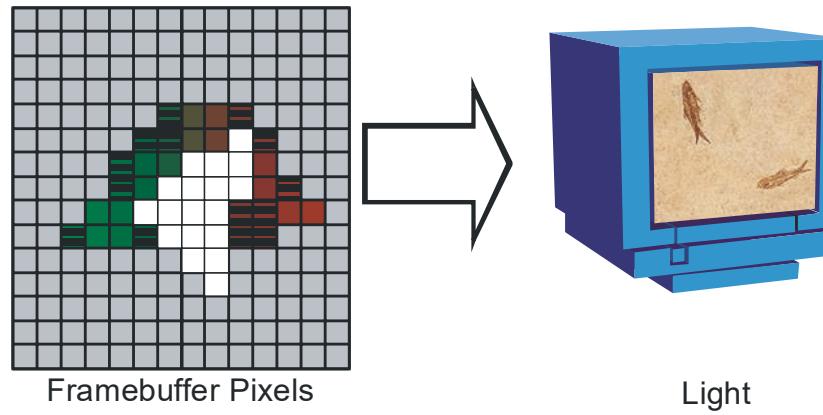
# Display Stage

- What happens after fragments are retired to the frame buffer?



# Display Stage Tasks

- Gamma correction
- Historically: digital to analog conversion
- Today: digital scan-out, HDMI encryption?

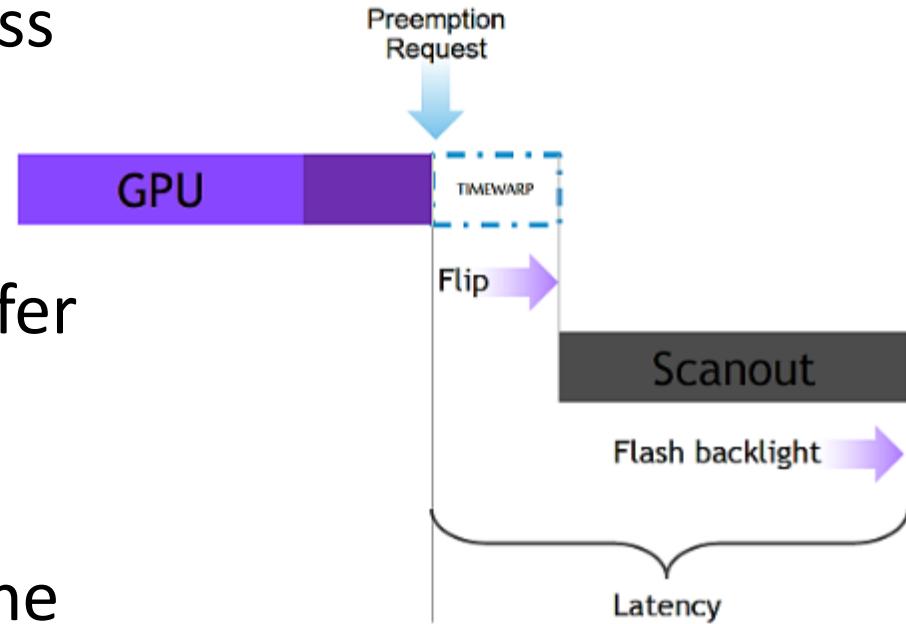


# Display Format

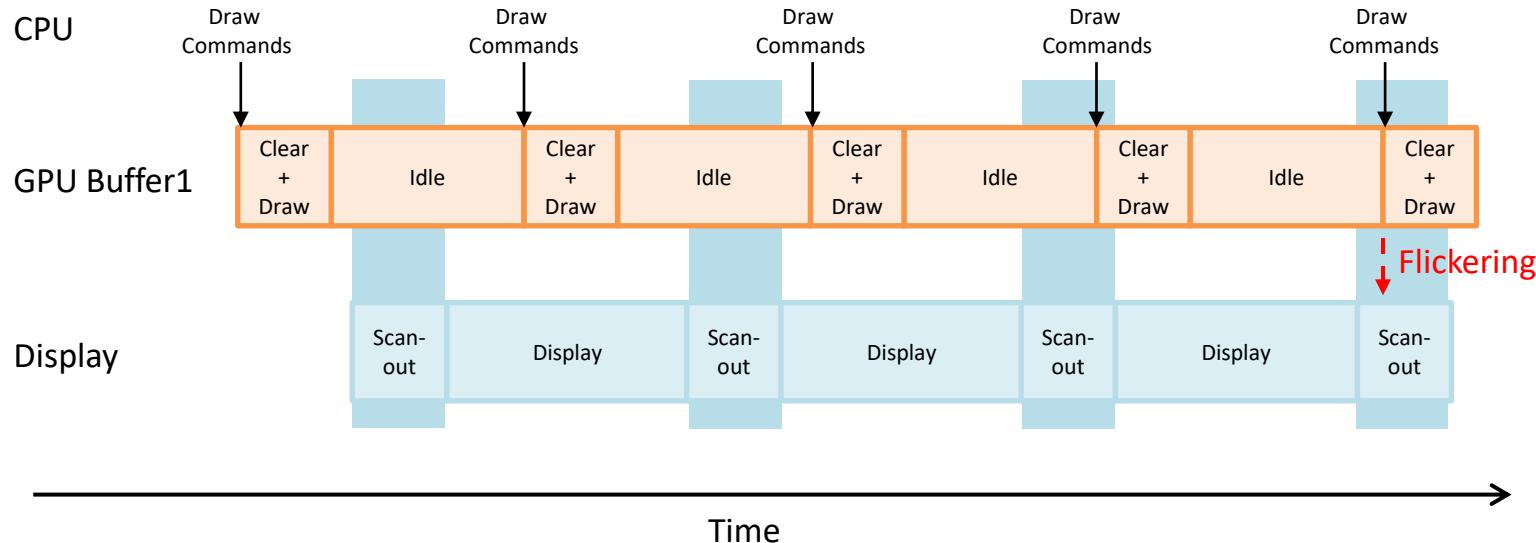
- Frame buffer pixel format:  
RGBA vs. index (obsolete)
- Bits: 16, 32, 64, 128 bit floating point, ...
- Double buffering for smooth animation
- Quad-buffering for stereo graphics
- Overlays (extra bitplanes)
- Auxilliary buffers: alpha, stencil, depth

# What is Display Synchronization?

- Need to synchronize access to frame buffer between GPU and display
- Cannot change frame buffer during display scan-out
- Need proper buffering in the whole graphics pipeline

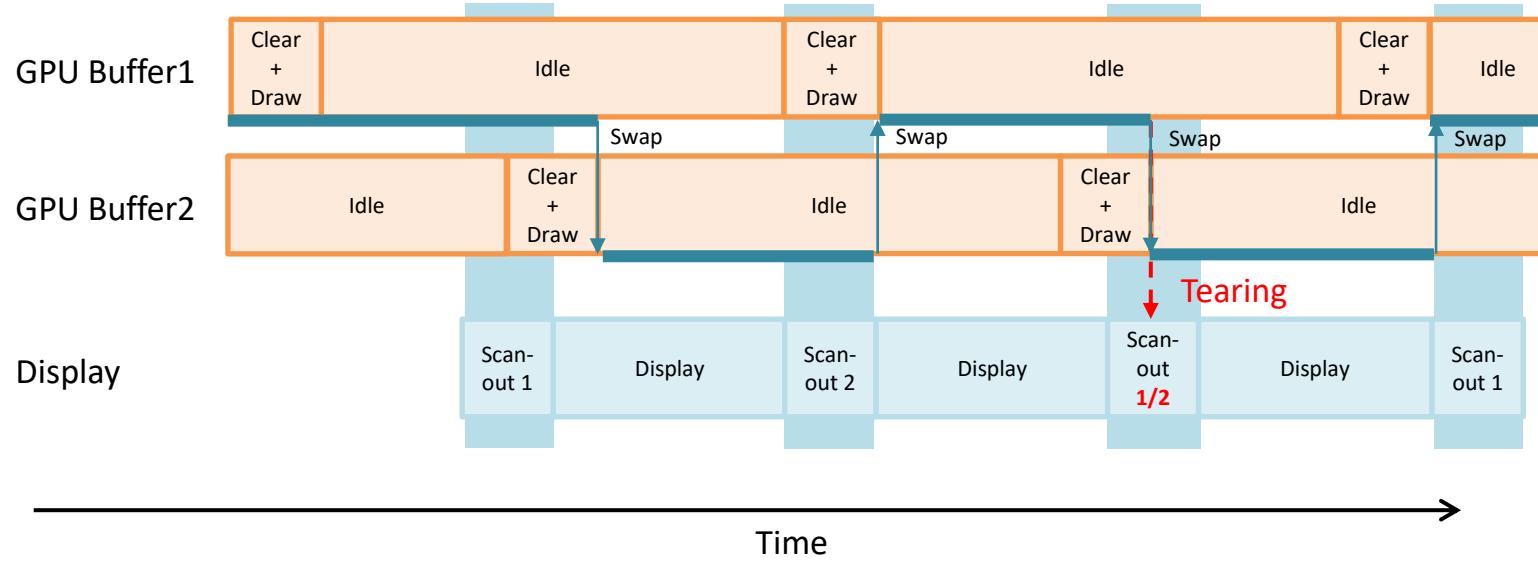


# Single Buffering



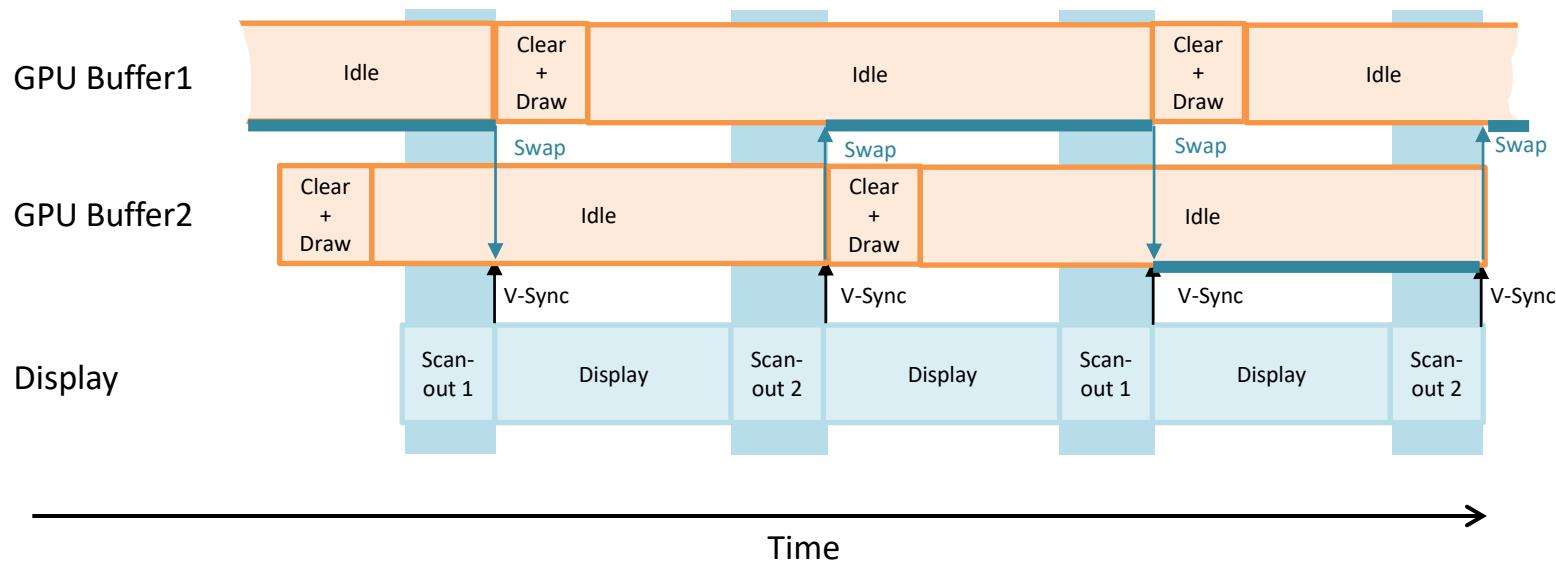
# Double Buffering without V-Sync

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



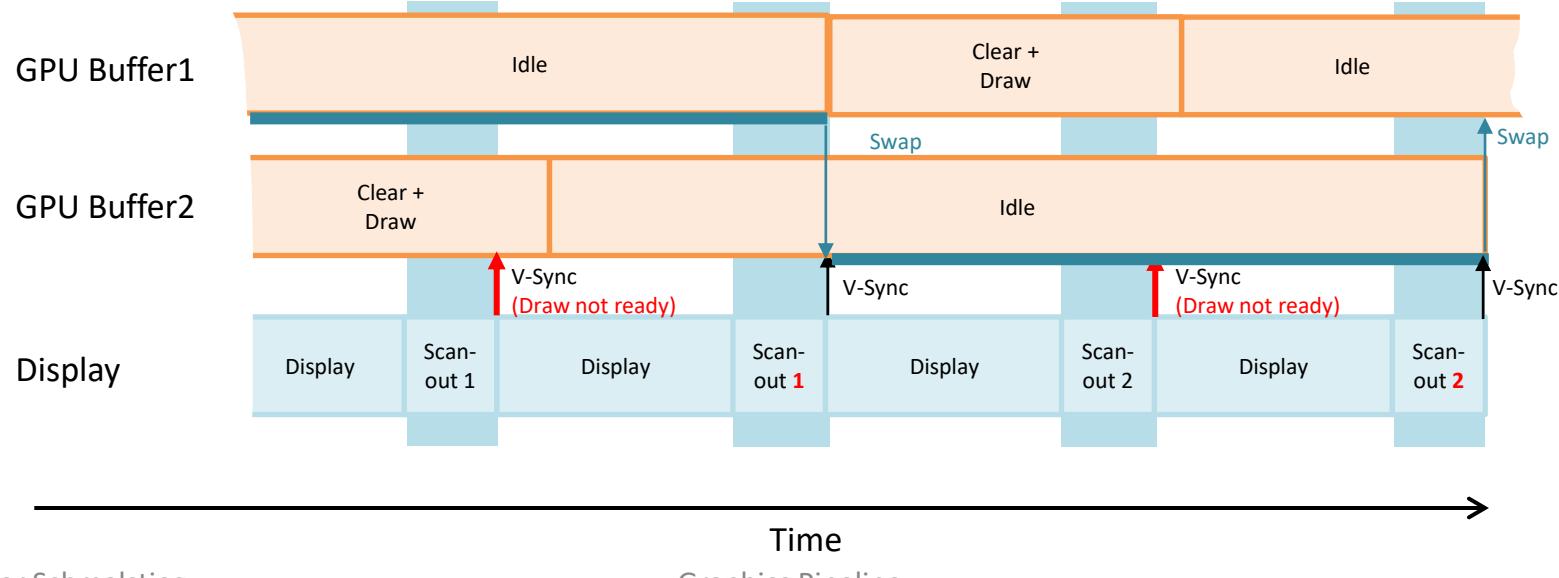
# Double Buffering with V-Sync, Fast

- 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



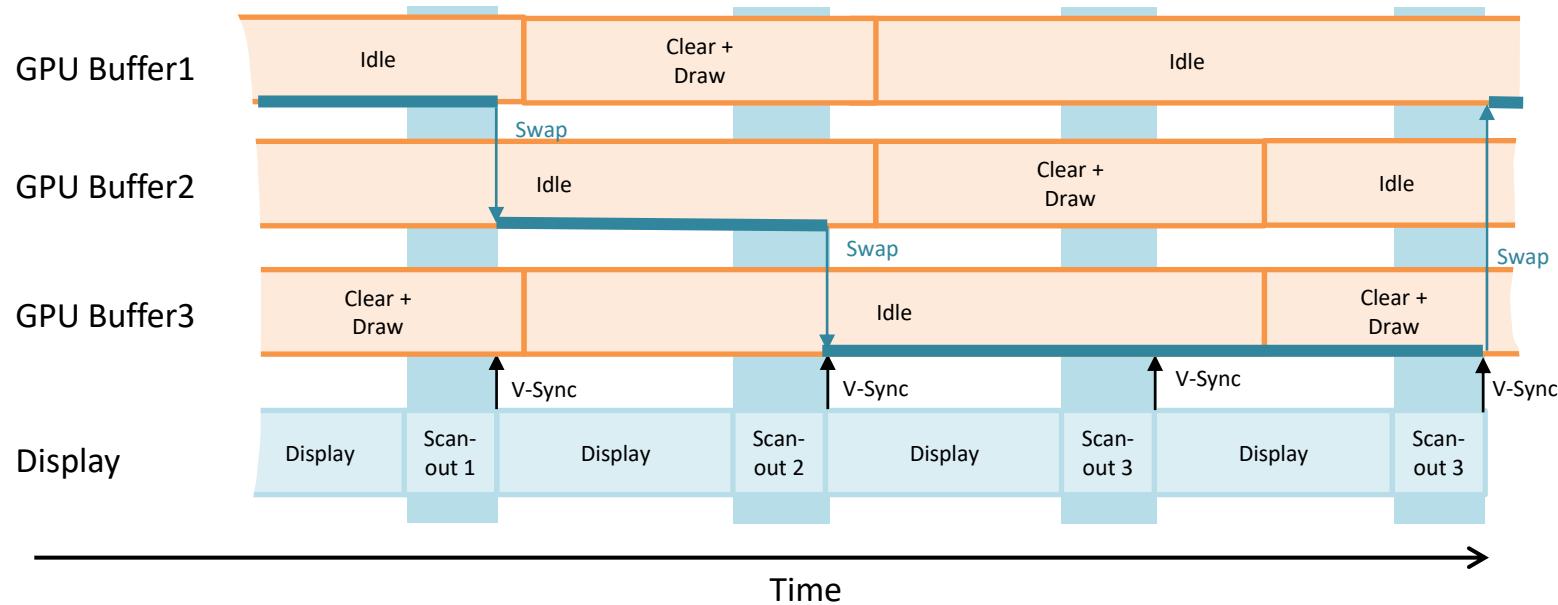
# Double Buffering with V-Sync, Slow

- 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



# Triple Buffering with V-Sync

- Triple buffering only makes sense together with V-Sync
- When rendering is **slow**, third buffer allows **continuous drawing**
- When rendering is **fast**, two back-buffers can be alternated (**wasted frames!**)

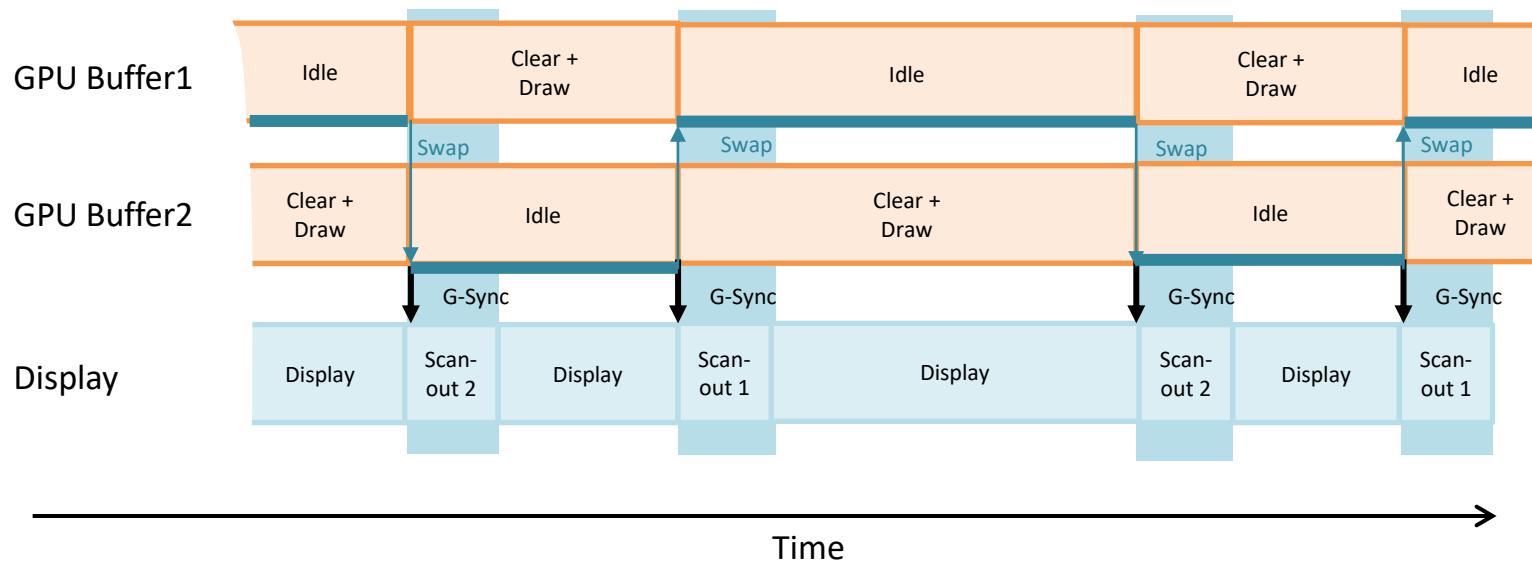


# G-Sync / FreeSync

- Display scan-out can be triggered by GPU
- Requires additional display hardware feature
- G-Sync (NVIDIA), FreeSync (AMD)

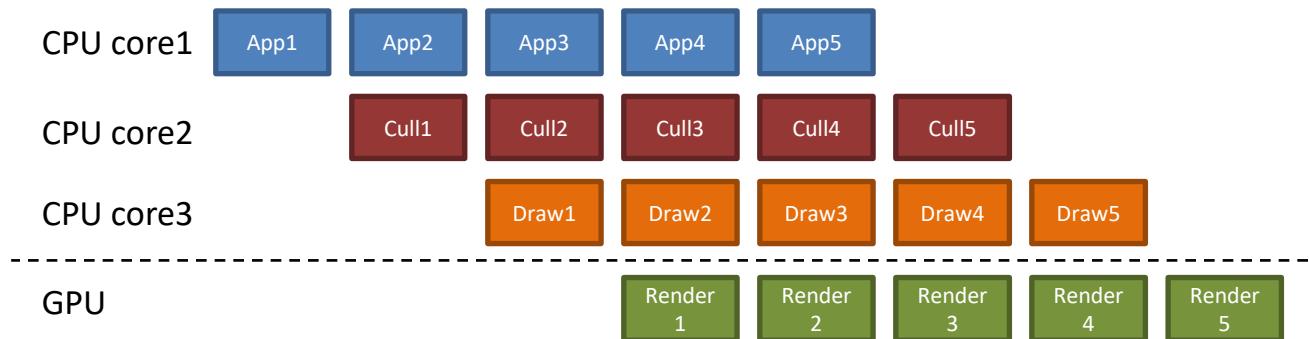


<https://youtu.be/5maHG9Kpjic>



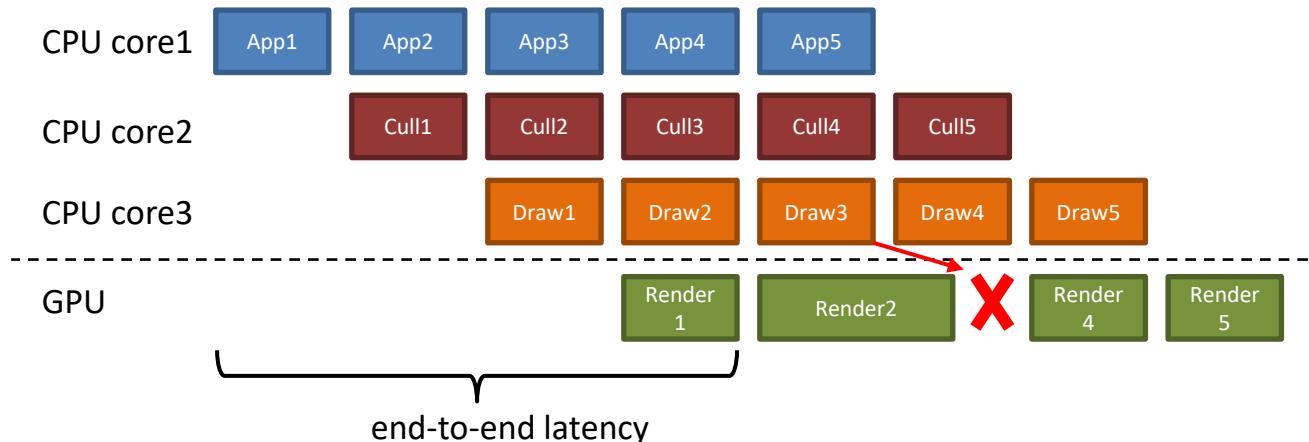
# Multi-Threaded Rendering Pipeline

- App stage: simulate 3D world
- Cull stage: determine object in view frustum
- Draw stage: issue OpenGL commands to driver (includes optimizations such as mode sorting)
- Everything must work at target frame rate!



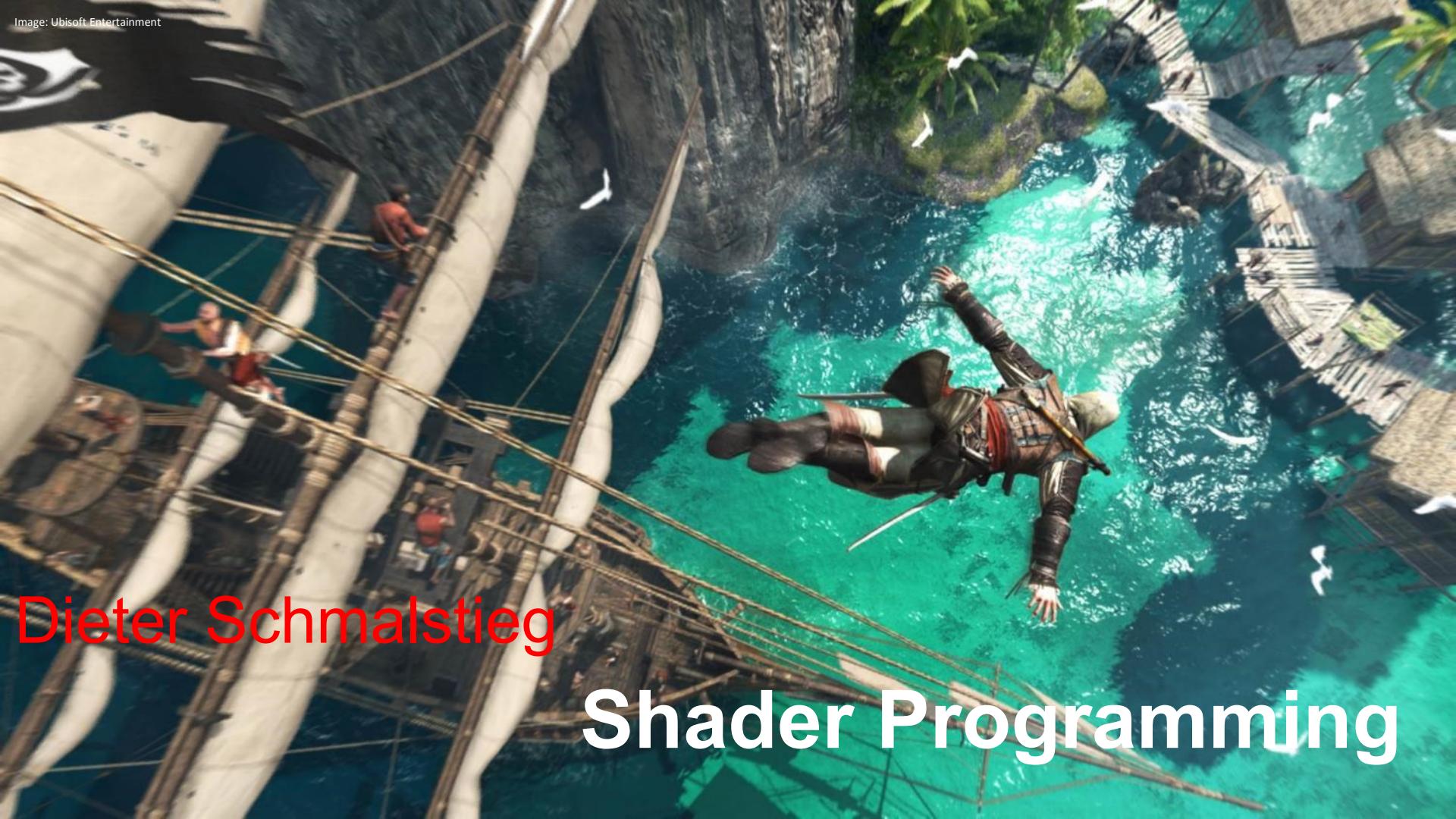
# Minimizing Pipeline Latency

- Always aim for minimal latency
- Fixed size buffer from stage to stage
- Never wait for (downstream) consumer!



# Thank You!

# Questions ?



Dieter Schmalstieg

# Shader Programming

# Today's Agenda

- The GPU execution model
- How to write shader programs

# Graphics Application Programmer Interface

- Hardware-vendor independent interface
  - Interface is hardware independent
  - But implementation is hardware dependent
- Defines
  - Abstract rendering device
  - Set of functions to operate the device

# API and Vendor Overview

## Graphics API

- DirectX (Microsoft)
- OpenGL
- OpenGL ES
- Vulkan (Khronos group)
- Metal (Apple)

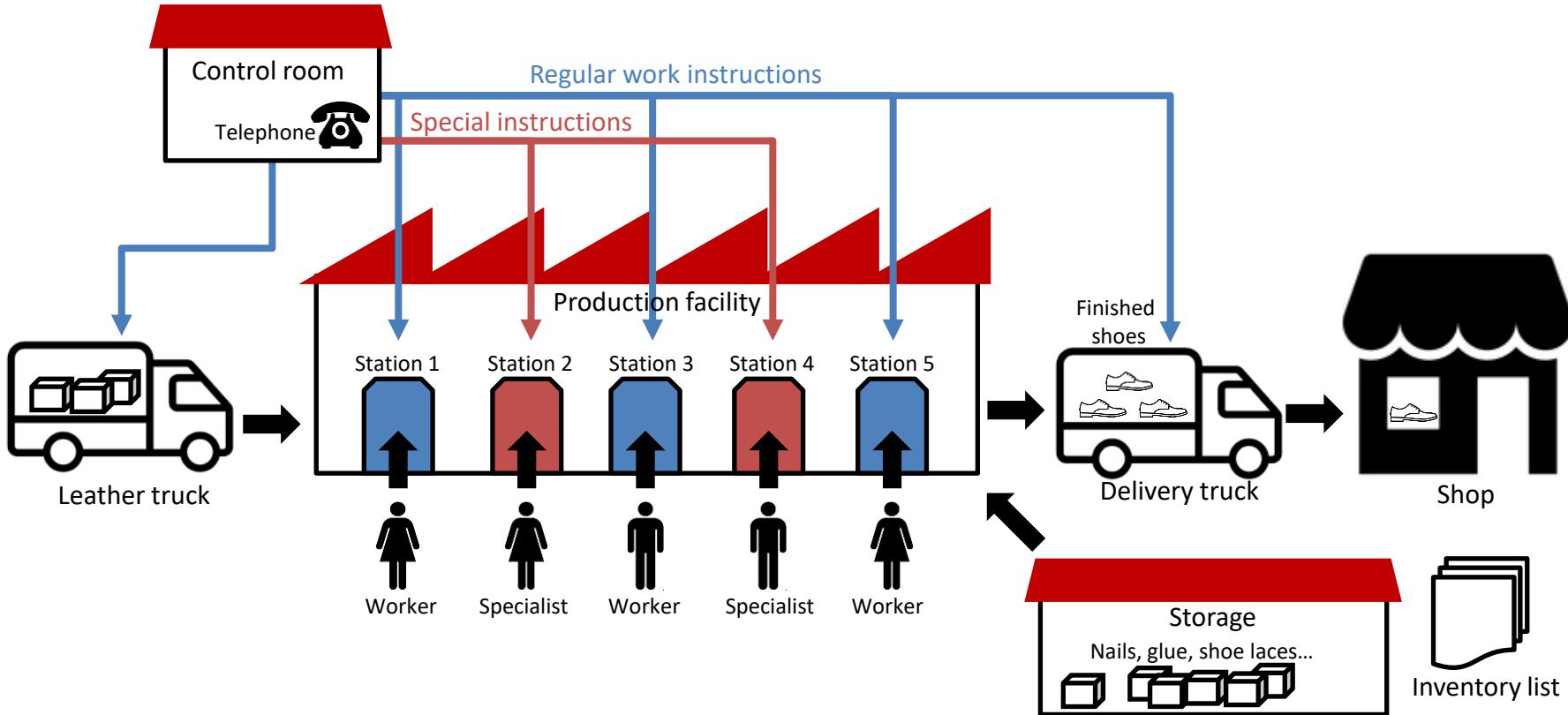
## GPU Hardware Vendors

- Intel (Iris, X<sup>e</sup>)
- NVIDIA (GeForce)
- AMD (Radeon)
- Qualcomm (Adreno)
- Imagination (PowerVR)
- Apple
- ARM (Mali, only design)

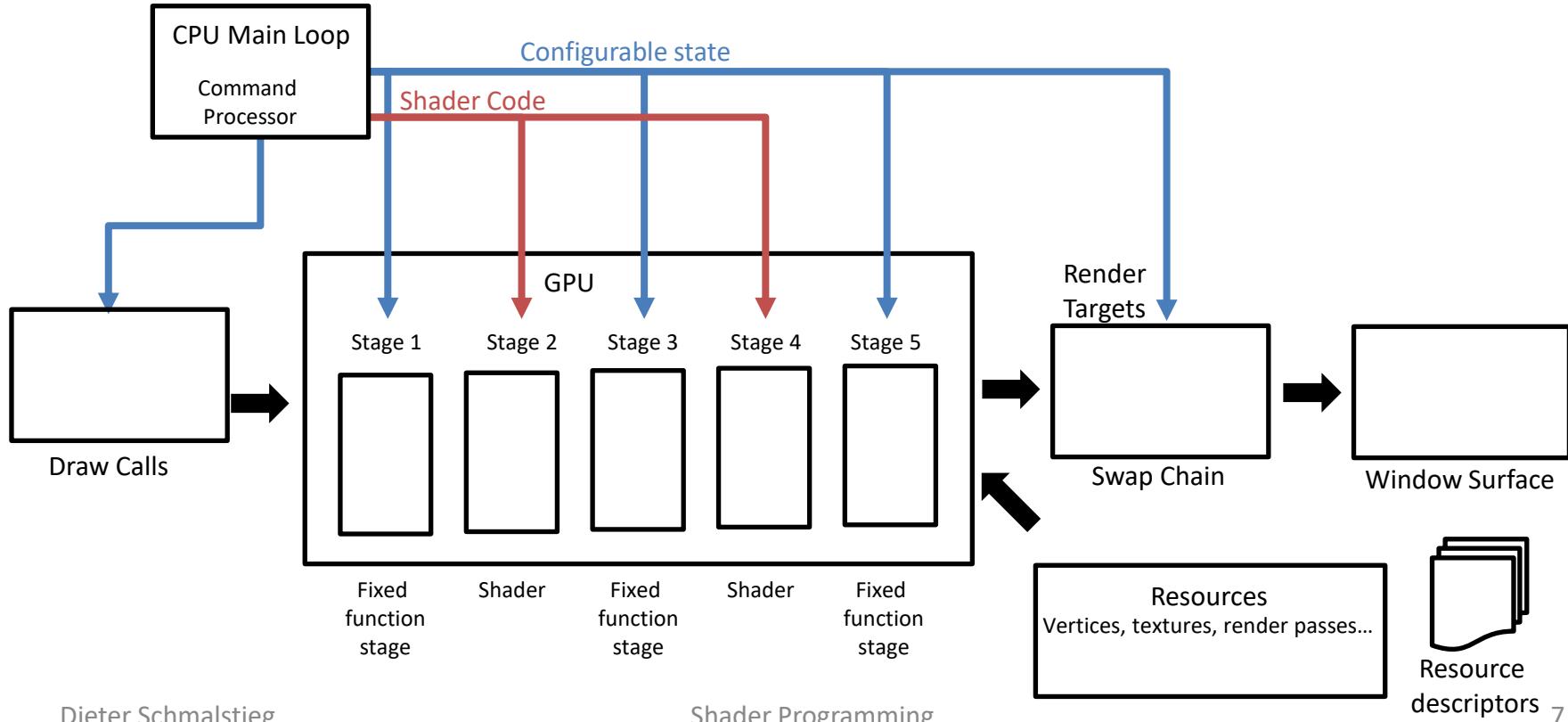
# A Story About GPU Programming

- Imagine you want to produce shoes
- CPU, single-threaded
  - The master shoemaker is alone in the shop
  - One task is done after the other
- CPU, multi-threaded
  - The master shoemaker has a few apprentices
  - They talk directly to one another to split the work
- GPU: 10000 worker threads...?

# A Big Shoe Factory



# A Big Graphics API

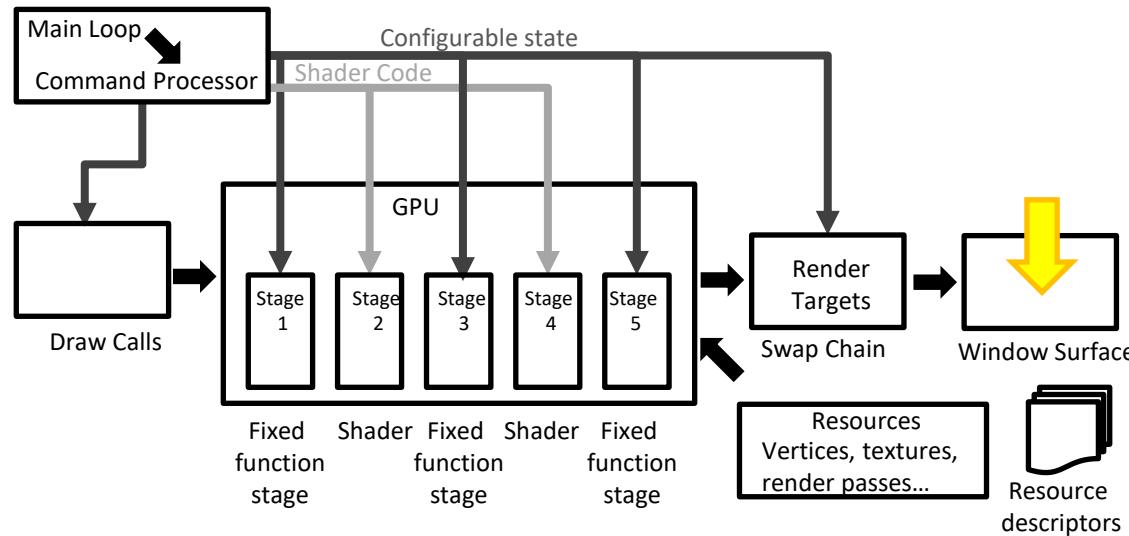


# Initialization

- Initialize the graphics system
- Get the “command processor” object
  - Context (OpenGL) or command queue (Vulkan) – see later slides
- Select a render device
  - Which physical GPU
    - If there are multiple
  - Which features are needed
    - Number of viewports, single or double float...
  - Which operation modes are needed
    - Graphics, compute, memory transfer

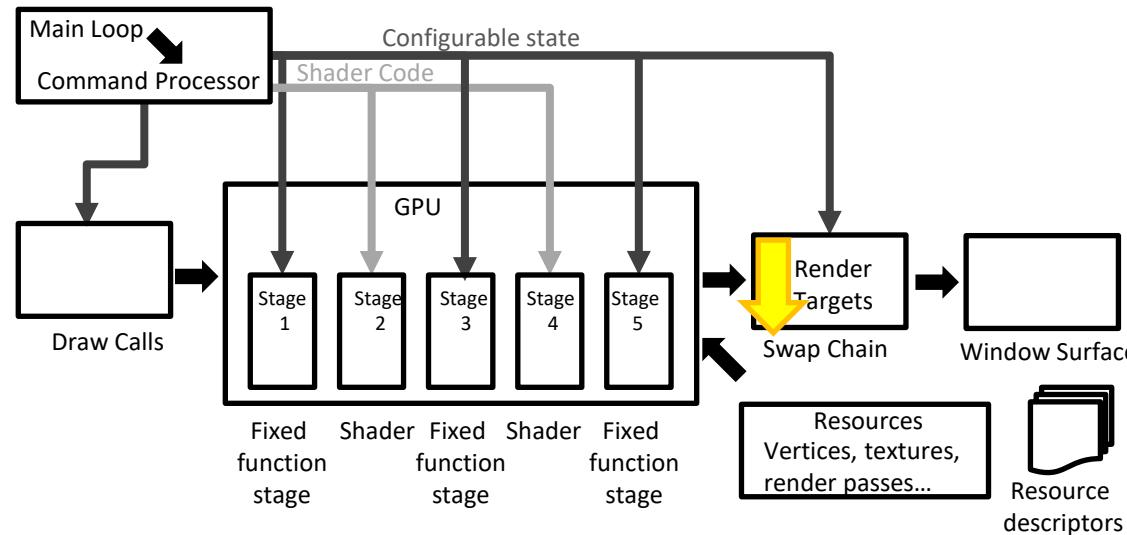
# Surface

- Object connected to window or entire screen
- Requires a platform-specific window manager library (e.g., GLFW)



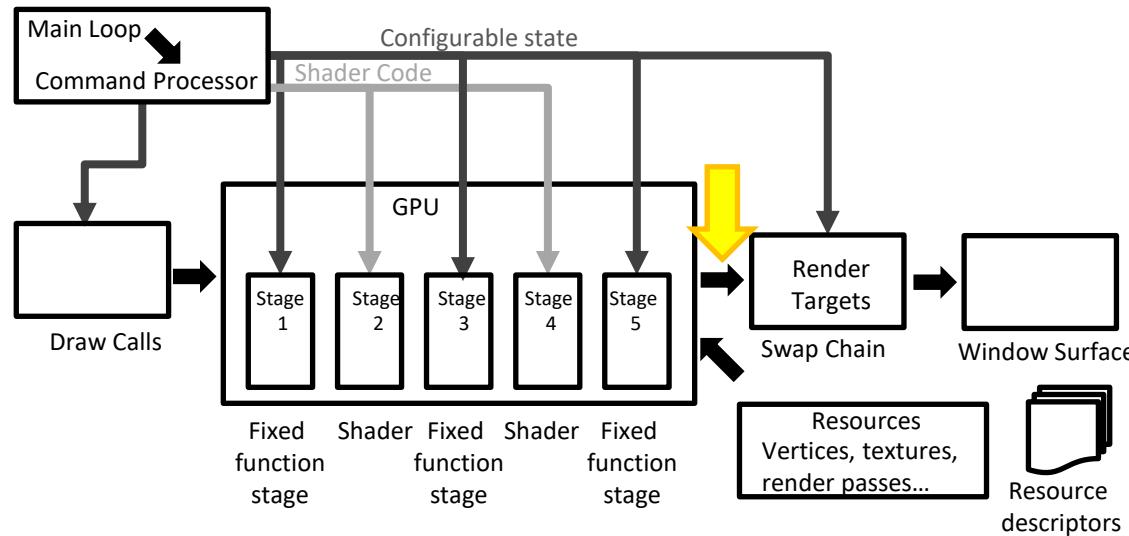
# Swap Chain

- Contains render targets (images) waiting to be shown
- Single, double, triple buffering



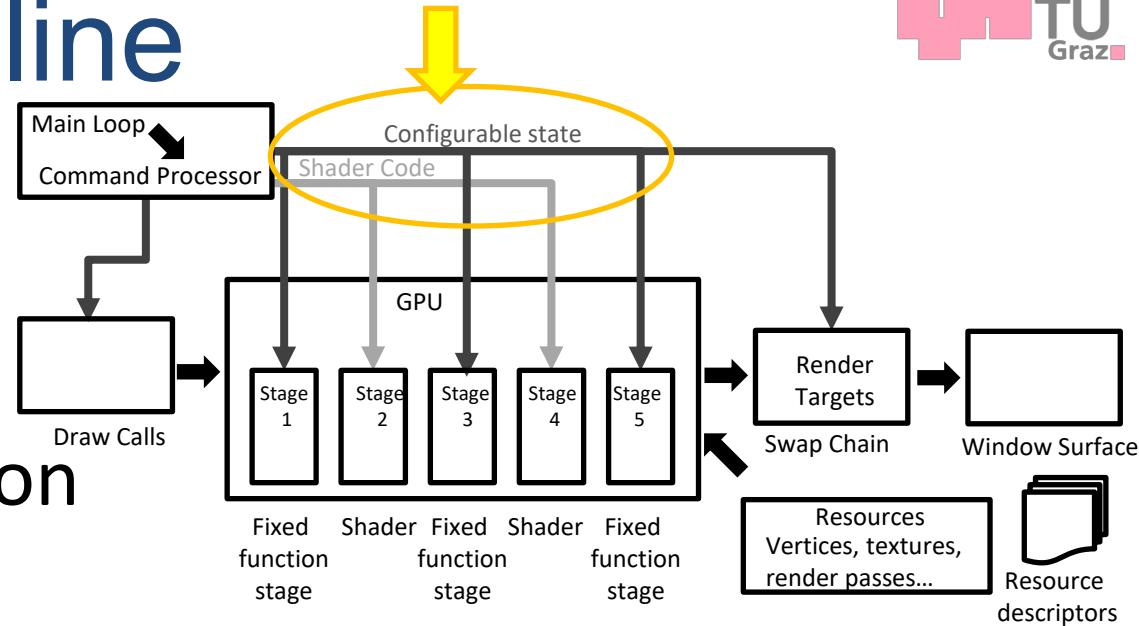
# Framebuffer

- Points to render targets for color, depth, stencil



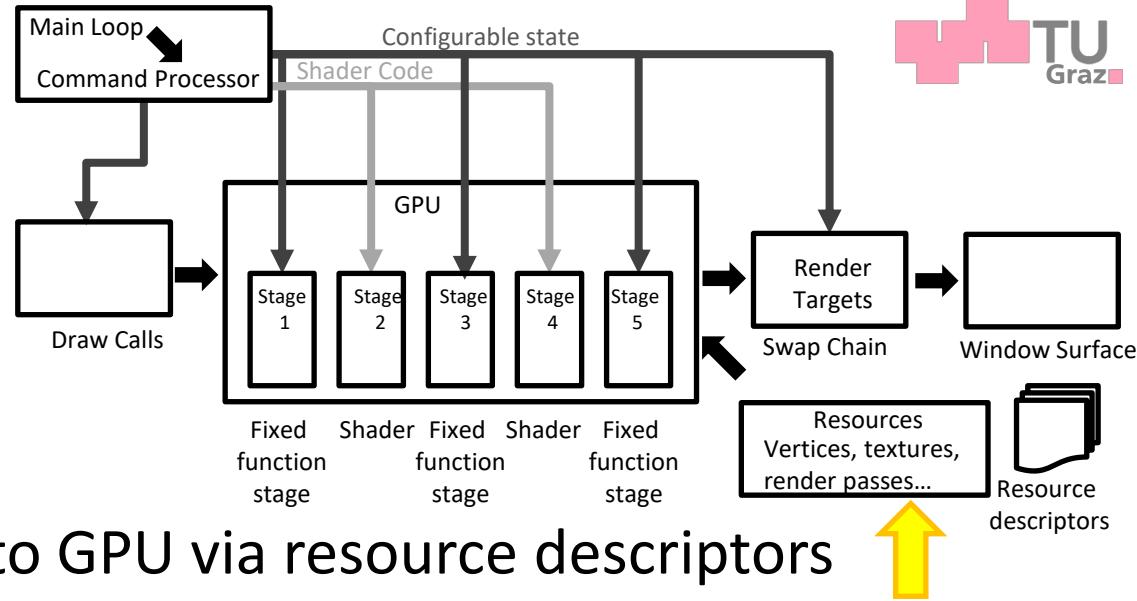
# Graphics Pipeline

- Pipeline stores specific configuration of render state
- Configurable state: viewport size, depth buffer etc.
- Programmable state: shader programs



# Resources

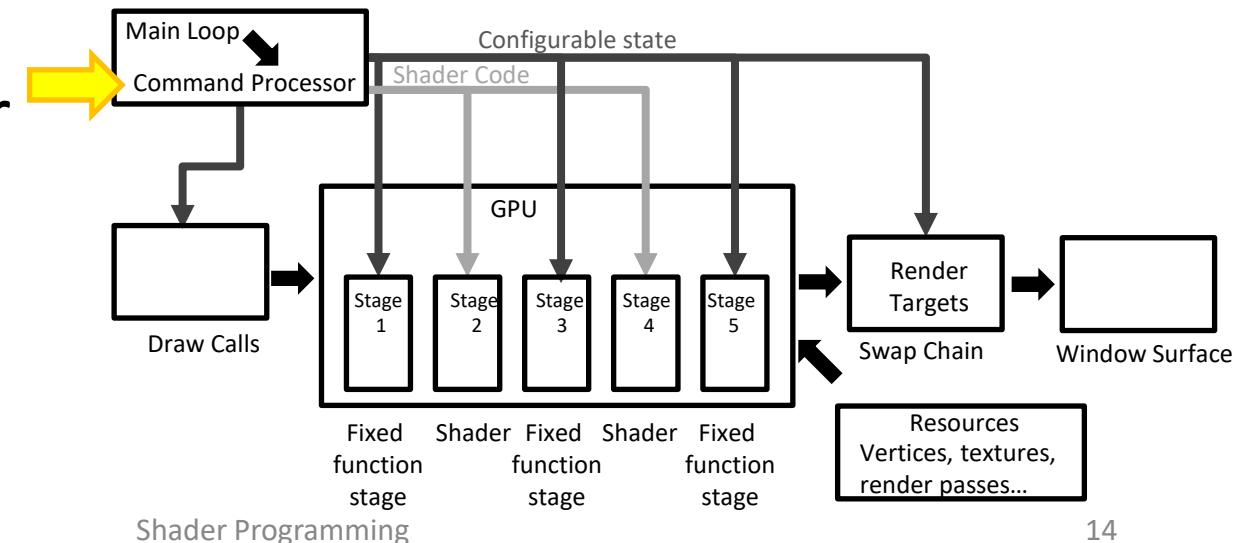
- Data stored in GPU local memory
- E.g, vertices, textures, other buffers
- Resource are described to GPU via resource descriptors
- Before use
  - Generate or download resource
  - Specify resource descriptors
  - Bind (=activate) the chosen resource
- Configurable state has “slots” for binding various resource types



# Command Processor

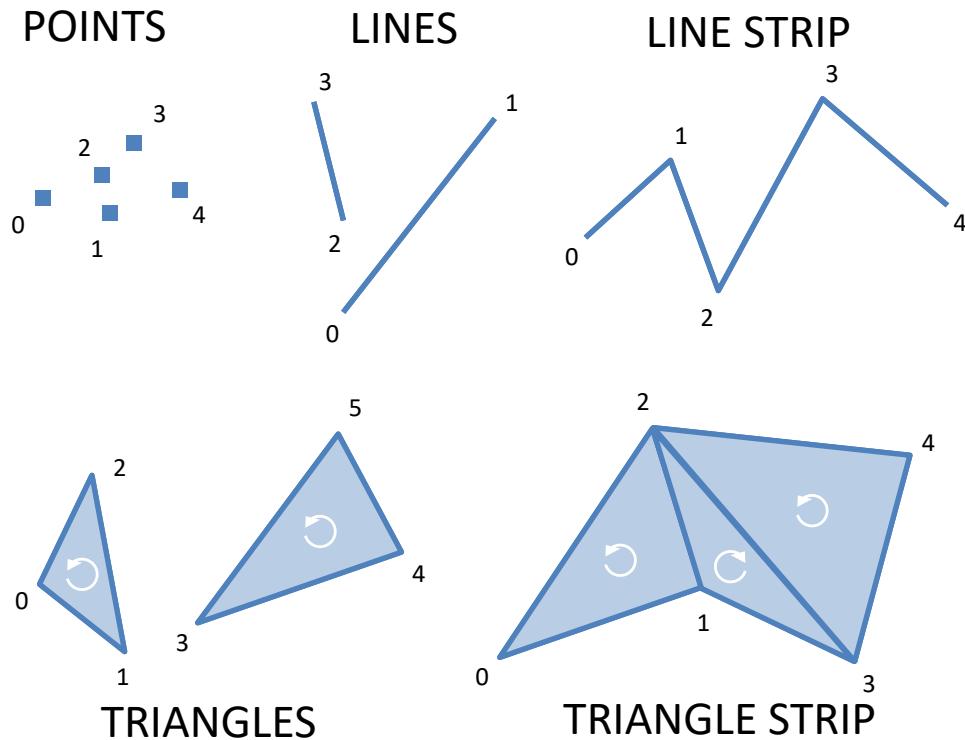
## Example command sequence

- Begin render pass
- Bind pipeline (Vulkan) or pipeline components (OpenGL)
- Bind framebuffer
- Draw vertices (=draw call)
- End render pass



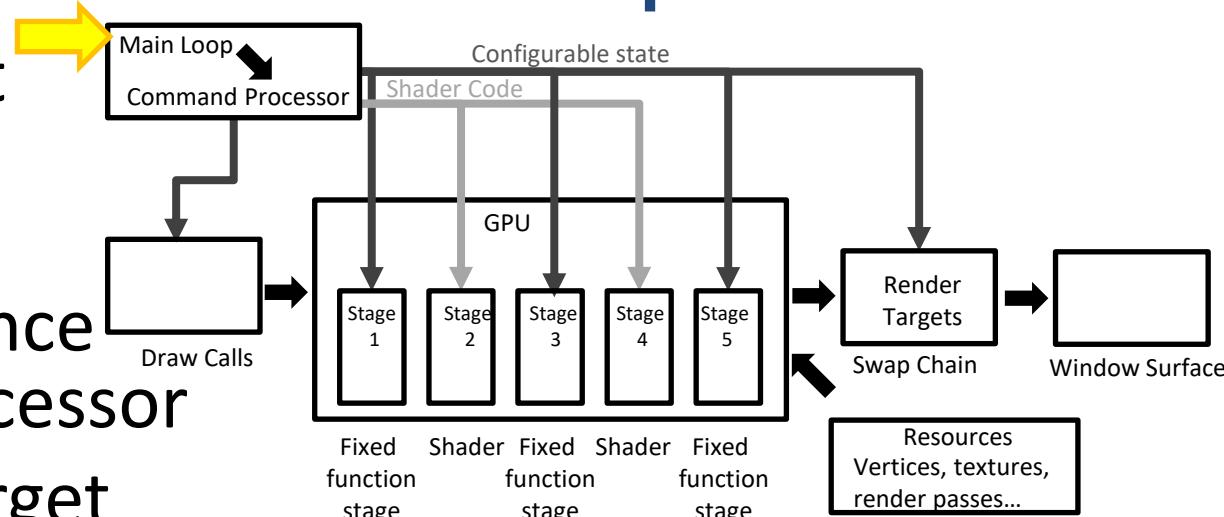
# Draw Calls

- Specify
  - Which primitive type
  - Which vertex buffer
  - Start index in buffer
  - Stop index in buffer



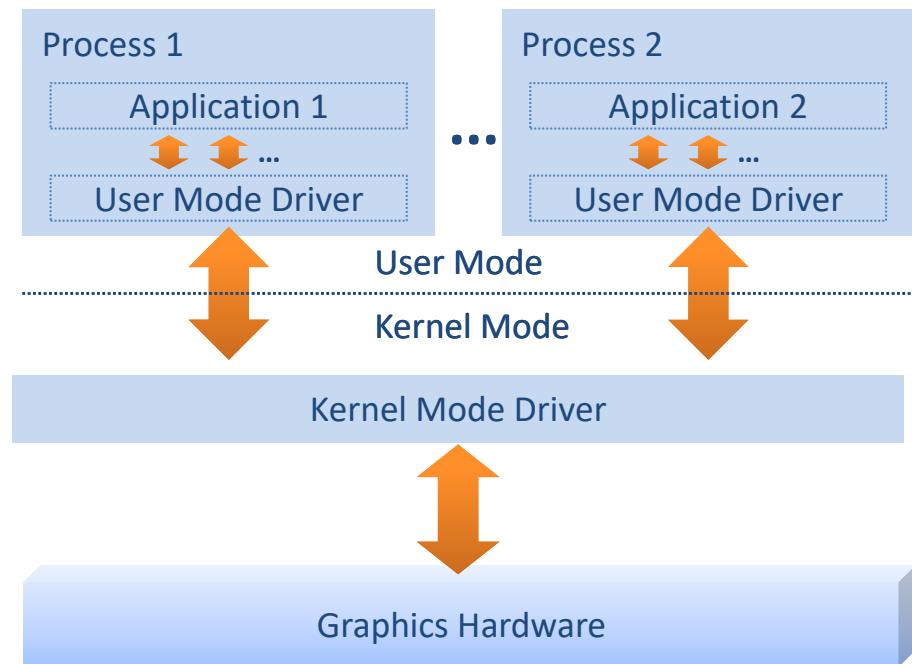
# CPU Main Loop

- Get render target from swap chain
- Submit chosen command sequence to command processor
- Return render target to swap chain
- Command submission is very different between traditional and modern API



# Graphics Driver Architecture

- User mode graphics driver
  - Minimize number of mode switches
  - Translation of graphics commands to instructions for the hardware
  - Batching, optimization, validation
  - Fine grained memory management
- Kernel mode graphics driver
  - Schedule access to hardware
  - Microkernel pattern, stability
  - Coarse grained memory management
  - Submits command buffers to GPU



# 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

# Pipeline State

- Inside a context, one must use commands to configure GPU **pipeline state** before rendering
- Pipeline state consists of
  - Programmable state (shaders)
  - Configurable state: blend, depth, culling, etc.
  - Layout: how to map settings at each stage's shader

# Problems with Traditional API

- Pipeline execution is largely asynchronous
  - CPU sends commands into a “black hole” and
  - CPU does not know exactly when commands are executed
- Configuration of state can only be done incrementally
  - Submitting configuration changes to driver requires immediate validation, conversion, buffering → high cost at runtime
  - Drivers must have per-game optimizations built in
- Pipeline state *not* made explicit in rendering context
  - Switching pipeline configurations is cumbersome
  - Must be done by sequences of state change commands
- Pipeline abstraction is single-threaded on CPU
  - Cannot multi-thread feeding the pipeline
  - Having too many draw calls becomes a bottleneck

# Modern API

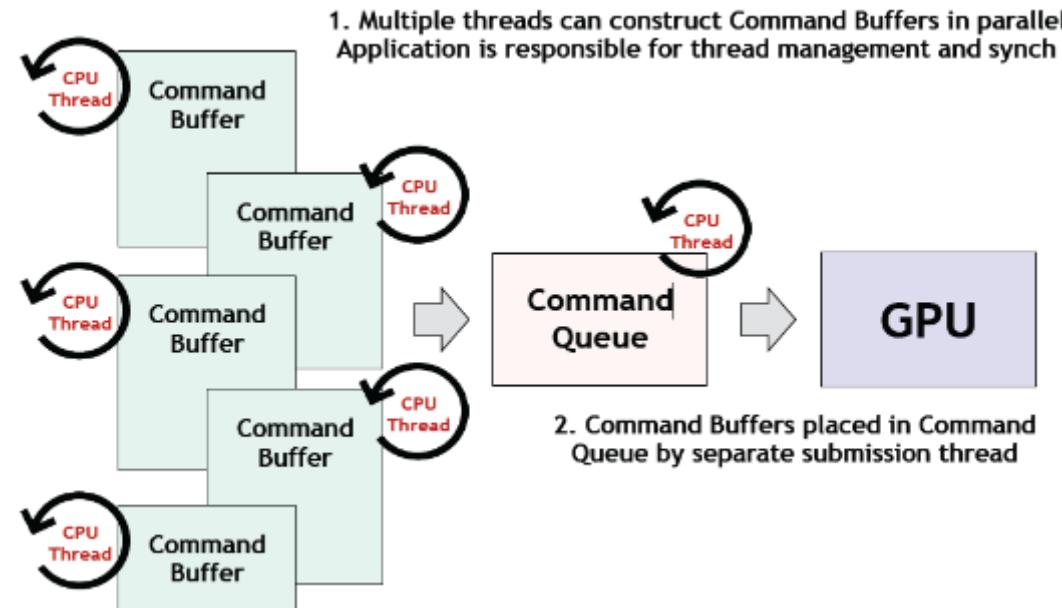
- DirectX 12, Vulkan (=OpenGL successor)
- Designed for modern GPU types (including mobile)
- Make (CPU driver side of) pipeline more programmable
- CPU multi-threading possible (at your own risk)
- Split rendering context into command buffers and queues
- Make pipeline state (and render passes) explicit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

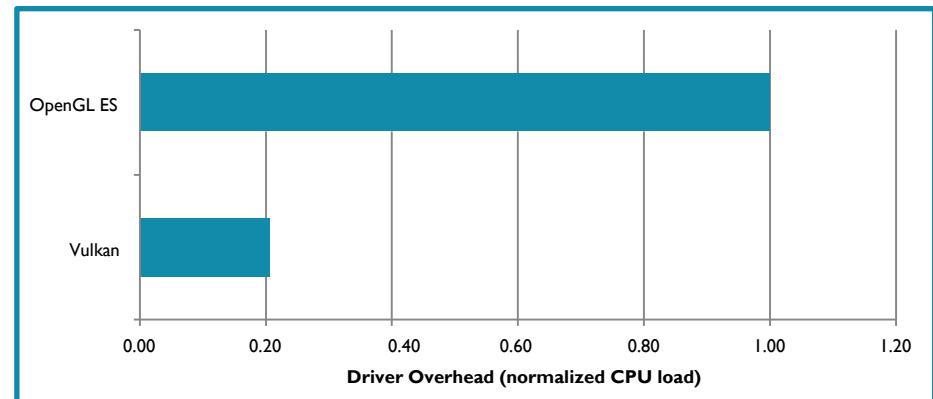
# Queues

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



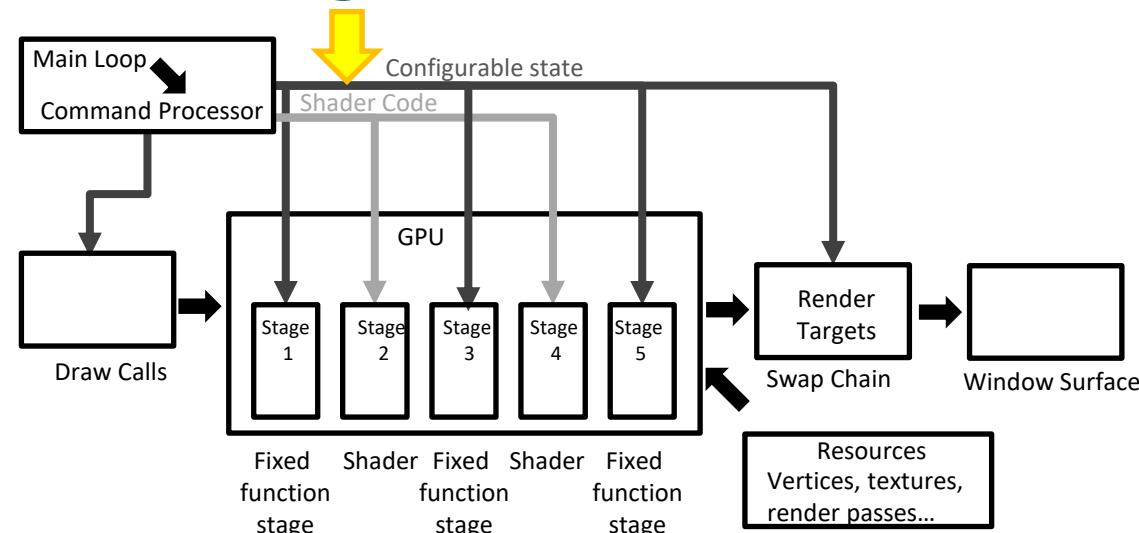
# Example: Multi-Threaded Drawing

- ARM Vulkan Benchmark
  - ARM Cortex A-15/7, Mali T-628 MP6
  - 1000 meshes, 3 materials
  - 79% less CPU



# Shader Programs

- For the programmable parts of the pipeline
- Compiled from shader language
  - HLSL, GLSL
- C-like syntax
- Stream execution model:
  - Shaders are like callbacks, no “main loop” code



# Language Elements

- Skalar data types: float, int, bool...
- Vector/matrix data types: vec2, vec3, vec4, ivec3, mat4...
- Struct, arrays (static size)
- Texture sampler: sample2D, sampler3D, ...
- Control flow: if, else, while, for
- Function calls (no recursion)
- Swizzle operators: color1.rgb = color2.bga
- Mask operator: pos1.z = pos2.x + pos2.y

# Build-In Functions

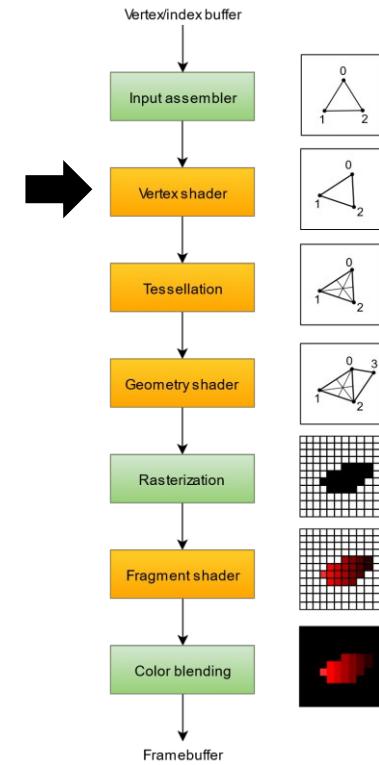
- Trigonometric
  - `sin()`, `cos()`, `radians()`, ...
- Logarithm, exponentiation
  - `log()`, `sqrt()`, ...
- Other
  - `min()`, `max()`, `mod()`, `floor()`, `abs()`, ...
- Geometric
  - `distance()`, `normalize()`, `dot()`, `length()`, ...
- Special functions
  - Linear interpolation
  - Reflection vector
  - Refraction vector

# Shader Compilation

- Compilation separated into front-end/back-end
- Front-end: HLSL, GLSL, OpenCL, etc.
- Back-end = Bytecode
  - Microsoft HLSL uses proprietary bytecode
  - Vulkan uses SPIR-V (standard portable intermediate representation)
- Applications ship with bytecode, not shader source
- Just-in-time compilation of bytecode
  - To hardware target platform (NVIDIA, AMD, ...)

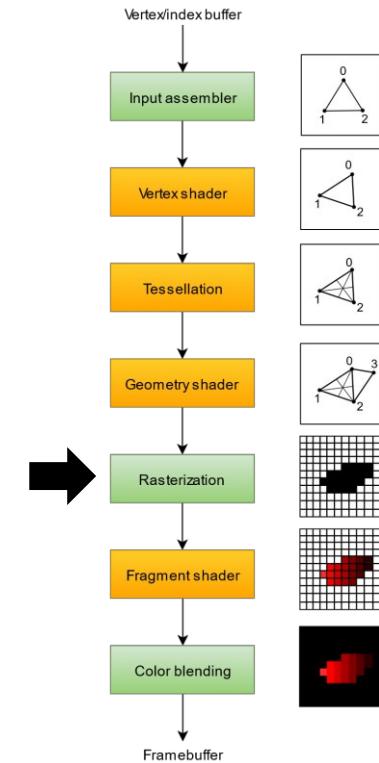
# Vertex Shader

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



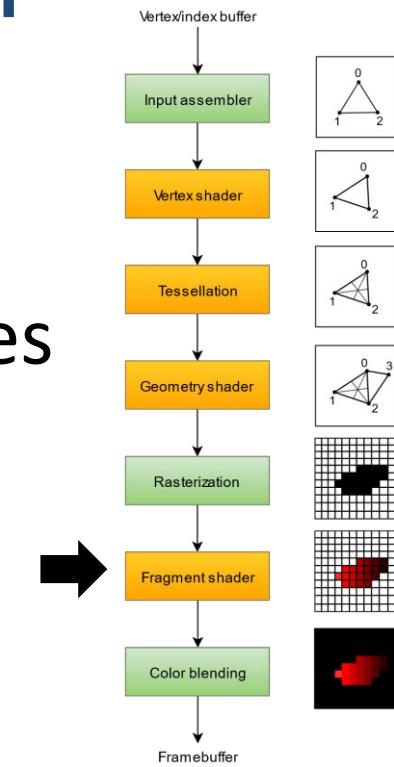
# Rasterizer

- Generates fragments covering a primitive
- Fixed-function unit
- Input: primitives, vertex attributes
- Output: fragments with interpolated vertex attributes

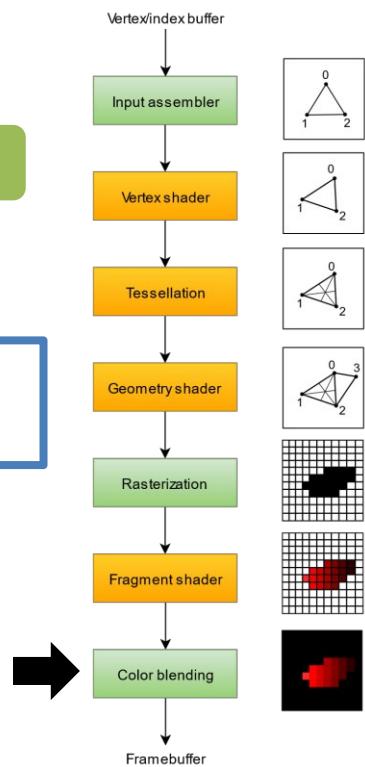
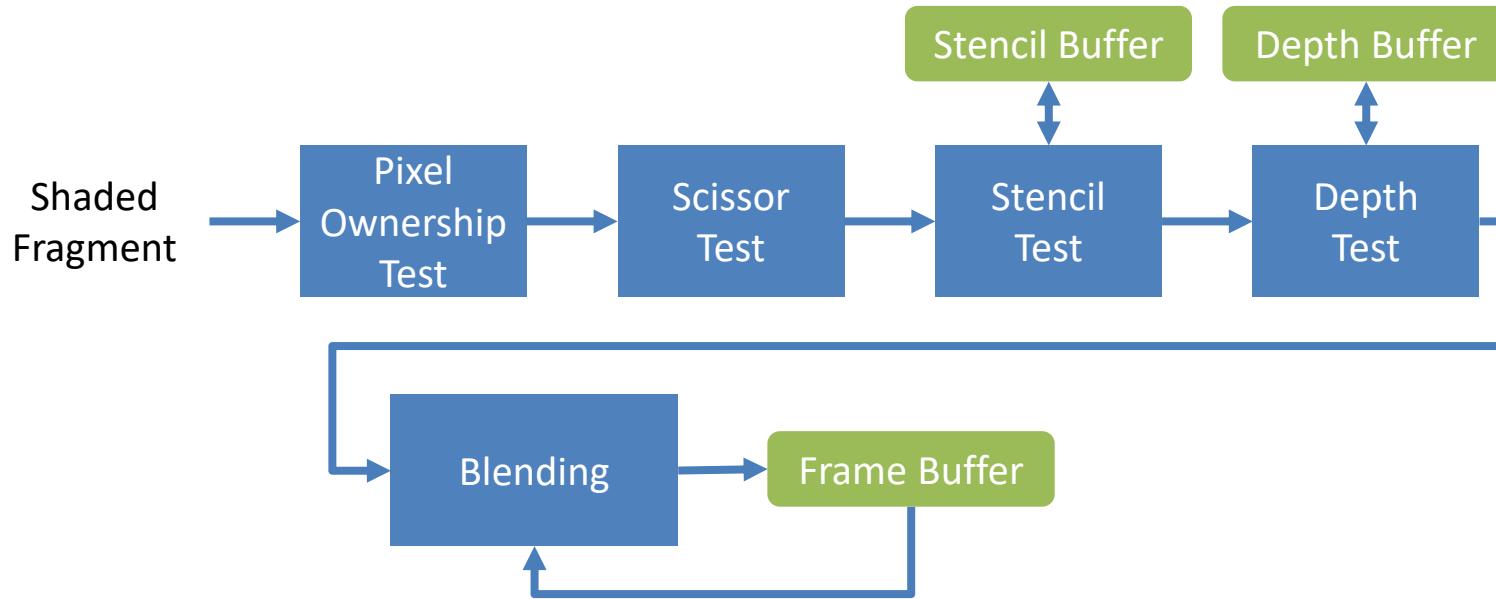


# Fragment Shader

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



# Fragment Merging



# Anatomy of a GLSL Shader

```
1 #version 330
2 // uniform inputs are constant for all shader invocations
3 uniform vec4 some_uniform;
4 // inputs are varying with each shader invocation
5 layout(location = 0) in vec3 some_input;
6 layout(location = 1) in vec4 another_input;
7 // outputs
8 out vec4 some_output;
9 void main()
10 {
11     //...
12 }
```

# Built-In Variables

- Interface to fixed-function parts of pipeline
  - E. g. vertex shader:
    - `in int gl_VertexID;`
    - `out vec4 gl_Position;`
  - E. g. fragment shader:
    - `in vec4 gl_FragCoord;`
    - `out float gl_FragDepth;`

# Example: Vertex Shader

```
1 #version 330
2
3 uniform mat4 mvp_matrix; // model-view-projection matrix
4
5 layout(location = 0) in vec3 vertex_position;
6 layout(location = 1) in vec4 vertex_color;
7
8 out vec4 color;
9
10 void main()
11 {
12     gl_Position = mvp_matrix * vec4(vertex_position, 1.0f);
13     color = vertex_color;
14 }
```

# Example: Fragment Shader

```
1 #version 330
2
3 layout(location = 0) in vec4 color; // interpolated color
4
5 out vec4 fragment_color;
6
7 void main()
8 {
9     fragment_color = color;
10 }
```

# Thank You!

# Questions ?

Image source: Epic Games

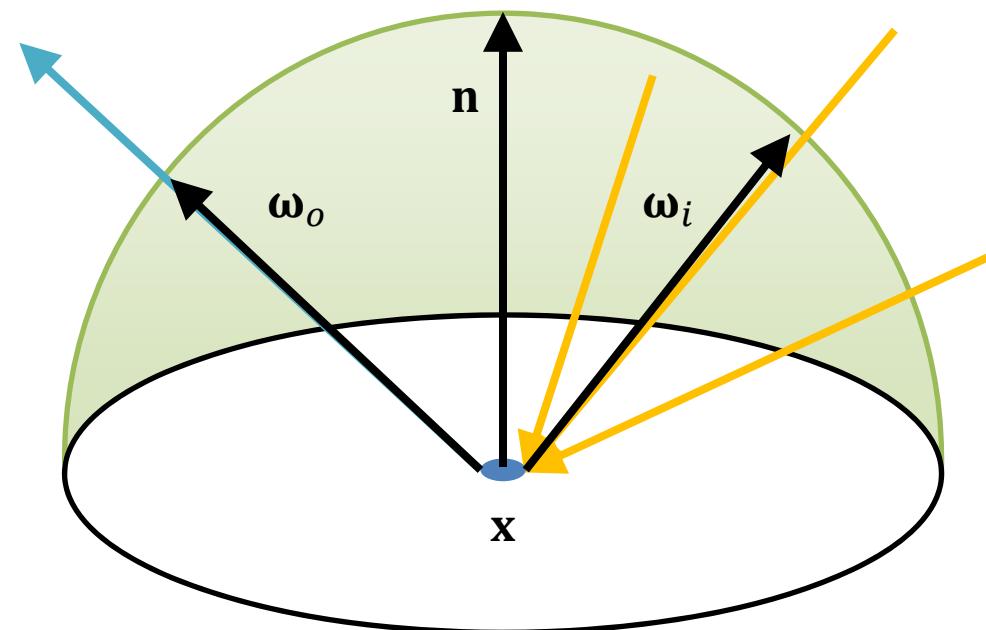


# Shading Models

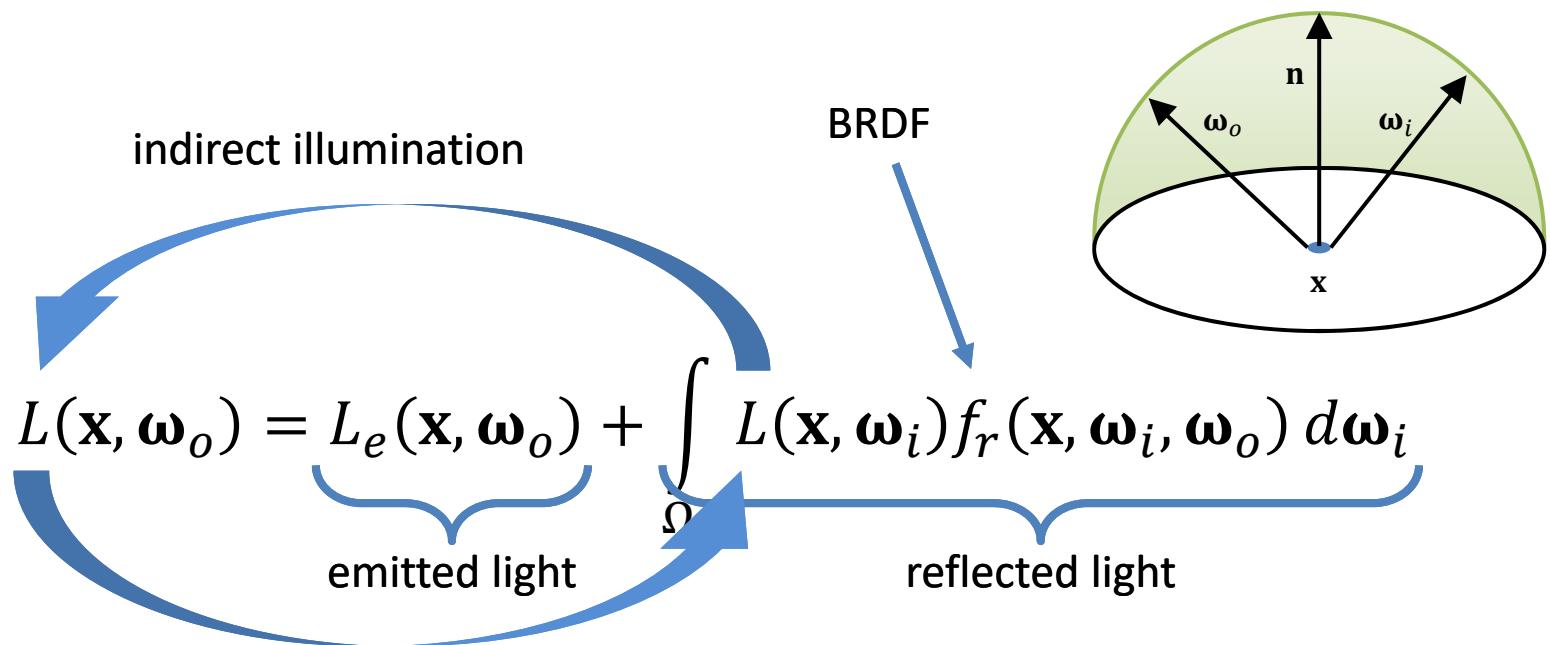
# Shading

- We know which pixels are occupied by which object
  - E. g., from rasterization
- What color should the pixels have?
  - How is light reflected by an object?

# What Happens at a Surface?



# Rendering Equation



# BRDF

- **Bidirectional Reflectance Distribution Function**
- Describes how a surface reflects light
  - At location  $\mathbf{x}$
  - From incoming direction  $\omega_i$
  - Into outgoing direction  $\omega_o$

$$f_r(\mathbf{x}, \omega_i, \omega_o)$$

# BRDF Properties

- Reciprocity

$$f_r(\mathbf{x}, \boldsymbol{\omega}_1, \boldsymbol{\omega}_2) = f_r(\mathbf{x}, \boldsymbol{\omega}_2, \boldsymbol{\omega}_1) \quad \forall \boldsymbol{\omega}_1, \boldsymbol{\omega}_2$$

- Energy conservation

$$\int_{\Omega} f_r(\mathbf{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) d\boldsymbol{\omega}_i \leq 1 \quad \forall \boldsymbol{\omega}_o$$

- Positivity

$$f_r(\mathbf{x}, \boldsymbol{\omega}_1, \boldsymbol{\omega}_2) \geq 0$$

# Light Sources

- Most general light source: area light
- Problem: at every location, light is coming from a range of directions
  - Integration needed
  - Analytic solution only for extremely trivial cases
- Simplification needed

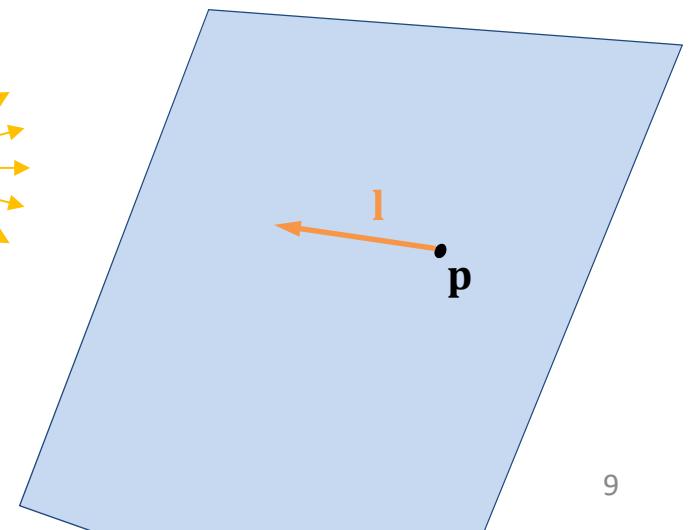
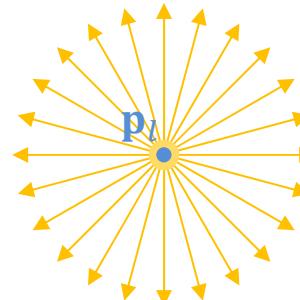


# Light Sources in Real-Time

- Local shading
  - Direct illumination only
  - $O(n^\infty)$  reduces to  $O(n)$
- Consider analytical light sources only
  - Point light (infinitely small)
  - Directional light (infinitely far away)
  - Light from a single direction  $\mathbf{I} \rightarrow$  integral vanishes

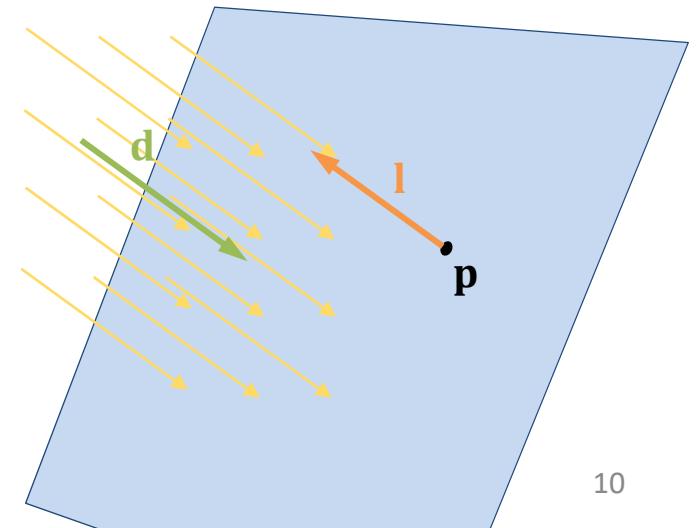
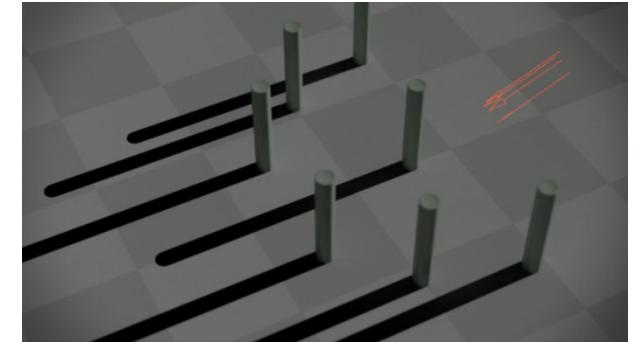
# Point Light

- Simplification
  - Light source infinitesimally small
  - At every location  $\mathbf{p}$ , light incoming from a single direction only
- Parameters:
  - $\mathbf{p}_l$  light position



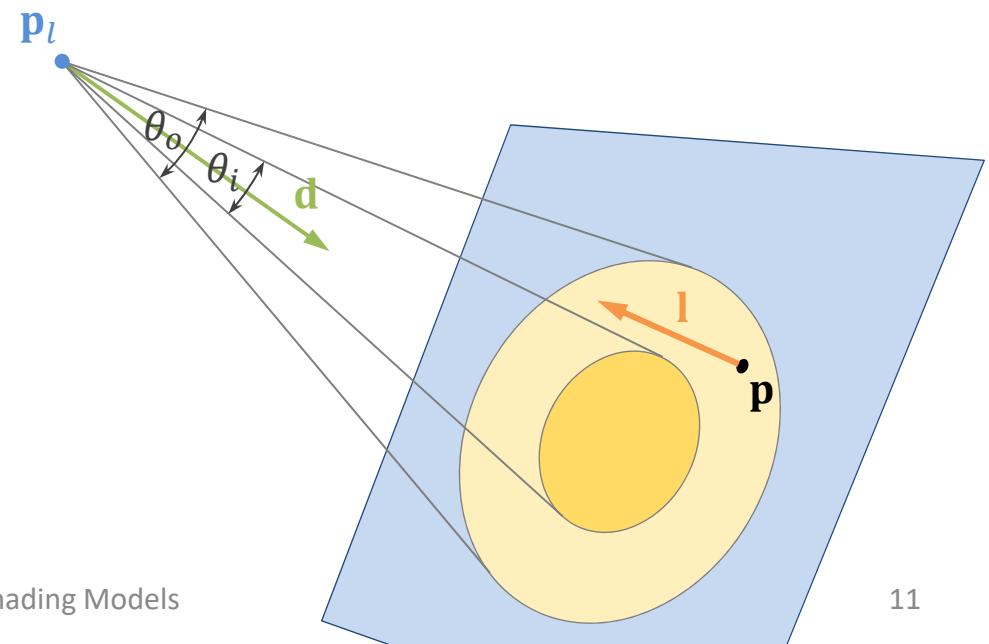
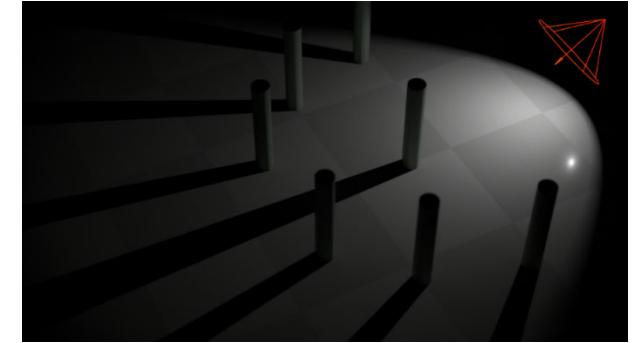
# Directional Light

- Simplification
  - Light source infinitely far away
  - At every location  $\mathbf{p}$ , light from the same direction only
- Parameters:
  - $\mathbf{d}$  light direction



# Spot Light

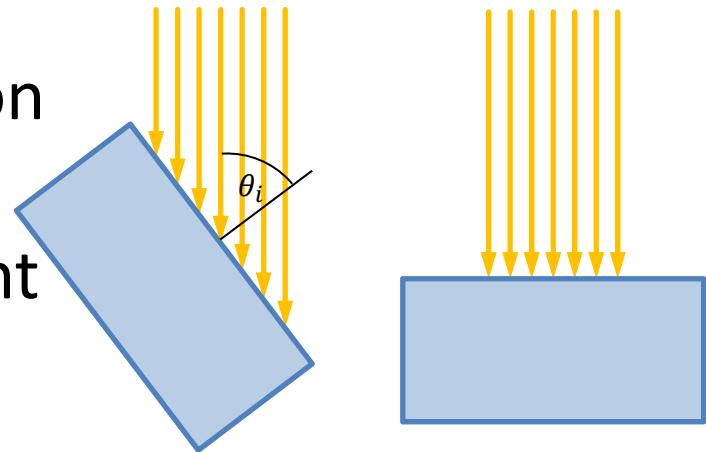
- Like point light
- Volume constrained to a cone
- Parameters:
  - $p_l$  light position
  - $d$  cone direction
  - $\theta_i$  inner cone angle
  - $\theta_o$  outer cone angle



# Lambert Model

- Assumption of a *perfectly diffuse* reflector
- Scatters light evenly in all directions
  - View independent
  - Depends only on orientation of surface towards light
  - Surface irradiance from light

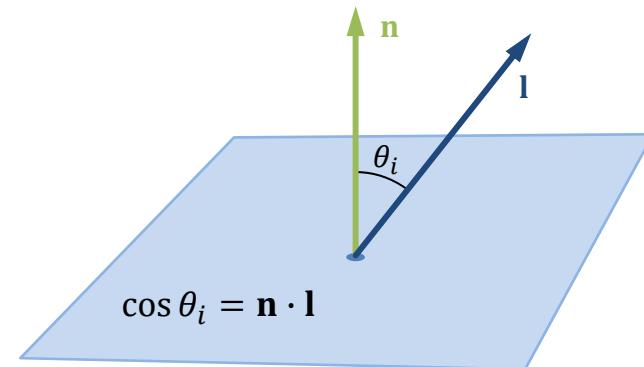
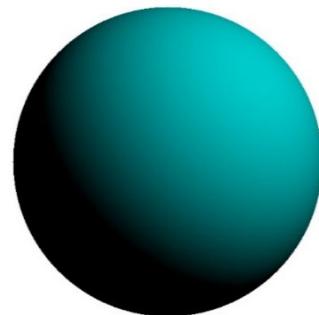
$$E_L \propto \cos \theta_i$$



# Lambert Shading Computation

$$L_o = c_d \circ \max(\mathbf{n} \cdot \mathbf{l}, 0) \circ I_L$$

- $c_d$  Diffuse reflectance („albedo“)
- $I_L$  Irradiance from light



# Diffuse vs Specular

Diffuse + specular



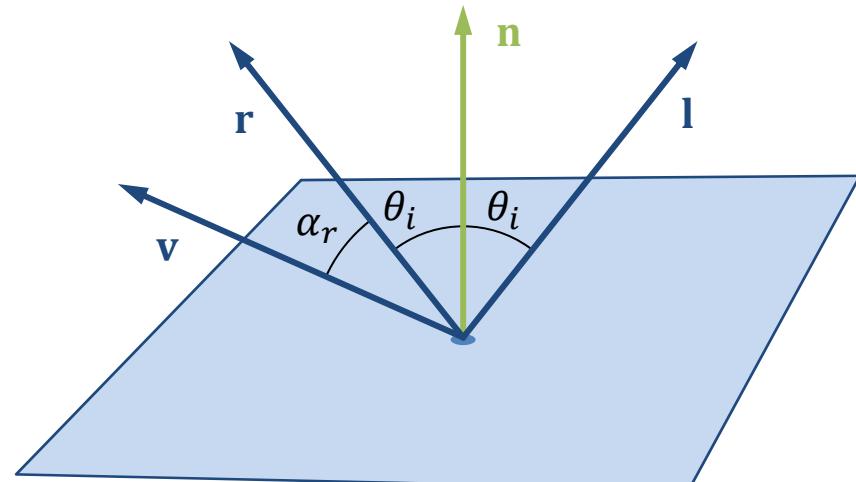
Diffuse



# Phong Model

- $\mathbf{L}_o = \mathbf{c}_e + (\mathbf{c}_d \circ \cos \theta_i + \mathbf{c}_s \circ (\cos \alpha_r)^m) \circ \mathbf{B}_L$

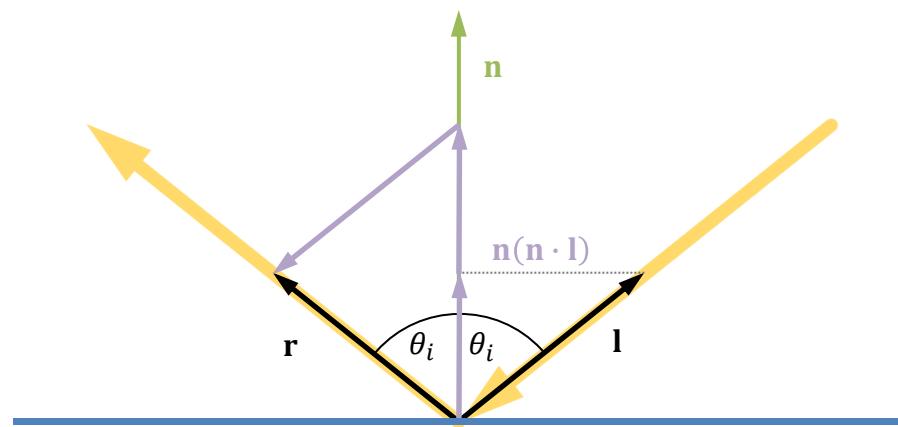
- $\mathbf{c}_e$  Emissive color
- $\mathbf{c}_d$  Diffuse color
- $\mathbf{c}_s$  Specular color
- $m$  Specular power
- $\mathbf{B}_L$  Light color



$$\mathbf{r} = 2\mathbf{n}(\mathbf{n} \cdot \mathbf{l}) - \mathbf{l}$$

All vectors assumed to be normalized!

# Reflection



$$\mathbf{r} = 2\mathbf{n}(\mathbf{n} \cdot \mathbf{l}) - \mathbf{l}$$

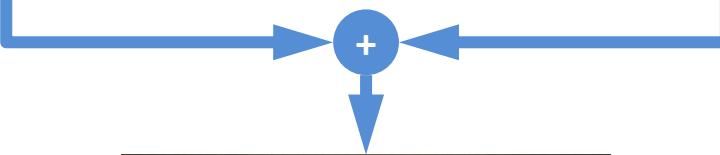
All vectors assumed to be normalized!



# Diffuse and Specular Combined 1

Diffuse

Specular

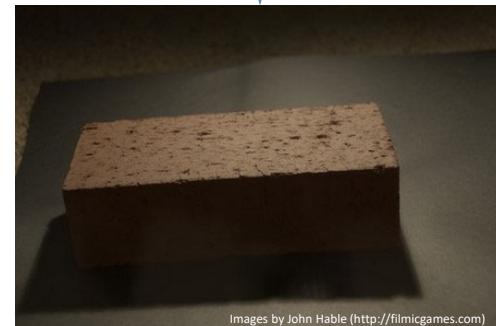


# Diffuse and Specular Combined 2

Diffuse

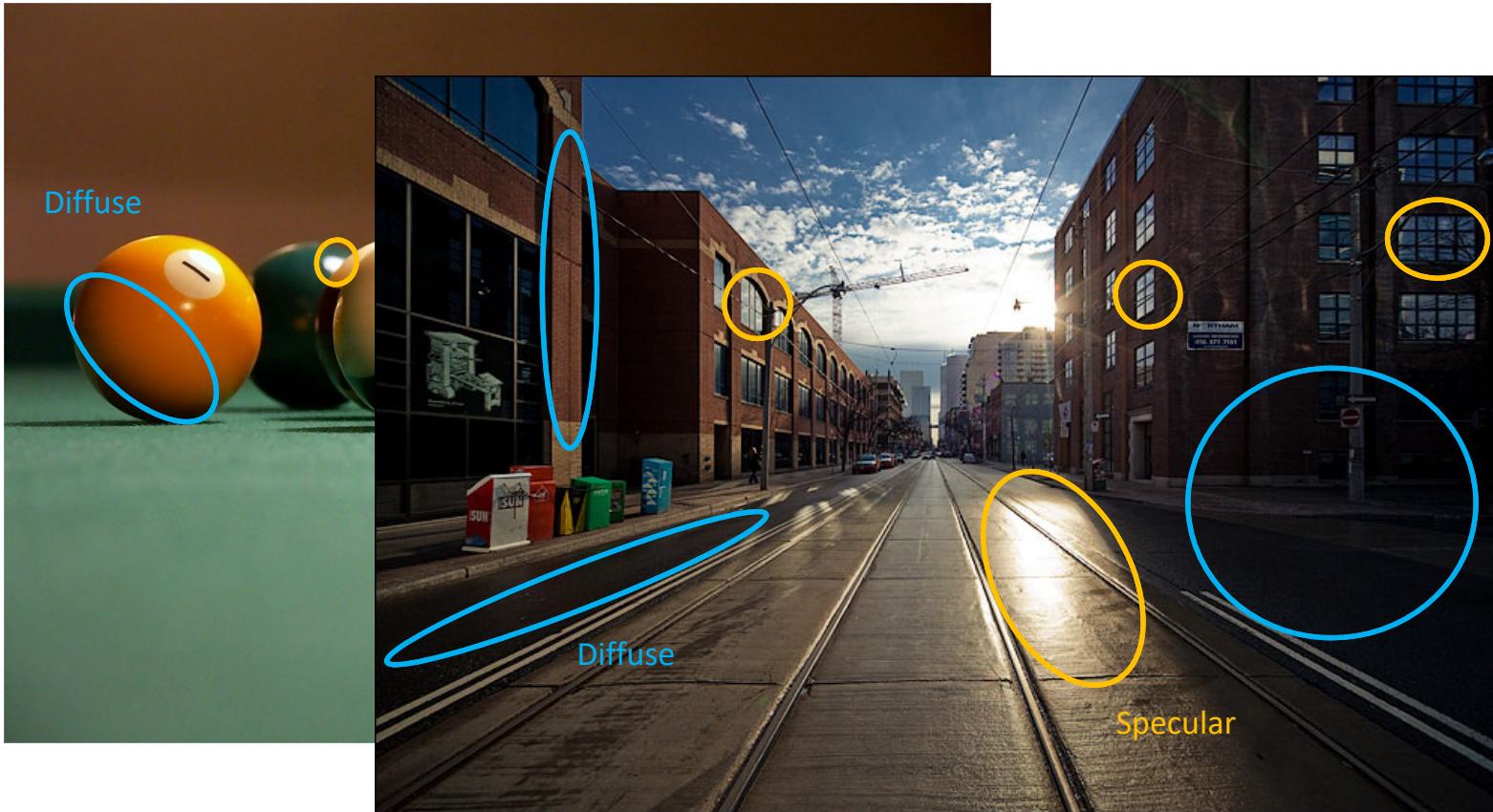
Images by John Hable (<http://filmicgames.com>)

Specular

Images by John Hable (<http://filmicgames.com>)Images by John Hable (<http://filmicgames.com>)

Specular reflection  
on brick is rather white  
(from light), not red

# Example Images

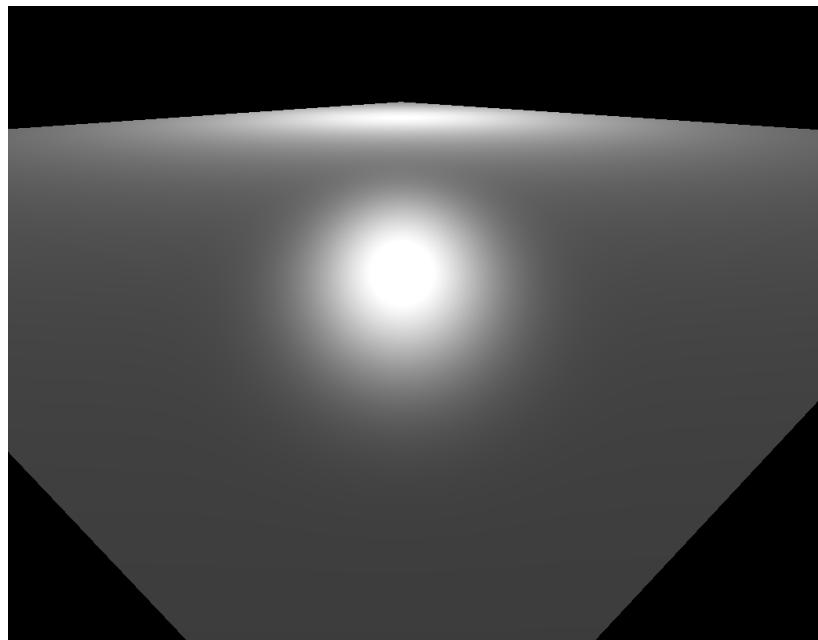


# Phong Shading Properties

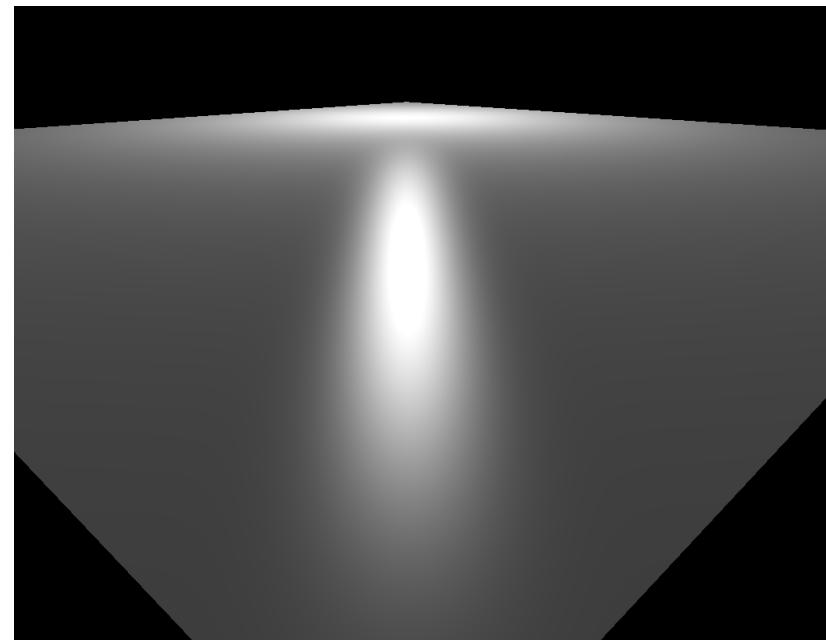
- Phenomenological model
  - Not a physically meaningful BRDF
- Highlights always circular
- Energy conservation ignored
  - Large  $m$  should lead to smaller, but stronger highlight
  - Normalization would be needed

# Highlight Shape

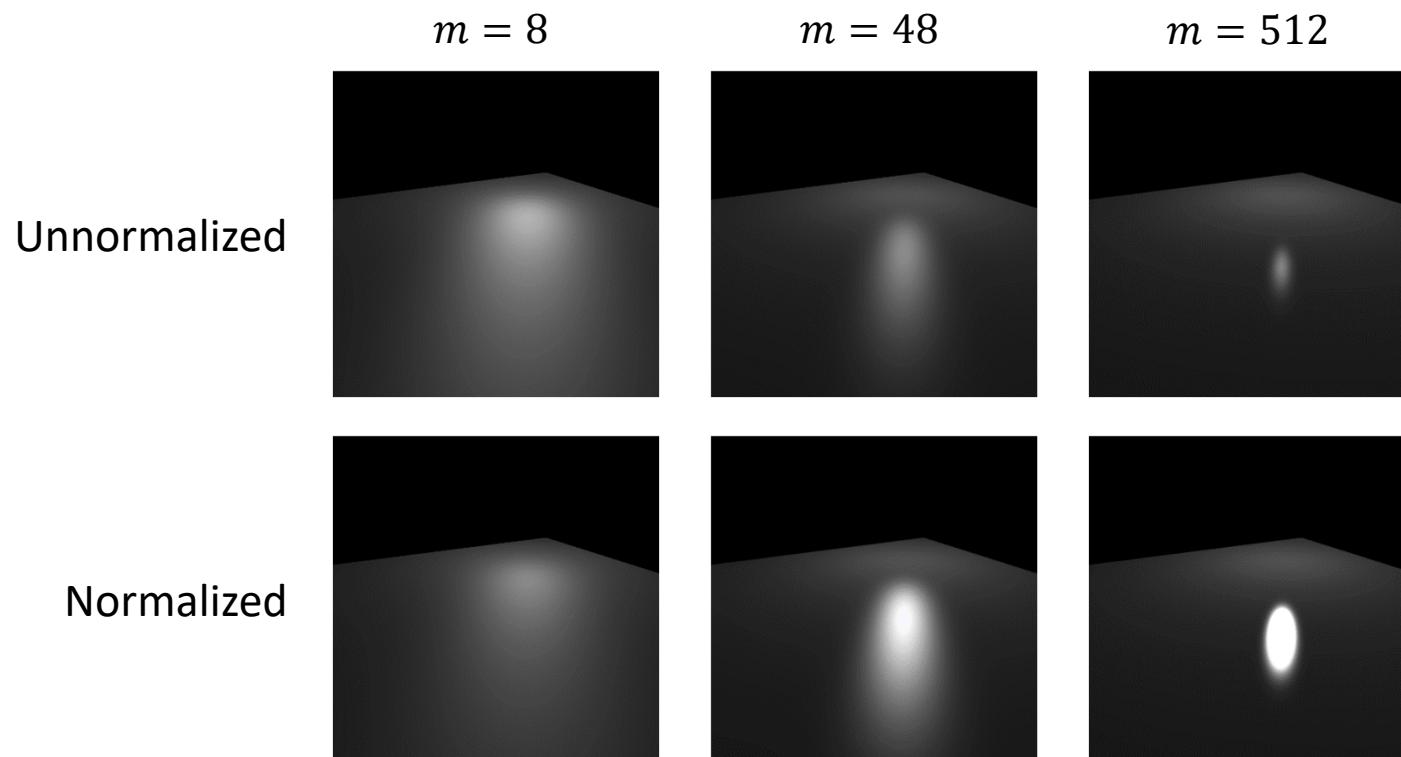
Phong



Blinn-Phong



# Normalization



# Energy Conservation

Specular tweaked for glossy



Specular tweaked for dull



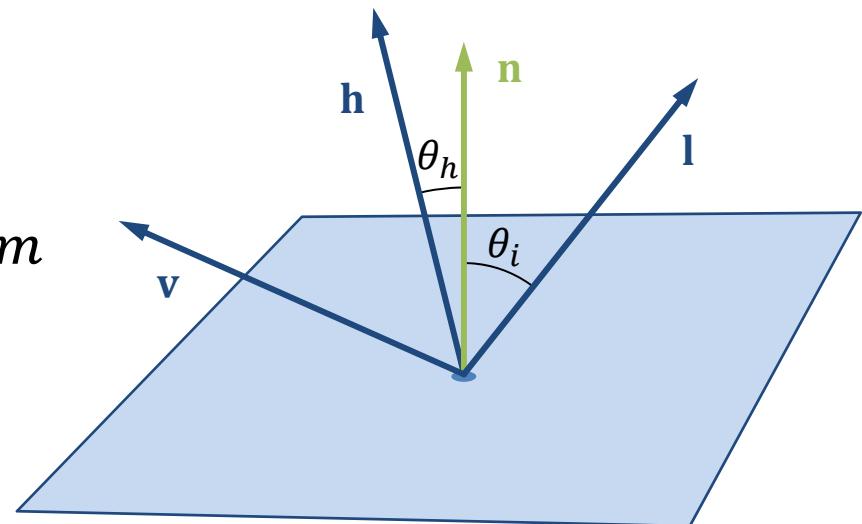
Energy conserving



# 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

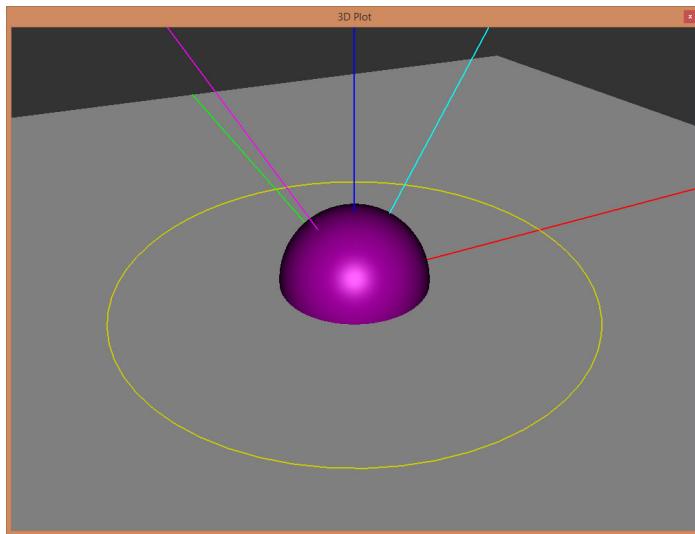
All vectors assumed to be normalized!



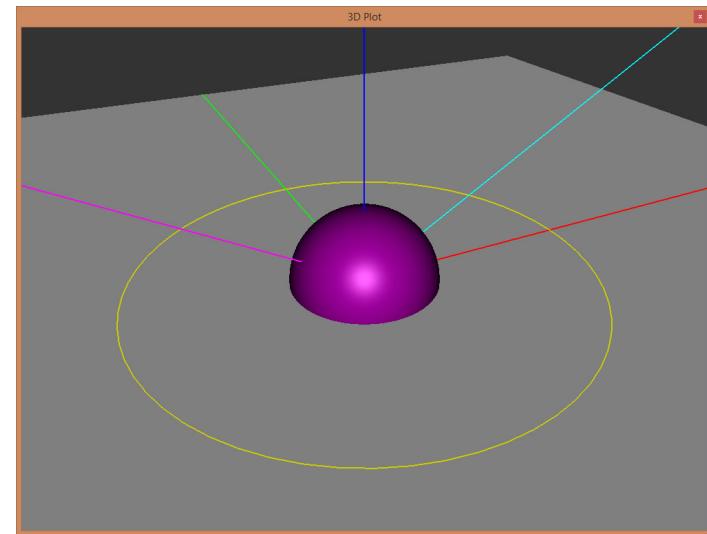
$$\mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}$$

# Lambert BRDF

$$\theta_i = 30^\circ$$

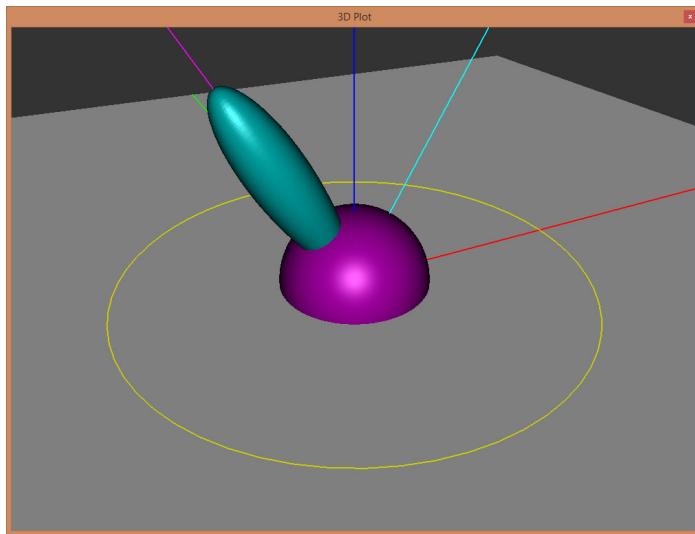


$$\theta_i = 60^\circ$$

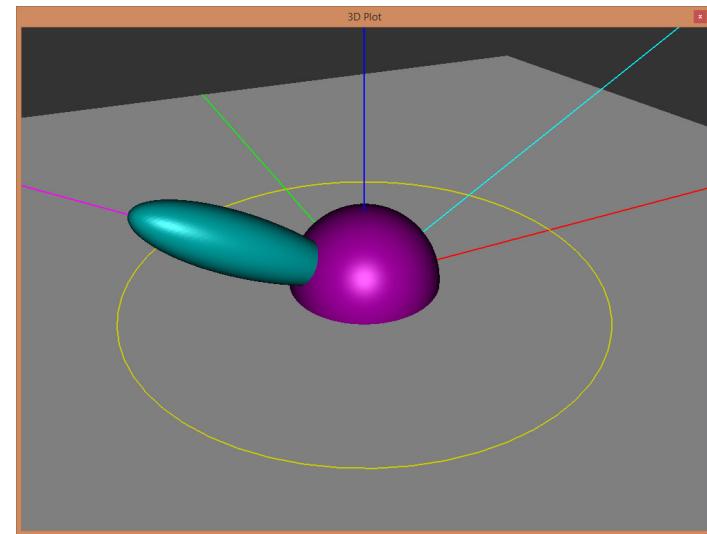


# Phong BRDF 1

$$\theta_i = 30^\circ$$

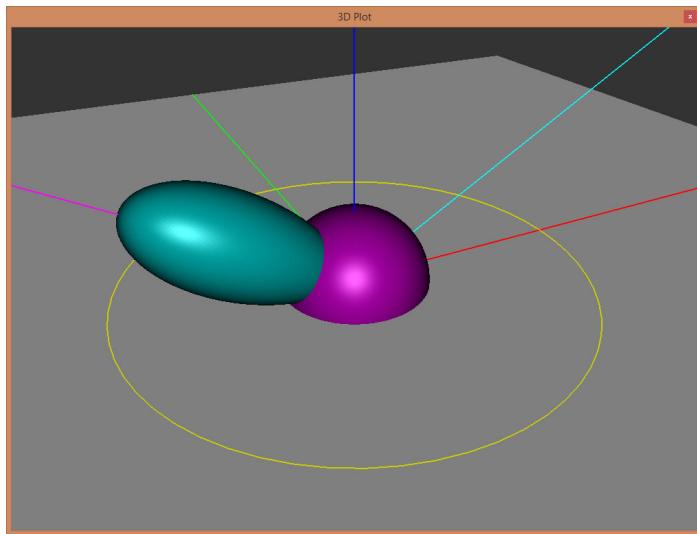


$$\theta_i = 60^\circ$$

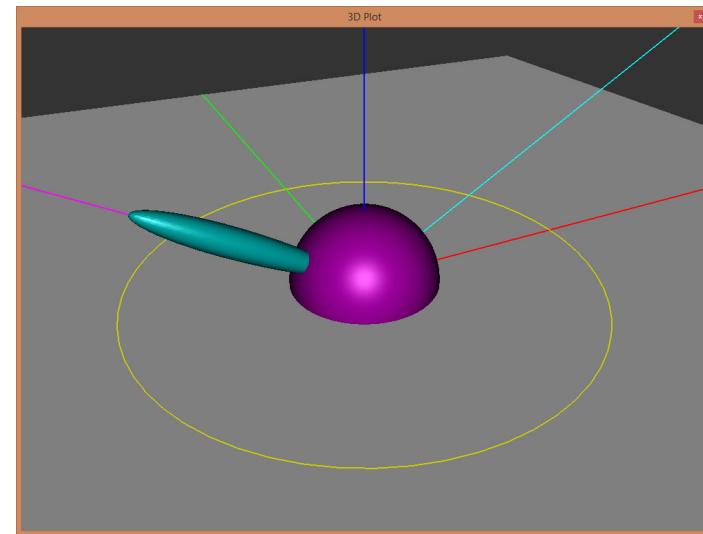


# Phong BRDF 2

$m = 8$

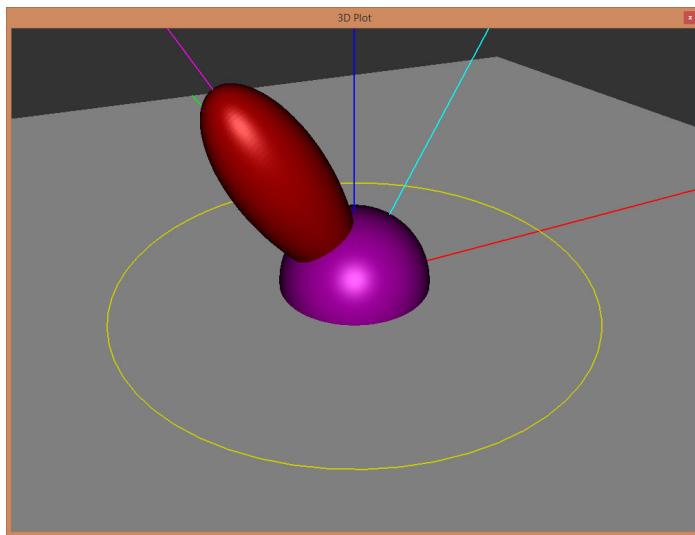


$m = 100$

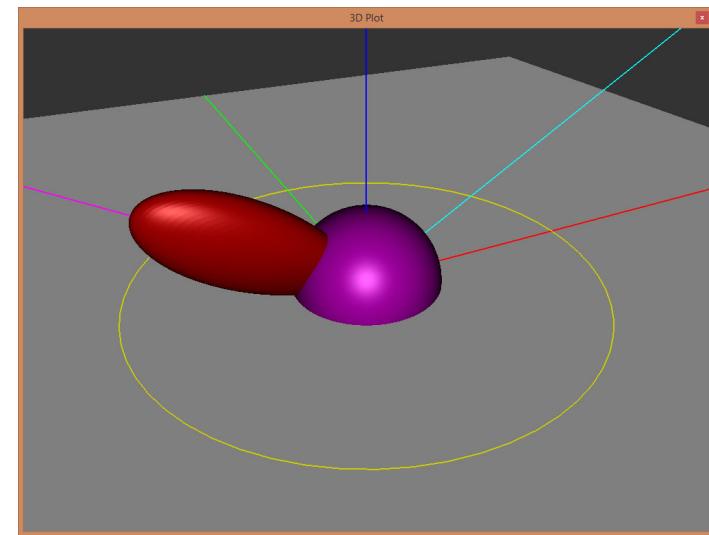


# Blinn-Phong BRDF 1

$$\theta_i = 30^\circ$$

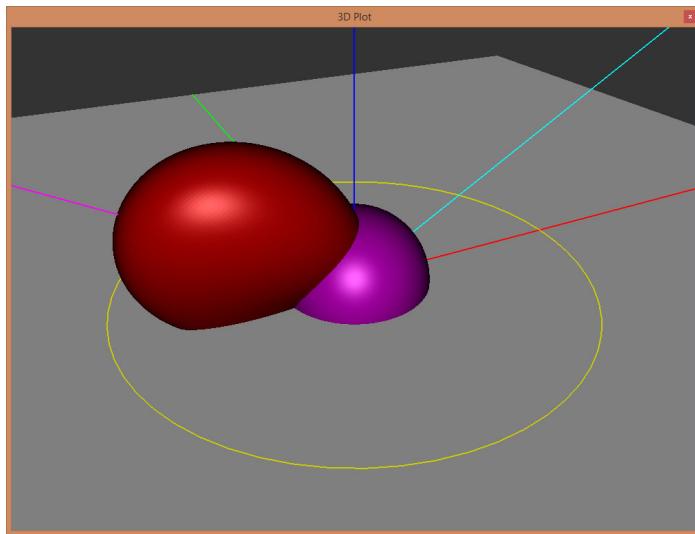


$$\theta_i = 60^\circ$$

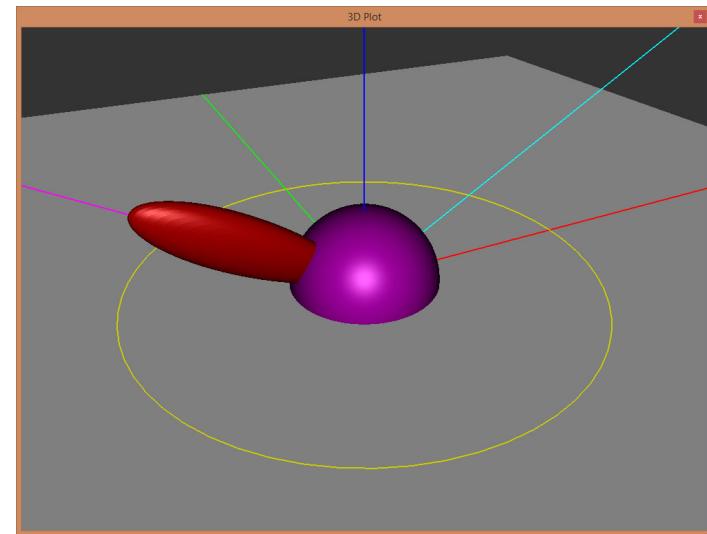


# Blinn-Phong BRDF 2

$m = 8$



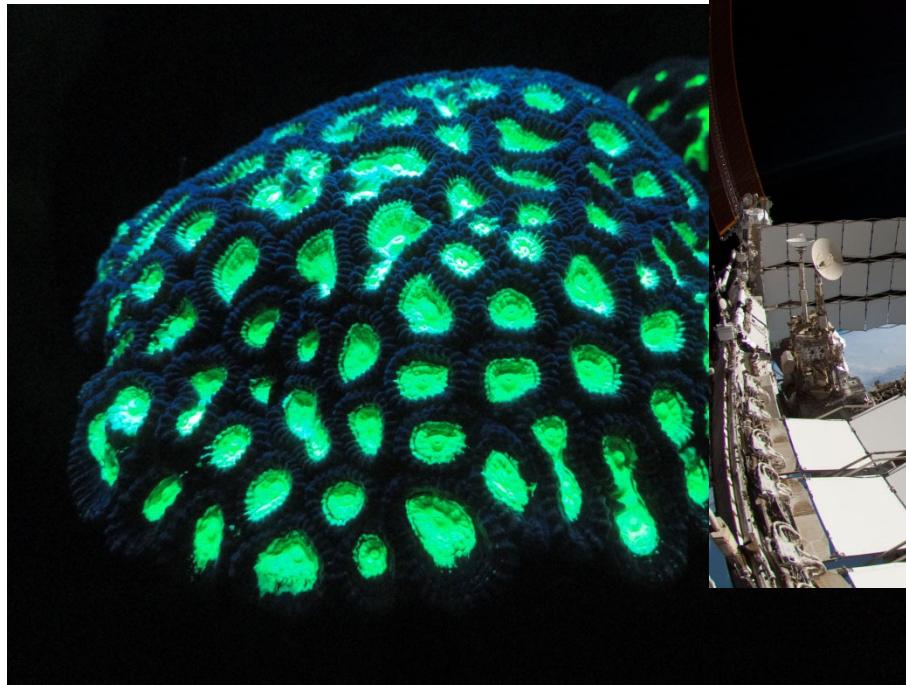
$m = 100$



# Physically-Based Rendering

- Derive shading from (simplified) physics instead of just re-creating the phenomena
- All light-matter interaction boils down to
  - Emission
  - Scattering
  - Absorption

# Emission

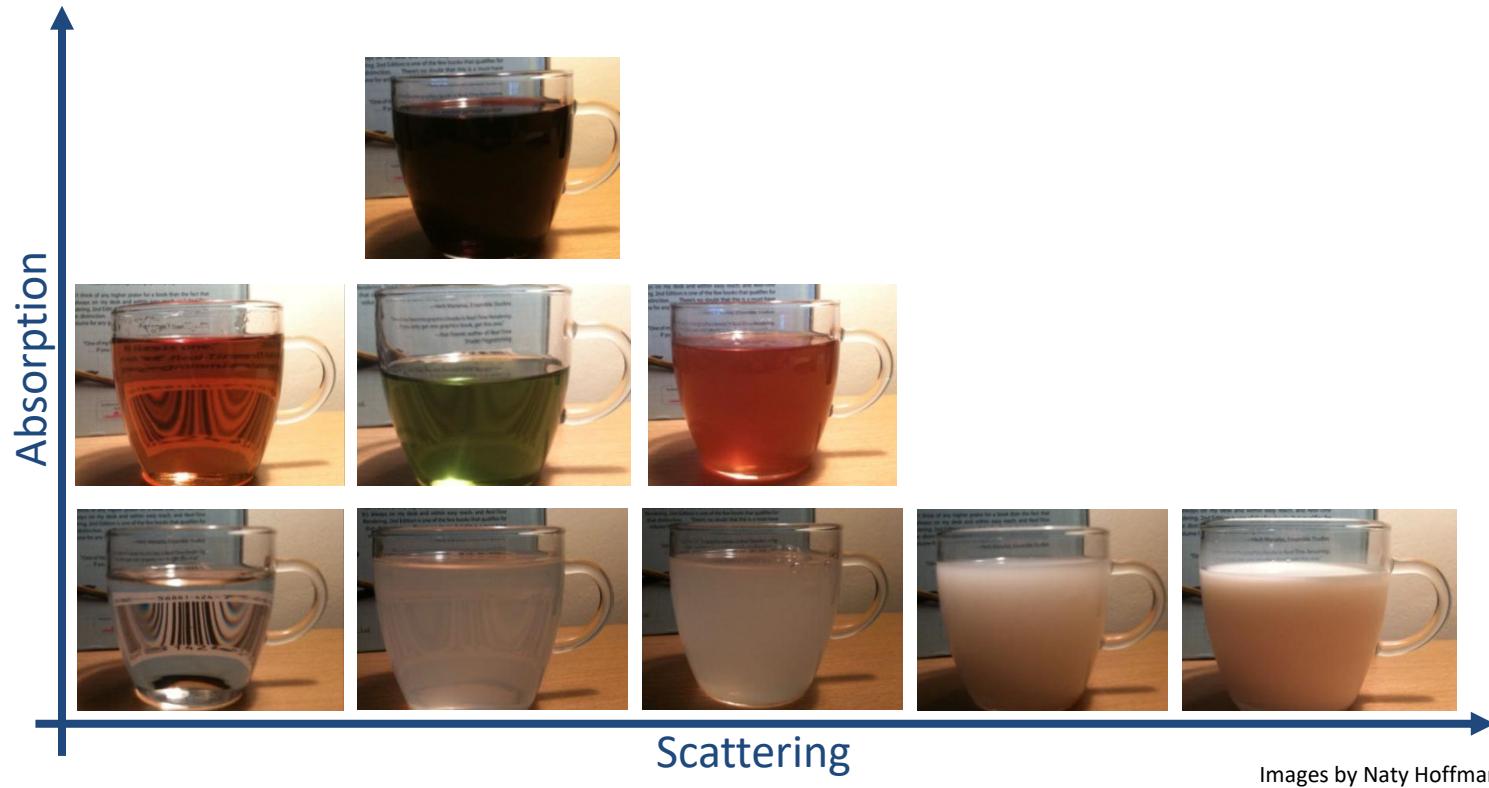


Dieter Schmalstieg

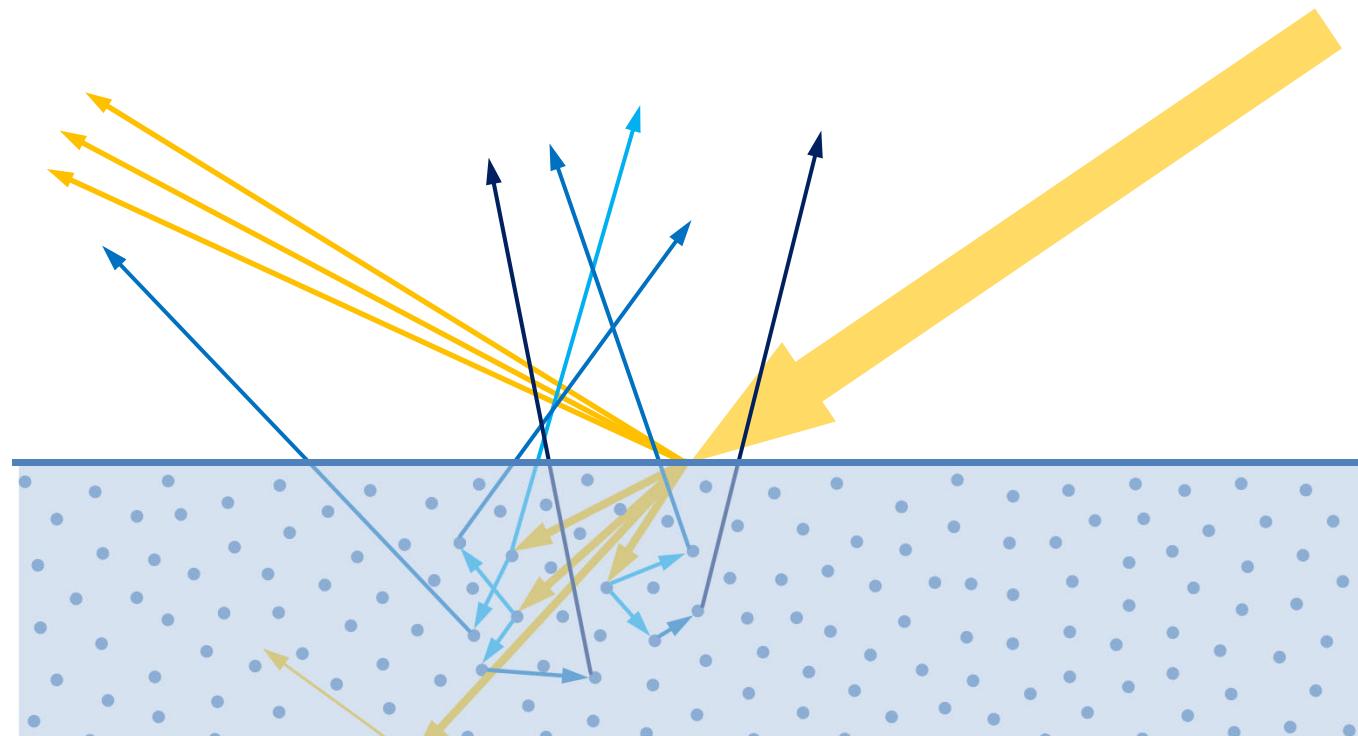


Shading Models

# Absorption vs. Scattering



# Nature

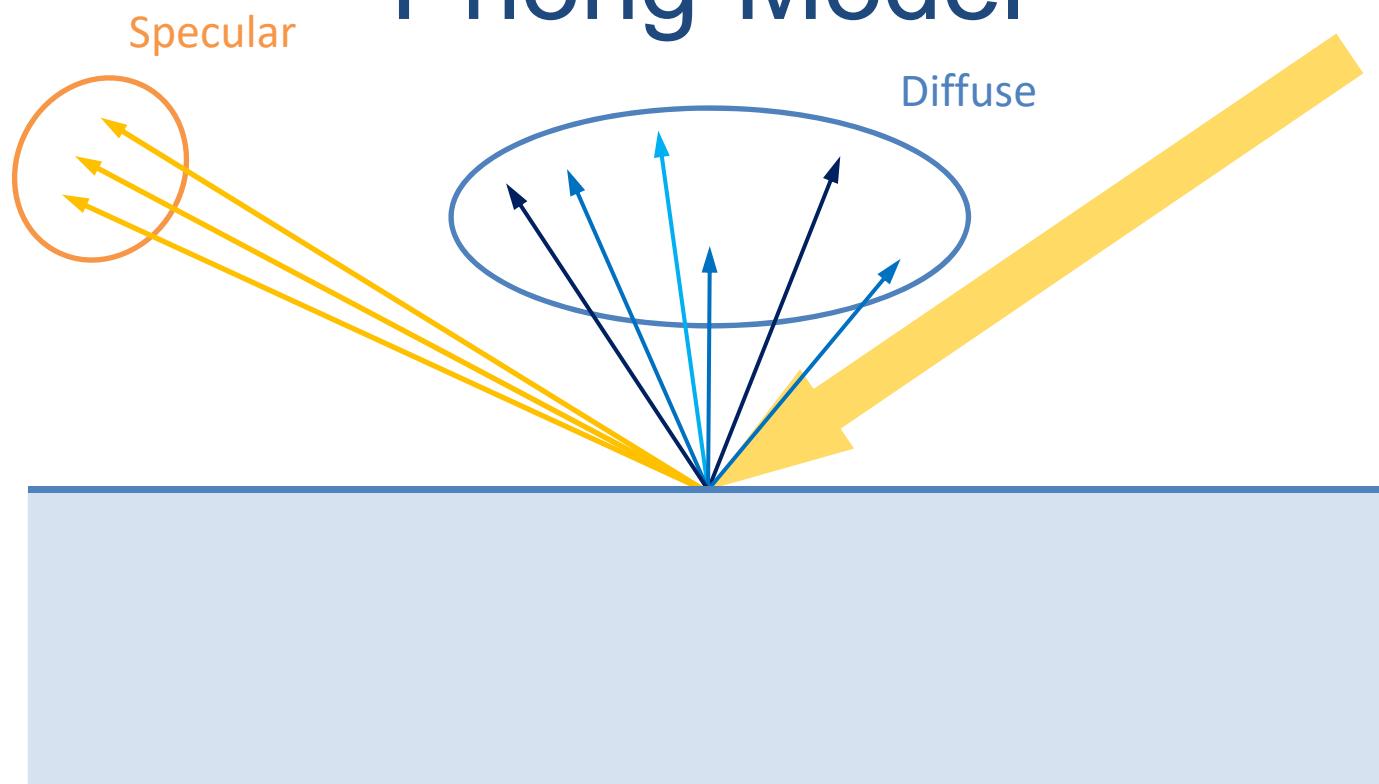


Dieter Schmalstieg

Shading Models

14

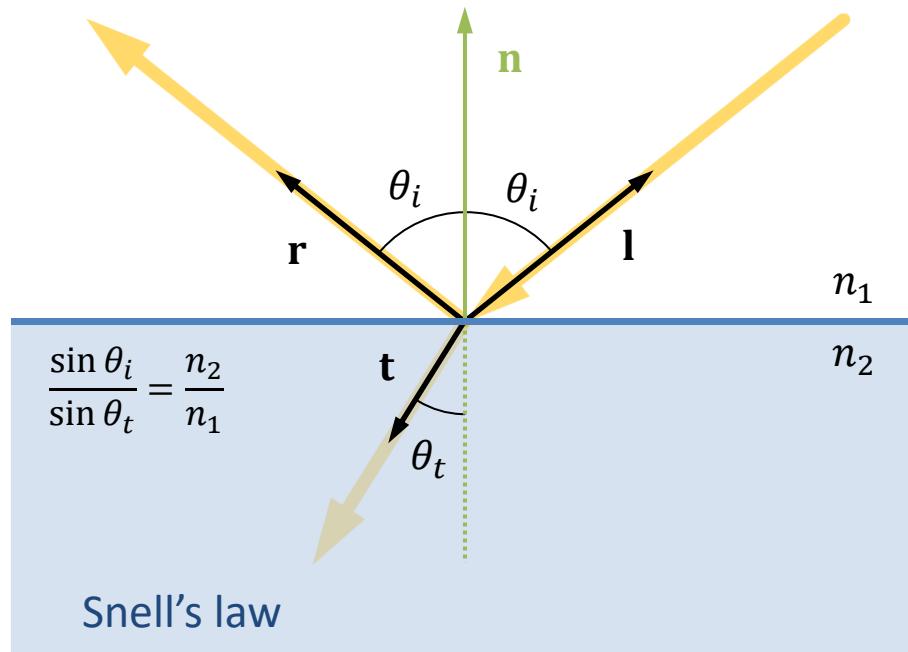
# Phong-Model



# 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.*

# Fresnel Equations



$$R_s = \left| \frac{n_1 \cos \theta_i - n_2 \sqrt{1 - \left( \frac{n_1}{n_2} \sin \theta_i \right)^2}}{n_1 \cos \theta_i + n_2 \sqrt{1 - \left( \frac{n_1}{n_2} \sin \theta_i \right)^2}} \right|^2$$

$$R_p = \left| \frac{n_1 \sqrt{1 - \left( \frac{n_1}{n_2} \sin \theta_i \right)^2} - n_2 \cos \theta_i}{n_1 \sqrt{1 - \left( \frac{n_1}{n_2} \sin \theta_i \right)^2} + n_2 \cos \theta_i} \right|^2$$

$R, T$  depend only on  $n_1, n_2$ !

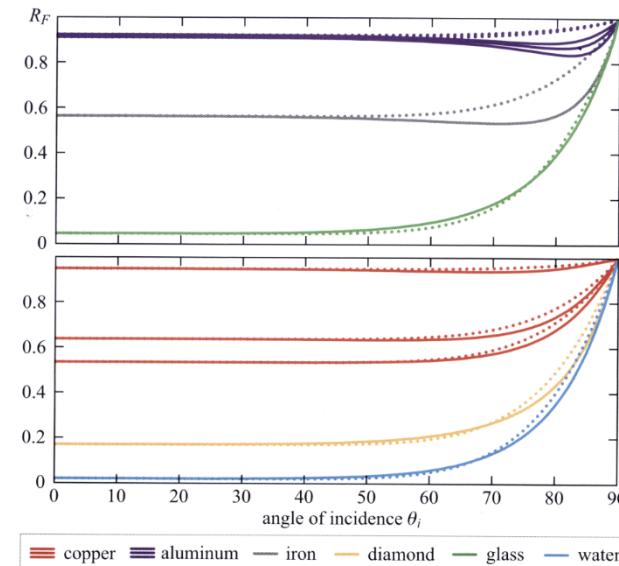
Specular  $R = \frac{R_s + R_p}{2}$

$T = 1 - R$  Diffuse

# Schlick's Approximation of Reflection

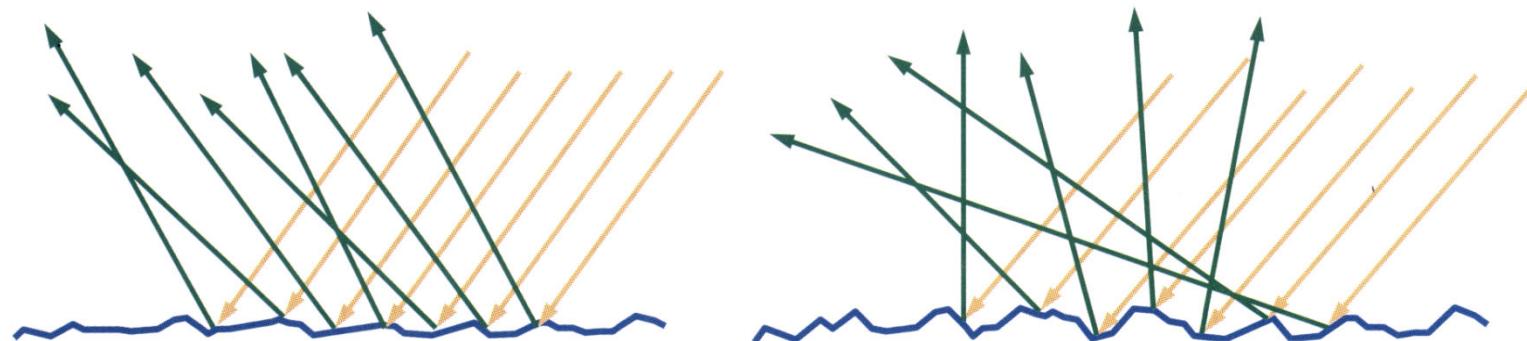
Easier (faster) to compute than original Fresnel equations

$$R = R_0 + (1 - R_0)(1 - \cos \theta_i)^5$$



[Schlick 1994]

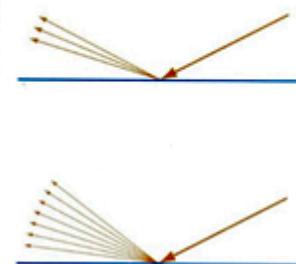
# Surface Roughness



Metallic surfaces: diffuse model not really working



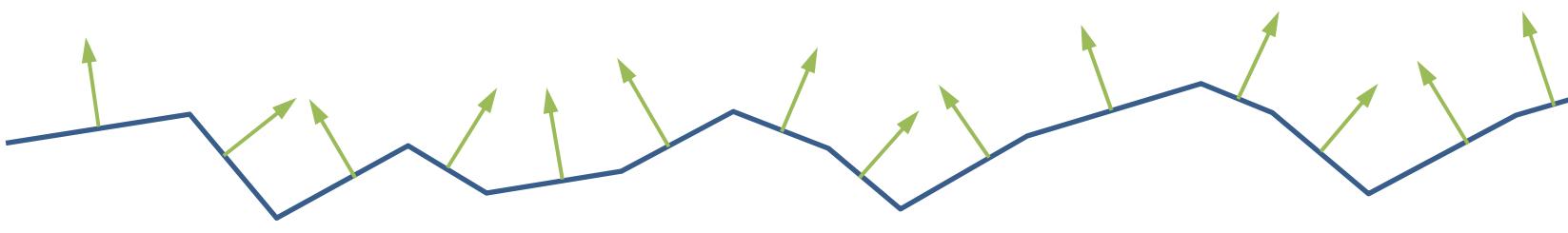
Model surface roughness instead



Images from [Akenine-Möller et al. 2008]

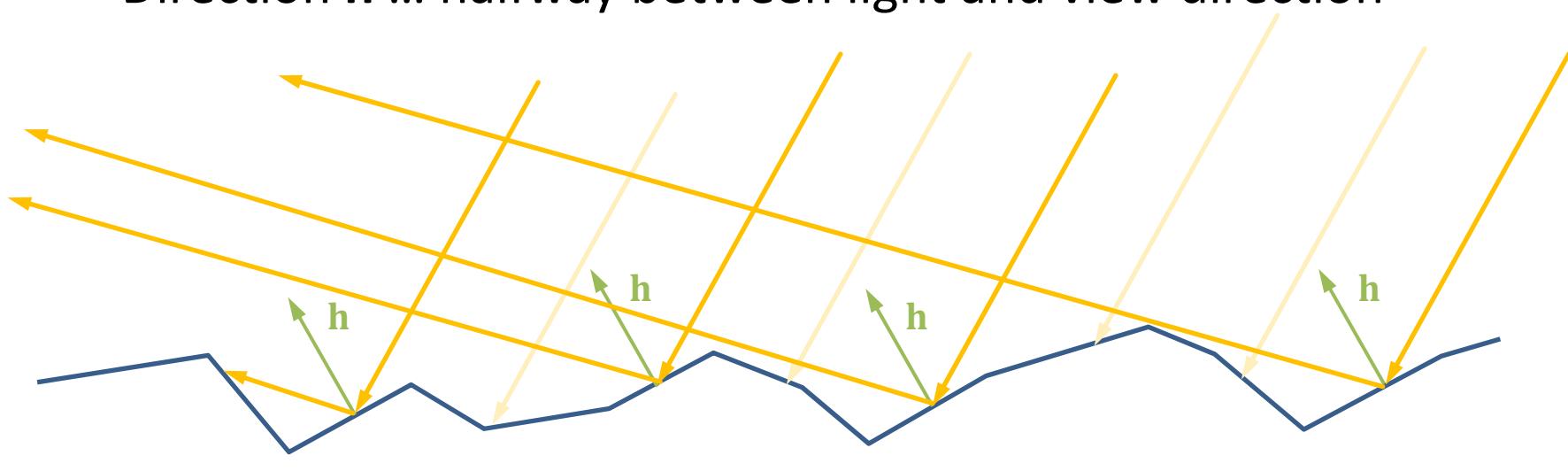
# Microfacets

- 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

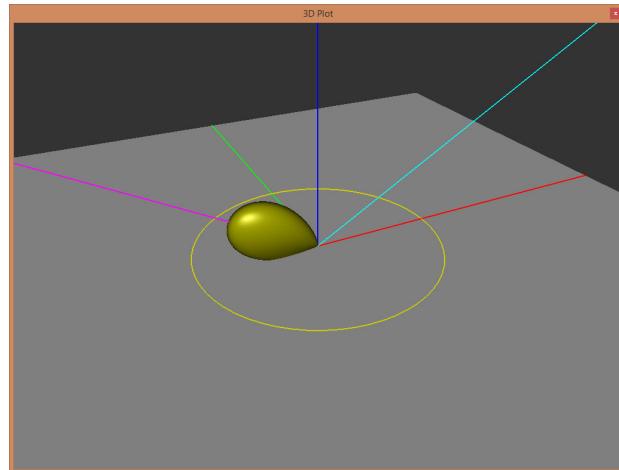
$$D(\mathbf{n}, \mathbf{h}, m) = \frac{1}{4m^2(\mathbf{n} \cdot \mathbf{h})^4} \exp\left(\frac{(\mathbf{n} \cdot \mathbf{h})^2 - 1}{m^2(\mathbf{n} \cdot \mathbf{h})^2}\right)$$

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

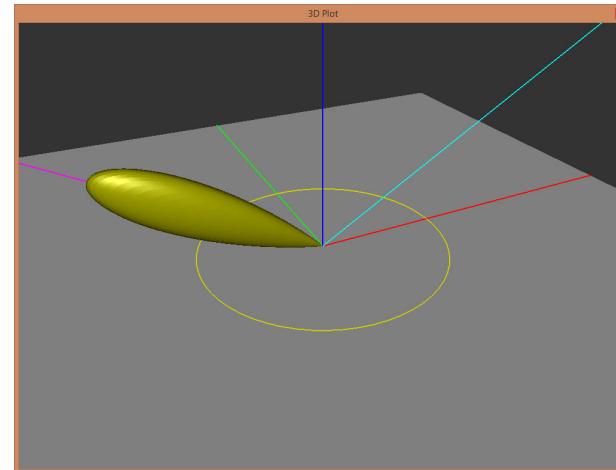
[Beckmann, Spizzichino 1963]

# Beckmann Distribution

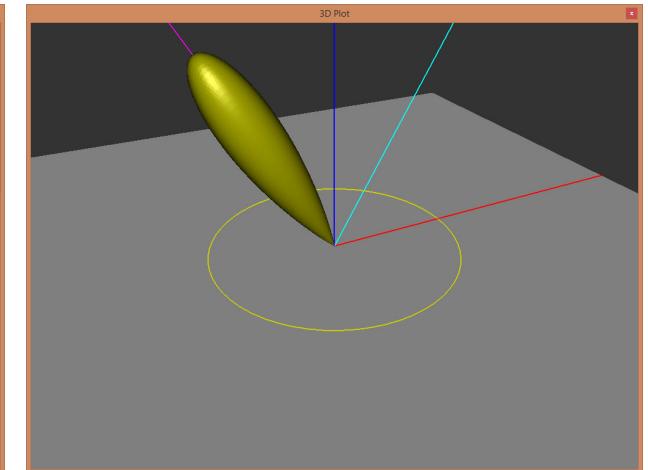
$\theta_i = 60^\circ, m = 0.24$



$\theta_i = 60^\circ, m = 0.069$



$\theta_i = 30^\circ, m = 0.069$



# 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

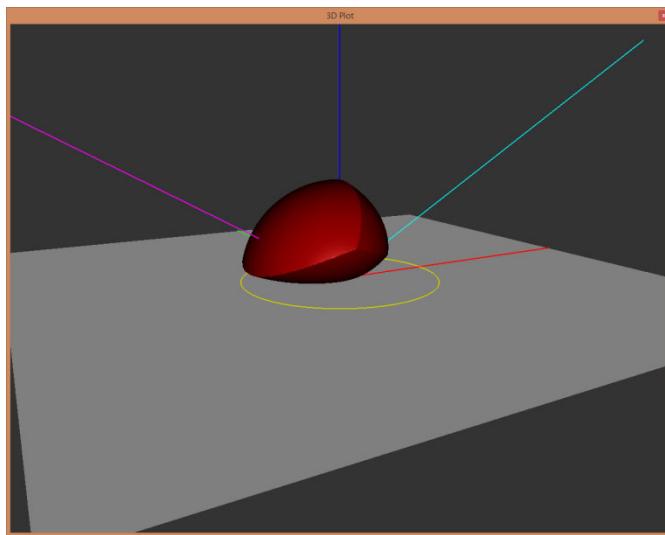
$$G(\mathbf{l}, \mathbf{v}) = \min\left(1, \frac{2 \cos \theta_h \cos \theta_o}{\cos \alpha_h}, \frac{2 \cos \theta_h \cos \theta_i}{\cos \alpha_h}\right)$$

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

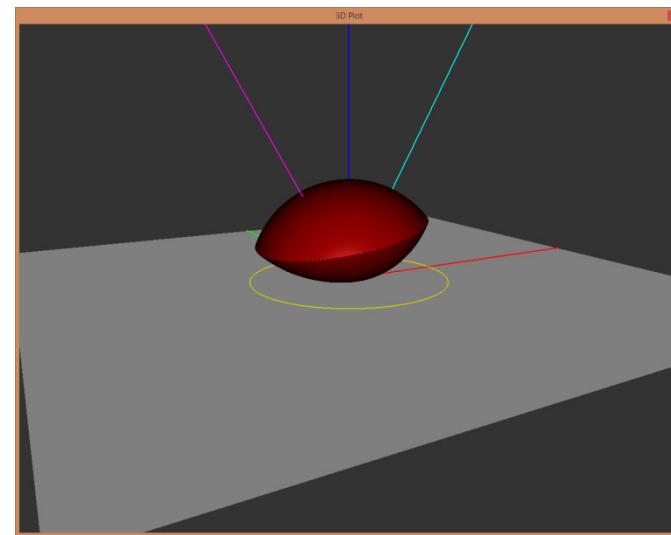
[Torrance, Sparrow 1967]

# Torrance-Sparrow Geometric Attenuation

$$\theta_i = 60^\circ$$



$$\theta_i = 30^\circ$$



# Cook-Torrance Model

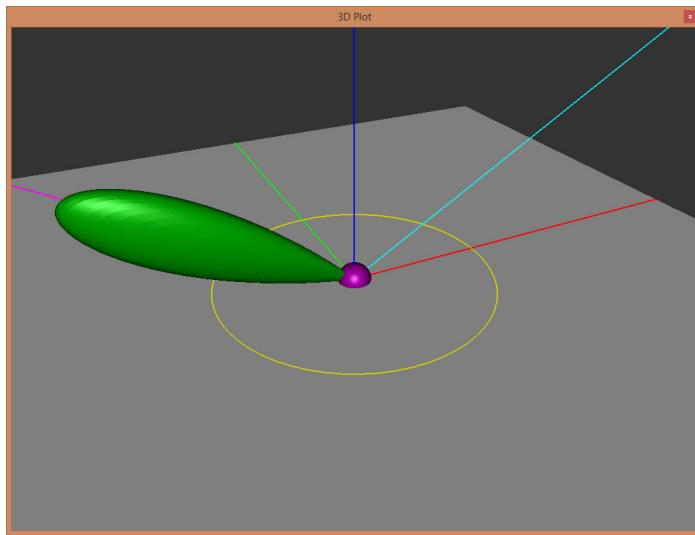
Popular shading model which is putting all these components together

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

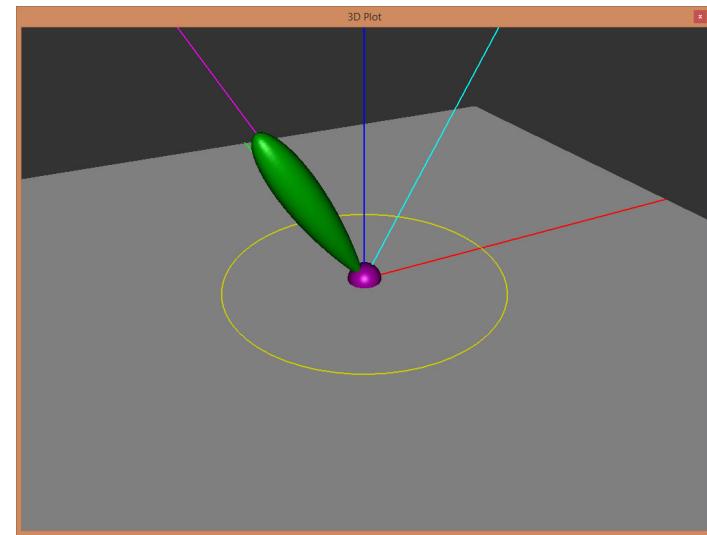
Microfacet distribution  
 (accounts for surface roughness)      Fresnel  
 reflectance      Geometric attenuation  
 (amount of self-shadowing)

# Cook-Torrance BRDF 1

$$\theta_i = 60^\circ, m = 0.069$$

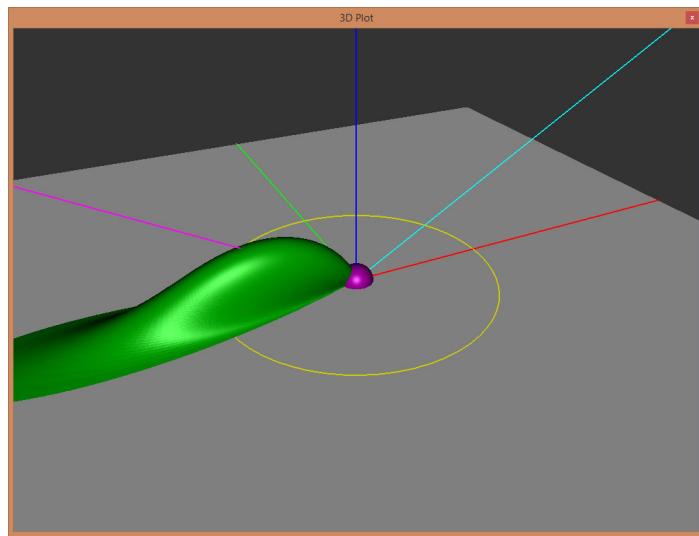


$$\theta_i = 30^\circ, m = 0.069$$

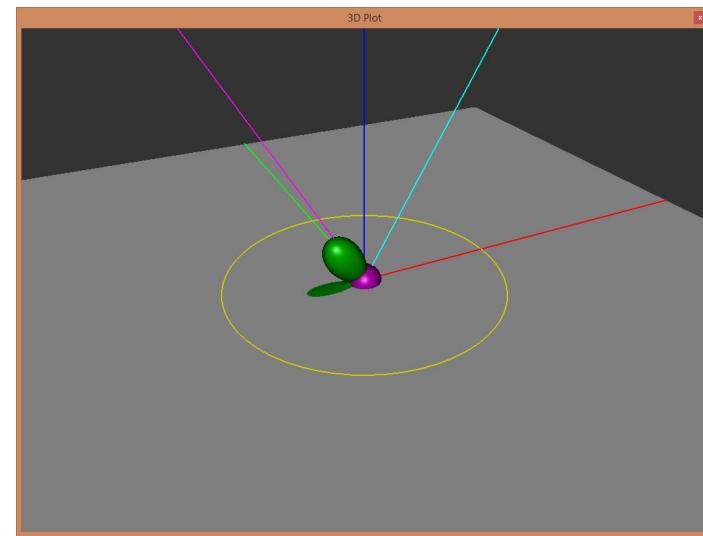


# Cook-Torrance BRDF 2

$\theta_i = 60^\circ, m = 0.24$



$\theta_i = 30^\circ, m = 0.24$



# Examples

Phong diffuse



Phong specular



Cook-Torrance



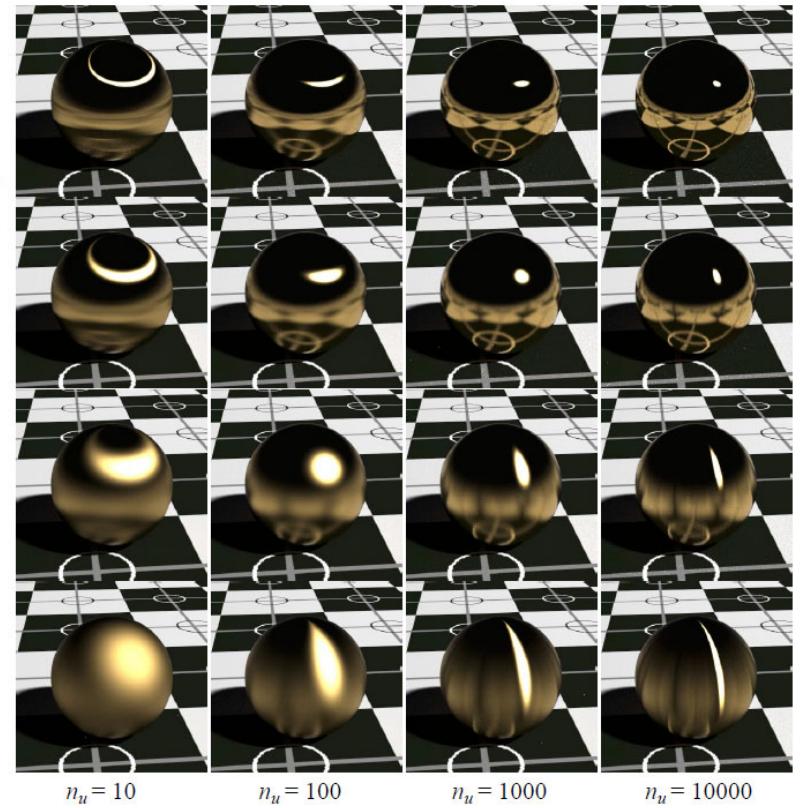
<https://www.youtube.com/watch?v=iVOfKIVtuVg>

# Anisotropic Reflection

- Viewing direction considered
  - Not just inclination
- Example
  - Brushed metal

[Ashikhmin, Shirley 2000]  $n_v = 10$

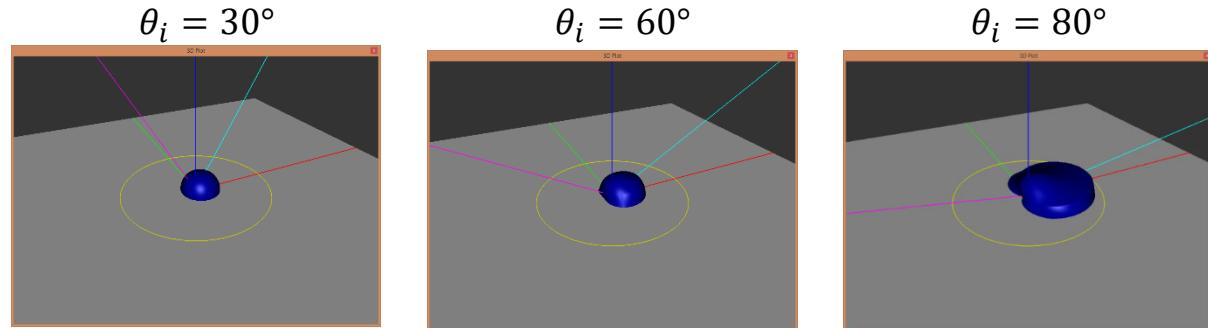
Shading Models



39

# Retro-Reflection

- Deep cavities on surface reflect light back to light source
- Brighter at steep angles (near silhouette of object)
- Example:  
pottery



Photograph



Lambertian



Oren-Nayar



[Oren, Nayar 1994]

# Implementation in GLSL

- BRDF:

$$f(\mathbf{l}, \mathbf{v}) = \frac{dL_o(\mathbf{v})}{dE_i(\mathbf{l})}$$

outgoing radiance

- Irradiance from light source:

$$E_L = \frac{I_L}{r^2}$$

light source intensity

irradiance incident on surface

incoming irradiance

squared distance to light source

- Radiance towards viewer:

$$L_o(\mathbf{v}) = \sum_{k=1}^n f(\mathbf{l}_k, \mathbf{v}) \cdot E_{L_k} \cos \theta_{i_k}$$

sum up contributions of each light source

Shading Models

# Example: Blinn-Phong Vertex Shader

```
1 #version 330
2
3 uniform mat4x4 PV; // view projection matrix
4
5 layout(location = 0) in vec3 vertex_position;
6 layout(location = 1) in vec3 vertex_normal;
7
8 out vec3 p;
9 out vec3 normal;
10
11 void main()
12 {
13     gl_Position = PV * vec4(vertex_position, 1.0f);
14     p = vertex_position;
15     normal = vertex_normal;
16 }
```

# Example: Phong Fragment Shader

```

1 #version 330
2
3 uniform vec3 camera_position;
4 uniform vec3 light_direction;
5 uniform vec3 B_L;
6 uniform vec3 c_d;
7 uniform vec3 c_s;
8 uniform float m;
9
10 in vec3 p;
11 in vec3 normal;
12
13 layout(location=0) out vec4 fragment_color;
14
15 void main()
16 {
17     vec3 n = normalize(normal);
18     vec3 v = normalize(camera_position - p);
19     vec3 l = -light_direction;
20     vec3 r = 2.0f * n * dot(n, l) - l;
21
22     float lambert = max(dot(n, l), 0.0f);
23     float specular = pow(max(dot(v, r), 0.0f), m);
24
25     fragment_color.rgb = (c_d * lambert + c_s * specular) * B_L;
26     fragment_color.a = 1.0f;
27 }
```

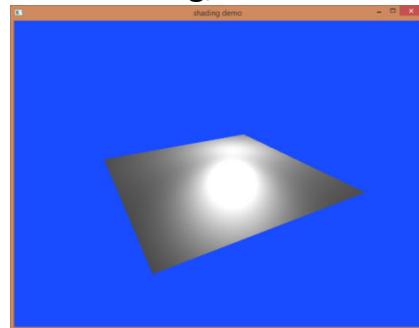
# Example: Blinn-Phong Fragment Shader

```

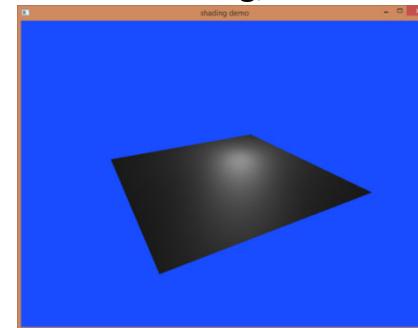
1 #version 330
2
3 const float pi = 3.14159265358979f;
4
5 uniform vec3 camera_position;
6 uniform vec3 light_direction;
7 uniform vec3 I_L;
8 uniform vec3 c_d;
9 uniform vec3 c_s;
10 uniform float m;
11
12 in vec3 p;
13 in vec3 normal;
14
15 layout(location=0) out vec4 fragment_color;
16
17 void main()
18 {
19     vec3 n = normalize(normal); // renormalize interpolated normal
20     vec3 v = normalize(camera_position - p);
21     vec3 l = -light_direction;
22     vec3 h = normalize(v + l);
23     float lambert = max(dot(n, l), 0.0f);
24     float specular = (m + 8) / (8 * pi) * pow(max(dot(n, h), 0.0f), m);
25     fragment_color.rgb = (c_d / pi + c_s * specular) * lambert * I_L;
26     fragment_color.a = 1.0f;
27 }
```

# Results

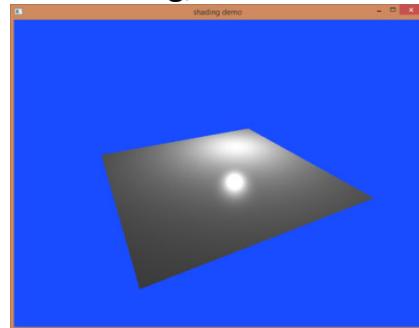
Phong,  $m = 8$



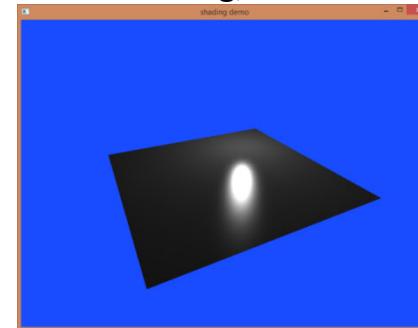
Blinn-Phong,  $m = 8$



Phong,  $m = 100$

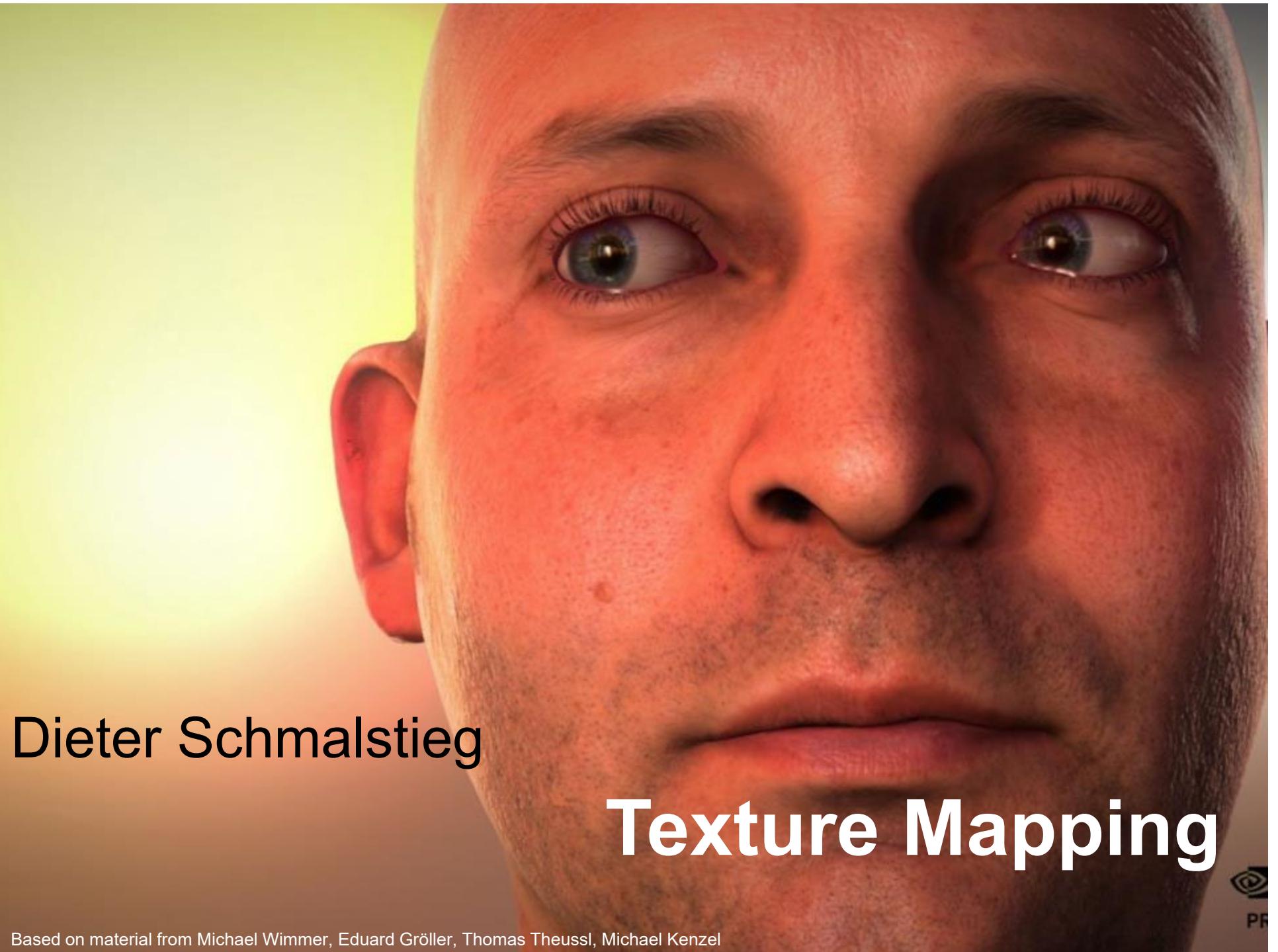


Blinn-Phong,  $m = 100$



# References

- [Akenine-Möller et al. 2008] Tomas Akenine-Möller, Eric Haines, Naty Hoffman (2008). Real-Time Rendering. 3rd ed. AK Peters.
- [Lengyel 2004] Eric Lengyel (2004). Mathematics for 3D Game Programming & Computer Graphics. 2nd ed. Charles River Media.
- [Phong 1975] Bui Tuong Phong (1975). “Illumination for computer generated pictures”. *Communications of the ACM*, vol. 18, no. 5.
- [Blinn 1977] James F. Blinn (1977). “Models of light reflection for computer synthesized pictures”. Proc. 4th annual conference on computer graphics and interactive techniques: 192–198.
- [Schlick 1994] Christophe Schlick. (1994). “An Inexpensive BRDF Model for Physically-based Rendering”. Computer Graphics Forum, vol. 13, no. 3: 233–246.
- [Torrance, Sparrow 1967] K. E. Torrance, E. M. Sparrow (1967). “Theory for Off-specular Reflection From Roughened Surfaces”. *Journal of the Optical Society of America*, vol. 57, no. 9: 32–41.
- [Beckmann, Spizzichino 1963] Petr Beckmann, André Spizzichino (1963). The scattering of electromagnetic waves from rough surfaces, Pergamon Press.
- [Ashikhmin, Shirley 2000] Michael Ashikhmin, Peter Shirley (2000). “An Anisotropic Phong BRDF Model”. *Journal of Graphics Tools*, vol. 7, no. 4: 61–68.
- [Oren, Nayar 1994] Michael Oren, Shree K. Nayar (1994). “Generalization of Lambert's reflectance model”. Proc. 21st annual conference on computer graphics and interactive techniques: 239–246.



Dieter Schmalstieg

# Texture Mapping

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

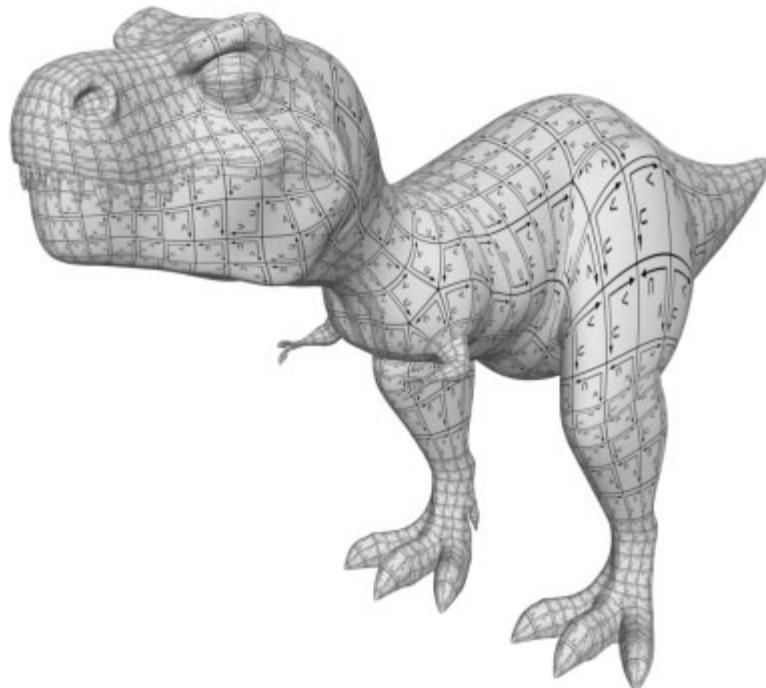


# Overview

- Basic texture mapping
  - Texture coordinate generation
  - Filtering
- Advanced texture mapping
  - Multitexture/multipass
  - Projective texturing
  - Environment mapping
  - Bump mapping
  - Per-pixel lighting

# Why Texturing?

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



Dieter Schmalstieg



Texture mapping

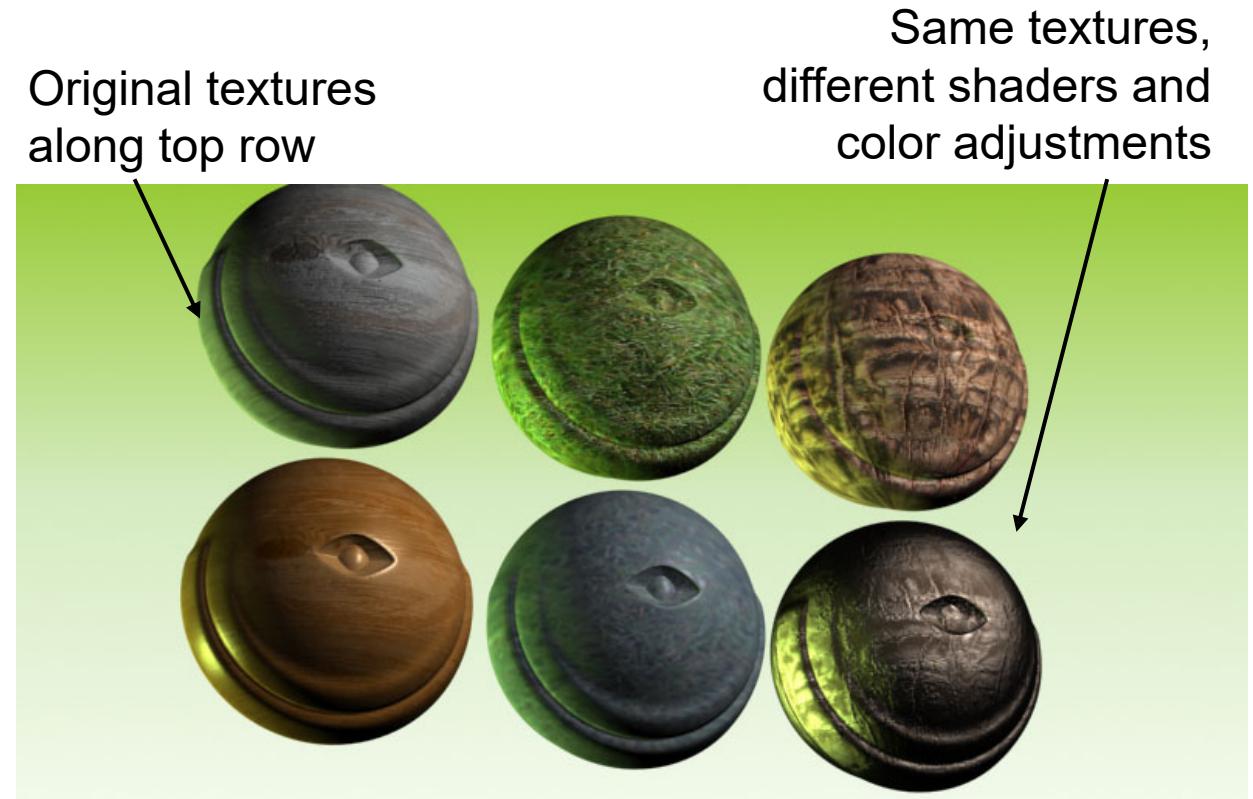
# What is a Texture?

- Provides look and feel of a surface
- Definition
  - A *function* that is evaluated for every *fragment* of a surface
  - Usually given as a 2D image
- Can hold arbitrary information
  - Texture data interpreted in fragment shader
  - Basis for most advanced rendering effects

# Textures as Patterns

Material properties simulation:

- Color
- Reflection
- Gloss
- Transparency
- Bumps

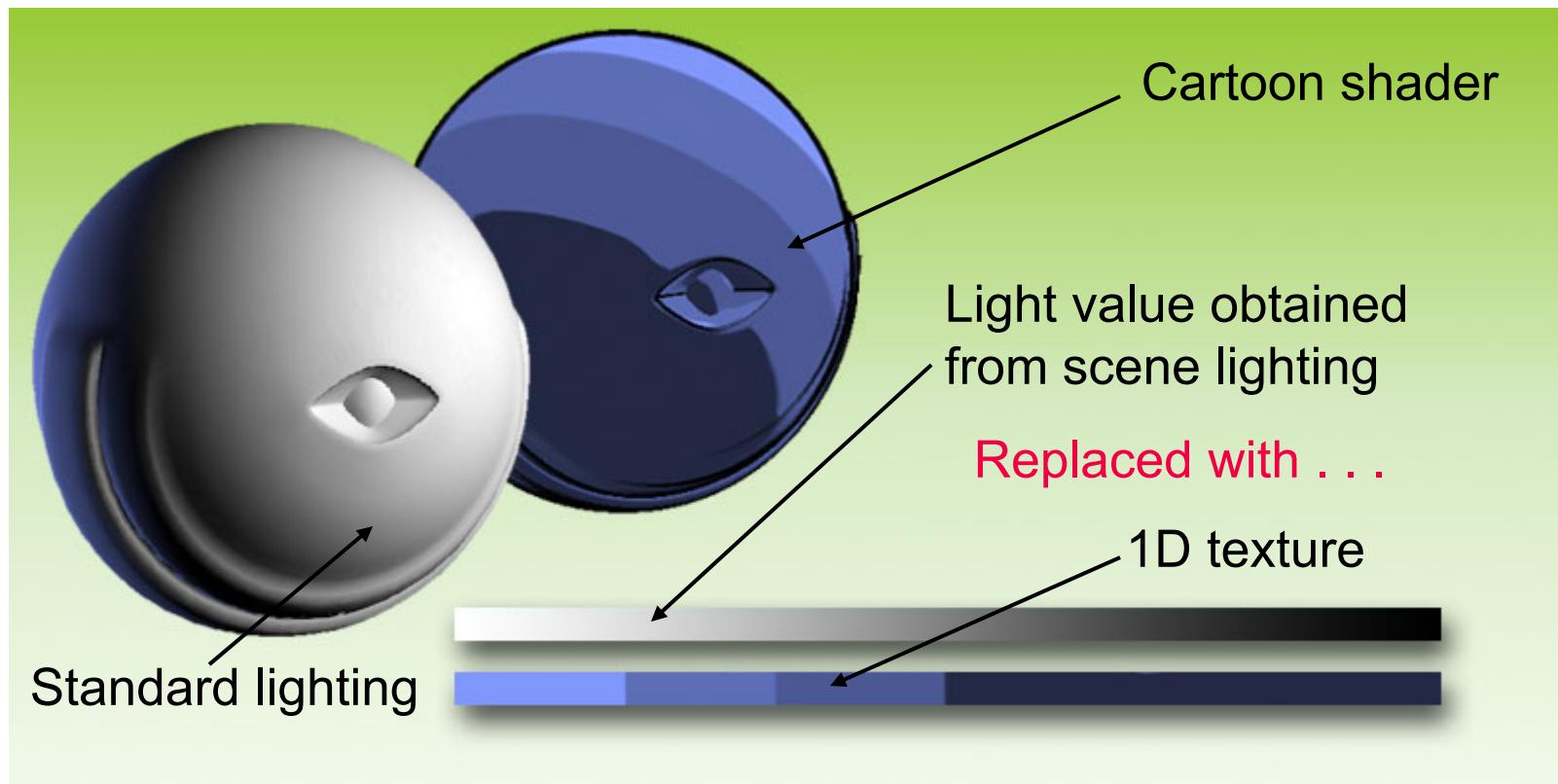


Textures are just color and value combinations, the same texture can be used in many different ways

# Texture Types

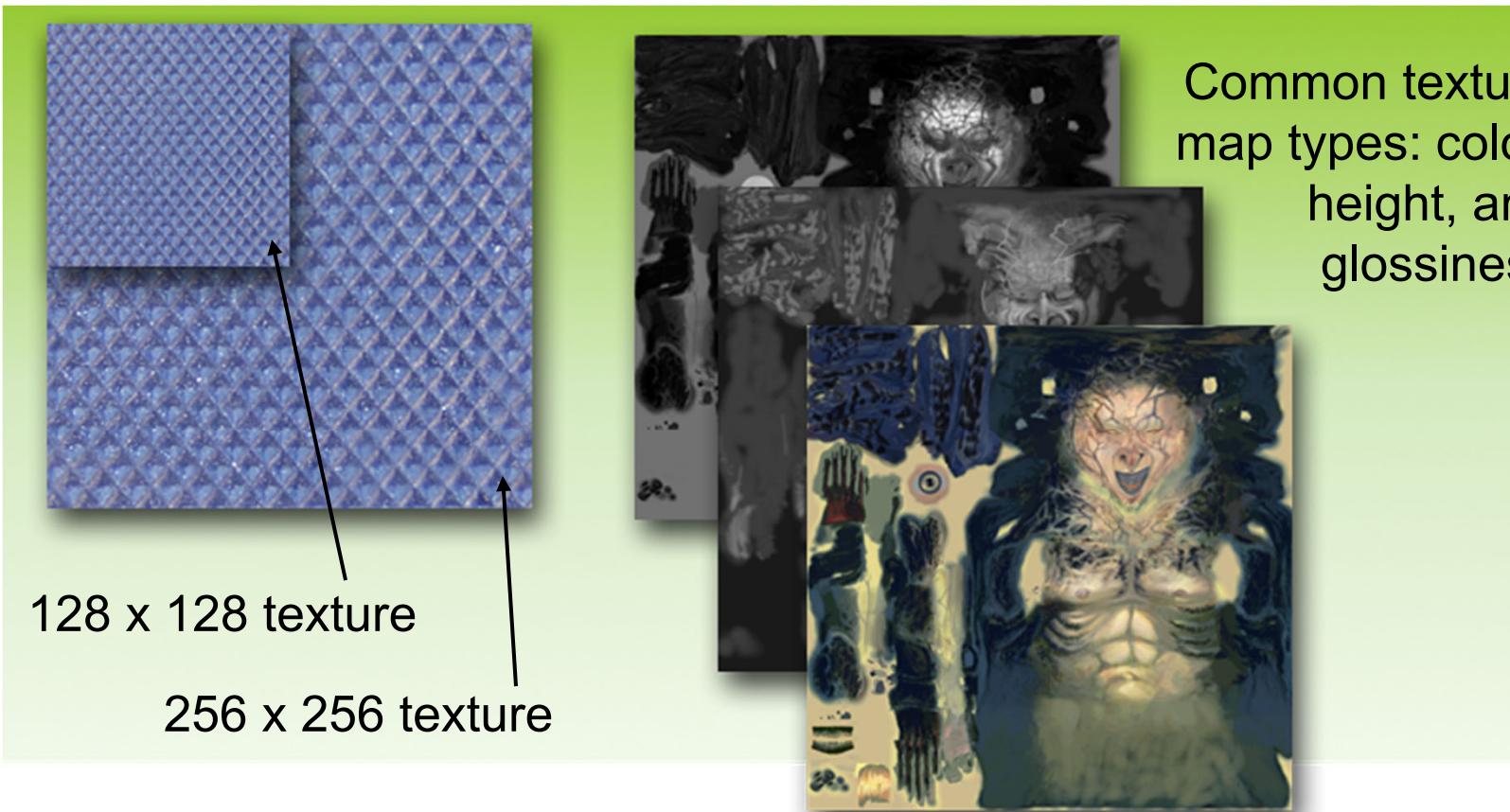
- One-dimensional functions
  - Parameter can have arbitrary domain (e.g., incident angle)
- Two-dimensional functions
  - Information is calculated for every  $(u,v)$
  - Many possibilities
- Raster images (“texels”)
  - Most often used method
  - Images from scanner or calculation
- Three-dimensional functions
  - Volume  $T(u,v,w)$

# 1D Textures



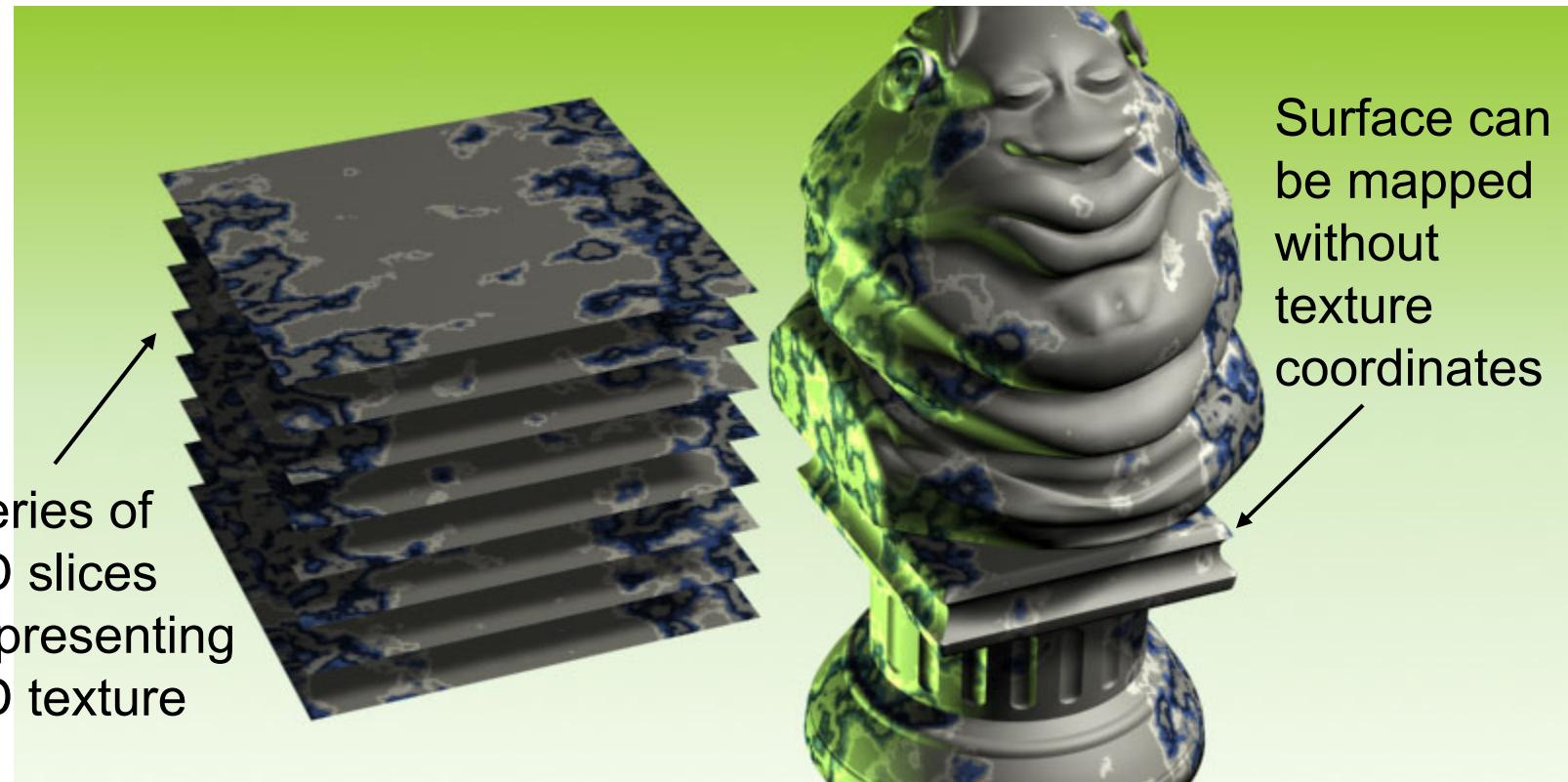
1D texture can be substituted for any linear value used in shader

# 2D Textures



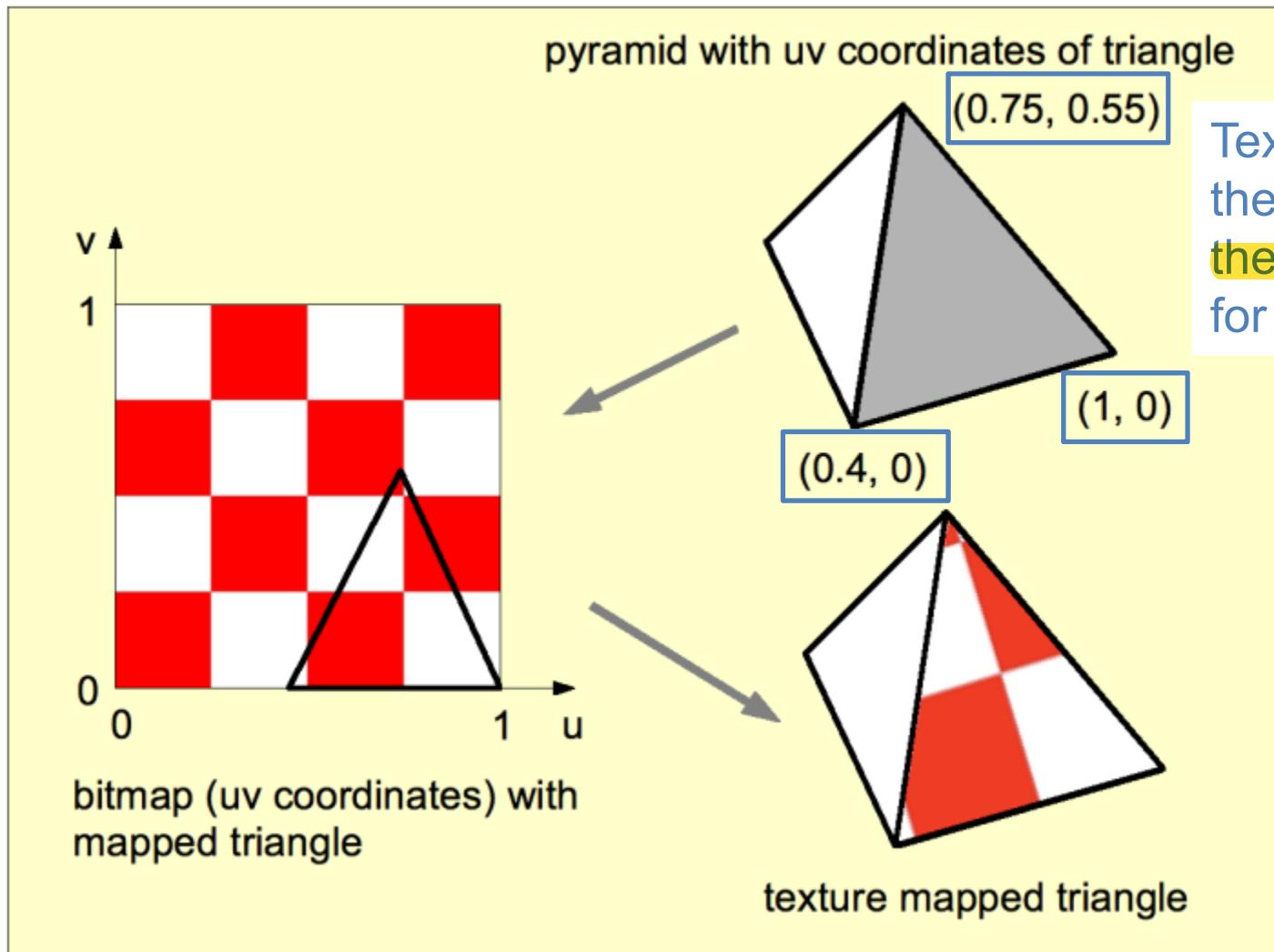
2D directly relate to surfaces

# 3D Textures



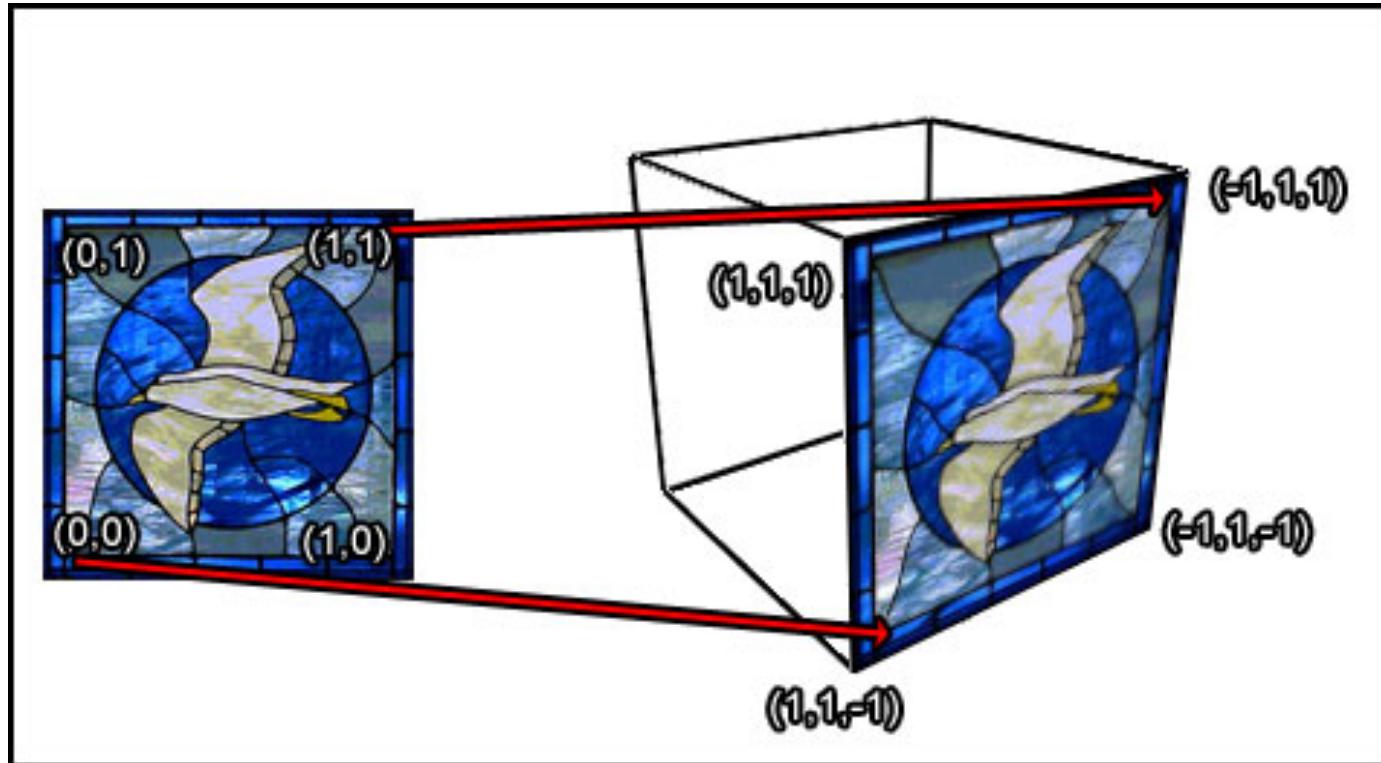
3D textures can represent procedural textures

# Texture Coordinates



# Parametrization

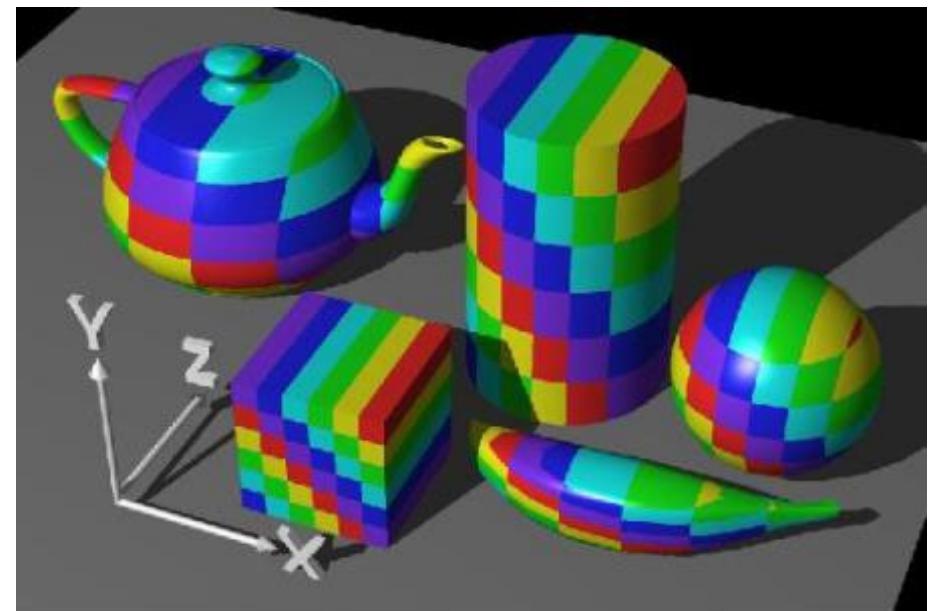
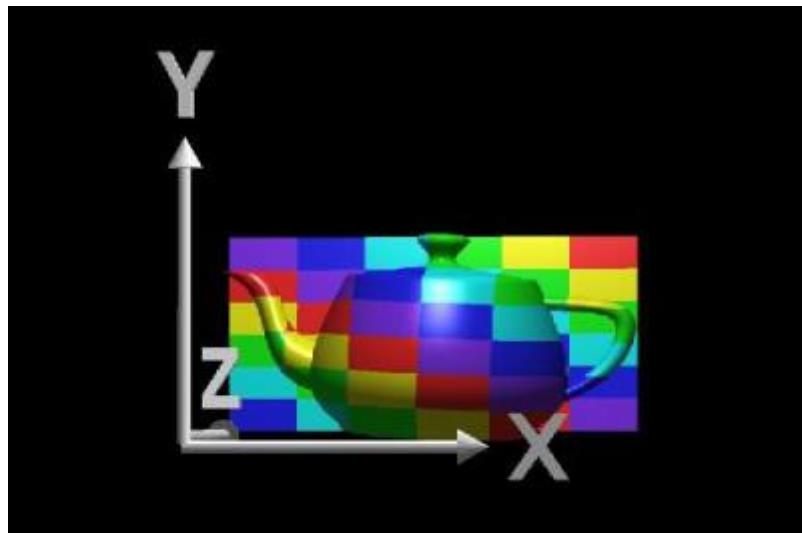
- How to do the mapping?



- Usually objects are not that simple

# Parametrization: Planar

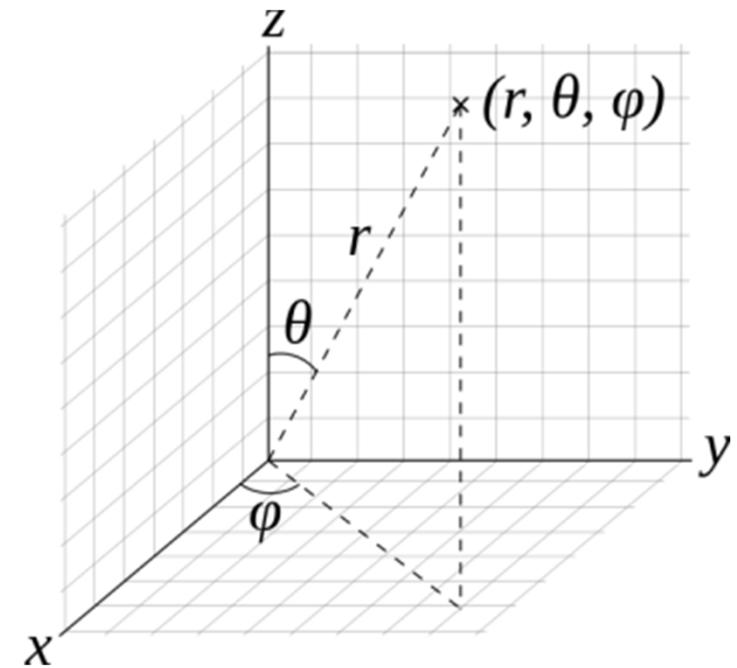
- Planar mapping: dump one of the coordinates



Only looks good from the front!

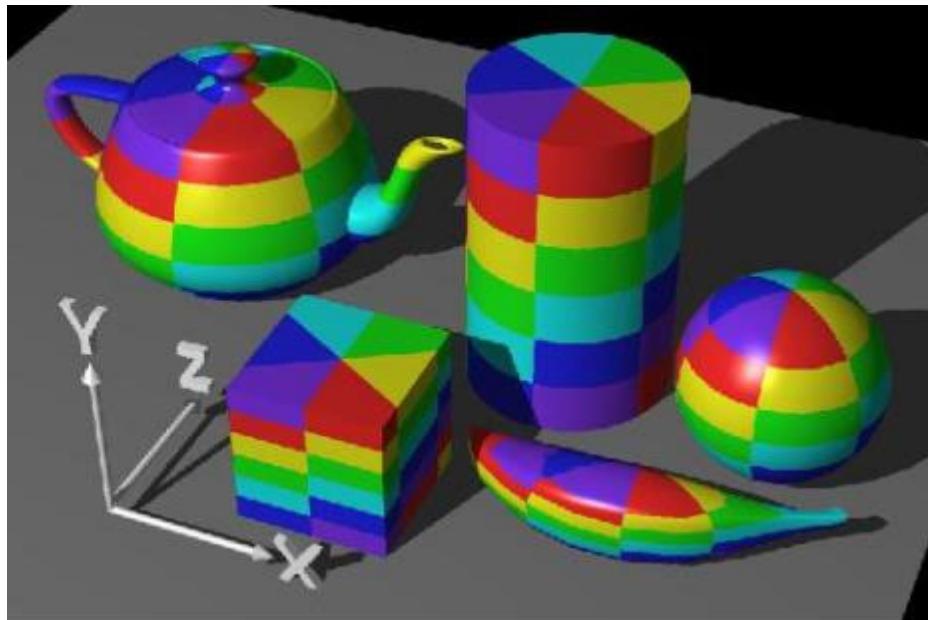
# Parametrization: Cylindrical

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

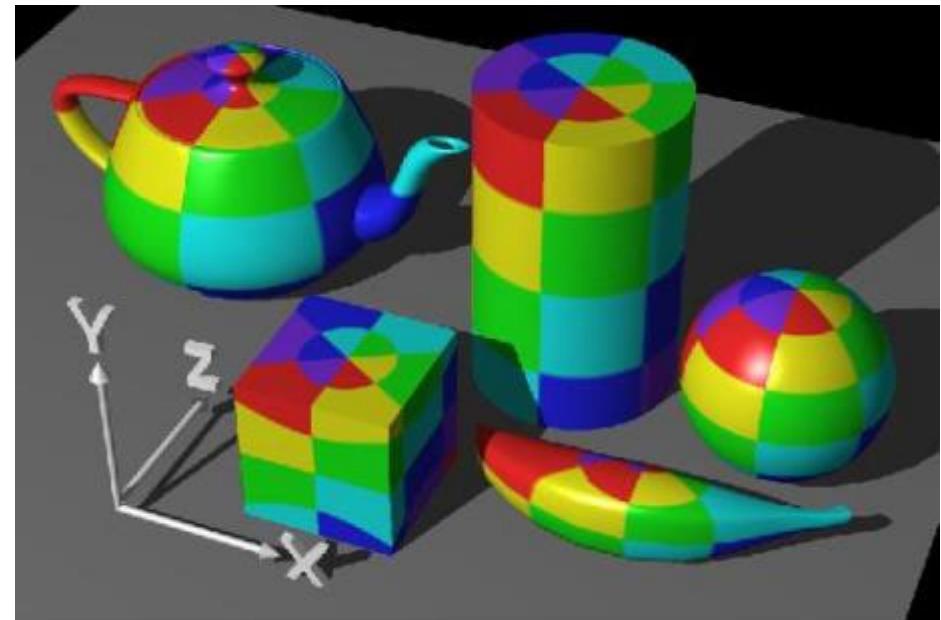


# Parametrization: Polar

- Cylindrical and spherical mapping



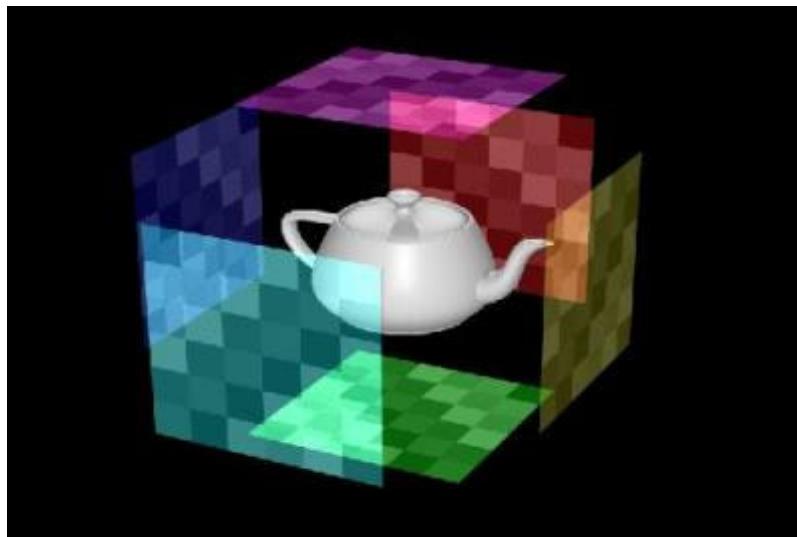
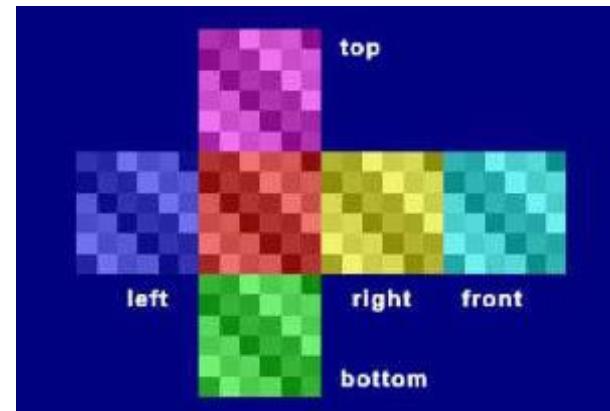
Cylindrical



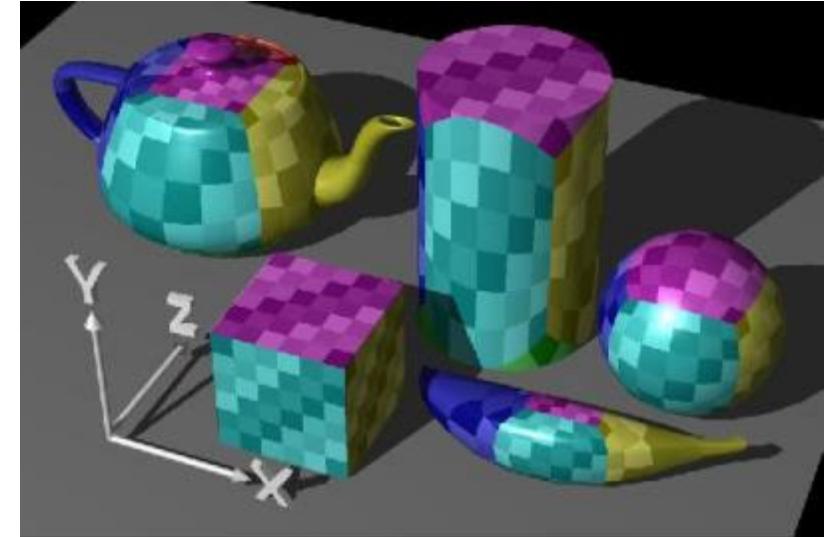
Spherical

# Parametrization: Box

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



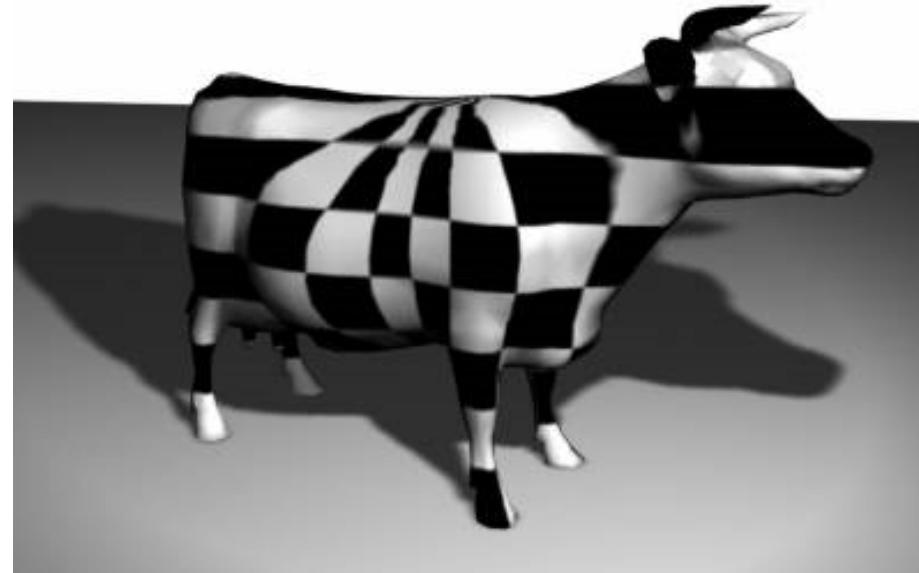
Dieter Schmalstieg



Texture mapping

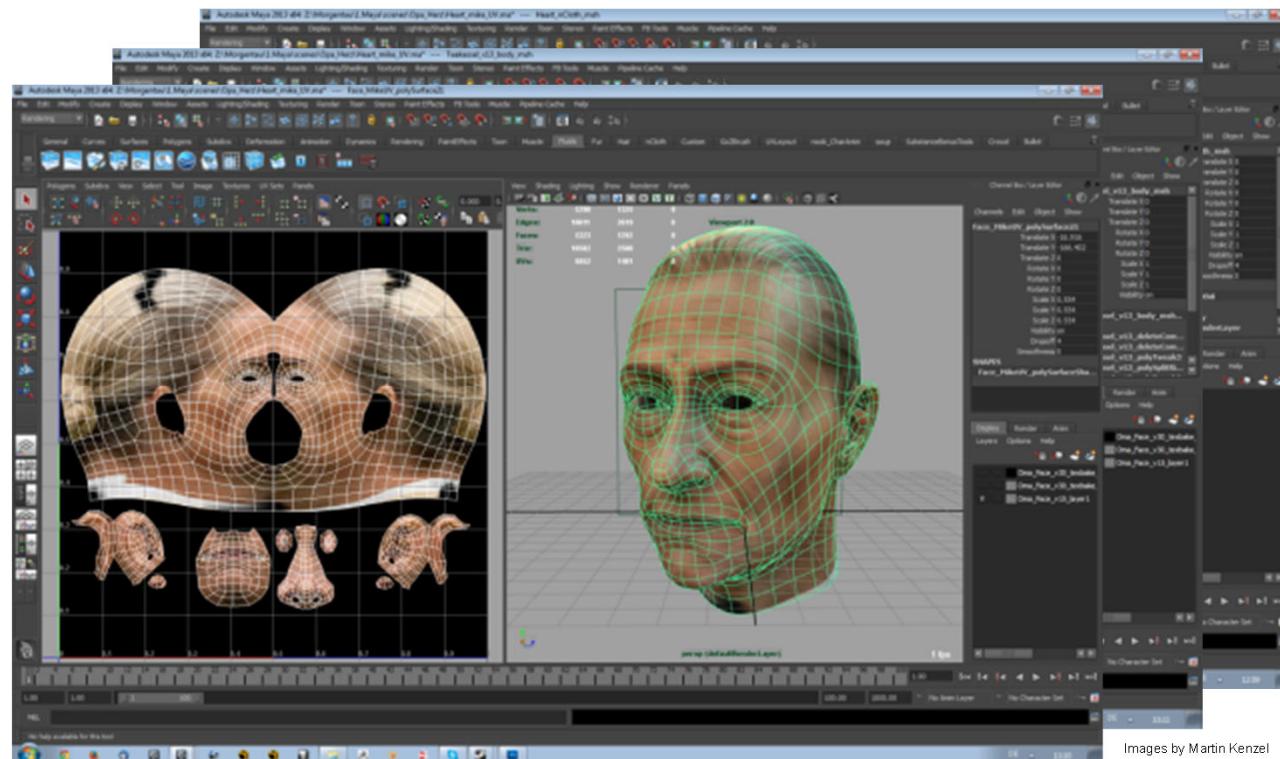
# Parametrization Problems

- All mappings have distortions and singularities
- Often they need to be fixed manually (CAD software)



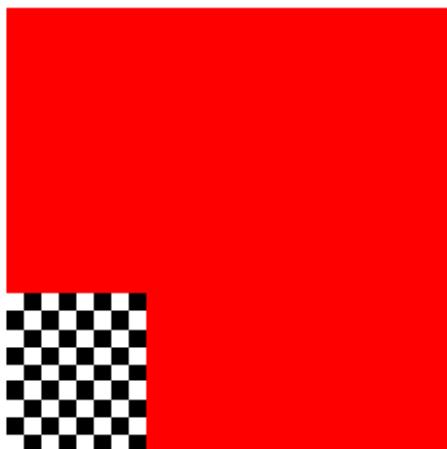
# Manual Parametrization

- Manual mapping using CAD Software
  - “Unwrapping”



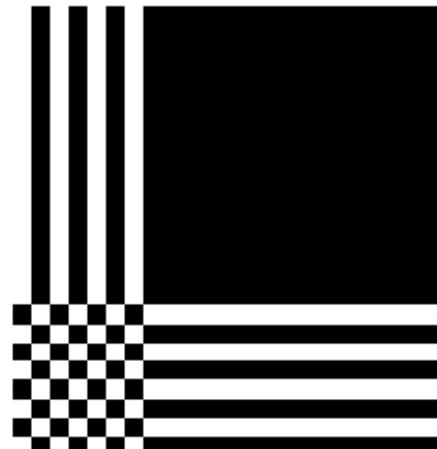
# Texture Adressing

- What happens outside  $[0,1]$ ?
- Border, repeat, clamp, mirror
- Also called texture addressing modes



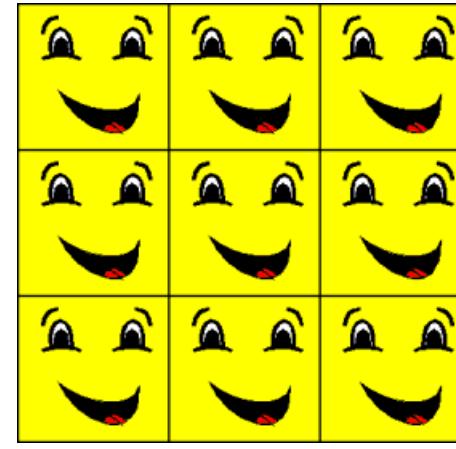
Texture with red border applied to primitive

**Static color**

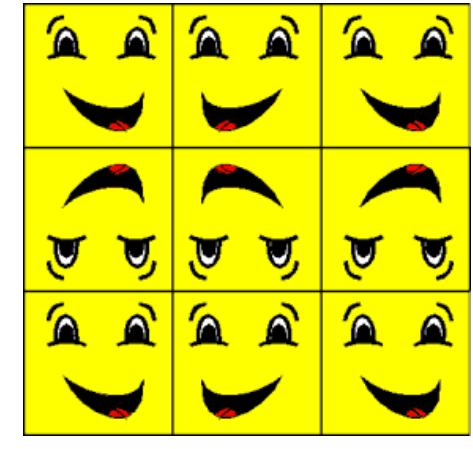


Clamped texture applied to primitive

**Clamped**



**Repeated**



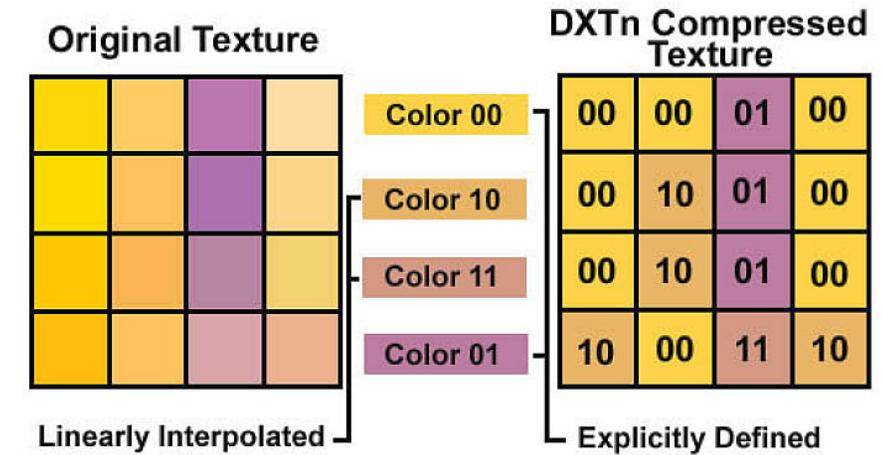
**Mirrored**

# Texture Animation

- Texture Matrix
  - Can specify an arbitrary 4x4 Matrix, each frame!
  - `glMatrixMode(GL_TEXTURE);`
  - There is also a texture matrix stack!
  - Possibilities: moving water or sky
- Custom shader programs
  - Can use video as texture
  - Allows arbitrary texture animations

# Texture Compression

- S3TC texture compression
  - Represent 4x4 texel block by two 16bit colors (RGB 5/6/5bit)
  - Store 2 bits per texel
- Uncompress
  - Create 2 additional colors between  $c_1$  and  $c_2$
  - Use 2 bits to index into color map
- Compression ratio 4:1 or 6:1



using color values based on the graphics chips color palette. It reduces the size of texture by reducing the number of colors contained within the texture without compromising quality

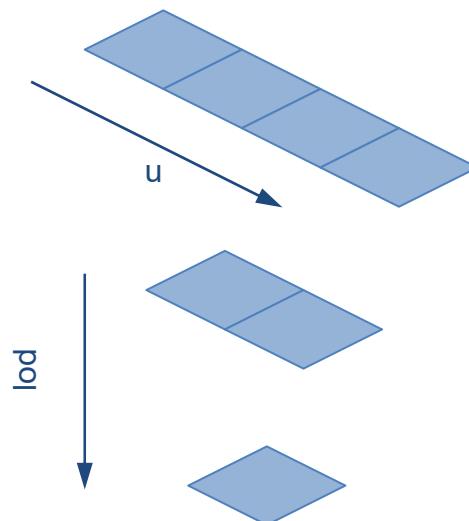
# Texture Representation on GPU

- Texture Image = **array of *texels***
  - Certain dimensionality
  - Certain format (e. g., GL\_RGBA8)
- Texture Objects
  - **Group of one or more texture images**
  - Images can represent mip levels, array slices...
- Sampler Objects
  - Configure texture sampling (interpolation mode...)

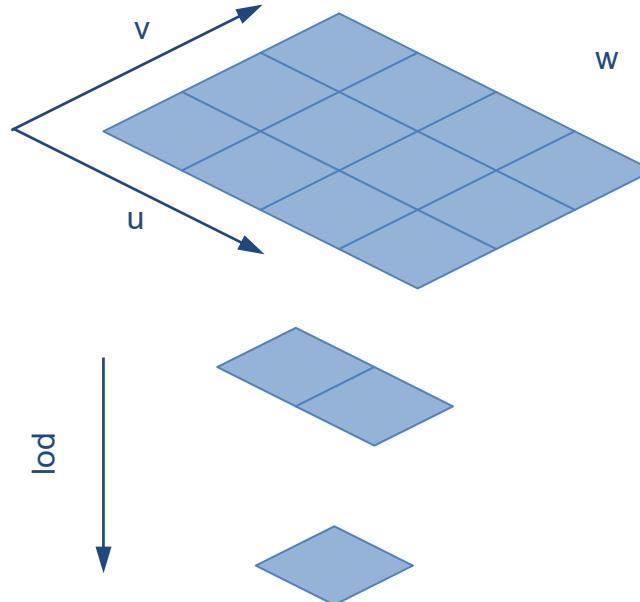
Ein Sampler Object ist ein Objekt, das die Einstellungen für die Art und Weise speichert, wie eine Textur auf ein 3D-Modell angewendet wird. Dazu gehören Einstellungen wie die Filterung der Textur (z.B. skalieren oder glätten).

# Texture Dimensions

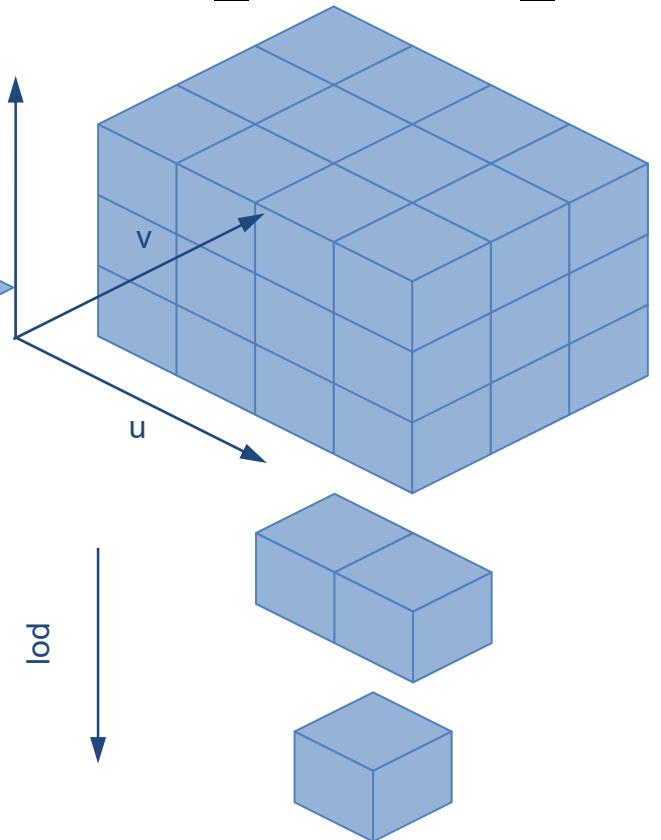
GL\_TEXTURE\_1D



GL\_TEXTURE\_2D



GL\_TEXTURE\_3D



# Texture Format and Specification

## Format

- Numerical format
  - Integer, float, double
- Bits per number
  - 8, 16, 32, 64
- Channels
  - Red, green, blue, alpha, depth, stencil

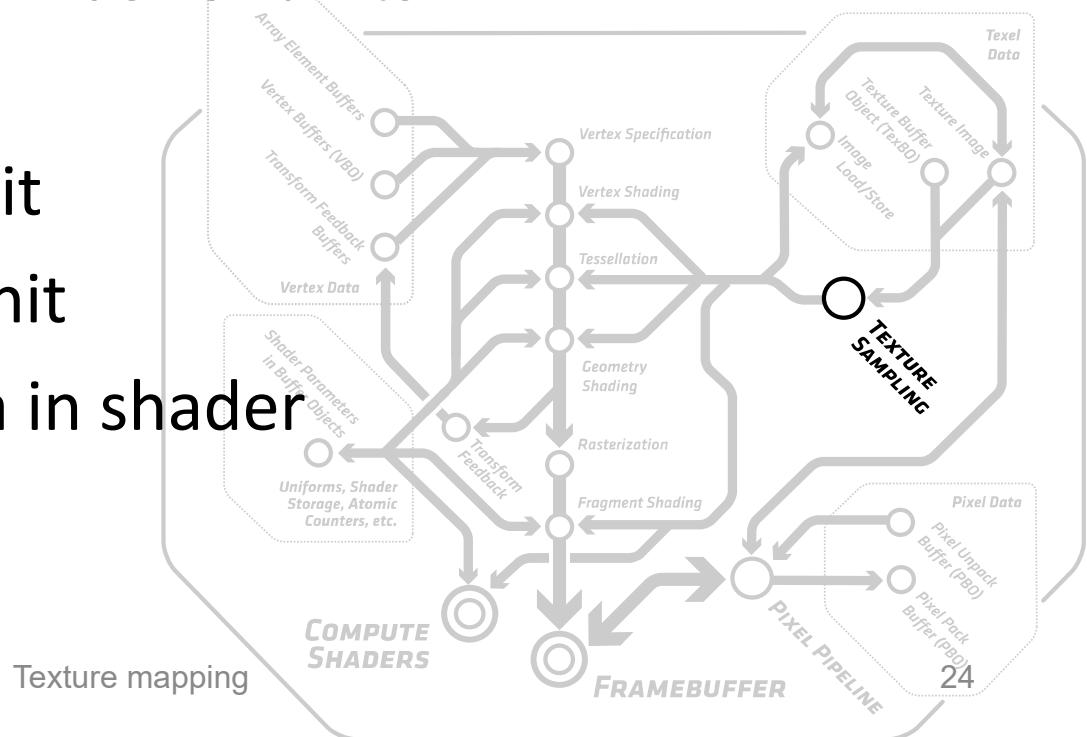
## Specification

- Texture id
- Dimension
- External format
- Internal format
- Width, height
- Pointer to raw data

# Sampler Objects

- Samplers configure texture units
  - Filtering, address mode, ...
- How texture units work
  - There is a limited number of units
  - Activate the unit
  - Bind texture to a unit
  - Bind sampler to a unit
  - Sample texture data in shader

Zusammen arbeiten Sampler Objects und Texture Units, um die Texturen auf 3D-Modelle anzuwenden und dafür zu sorgen, dass die dargestellten Modelle realistisch und ansprechend aussehen.

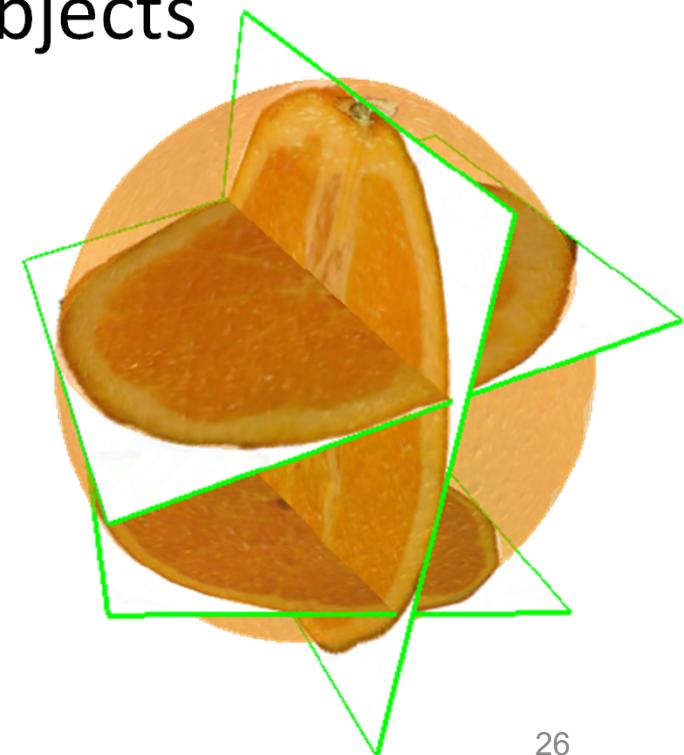


# 2D Texture Example: Fragment Shader

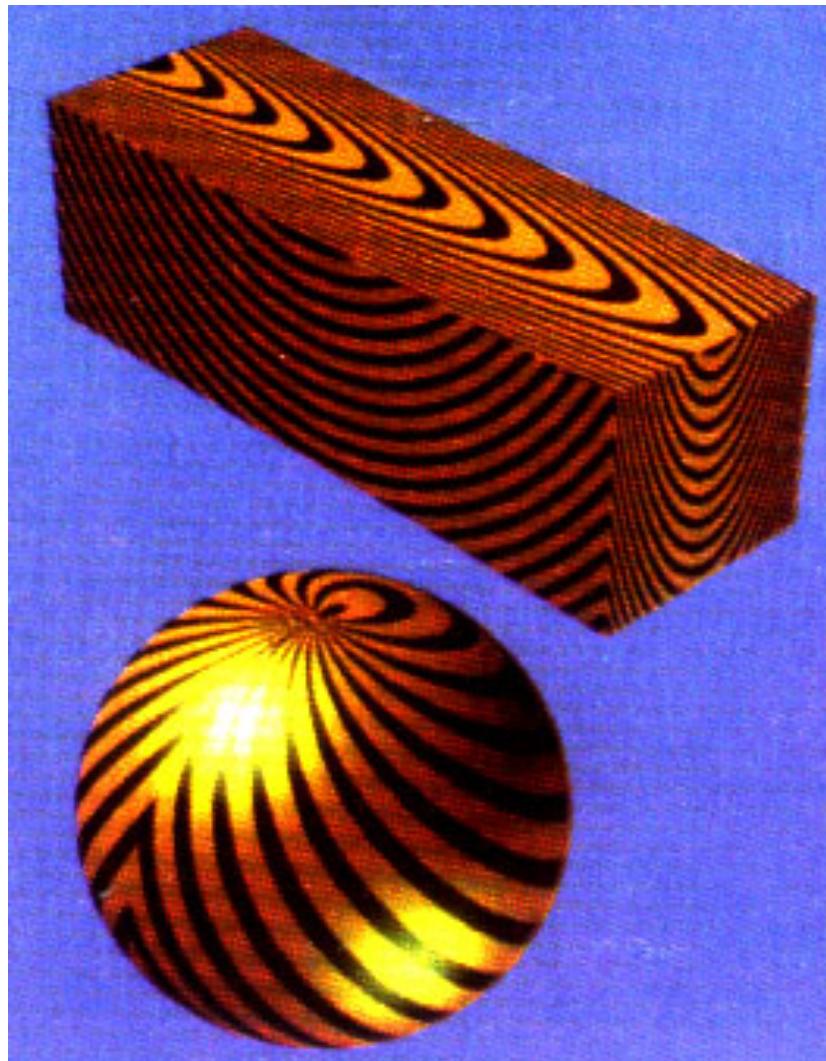
```
1 #version 330
2
3 uniform sampler2D my_texture;
4
5 in vec2 tex_coords;
6
7 layout(location = 0) out vec4 fragment_color;
8
9 void main()
10 {
11     fragment_color = texture(my_texture, tex_coords);
12 }
```

# 3D Texture Mapping

- Often referred to as *solid texturing*
- Directly map object space  $T(x,y,z)$  to  $(u,v,w)$
- Possible to slice/carve/open objects

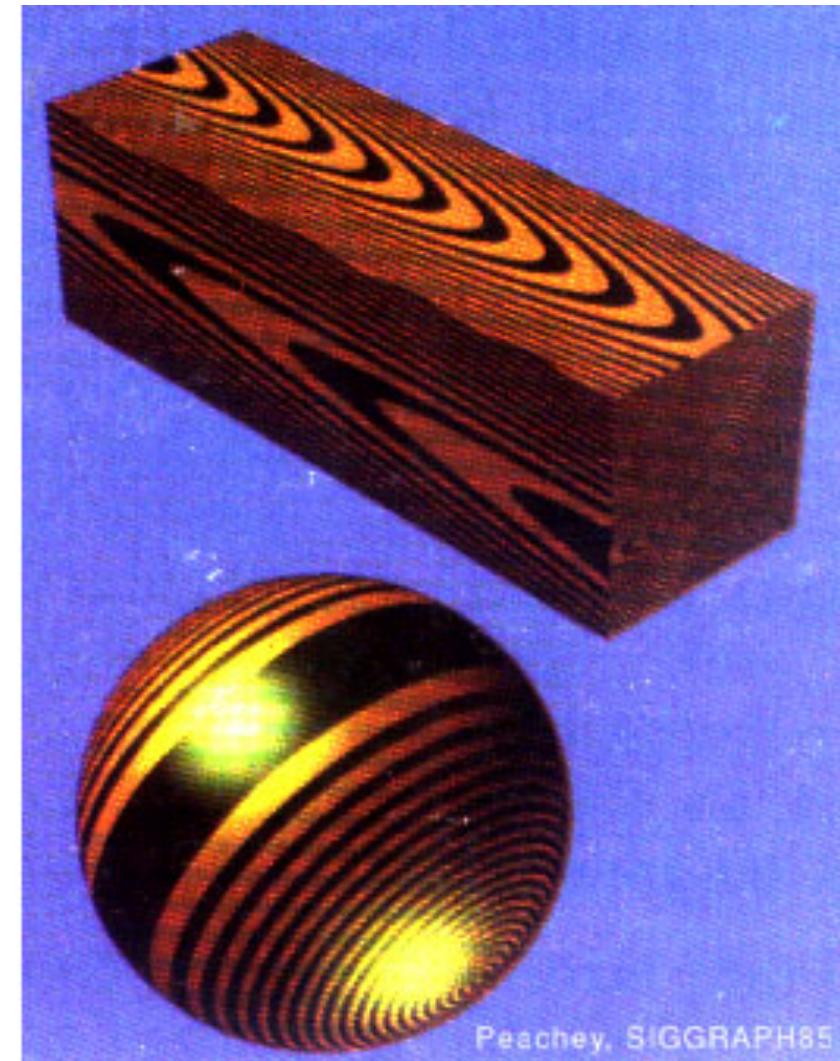


# Mapping vs Solid Texturing



Dieter Schmalstieg

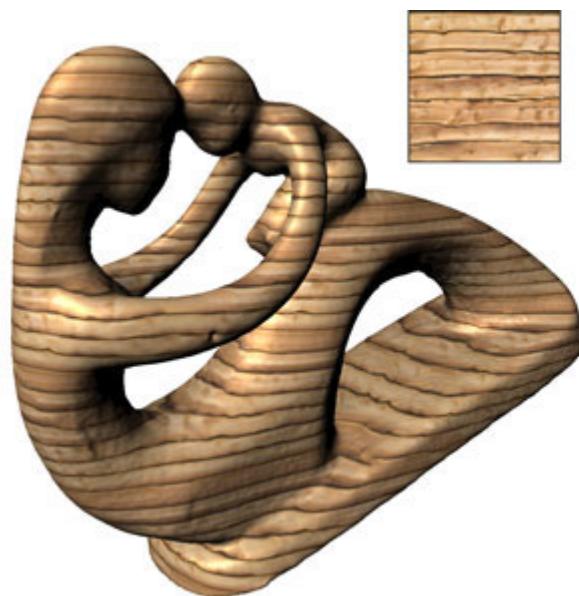
Texture mapping



Peachey, SIGGRAPH85

# Solid Texturing Examples

- Carved objects: wood, marble, stone...



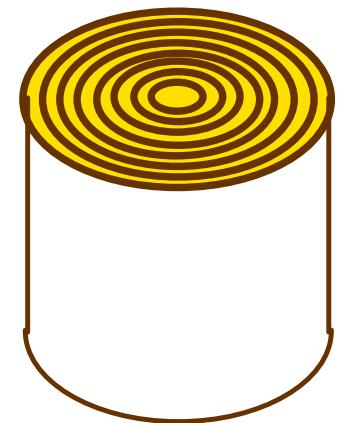
# Procedural Solid Textures

- **3D functions**

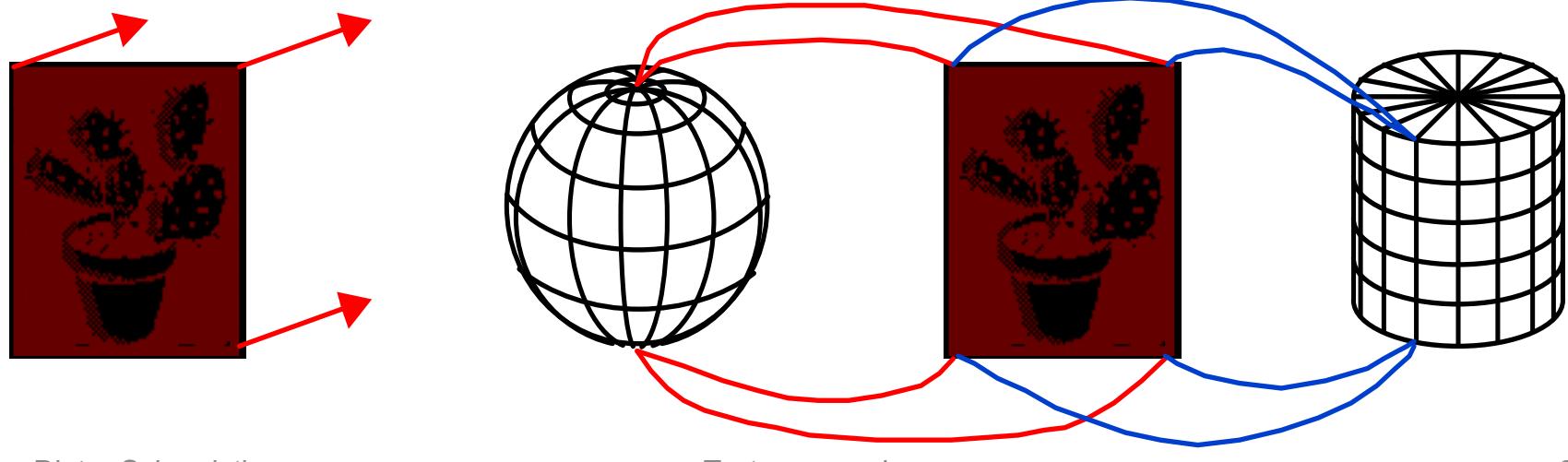
e.g.,  $0 \leq x^2+y^2 < 1 \Rightarrow$  light brown

$1 \leq x^2+y^2 < 2 \Rightarrow$  dark brown

$2 \leq x^2+y^2 < 3 \Rightarrow$  yellow etc.



- **Projections of 2D data**



# Volumetric Textures

- 3D „voxel“ array of texture values
- Explicit, sampled function
- E.g., from medical scanners (CT, MRT, ...)
  - + Works in real time
  - Lots of memory needed...



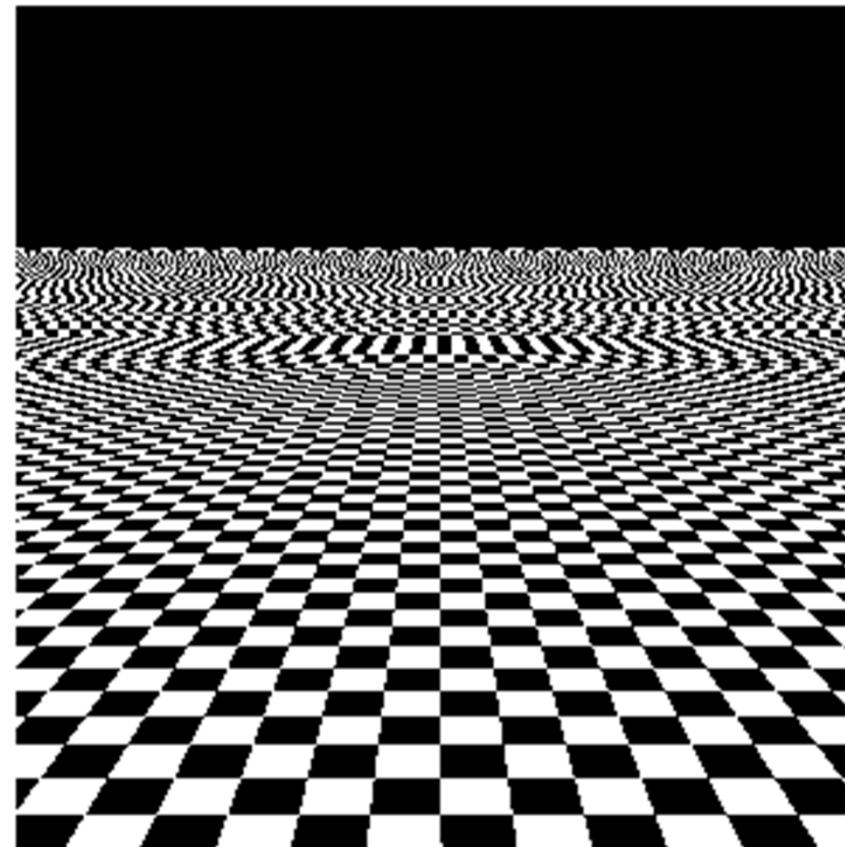
Textur ist 3D

# Volumetric Fog



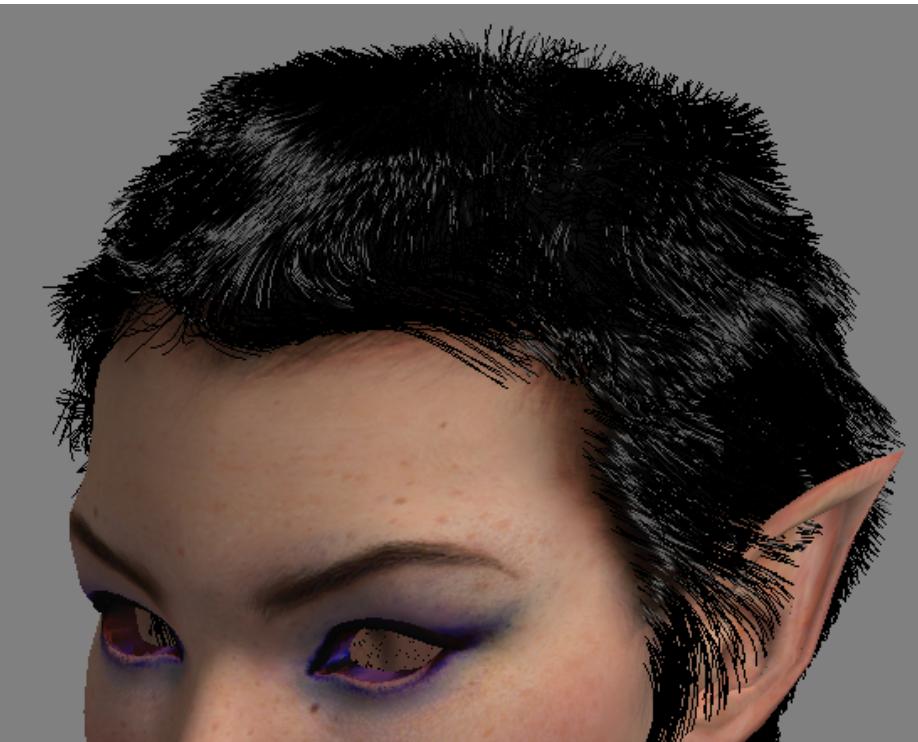
# Aliasing

One screen pixel maps to many texels → Aliasing!

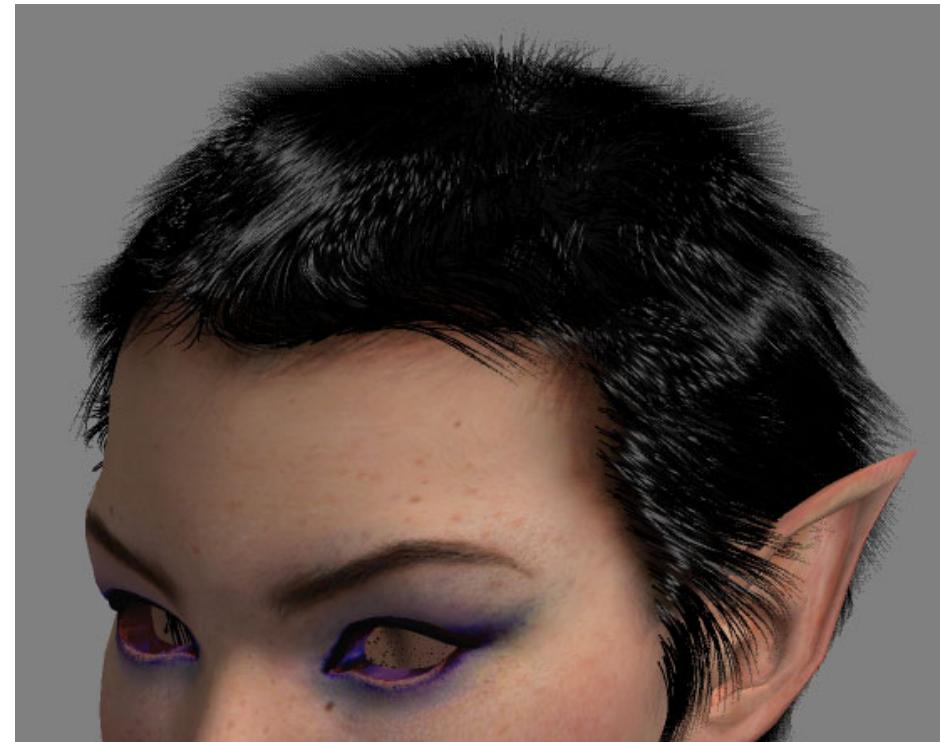


In der Entfernung -> Kantenbildung

# Hair



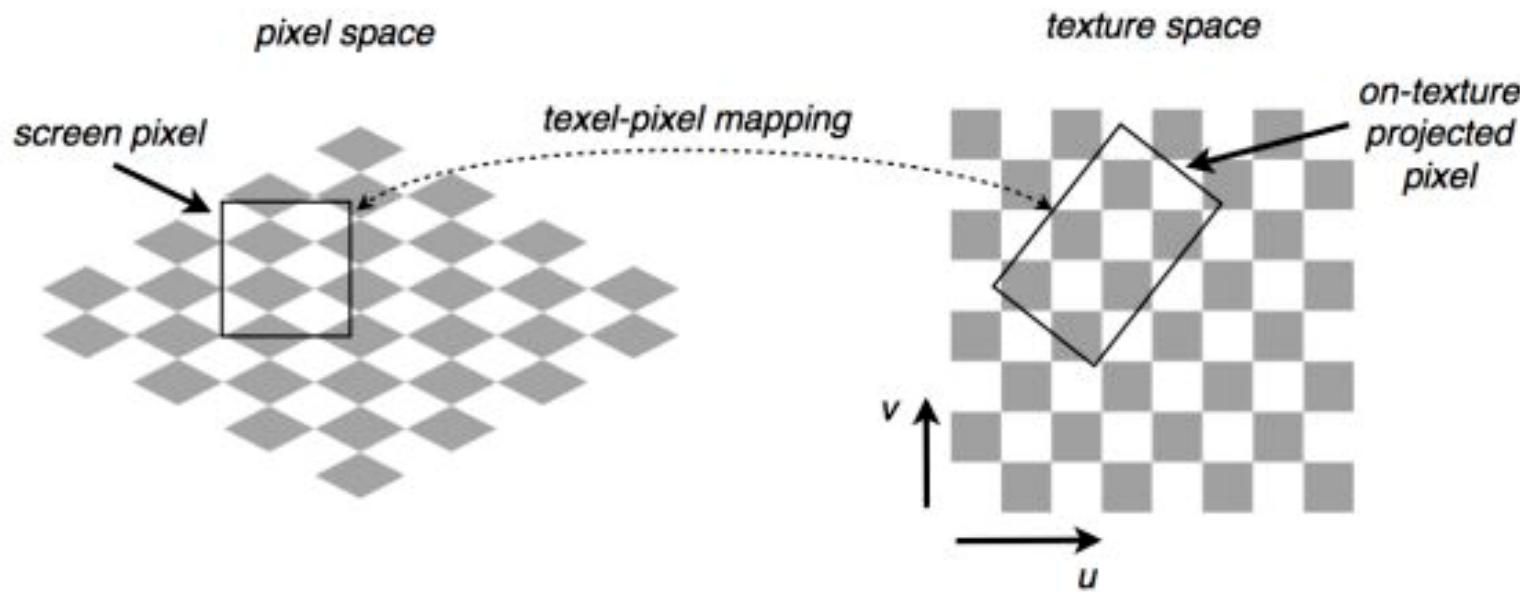
Without antialiasing



With antialiasing

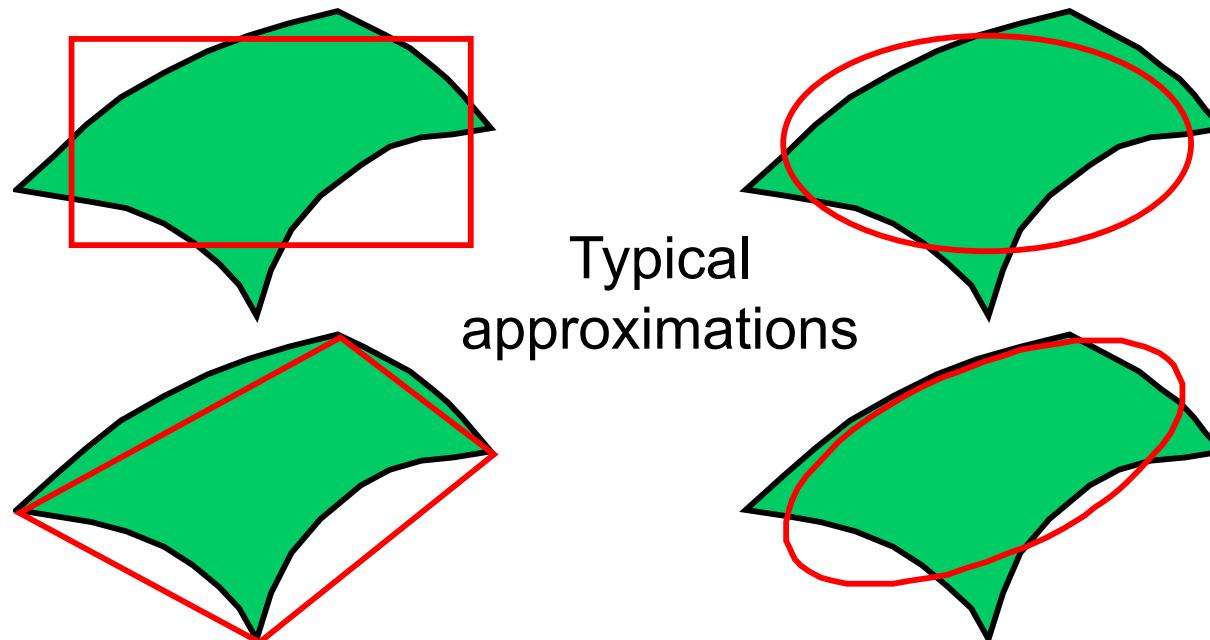
# Antialiasing of Textures

- Correct pixel value is (weighted) mean of pixel area projected (back) into texture space
- Two approaches:
  - Direct convolution
  - Prefiltering



# Direct Convolution

Calculate weighted mean of relevant texels



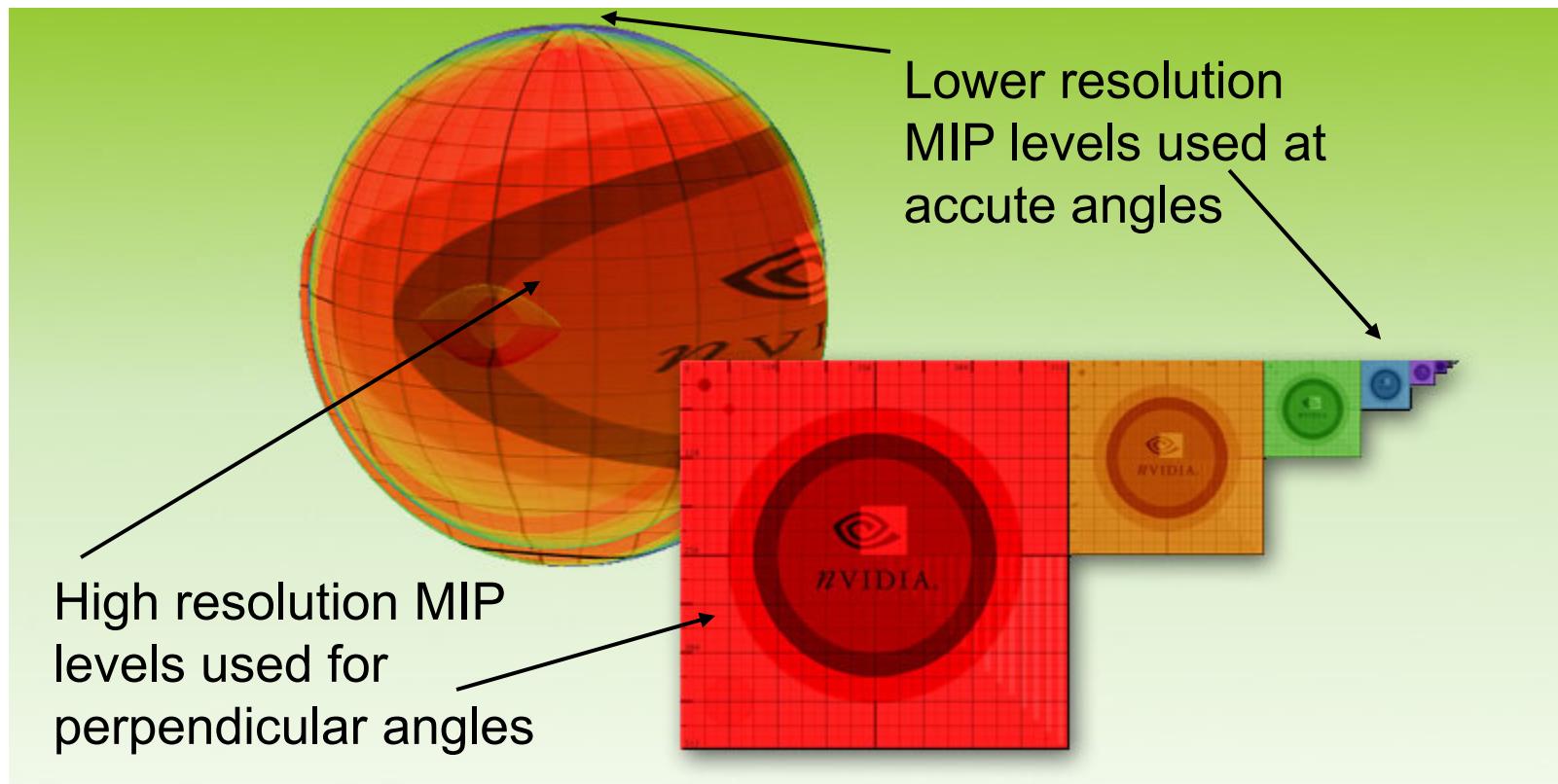
Real-time: approximate with fixed sample num.

# 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

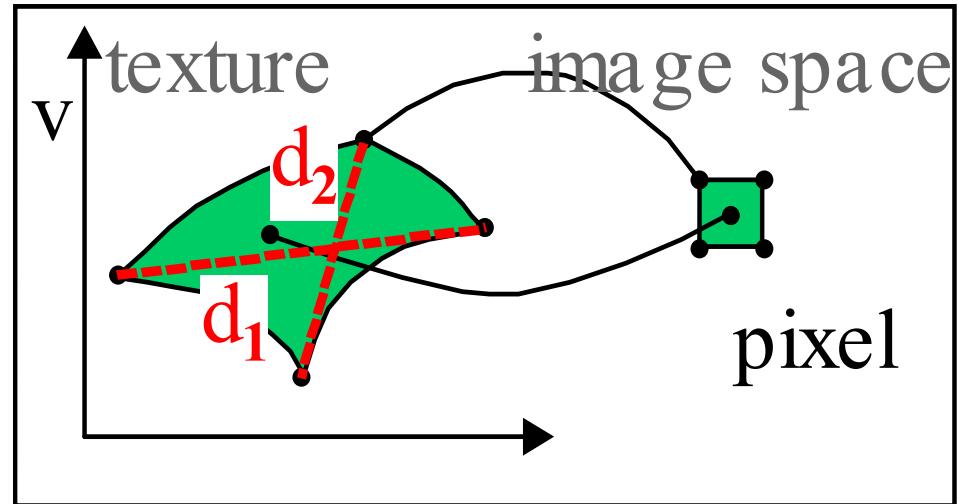


Mip Maps are used to reduce bandwidth requirements of distant textures and minimize scintillating pixels

# Mip Mapping Algorithm

$$D := \lfloor d(\max(|d_1|, |d_2|)) \rfloor$$

D ... Detailierungsgrad LoD  
d ... Distanz zwischen  
Eckpunkten auf der Textur



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

$T_1 :=$  value from table  $D_1 = D_0 + 1$

T0 und T1 sind Indizes für die verschiedenen Mip-Map-Stufen und Bilinear Interpolation wird verwendet, um die Textur des Pixels auf der Grundlage dieser Mip-Map-Stufen und des Detaillierungsgrades zu berechnen.

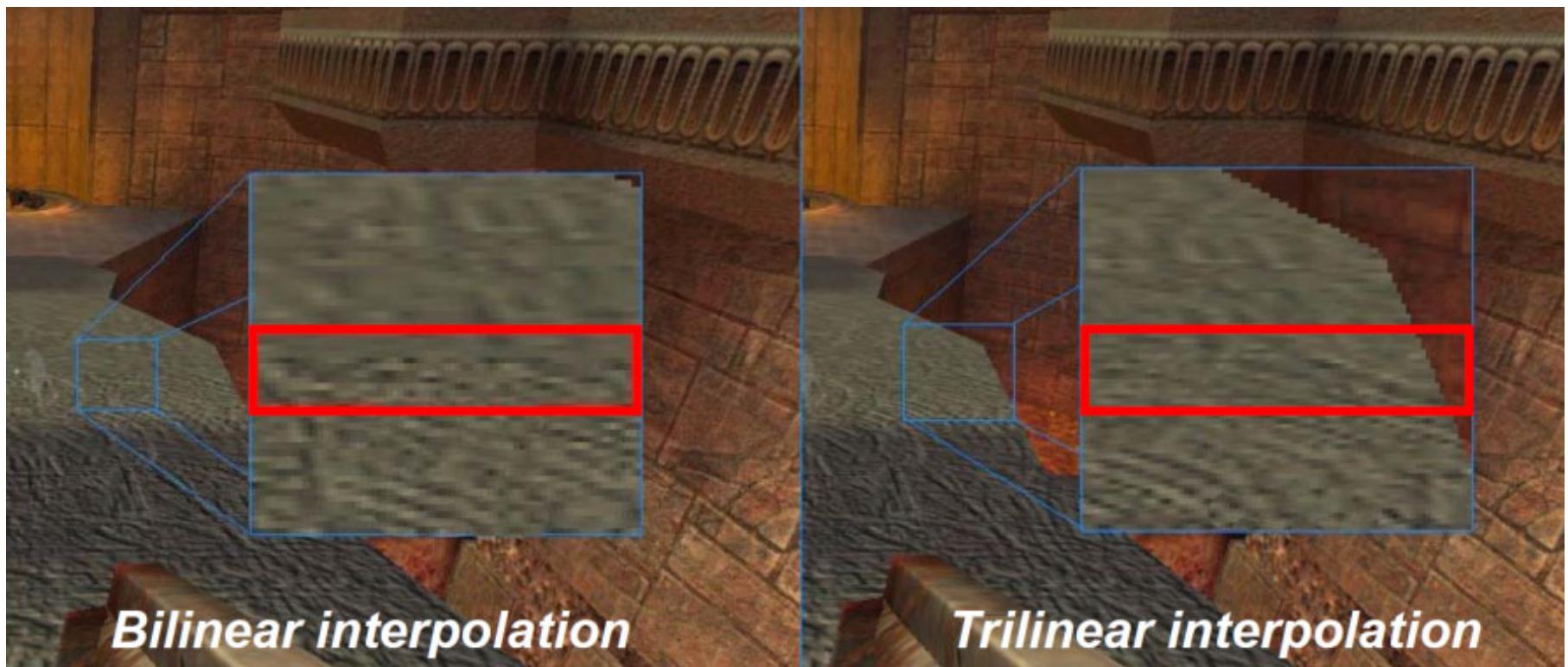
- $T_0, T_1$  obtained with bilinear interpolation
- How to combine  $T_0, T_1$ ?

# Bilinear vs Trilinear

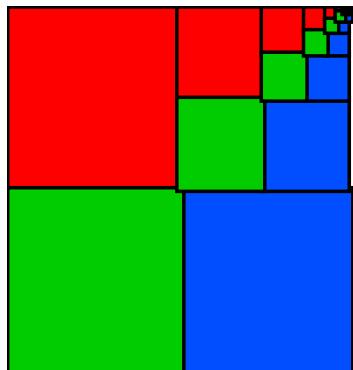
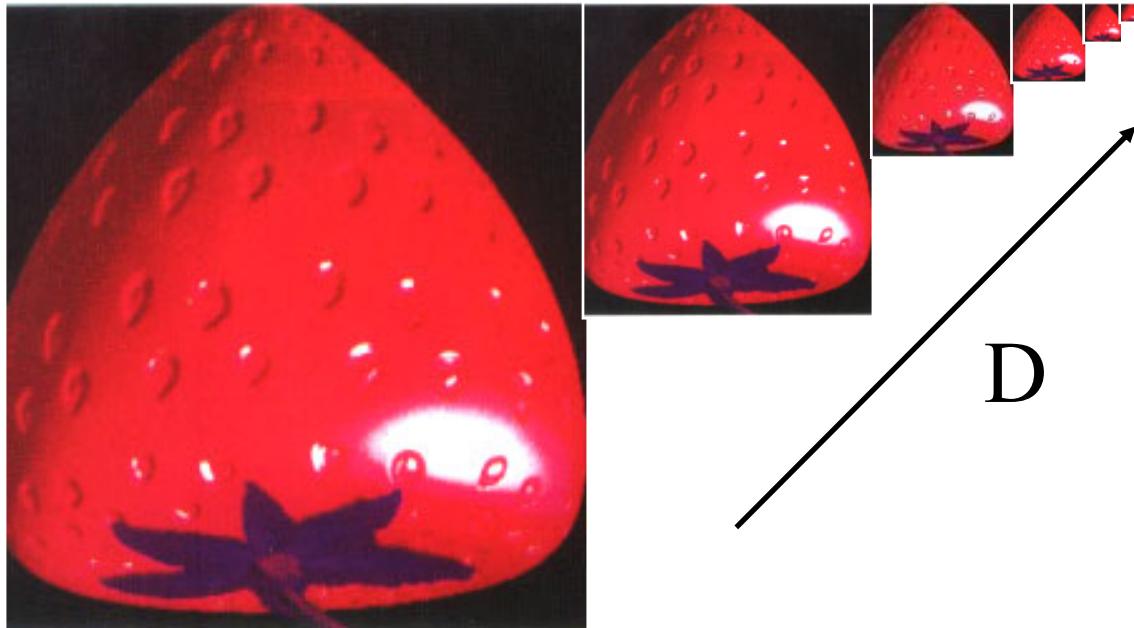
- **Bilinear** :=  $(D_1 - D > 0.5) ? T_1 : T_0$
- **Trilinear** :=  $(D_1 - D) \cdot T_0 + (D - D_0) \cdot T_1 =$   
 $= (D_1 - D) \cdot (T_0 - T_1) + T_1$

**Bilinear:**  
4 benachbarte Pixel auf gleicher Ebene Interpoliert

**Trilinear:**  
8 benachbarte Pixel auf 2 Ebenen interpoliert

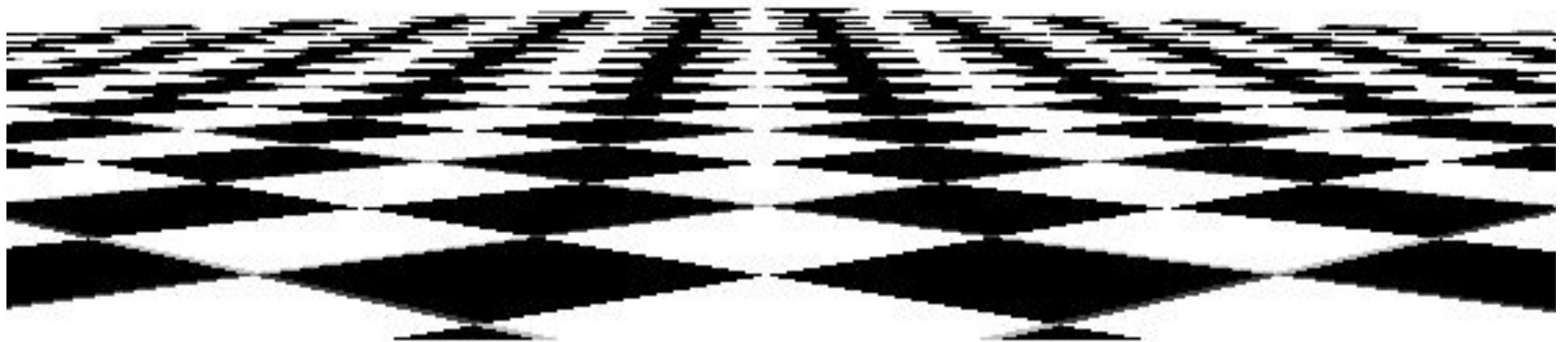


# Mip Mapping Memory Layout



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

# Comparison: Nearest Neighbor



Simple, but bad quality

# Comparison: Linear Interpolation



Fixes some aliasing in the front,  
but in the back it's still not good

# Comparison: Mip Mapping



Replaces objectionable artifacts with blur

# What Causes this Problem?

- Problem with mip mapping
  - Blurs in every direction equally
- Not optimal if distortion due to projection is not uniform



ok →



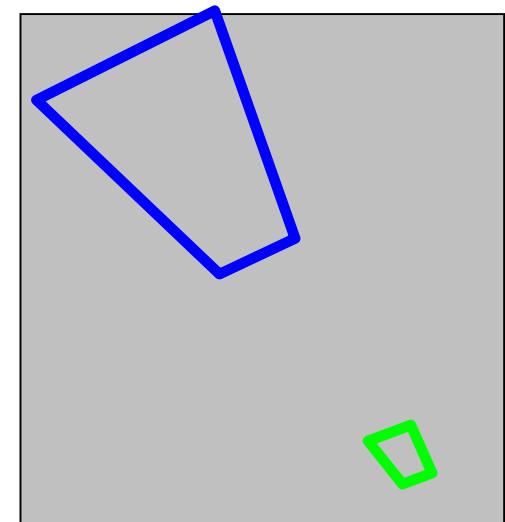
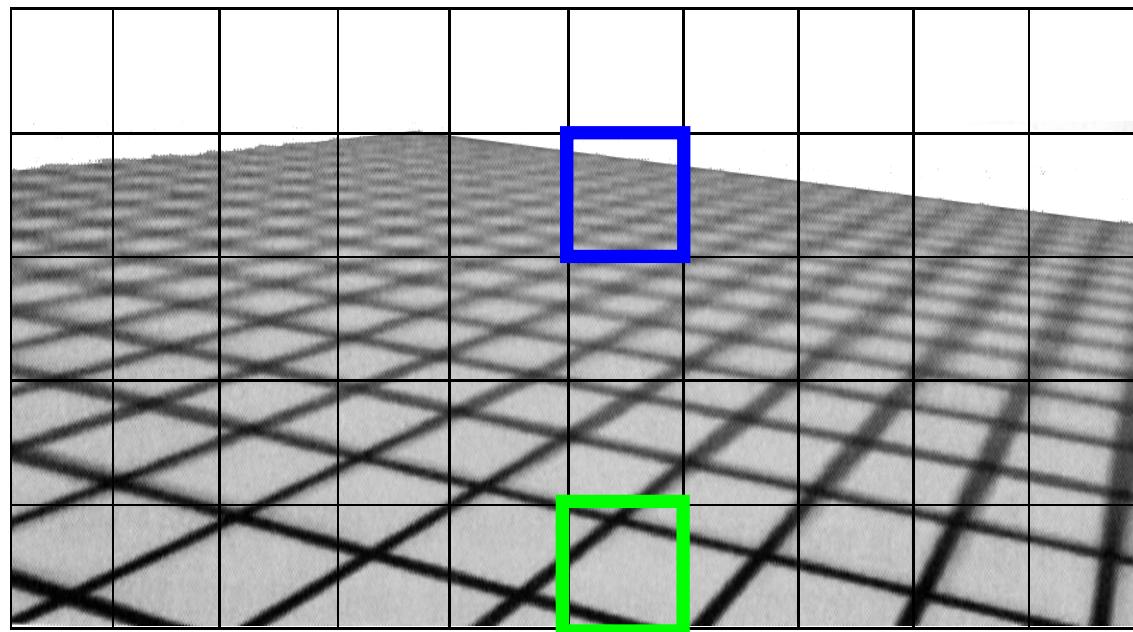
not ok →



# Anisotropic Filtering

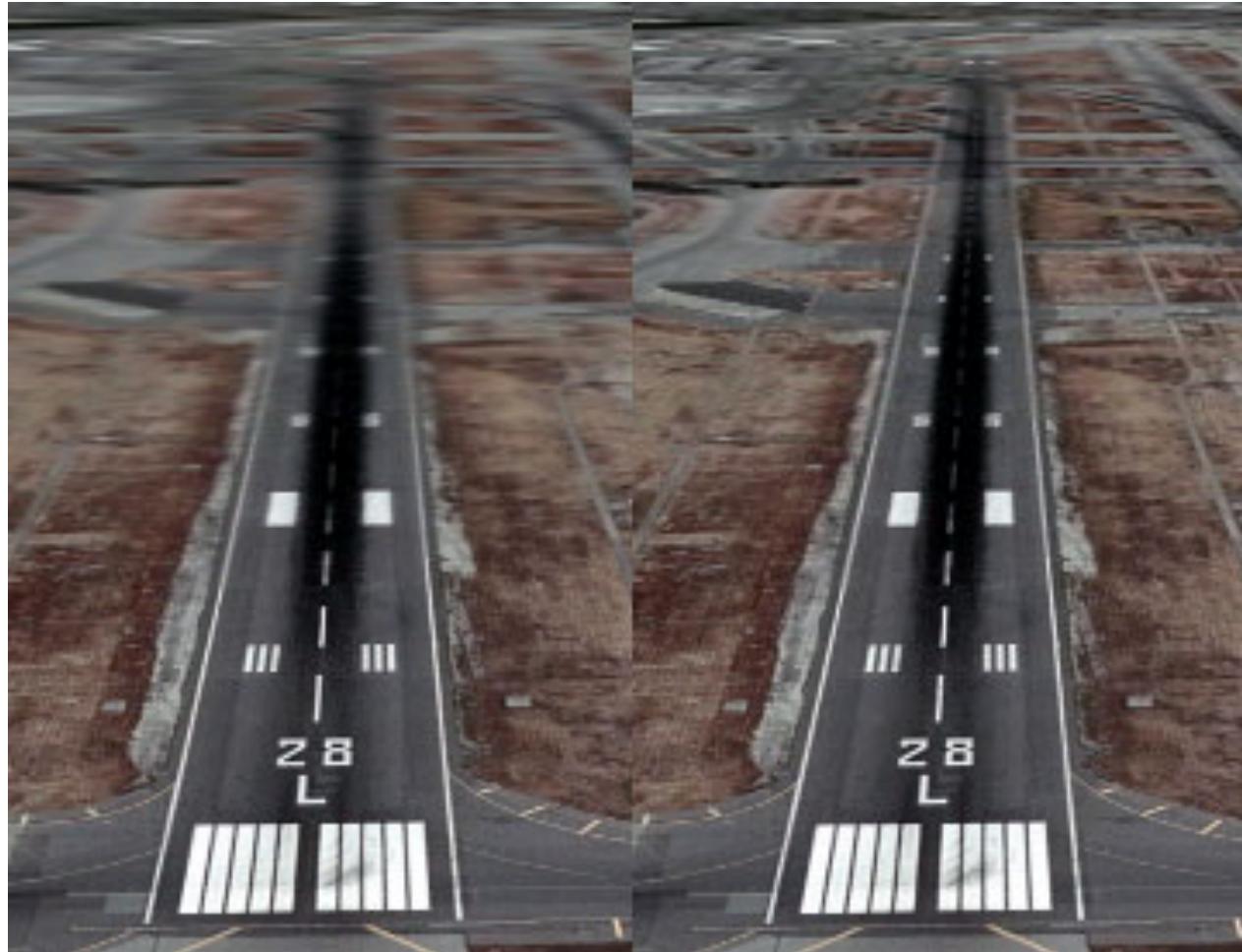
More sharpness and clarity in the distance. Anisotropic filtering works by taking multiple texture samples on the surface and combining them to create higher quality textures.

Needs a view-dependent filter kernel



**Texture space**

# Anisotropic Filtering Example

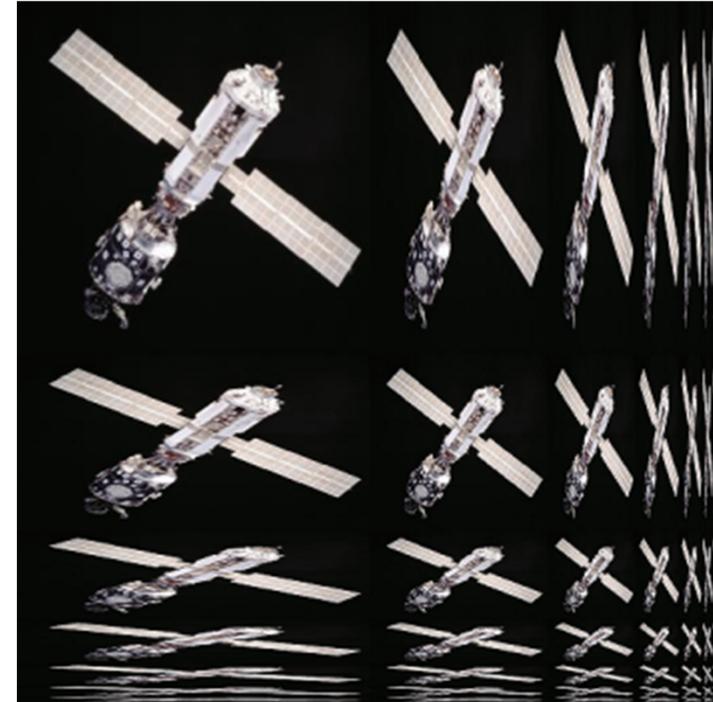


Mip mapped

With anisotropic filtering

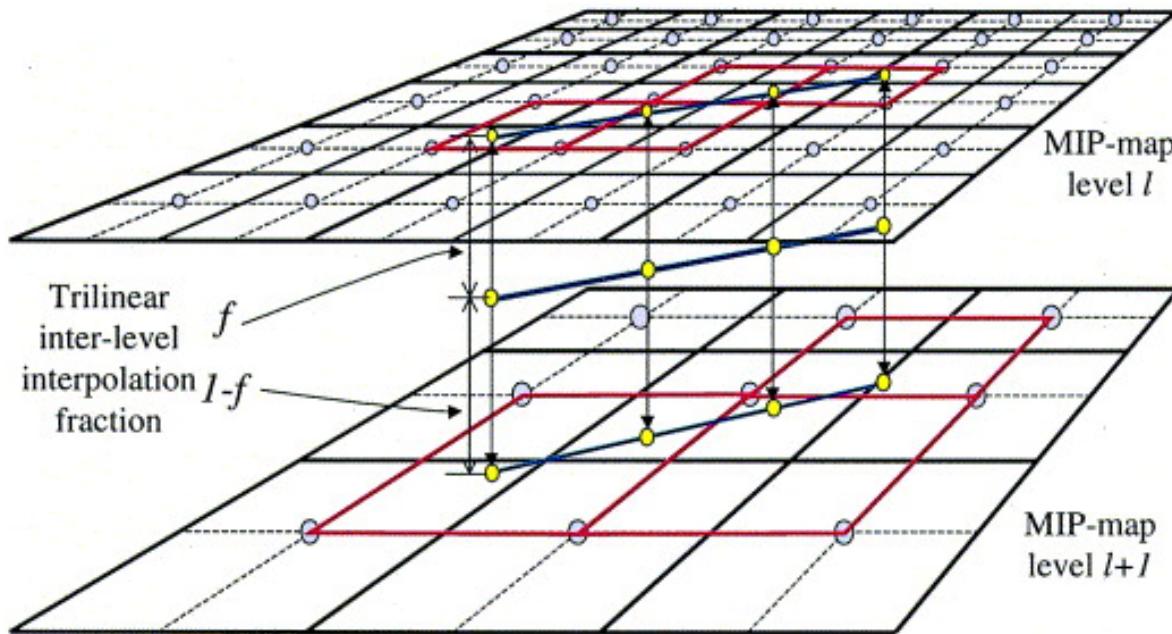
# Rip Mapping

- Anisotropic filtering as extension to mip map
- Prefilter also anisotropic (non-uniform) scales
- Computed offline
- Needs 4x the memory



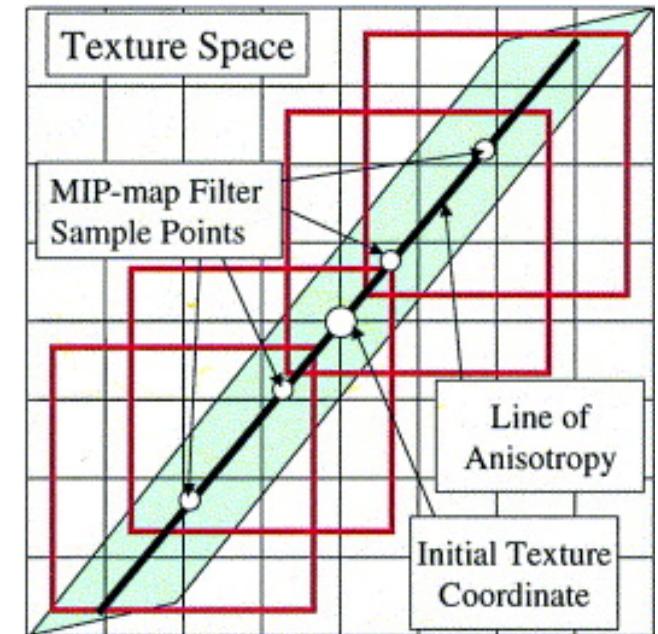
# Anisotropic Real-Time Sampling

- Determine major line of anisotropy (=longer axis)
- Fixed number of sample along line of anisotropy
- Trilinear interpolation for each sample



Dieter Schmalstieg

Texture mapping



48

# Mip Mapping Upsampling

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



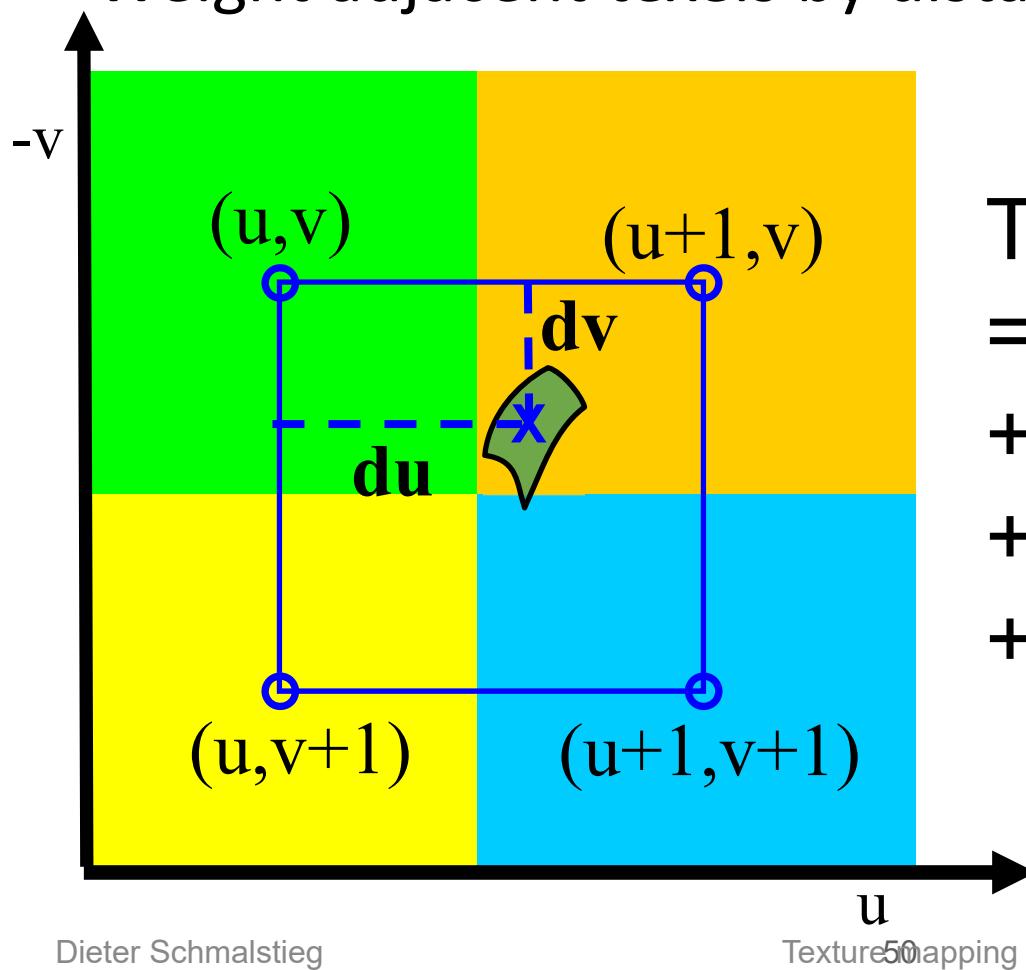
Dieter Schmalstieg



Texture mapping

# Bilinear Upsampling

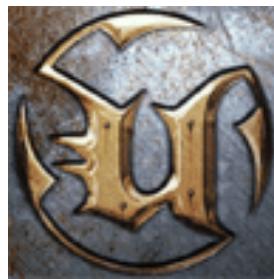
- Bilinear reconstruction for texture magnification ( $D<0$ )
- Weight adjacent texels by distance to pixel position



$$\begin{aligned}
 T(u+du, v+dv) \\
 &= du \cdot dv \cdot T(u+1, v+1) \\
 &+ du \cdot (1-dv) \cdot T(u+1, v) \\
 &+ (1-du) \cdot dv \cdot T(u, v+1) \\
 &+ (1-du) \cdot (1-dv) \cdot T(u, v)
 \end{aligned}$$

Der Prozess besteht darin, jeden Pixel in der niedrig aufgelösten Textur mit den vier umgebenden Pixeln in der höher aufgelösten Textur zu interpolieren. Dies wird erreicht, indem die Farbwerte der umgebenden Pixel gewichtet nach ihrer Entfernung zum Zielpixel berechnet werden.

# Bilinear Upsampling Result



Original image



Nearest neighbor

Dieter Schmalstieg



Bilinear filtering

Texture Sampling

# 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)

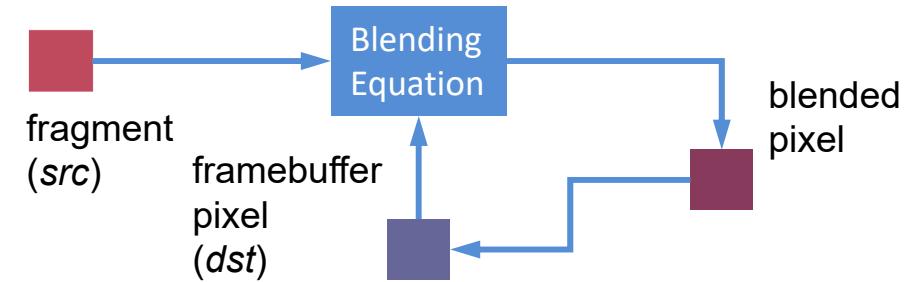
1. Frame buffer with color attachment is bound
2. Scene (or subset) is rendered,  
possibly using a custom shader program
3. Frame buffer is unbound  
Color attachment is bound as texture
4. Scene is rendered to the back buffer,  
Rendered texture can be used either by  
fixed function pipeline or a custom shader

# Example: Fragment Shader Multiple Outputs

```
1 #version 330
2
3     layout(location = 0) out vec4 color0;
4     layout(location = 1) out vec4 color1;
5
6     void main()
7     {
8         color0 = ...
9         color1 = ...
10    }
```

# 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

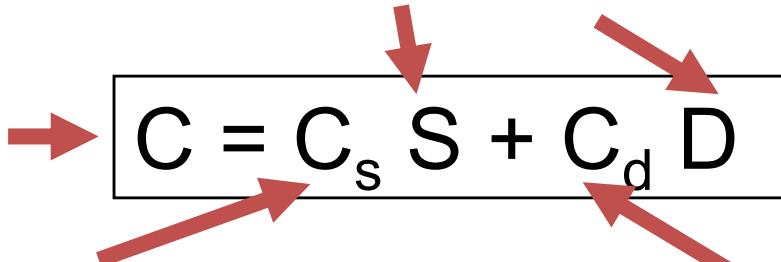


# Multipass – Framebuffer Blending

- 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!

# Other Blend Equations

- **GL\_FUNC\_ADD**

$$\mathbf{c}_d \leftarrow \mathbf{c}_s \otimes \mathbf{w}_s + \mathbf{c}_d \otimes \mathbf{w}_d$$

- **GL\_FUNC\_SUBTRACT**

$$\mathbf{c}_d \leftarrow \mathbf{c}_s \otimes \mathbf{w}_s - \mathbf{c}_d \otimes \mathbf{w}_d$$

- **GL\_FUNC\_REVERSE\_SUBTRACT**

$$\mathbf{c}_d \leftarrow \mathbf{c}_d \otimes \mathbf{w}_d - \mathbf{c}_s \otimes \mathbf{w}_s$$

- **GL\_FUNC\_MIN**

$$\mathbf{c}_d \leftarrow \min(\mathbf{c}_s, \mathbf{c}_d)$$

- **GL\_FUNC\_MAX**

$$\mathbf{c}_d \leftarrow \max(\mathbf{c}_s, \mathbf{c}_d)$$

# 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)

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

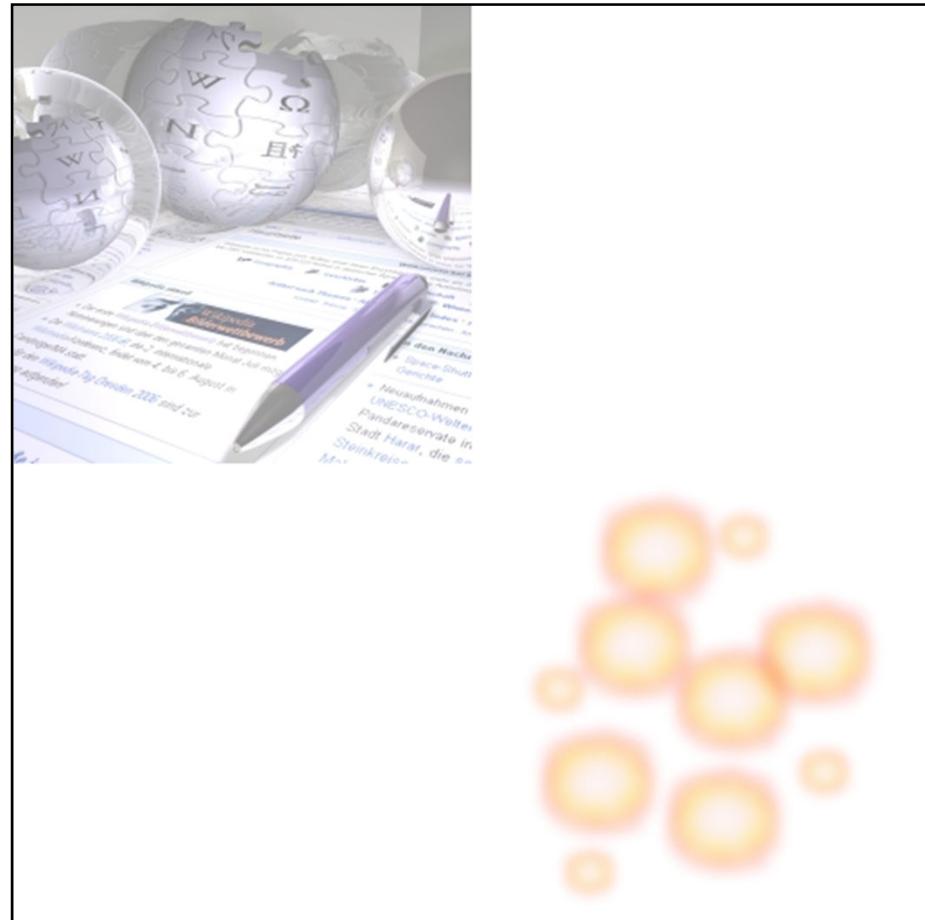
# Alpha Buffer

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**

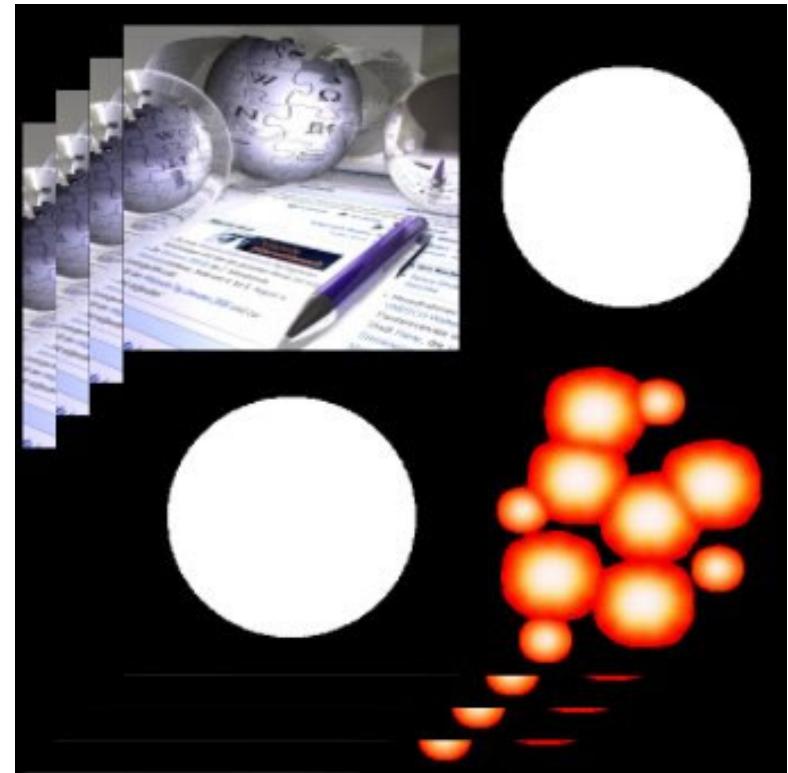
# Blending Examples (1)

Texture with alpha  
channel for the  
following examples...



# Blending Examples (2)

- `glBlendFunc(GL_ONE, GL_ZERO);`
  - Everything works as if blending was disabled
  - Alpha values also ignored

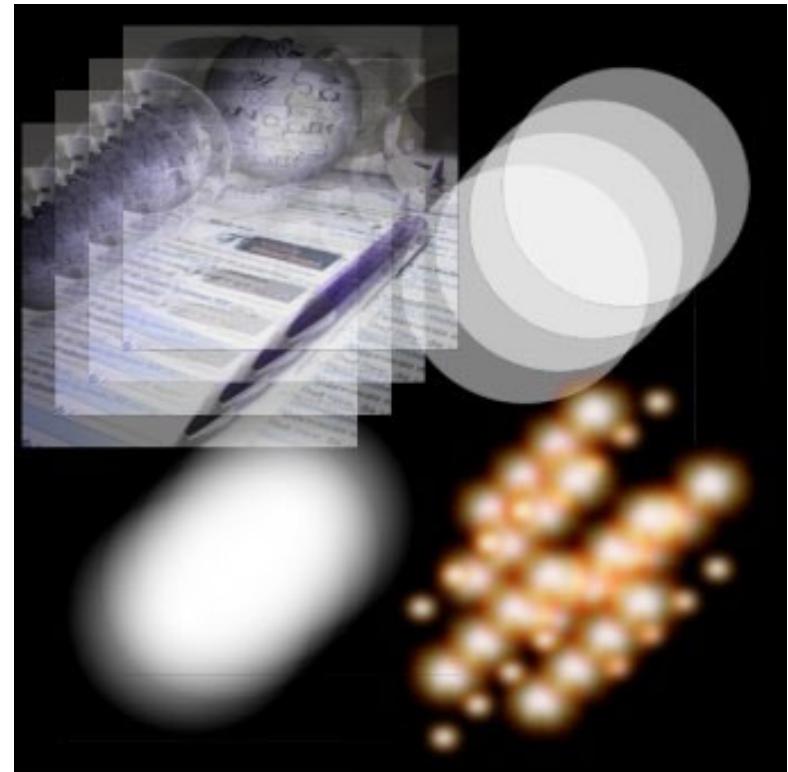


# Blending Examples (3)

- `glBlendFunc(GL_ZERO, GL_ONE);`
  - Nothing is drawn
  - Not really useful...

# Blending Examples (4)

- `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`
  - Most common blending method
  - Simulates traditional transparency
  - Source color's alpha (usually from texture) defines opaqueness of object to render



# Blending Examples (5)

- `glBlendFunc(GL_ONE, GL_ONE);`
  - Source and destination color are simply added
  - Values >1.0 are clamped to 1.0 (saturation)
  - Good for fire effects



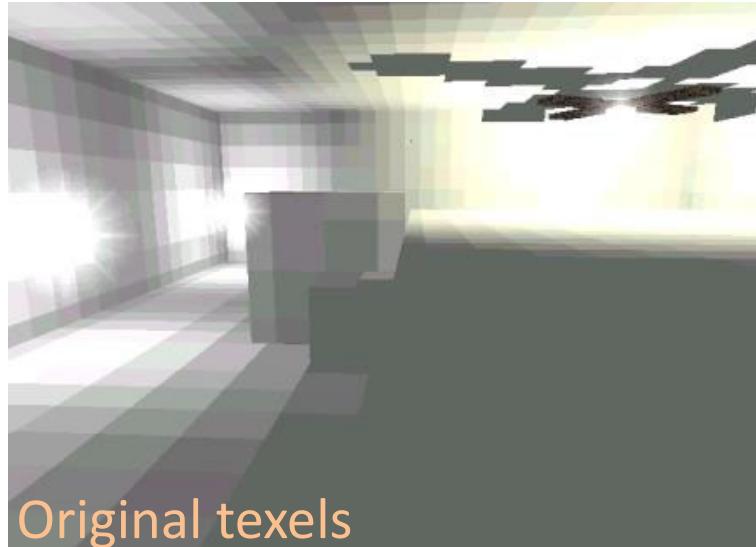
# Multitexturing

- Idea: Apply multiple textures in one pass
- Textures can be combined arbitrarily (add, mul, ...)
- Indirect lookup
  - 1st texture lookup is used to index into 2nd texture
- Today, just use multiple *texture samplers* in a fragment shaders
- Texture samplers correspond to dedicated hardware units for accessing texture memory

# 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
- 
- =

# Light Mapping Example 1



Original texels



Gouraud shaded

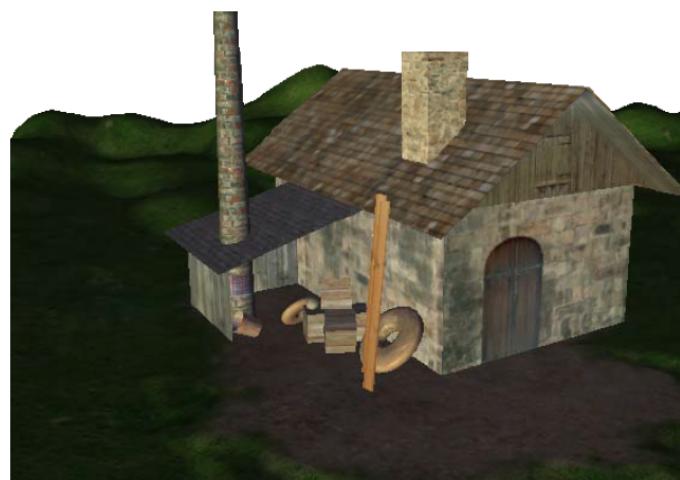
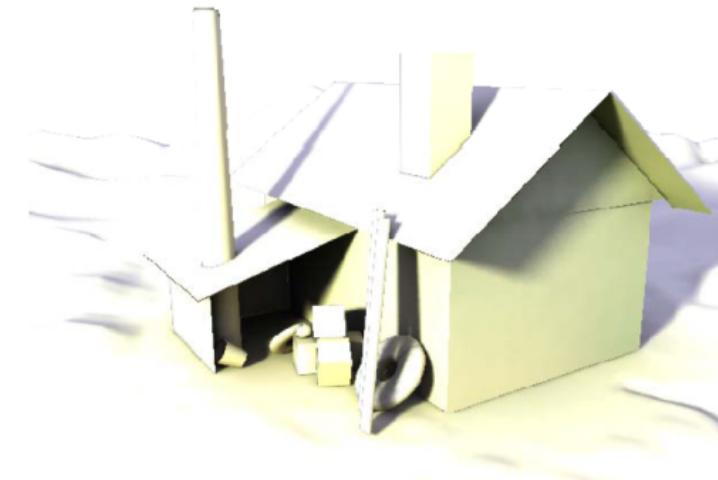
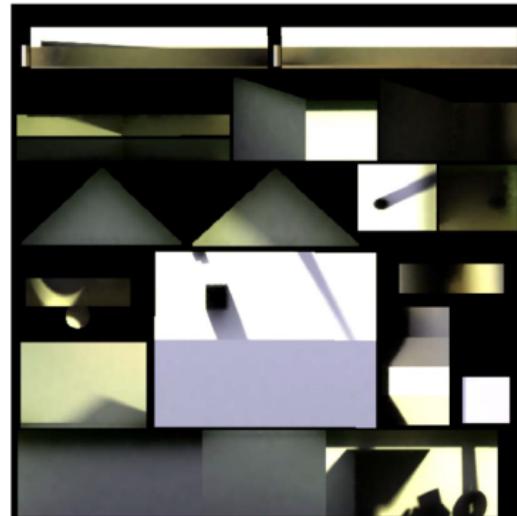


Dieter Schmalstieg



Texture mapping

# Light Mapping Example 2

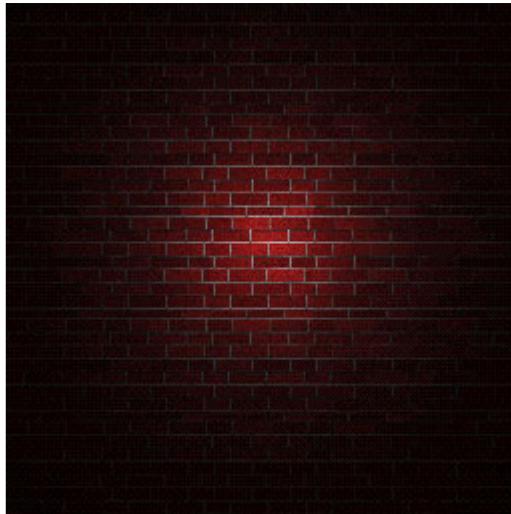


# Light Mapping Procedure

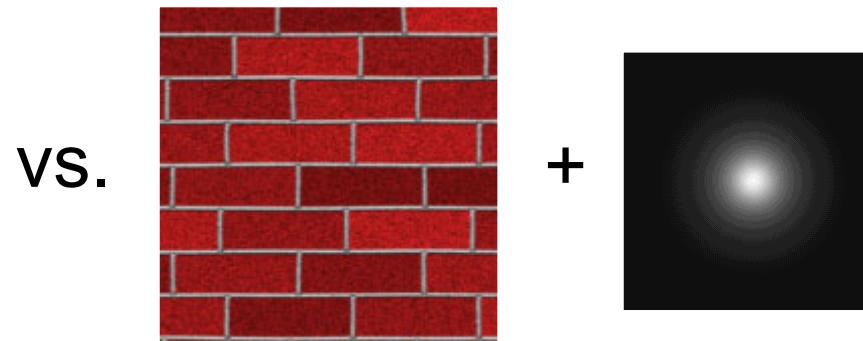
- Map generation
  - Use single map for group of coplanar polys
  - Map back to worldspace to calculate lighting
- Map application
  - Premultiply textures by light maps
    - Large textures, no dynamics
  - Multitexturing at runtime
    - Fast, flexible

# Light Mapping Issues

- Why premultiplication is bad...



Full Size Texture  
(with Lightmap)

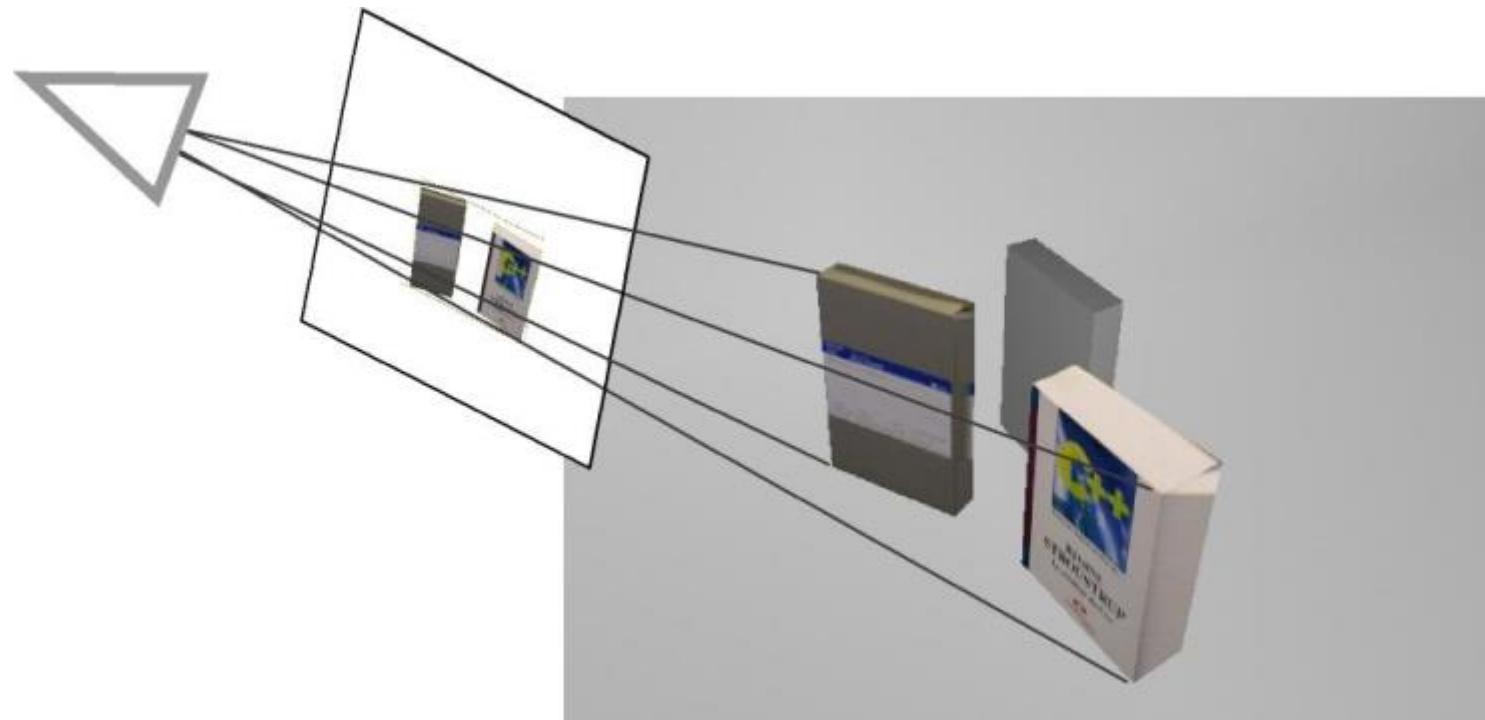


Tiled Surface Texture  
plus Lightmap

- Use small surface textures and small maps
- Maybe calculate low-res light maps on the fly

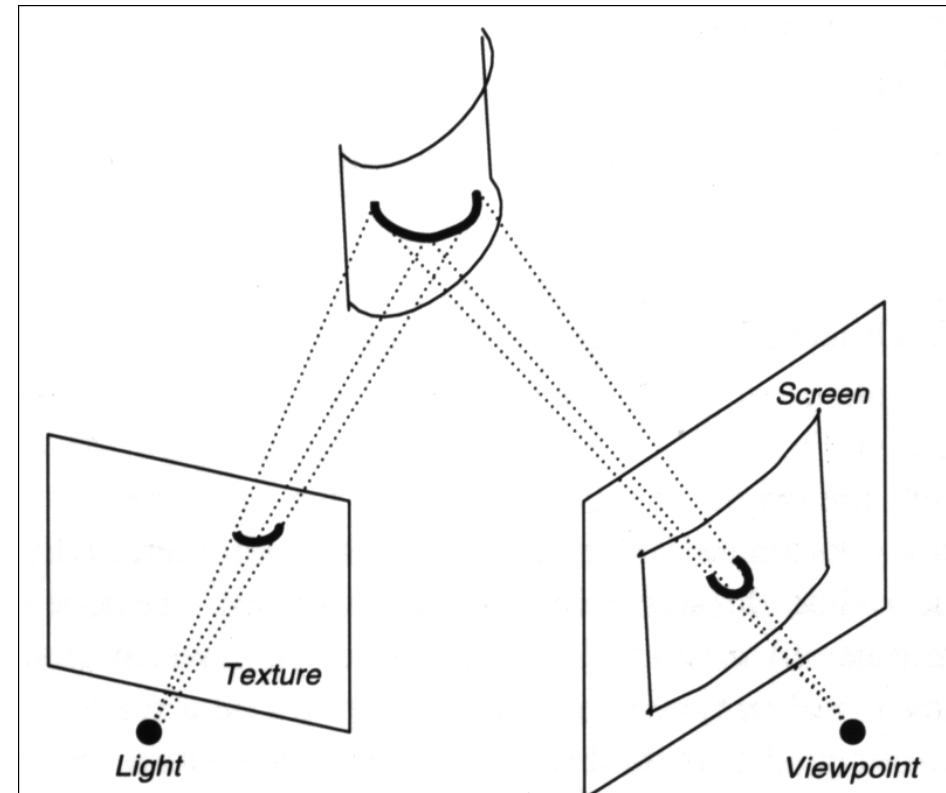
# Projective Texture Mapping

- PTM simulates a video projector
  - Or a flashlight, slide projector, ...
- Using perspective projection



# PTM: Geometry

- Map object coordinates to light frustum
- Express this as a projective transform
- Usually a 4x4 matrix
- Similarity to video projector observed by camera



# PTM: Flashlights Examples



STALKER

Dieter Schmalstieg

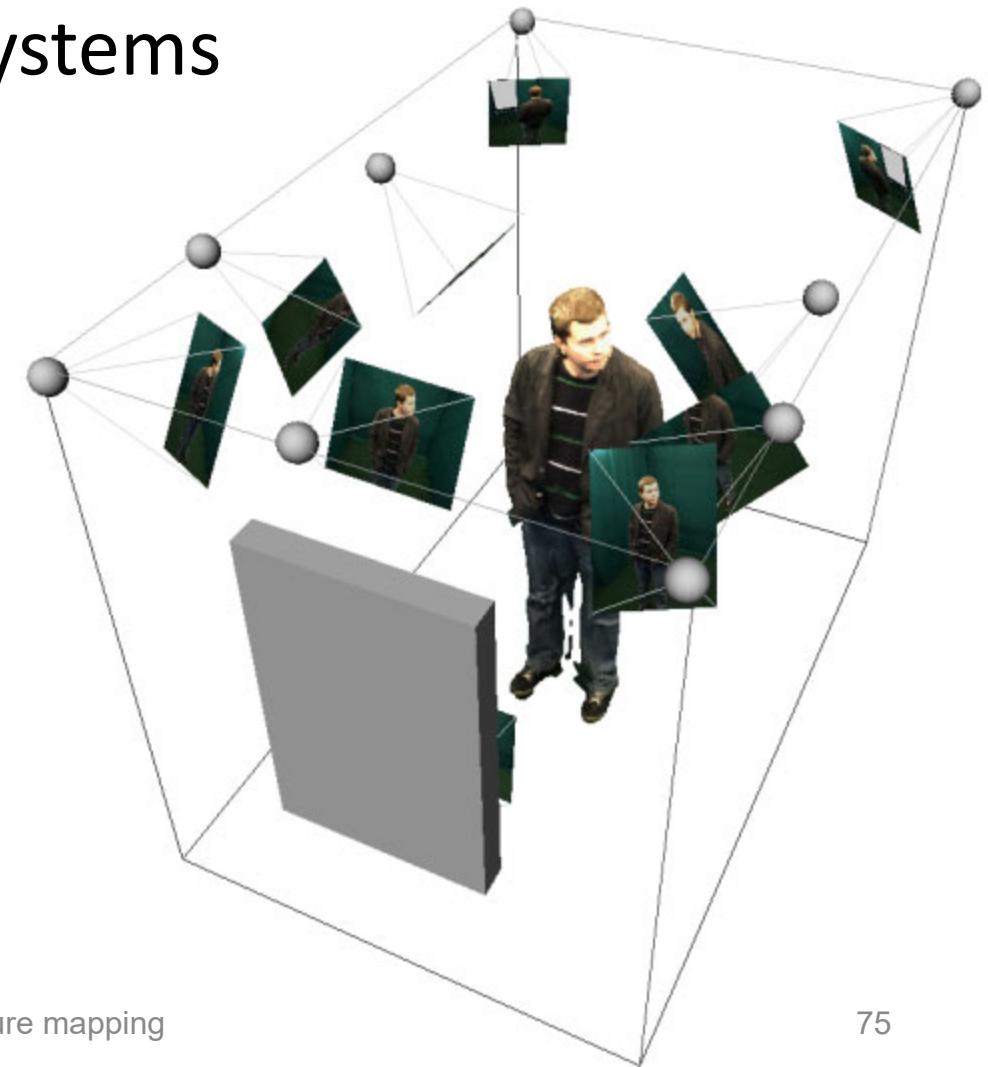


Doom 3

Texture mapping

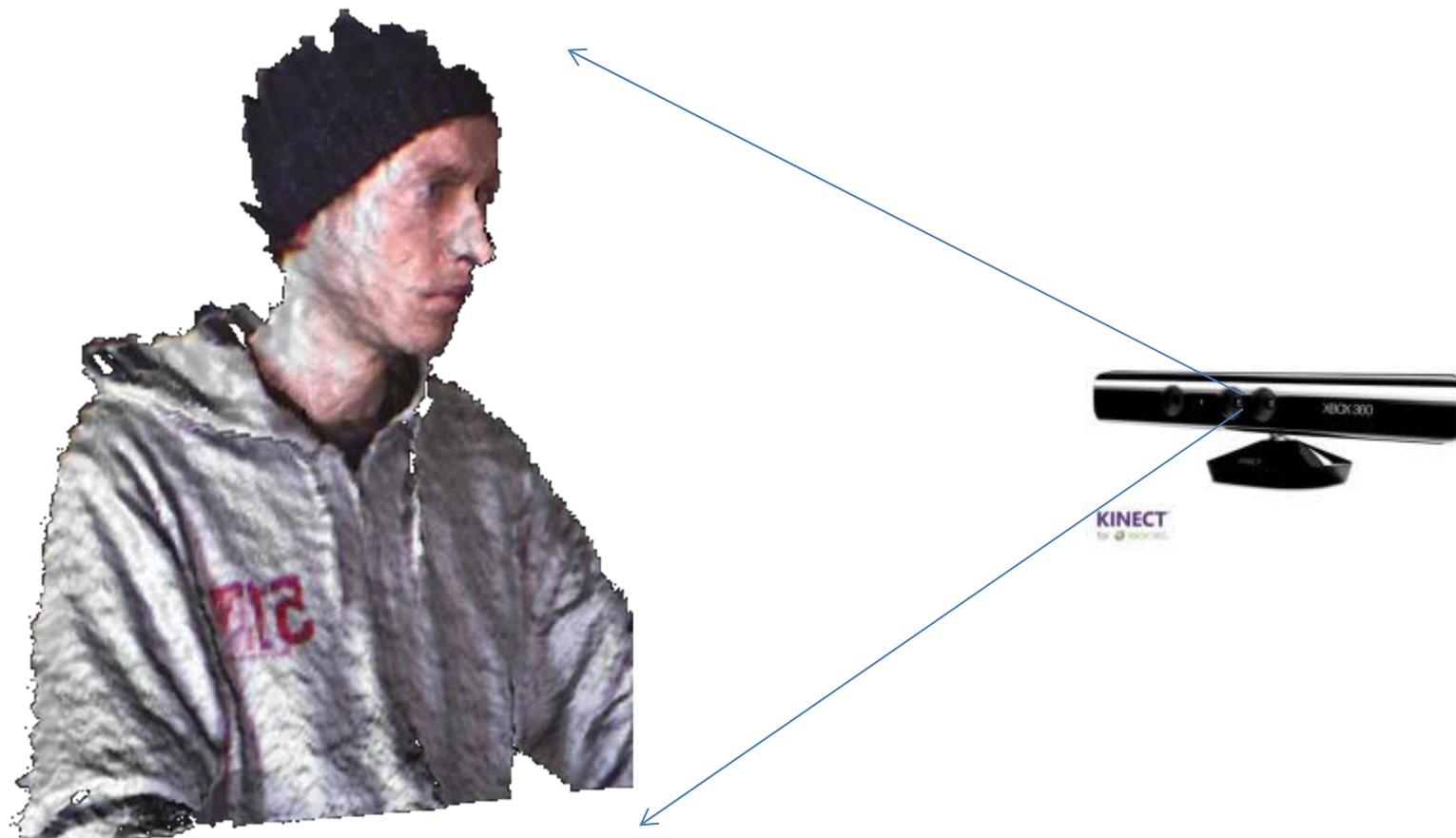
# PTM from Multi-Camera

User in multi-camera systems



# PTM from Depth Camera

Example: display a Microsoft Kinect point cloud

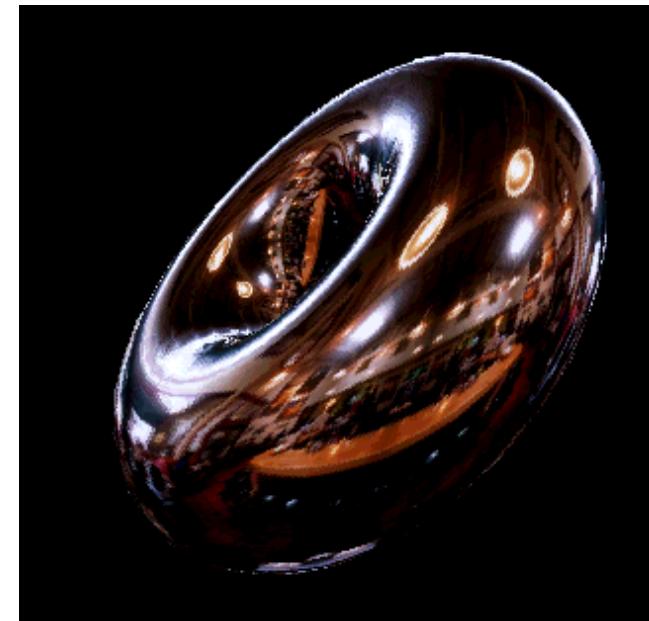


# Environment Mapping

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



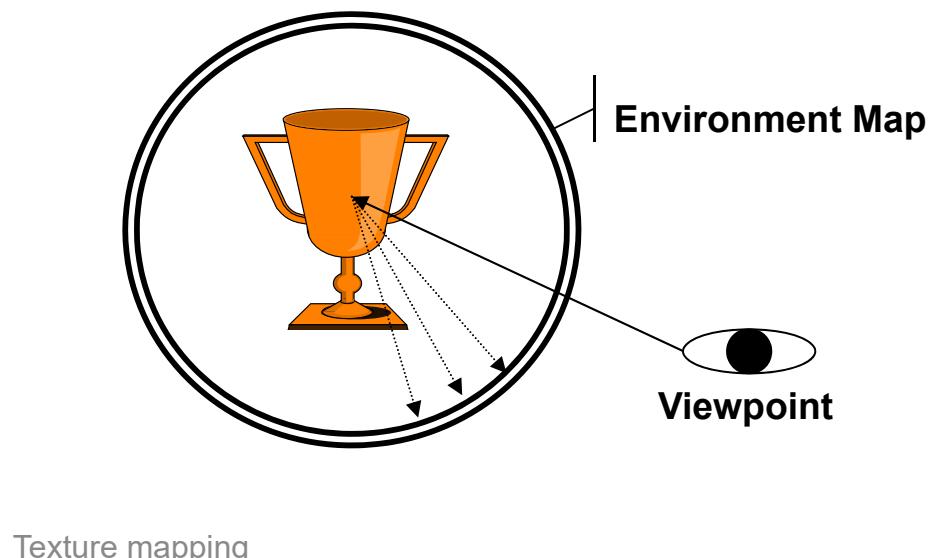
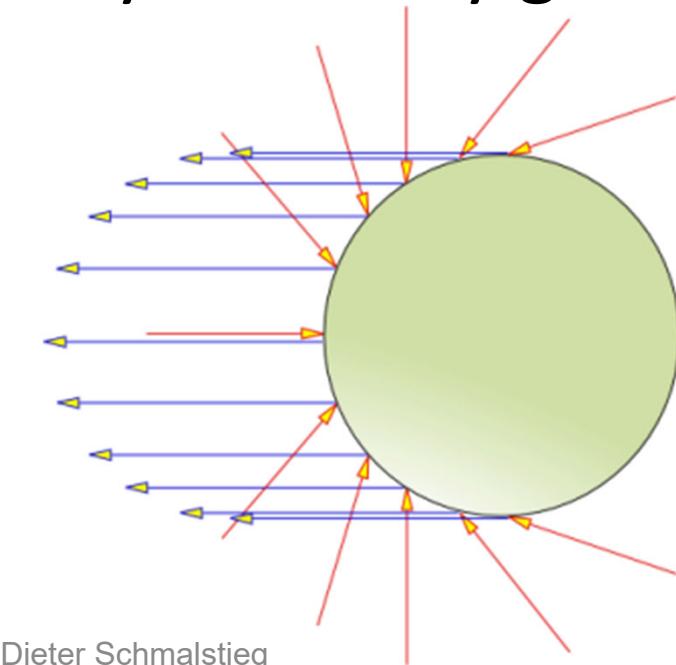
Dieter Schmalstieg



Texture mapping

# Environment Mapping Indexing

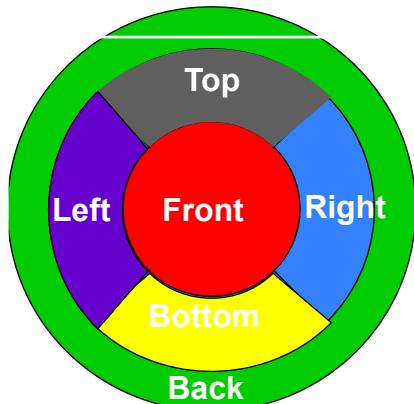
- Assumption: index envmap via orientation →
  - Reflecting object shrunk to a single point
  - Or: environment infinitely far away
- Eye not very good at discovering the fake



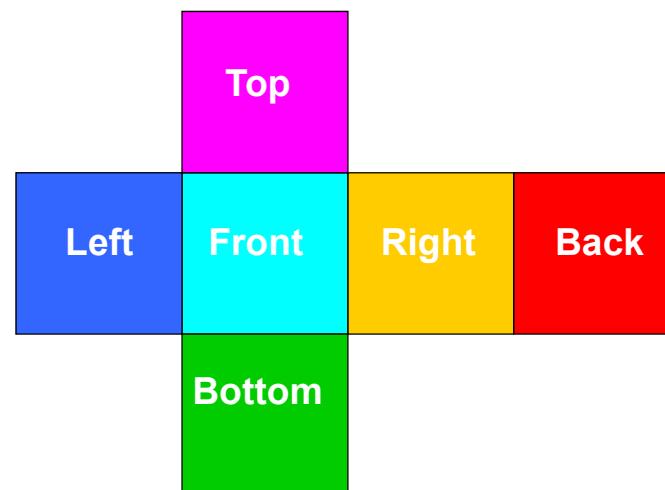
# Environment Mapping Types

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

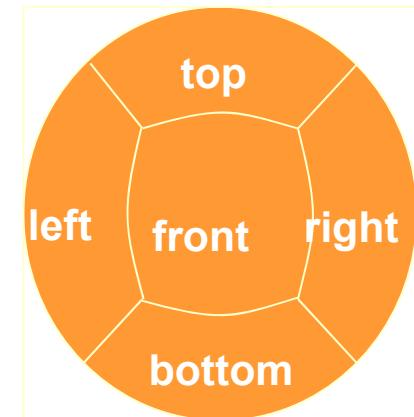
Sphere



Cube

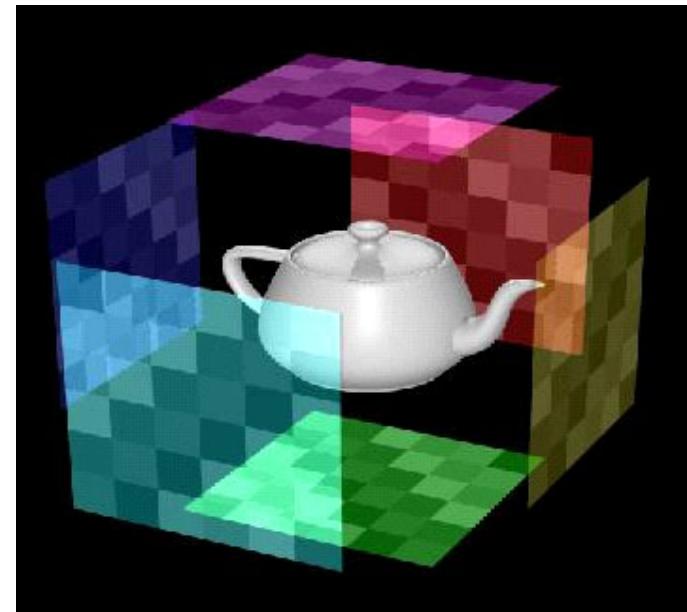
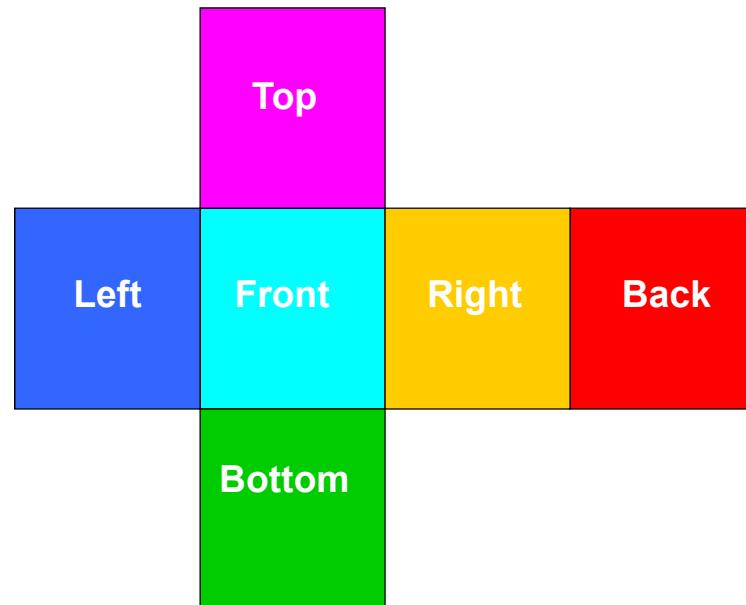


Dual paraboloid

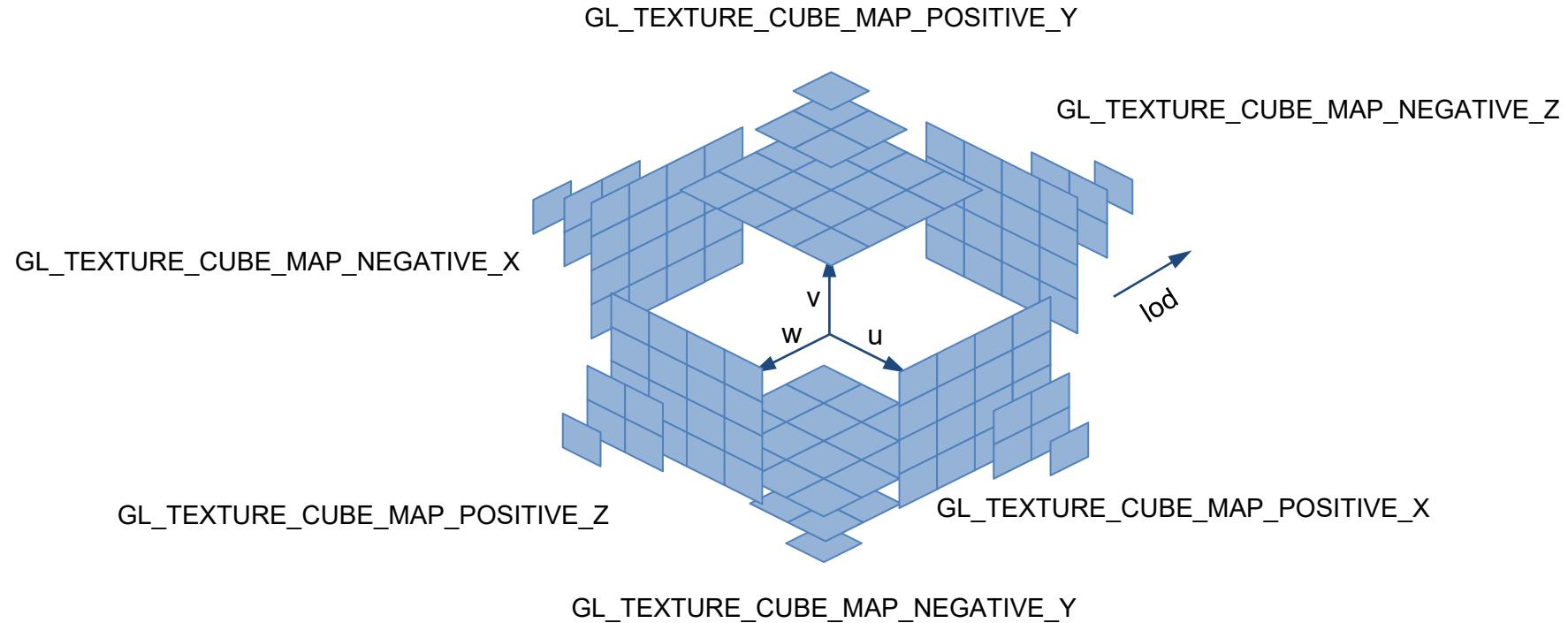


# Cube Mapping

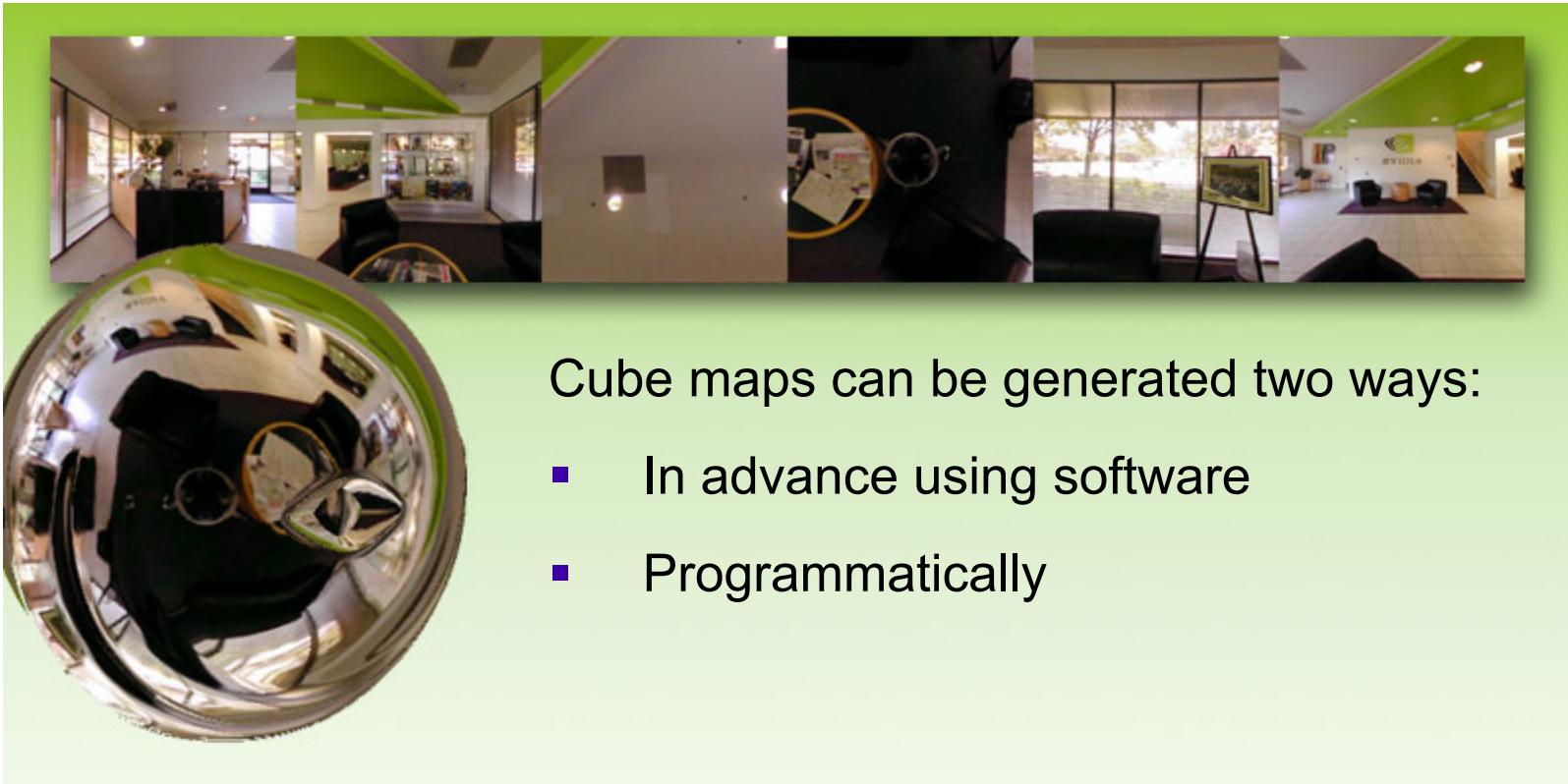
- Implemented in the OpenGL pipeline



# OpenGL Cube Map Targets



# Cube Map Generation



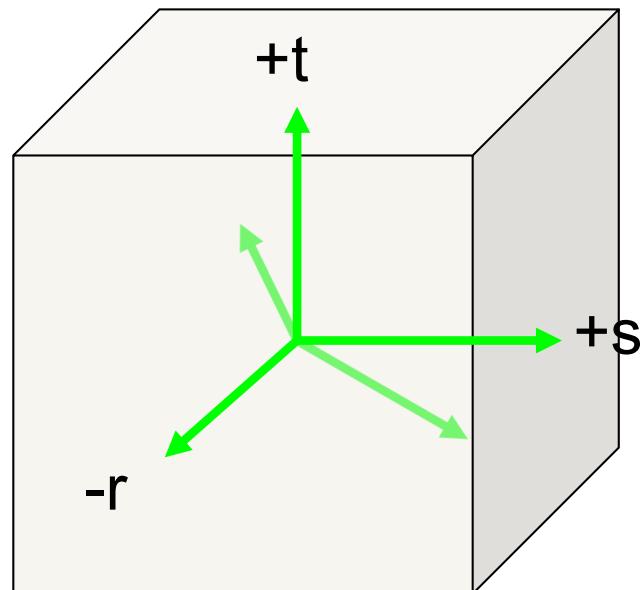
Cube maps can be generated two ways:

- In advance using software
- Programmatically

Cube maps are the primary method of generating reflections in real-time 3D

# Cube Map Addressing

- Cube map accessed via **vectors expressed as 3D texture coordinates ( $s, t, r$ )**



# Cube Map Address Calculation

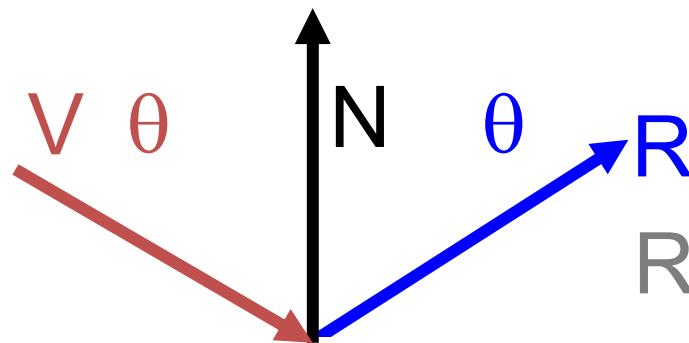
- 3D → 2D projection done by hardware
- Correct ray/quad intersection too slow, instead
  - Highest magnitude component selects which cube face to use (e.g.,  $-t$ )
  - Divide other components by this, e.g.:  
 $s' = s / -t$   
 $r' = r / -t$
  - $(s', r')$  select a texel from this face
- Does not define environment mapping itself
  - Need to generate useful texture coordinates

# Cubic Environment Mapping

- Generate views of the environment
  - One for each cube face
  - 90° view frustum
  - Use hardware to render directly to a texture
- Use reflection vector to index cube map
  - Use hardware mode for sampling in cube map

# Reflection Vector

- Angle of incidence = angle of reflection



$$R = V - 2 (N \ N^T) V$$

- OpenGL uses eye coordinates for R
  - Cube map needs reflection vector in world coordinates (where map was created)
- Load texture matrix with inverse 3x3 view matrix

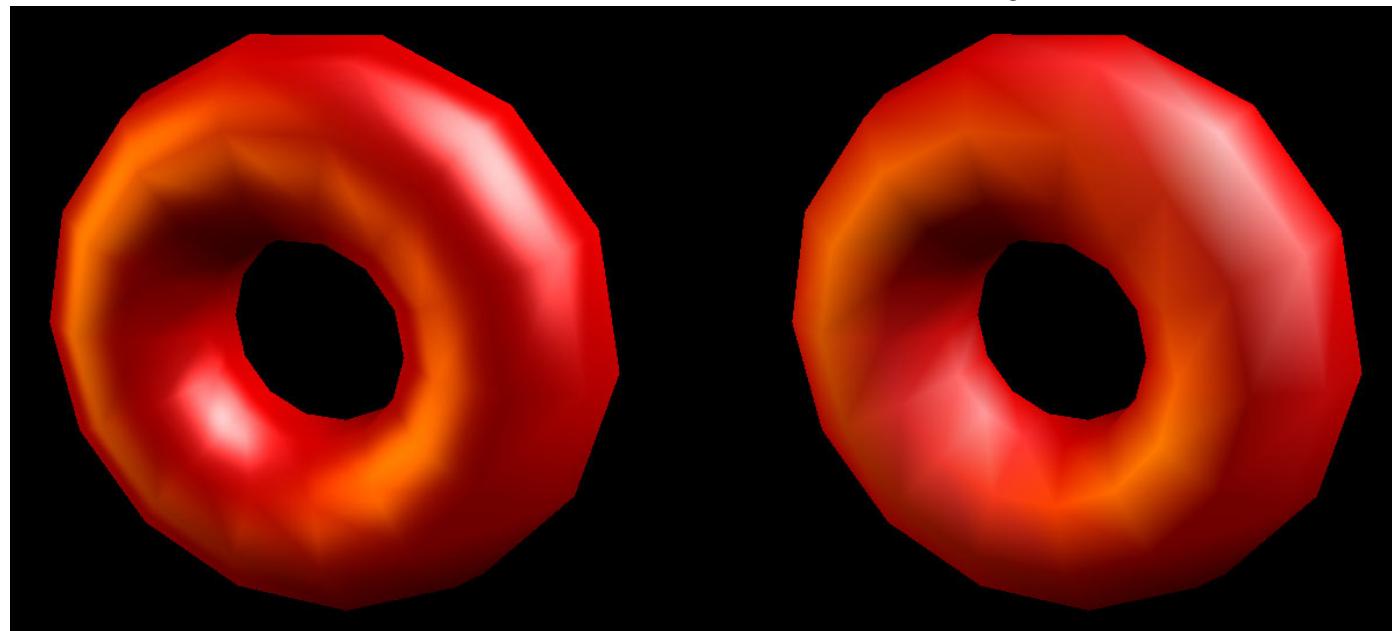
# Cube Mapping Discussion

- Advantages
  - Minimal distortions
  - Creation and map entirely hardware accelerated
  - Can be generated dynamically
- Optimizations for dynamic scenes
  - Need not be updated every frame
  - Low resolution sufficient

# Per-Pixel Lighting

- Simulating smooth surfaces by calculating illumination at each pixel (e.g., Phong shading)
- Advantage: better specular highlights

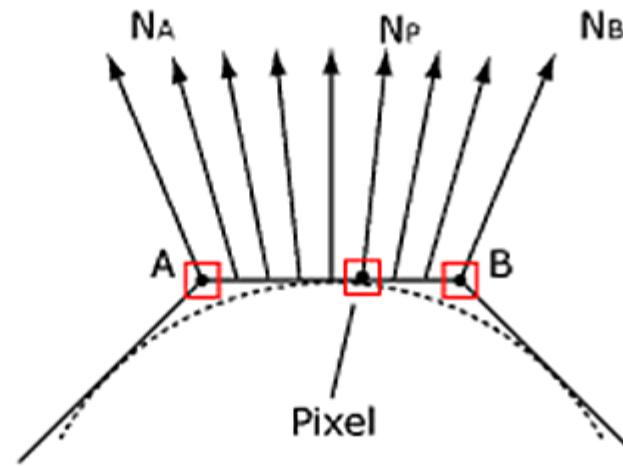
per-pixel evaluation    linear intensity interpolation



# Per-Pixel Lighting Algorithm

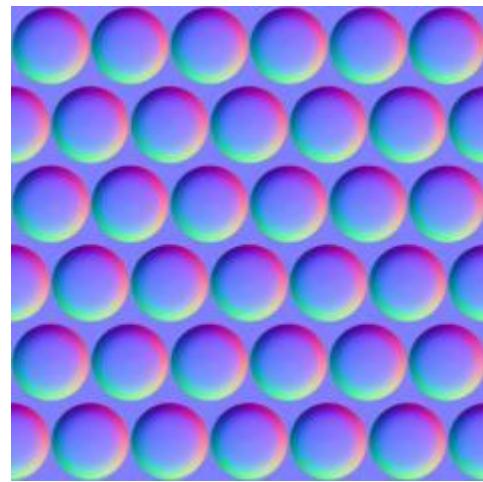
- Done in fragment shader programs
- Vertex normals are passed to fragment shader
- Rasterizer interpolates them

Phong-Shading

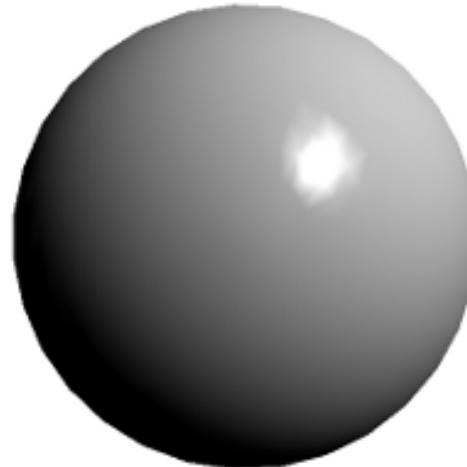


# Bump Mapping

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



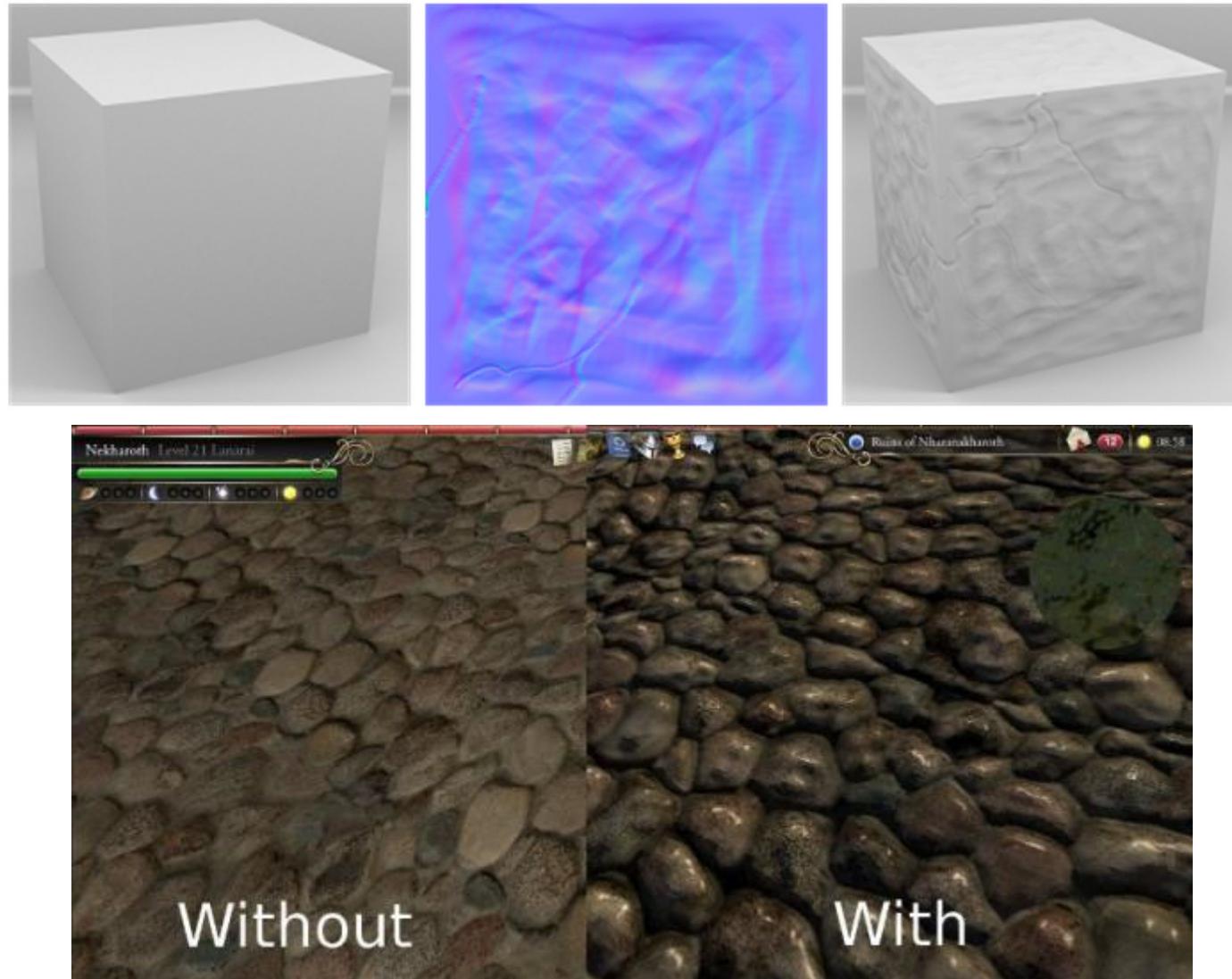
+



=

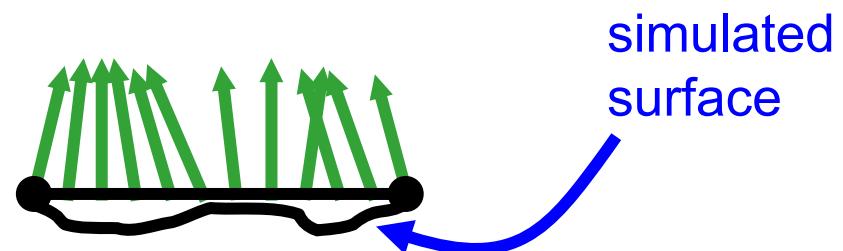


# Bump Mapping Examples



# 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

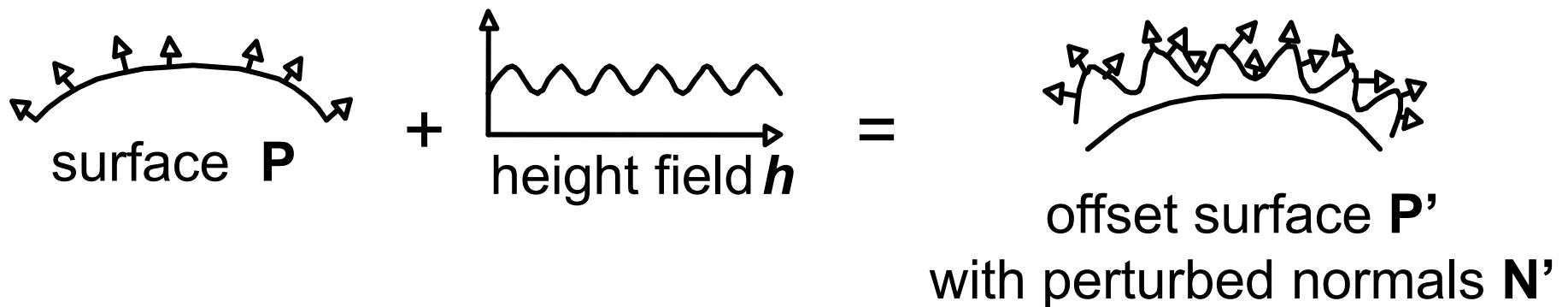


# Per-Pixel vs. Bump Mapping

- Where is the difference?
- Per-pixel lighting = bump mapping *without the bump map!*
- Per-pixel lighting is a special case where normal vector is interpolated instead of looked up

# 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:

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

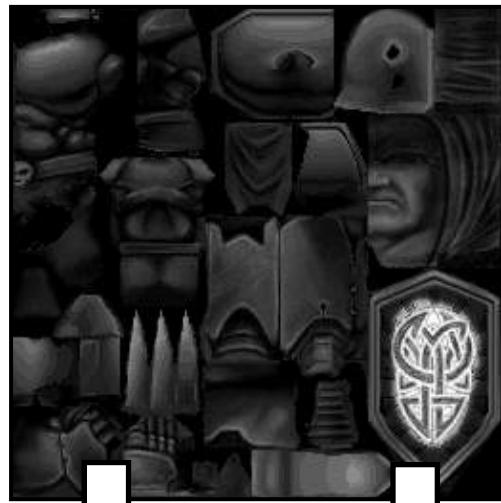
= 0 for locally flat base surface

after some computation...

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

# Multitexturing Example

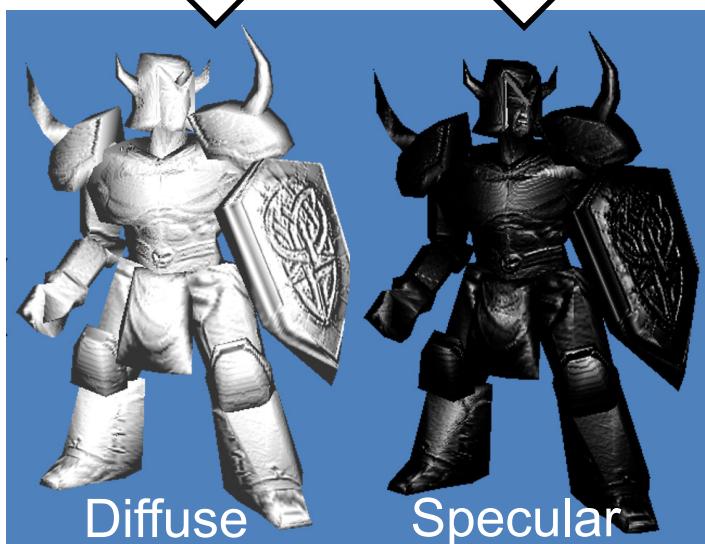
Bump Map



Diffuse Color



Per triangle  
gloss value



# Multitexturing Result



Note: Gloss map  
defines where to  
apply specular



**Final  
result!**

# 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

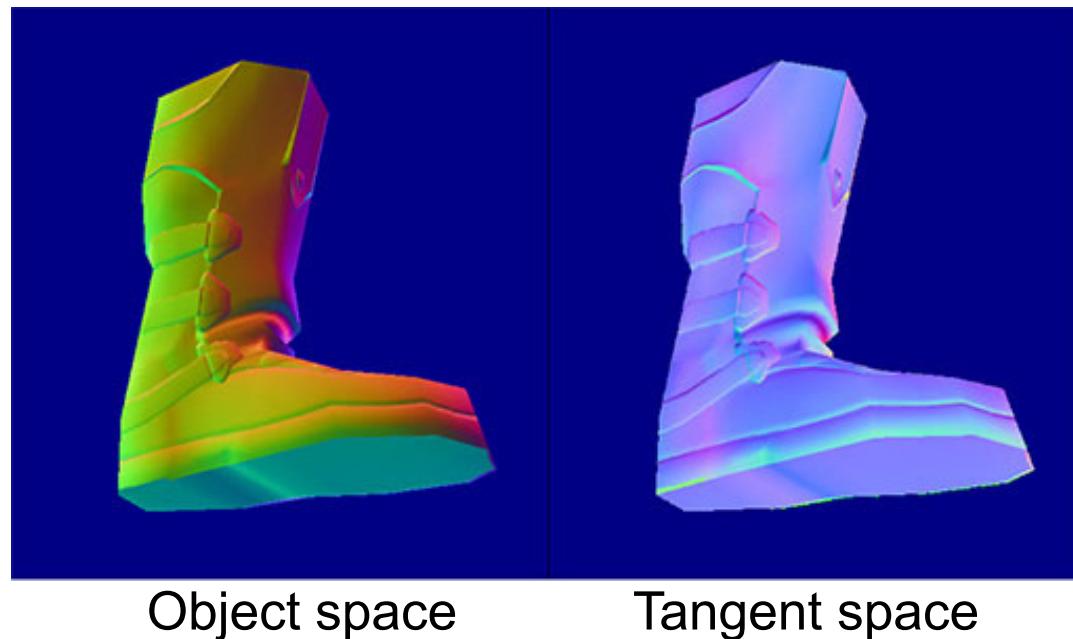
# Coordinate Systems

Problem: where to calculate lighting?

- **Object coordinates**
  - Native space for object coordinates ( $P$ )
- **World coordinates**
  - Native space for light vector ( $\mathbf{l}$ ), env-maps
  - Not explicit in OpenGL
- **Eye Coordinates**
  - Native space for view vector ( $\mathbf{v}$ )
- **Tangent Space**
  - Native space for bump/normal maps ( $\mathbf{n}$ )

# Tangent Space

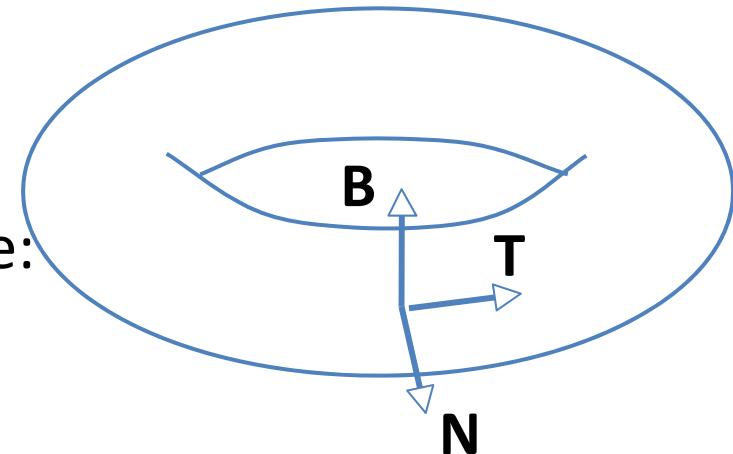
- Tangent space is **most often used**
- This variant is called **normal mapping**
- Normal maps usually blue-ish in RGB space, because perturbed normals are still mostly up-facing (0,0,1)



# Tangent Space Basics

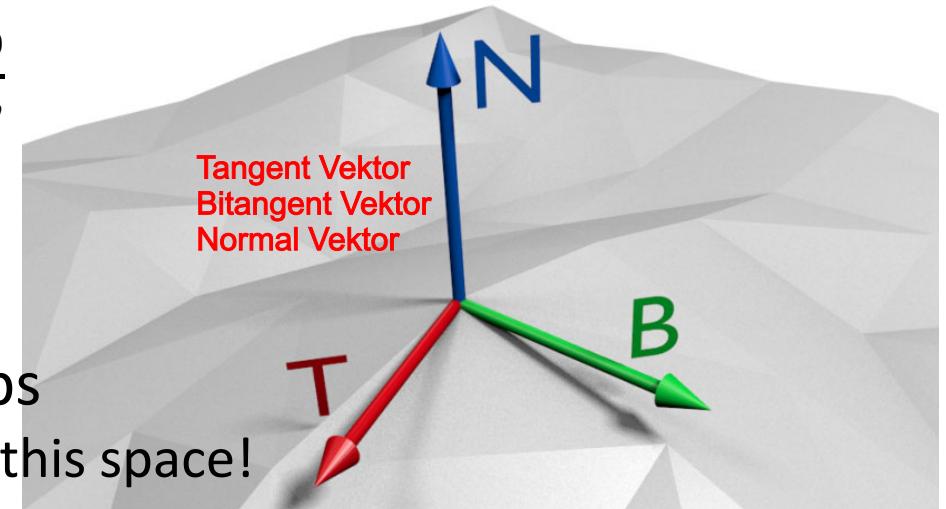
- Concept from differential geometry
- Set of all tangents on a surface
- Orthonormal coordinate system (frame) for each point on the surface:

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



$$\mathbf{t} = \frac{\partial \mathbf{p}}{\partial u} \quad \mathbf{b} = \frac{\partial \mathbf{p}}{\partial v}$$

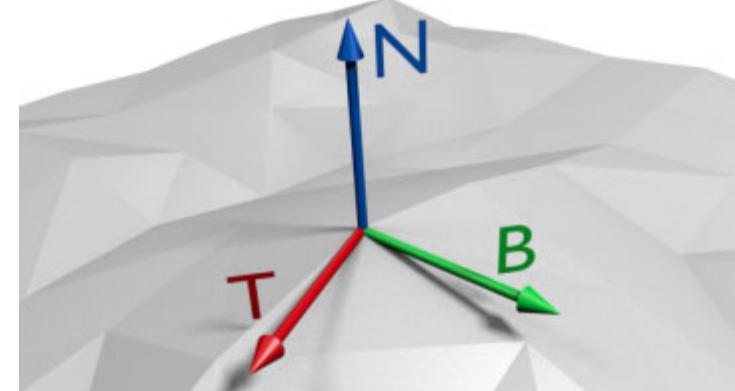
- Square patch assumption:
- $$\mathbf{b} = \mathbf{n} \times \mathbf{t}$$
- (watch out for handedness!)
- A natural space for normal maps
    - Vertex normal  $\mathbf{n} = (0, 0, 1)^T$  in this space!



# Tangent Space Matrix

- Every vertex needs these three vectors
  - E.g., in attribute arrays
- They are interpolated for each fragment
- Fragment program can set up tangent space matrix **TSM**

$$\mathbf{TSM} = \begin{pmatrix} T_x & B_x & N_x & 0 \\ T_y & B_y & N_y & 0 \\ T_z & B_z & N_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



# Tangent Space Algorithm

- For each vertex
  - Transform  $\mathbf{I}$  and  $\mathbf{v}$  with **TSM** and normalize
  - Compute normalized  $\mathbf{h}$  from  $\mathbf{v}$  and  $\mathbf{I}$
- For each fragment
  - Interpolate  $\mathbf{I}$  and  $\mathbf{h}$
  - Renormalize  $\mathbf{I}$  and  $\mathbf{h}$
  - Fetch  $\mathbf{n}' = \text{texture}(u,v)$
  - Compute  $\max(\mathbf{I} \cdot \mathbf{n}', 0)$  and  $\max(\mathbf{h} \cdot \mathbf{n}', 0)^s$
  - Combine using standard Phong equation

Der Tangent Space Algorithmus wird verwendet, um Normalenvektoren auf einer 3D-Oberfläche zu berechnen. Ein Normalenvektor ist ein Vektor, der senkrecht auf der Oberfläche eines Objekts verläuft und die Richtung anzeigt, in die die Oberfläche ausgerichtet ist. Diese Normalenvektoren sind für die Berechnung von Beleuchtung und Schattierung erforderlich, da sie bestimmen, wie Licht auf ein Objekt trifft und wie es reflektiert wird. Der Tangent Space Algorithmus berechnet die Normalenvektoren an einem gegebenen Punkt auf der Oberfläche, indem er die lokale Tangentialebene an diesem Punkt bestimmt und dann den Normalenvektor auf dieser Ebene berechnet.

# Reflective Bump Mapping

- EMBM (Environment Map Bump Mapping)



Dieter Schmalstieg



Texture mapping

# EMBM in World Space

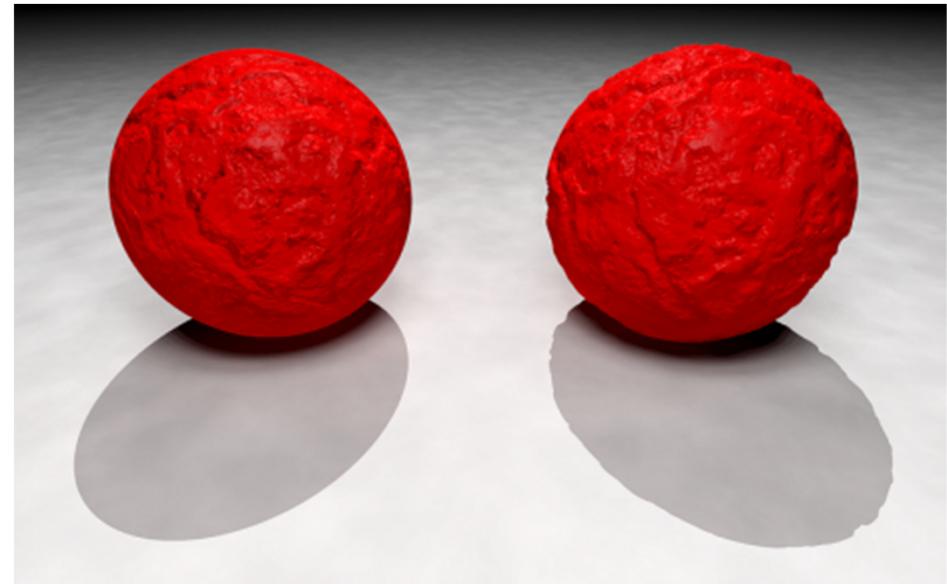
- For each vertex
  - Transform  $v$  to world space
  - Compute tangent space to world space transform ( $t, b, n$ )
- For each fragment
  - Interpolate and renormalize  $v$
  - Interpolate frame ( $t, b, n$ )
  - Lookup  $n' = \text{texture}(u, v)$
  - Transform  $n'$  from tangent space to world space
  - Compute reflection vector  $r$  (in world space) using  $n'$
  - Lookup  $c = \text{cubemap}(r)$
- Note: this is an example of dependent texturing

Environment Map Bump Mapping ist eine Technik in der Computergrafik, die verwendet wird, um die Illusion von Reliefstrukturen auf einer Fläche zu erzeugen. Es ermöglicht es, eine einzelne Textur auf eine Oberfläche zu projizieren und die Illusion von Tiefe und Schatten zu erzeugen.

Es funktioniert indem man eine Umgebungskarte (Environment Map) verwendet, die ein Abbild der Umgebung um das Objekt herum enthält und diese Karte wird auf die Oberfläche des Objekts projiziert. Dann wird eine Bump Map verwendet, die Höheninformationen enthält. Diese Höheninformationen werden verwendet, um die Normale der Oberfläche zu verändern und somit die Illusion von Reliefstrukturen zu erzeugen.

# 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



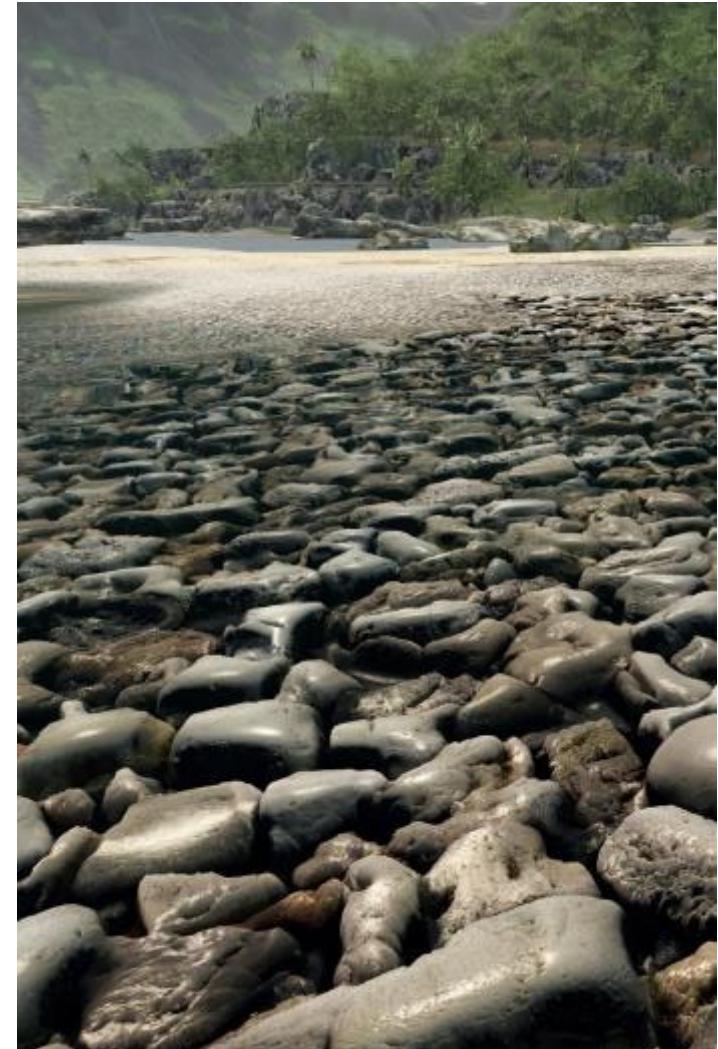
Als Parallaxe) bezeichnet man die scheinbare Änderung der Position eines Objektes durch verschiedene Positionen des Beobachters.

# Parallax Mapping

- Also called *Relief Mapping*
- Enhanced bump mapping
  - Texture lookup is displaced according to viewing angle

Parallax Mapping ist eine Technik in der Computergrafik, die verwendet wird, um Tiefeneffekte auf 2D-Texturen zu erzeugen. Es ermöglicht es, die Illusion von Tiefe und Volumen in 2D-Oberflächen zu erzeugen, indem es Schichten von Texturen verwendet, die auf verschiedenen Ebenen angeordnet sind. Dies ermöglicht es, die Tiefenillusion zu erzeugen, ohne dass eine 3D-Modellierung erforderlich ist. Es wird häufig in Spielen und anderen Anwendungen verwendet, um die visuelle Qualität zu verbessern.

## Example: Crysis



# Parallax Mapping Example

- Allows for more realism with motion parallax and steeper height maps with occlusions

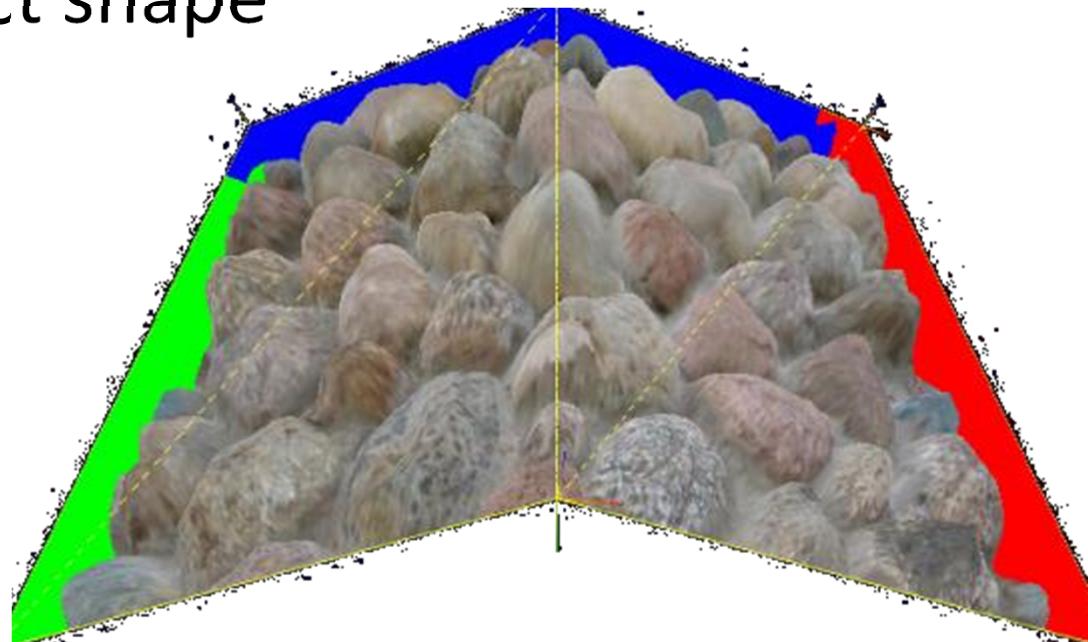


[https://commons.wikimedia.org/w/index.php?title=File%3AParallax\\_mapping.ogv](https://commons.wikimedia.org/w/index.php?title=File%3AParallax_mapping.ogv)

<https://youtu.be/6PpWqUqeqeQ>

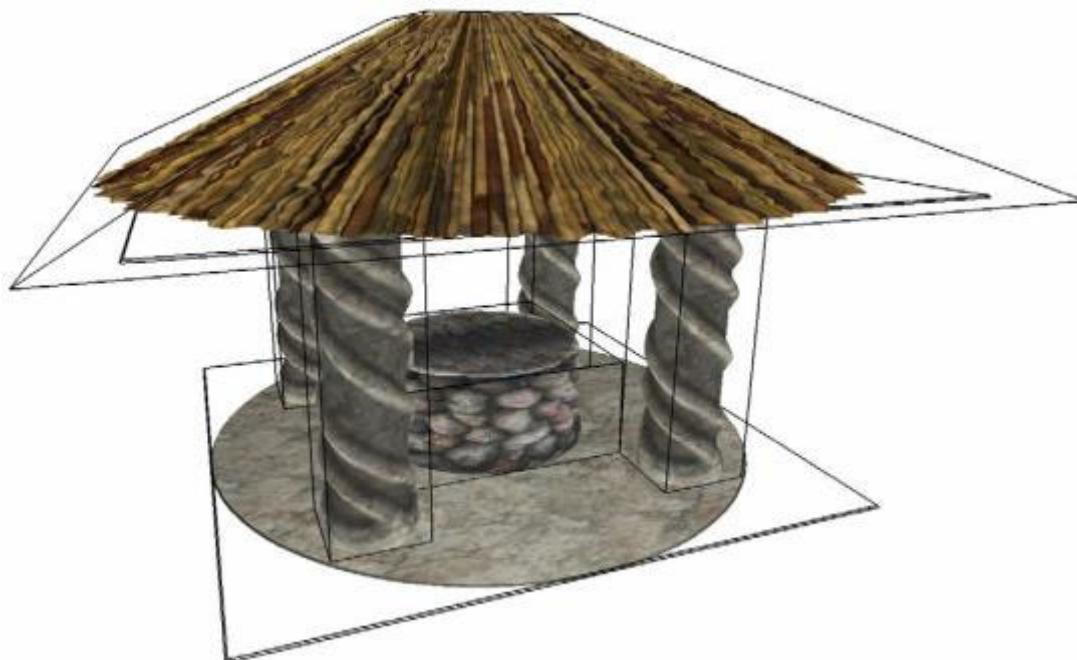
# Parallax Mapping Procedure

- Parallax occlusion mapping uses raycasting in the fragment shader
- Expensive, but silhouettes and shadows have correct shape



# Parallax Mapping Discussion

- Can replace almost all geometry with good relief maps
- Result: a raycasting engine



Dieter Schmalstieg

Texture mapping



110

# Thank You!

## Questions?



# Diligent Engine Texture Example

[https://github.com/DiligentGraphics/DiligentSamples/tree/master/Tutorials/Tutorial03\\_Texturing](https://github.com/DiligentGraphics/DiligentSamples/tree/master/Tutorials/Tutorial03_Texturing)



# Vertex Shader

```
cbuffer Constants { float4x4 g_WorldViewProj;  
};  
struct VSInput { float3 Pos : ATTRIB0;  
                float2 UV : ATTRIB1; // texture coordinates in  
};  
struct PSInput { float4 Pos : SV_POSITION;  
                float2 UV : TEX_COORD; // texture coordinates out  
};  
void main(in VSInput VSIn, out PSInput PSIn) {  
    PSIn.Pos = mul(float4(VSIn.Pos,1.0), g_WorldViewProj);  
    PSIn.UV = VSIn.UV; // just copy the input  
}
```

# Fragment Shader

```
Texture2D g_Tex; // texture object
SamplerState g_sampler; // sampler object
struct PSInput { float4 Pos : SV_POSITION;
                  float2 UV : TEX_COORD;
};
struct PSOutput { float4 Color : SV_TARGET; };
void main(in PSInput PSIn, out PSOutput PSOut) {
    PSOut.Color = g_Tex.Sample(g_sampler, PSIn.UV);
}
```

# Initializing the Pipeline State

- We need to define mutable resource variable `g_texture`

```
ShaderResourceVariableDesc Vars[] = {
    { SHADER_TYPE_PIXEL, "g_Texture", SHADER_RESOURCE_VARIABLE_TYPE_MUTABLE }
};
```

- Pass it to pipeline state object

```
PSOCreateInfo.PSODesc.ResourceLayout.Variables      = Vars;
PSOCreateInfo.PSODesc.ResourceLayout.NumVariables = _countof(Vars);
```

- Define an immutable sampler (sampling does not change)

```
SamplerDesc SamLinearClampDesc {
    FILTER_TYPE_LINEAR, FILTER_TYPE_LINEAR, FILTER_TYPE_LINEAR,
    TEXTURE_ADDRESS_CLAMP, TEXTURE_ADDRESS_CLAMP, TEXTURE_ADDRESS_CLAMP
};
ImmutableSamplerDesc ImtblSamplers[] = {
    { SHADER_TYPE_PIXEL, "g_Texture", SamLinearClampDesc }
};
PSOCreateInfo.PSODesc.ResourceLayout.ImmutableSamplers = ImtblSamplers;
PSOCreateInfo.PSODesc.ResourceLayout.NumImmutableSamplers= _countof(ImtblSamplers);
```

# Load Texture and Bind It

- Diligent loads jpg, png, tiff

```
TextureLoadInfo loadInfo;  
loadInfo.IsSRGB = true;  
RefCntAutoPtr<ITexture> Tex;  
CreateTextureFromFile("DGLogo.png", loadInfo, m_pDevice, &Tex)
```

- Get shader resource view from the texture

```
m_TextureSRV = Tex->GetDefaultView(TEXTURE_VIEW_SHADER_RESOURCE);
```

- Create shader resource binding object from PSO

```
m_pPSO->CreateShaderResourceBinding(&m_SRB, true);
```

- Set shader resource view

```
m_SRB->GetVariableByName(SHADER_TYPE_PIXEL, "g_Texture")->Set(m_TextureSRV);
```

# Render Textured Object

- ... (set vertex buffer etc. omitted here)
- Select our prepared pipeline

```
m_pImmediateContext->SetPipelineState(m_pPSO);
```

- Commit the shader resources

```
m_pImmediateContext->CommitShaderResources(m_SRB, RESOURCE_STATE_TRANSITION_MODE_TRANSITION);
```

- Draw Call (here: indexed draw)

```
DrawIndexedAttribs DrawAttrs = {...}; // This is an indexed draw
m_pImmediateContext->DrawIndexed(DrawAttrs);
```

# 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

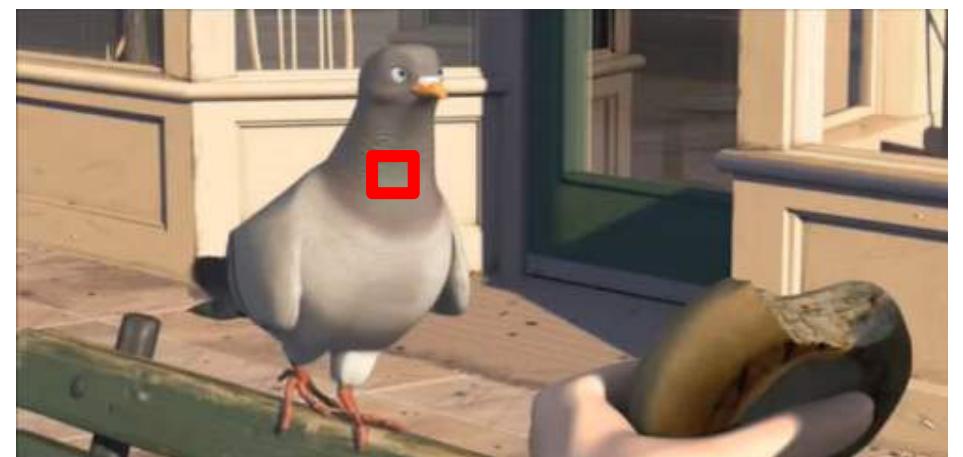
- Complex scenes, large virtual environments → overdraw
- Overdraw → redundant calculations
- Why shade a pixel, if it gets overdrawn in final image?
- Idea: perform shading at the end of rendering  
→ we need only one shading operation per pixel

Intermediate:



Dieter Schmalstieg

Final:

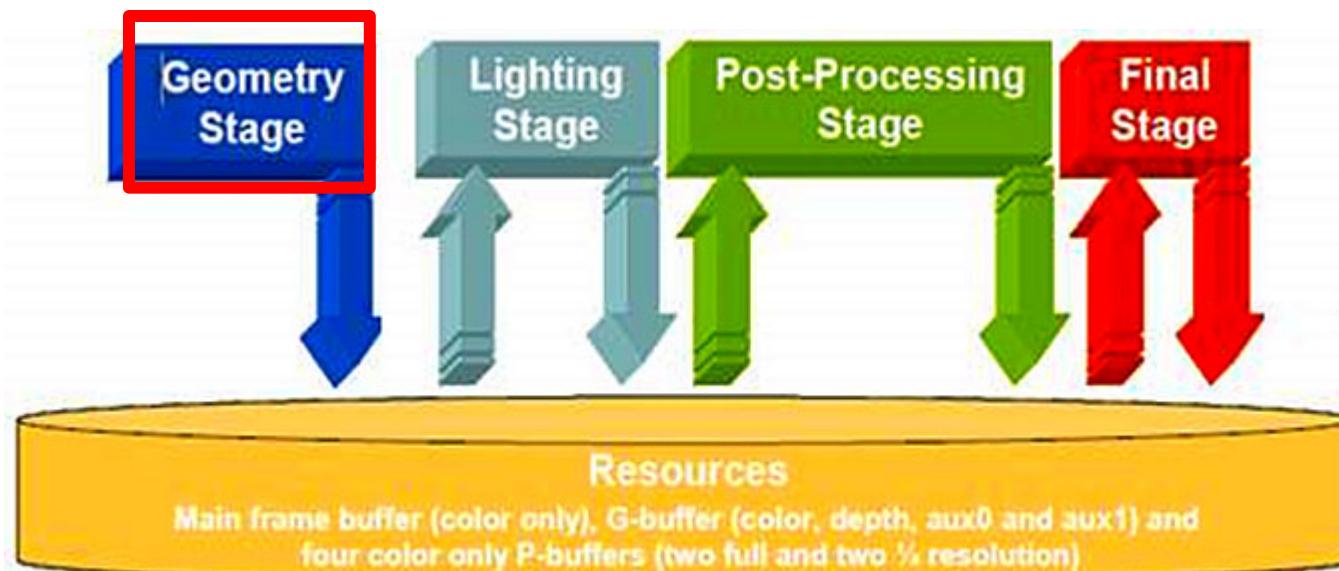


Deferred Shading

# Basic Deferred Rendering

Split rendering pipeline in two separate stages

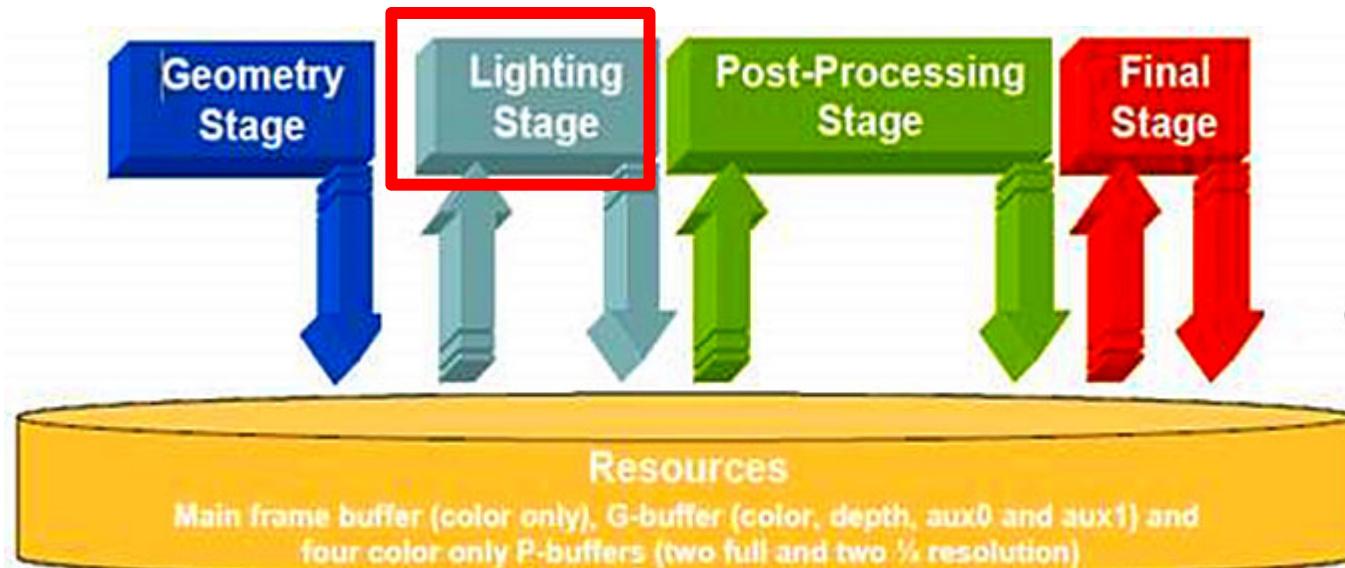
1. Geometry transformation and rasterization



# Lighting and Shading Stage

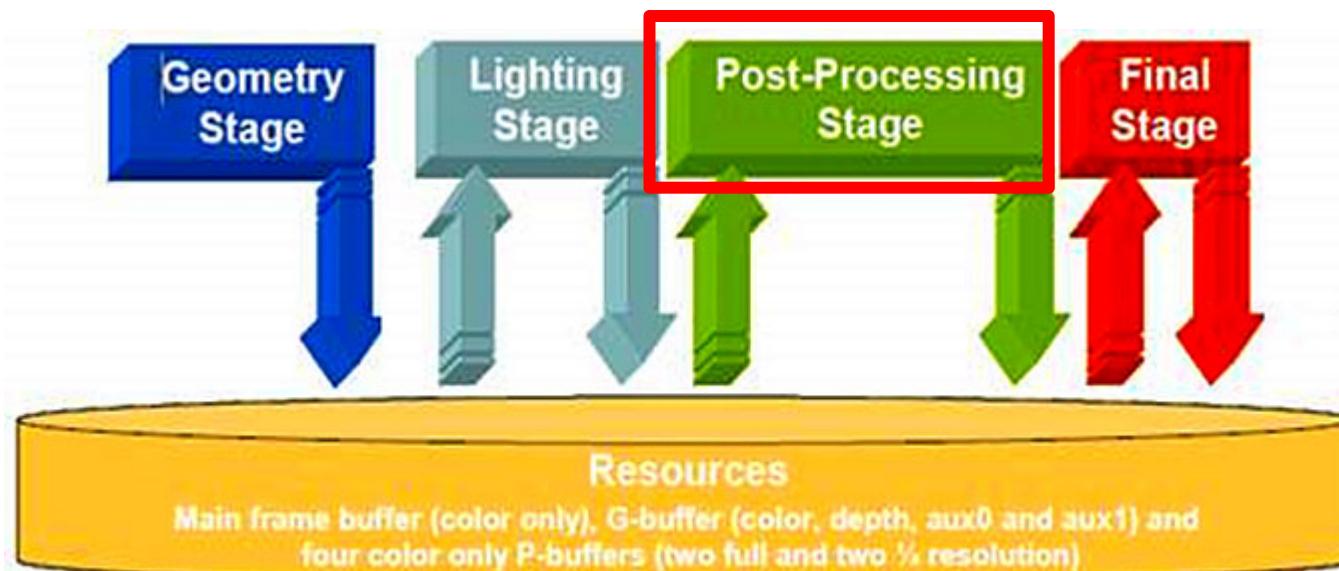
Split rendering pipeline in two separate stages

1. Geometry transformation and rasterization
2. Shading



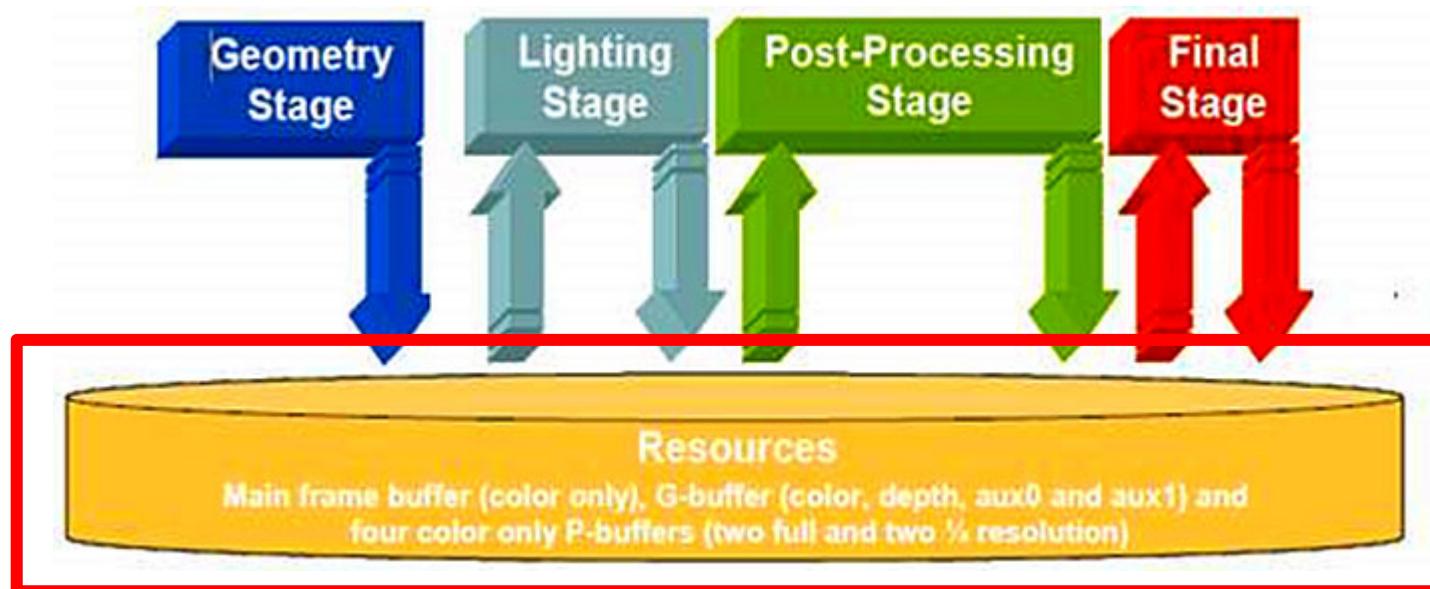
# Post-Processing

- In an optional final stage, apply image post-processing



# G-Buffers

- G-Buffers pass data from one stage to next



# Types of G-Buffers

G-Buffers store per pixel:

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

Normal



Color



Identity



Result

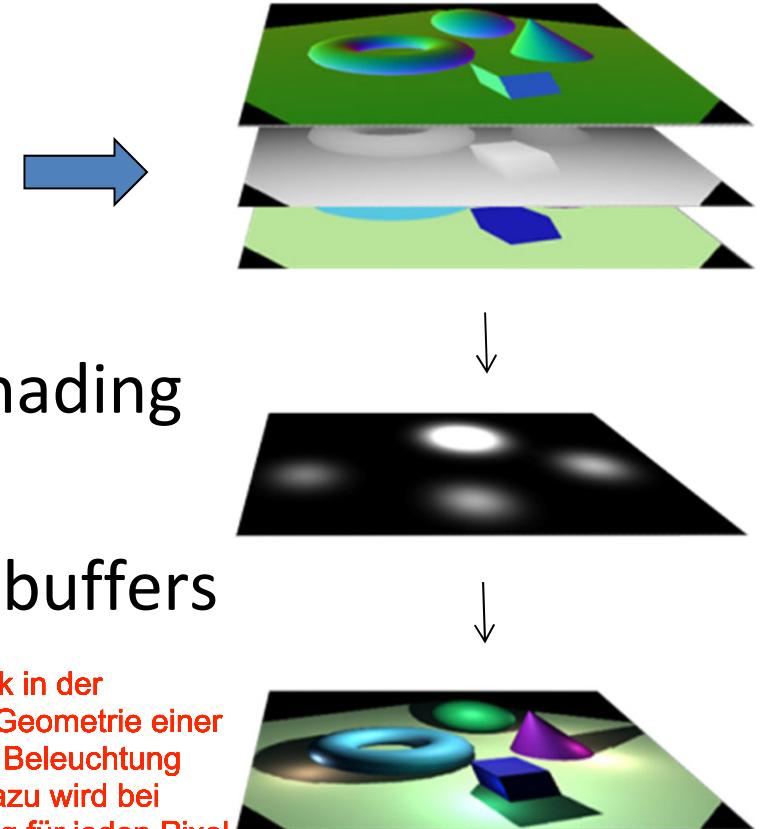


# Deferred Rendering Overview

- 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

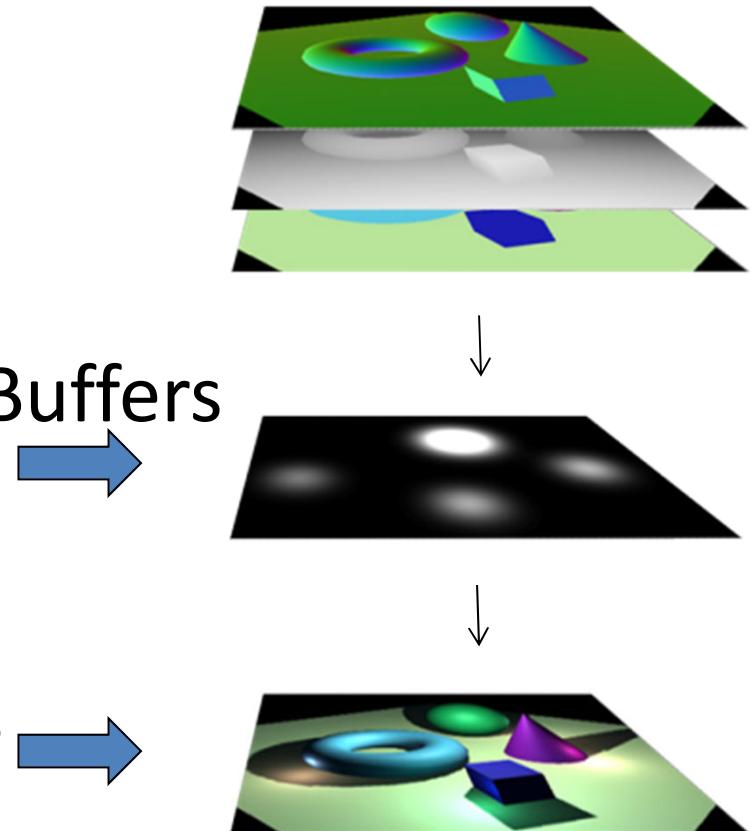
- Geometry stage as usual
  1. Geometry transformations
  2. Rasterization
  3. Material, texturing, but no shading
- Instead of shading,  
store intermediate results in G-buffers
  - Diffuse color
  - Depth
  - Position
  - Normal vectors



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.

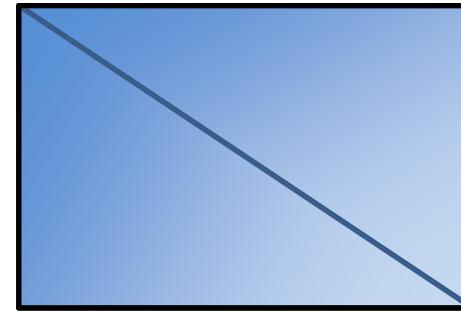
# Actual Shading Stage

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

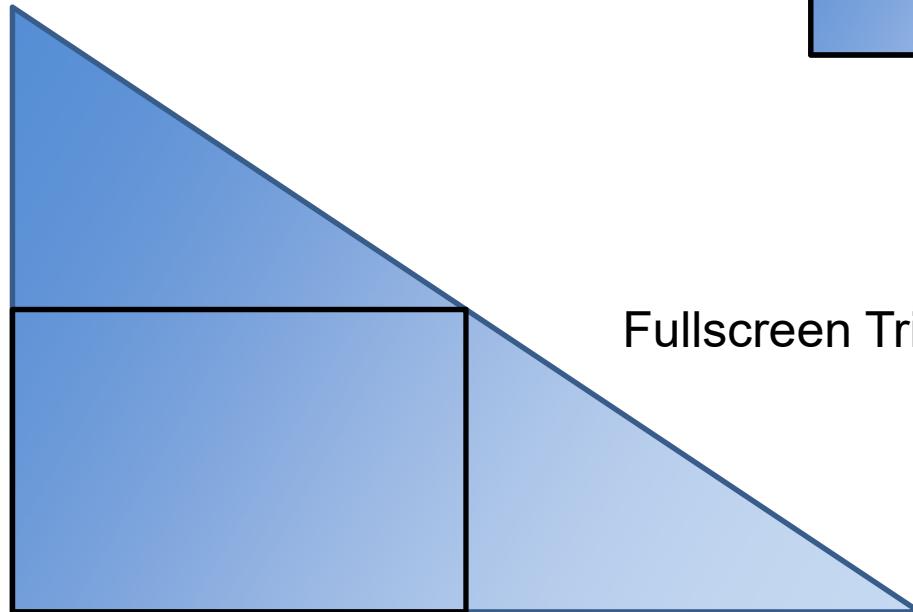


# Screen-filling Primitives

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

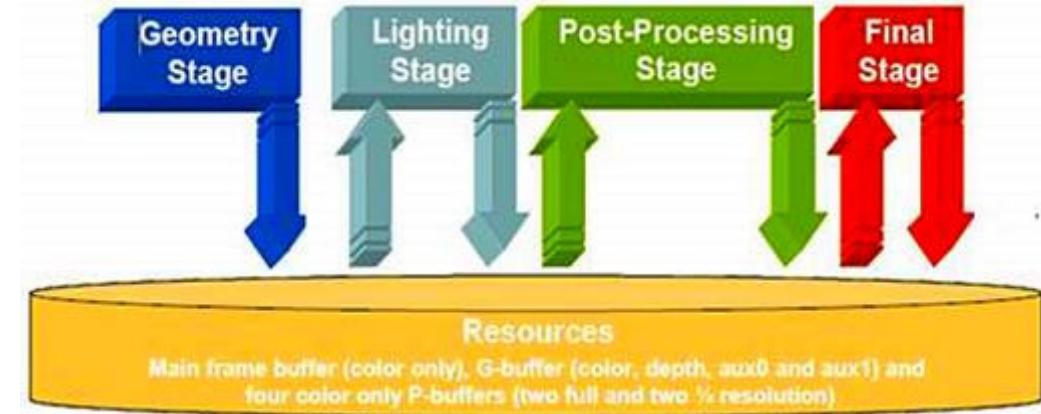
# Deferred Rendering Disadvantages

- 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**

# Motivation



- Utilize deferred rendering framework to add special effects *in image space* after rendering
- Independent of geometric scene complexity
- Often uses image processing techniques
- Can operate on G-buffer (not just color buffer)
- Can composite *many* G-buffers (e.g., per object)

# Postprocessing

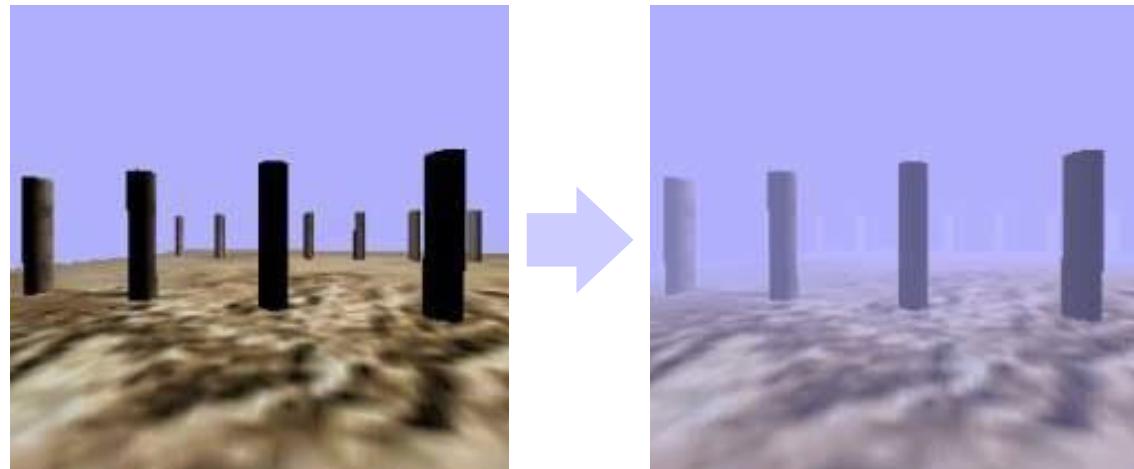


- **Postprocessing Idea**

- Run a fragment shader for every pixel
  - Apply some modification to pixel

- Example: fog

- Saturation decreases with distance to camera



# Distance Fog

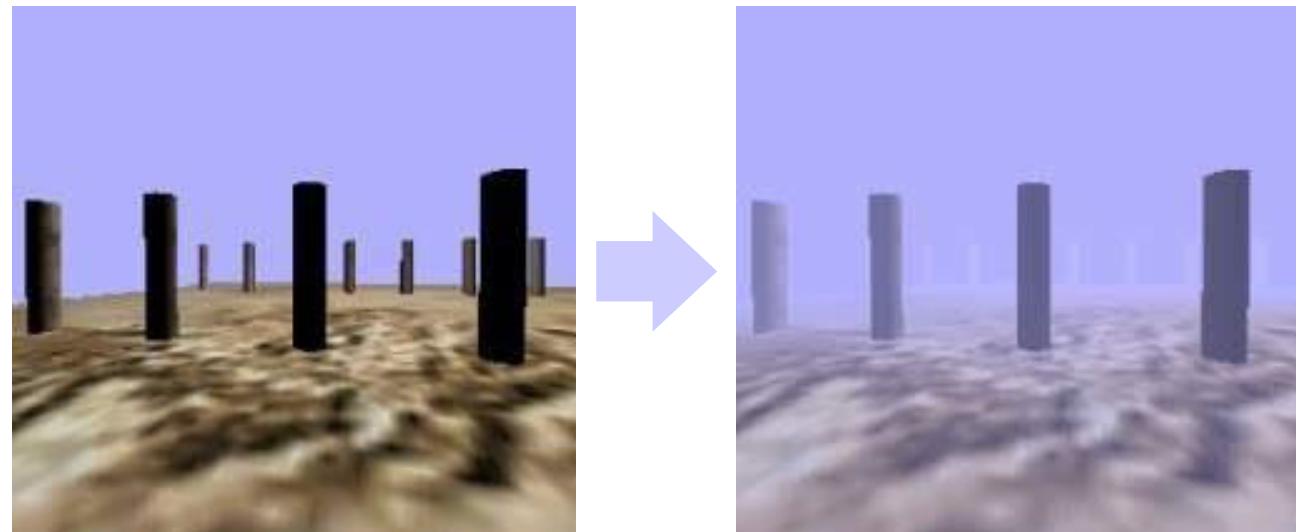
- Blend surface color with fog color

$$\mathbf{c} = f \mathbf{c}_s + (1 - f) \mathbf{c}_f$$

$\mathbf{c}_s$  surface color

$\mathbf{c}_f$  fog color

$f$  fog factor

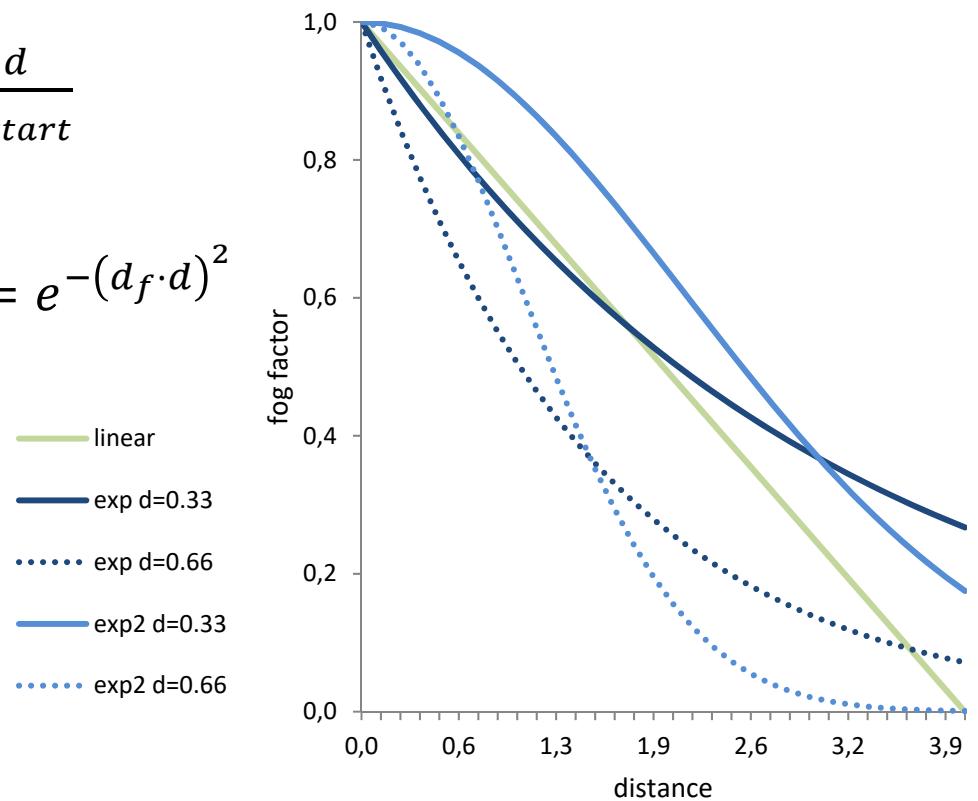


# Fog Function

- Blend surface color with fog color:  $\mathbf{c} = f\mathbf{c}_s + (1 - f)\mathbf{c}_f$   
 $\mathbf{c}_s$  surface color,  $\mathbf{c}_f$  fog color,  $f$  fog factor

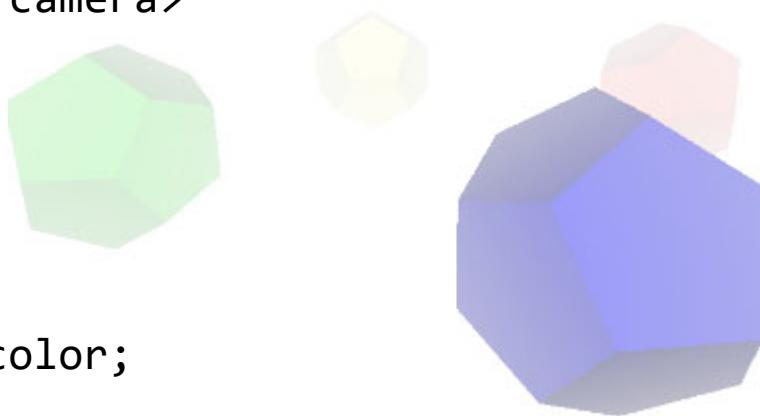
- Linear fog:  $f = \frac{d_{end} - d}{d_{end} - d_{start}}$
- Exponential fog:  $f = e^{-d_f \cdot d}$
- Squared exponential fog:  $f = e^{-(d_f \cdot d)^2}$

$d$  fragment distance  
 $d_{start}$  fog start  
 $d_{end}$  fog end  
 $d_f$  fog density



# Distance Fog – Implementation

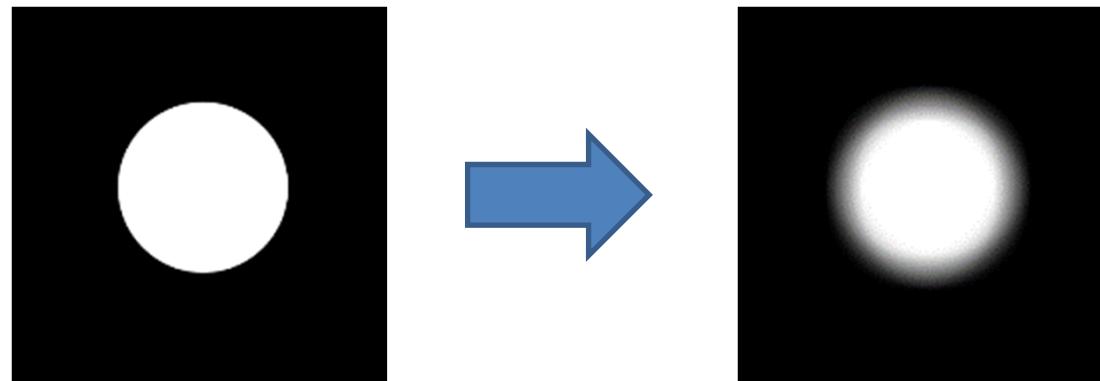
```
1 #version 330
2 #include <framework/utils/GLSL/camera>
3 uniform vec3 c_d;
4 uniform vec3 c_f;
5 uniform float d_f;
6
7 in vec3 p; // surface position
8 in vec3 c_s; // surface
9
10 layout(location = 0) out vec4 color;
11
12 void main()
13 {
14     vec3 v = camera.position - p;
15     float d = length(v);
16     float f = exp(-d_f * d);
17     color = vec4(f * c_s + (1.0f - f) * c_f, 1.0f);
18 }
```



# Image Processing



- Image Processing Idea
  - Compute some derivative function per pixel
- Example: Gaussian blur



# Gaussian Filter

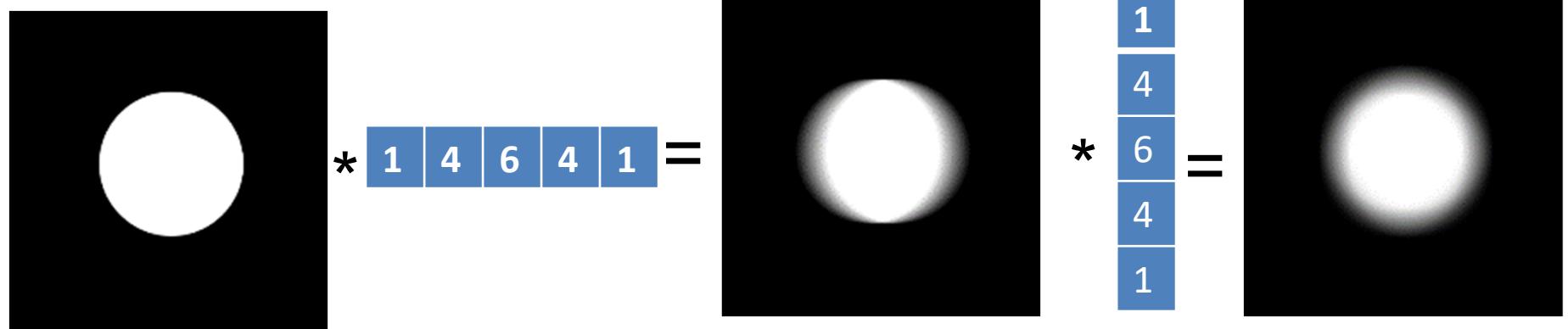
- Many effects based on Gaussian filter
- 5x5 Gaussian filter requires 25 texture lookups
  - Too slow and too expensive

\* 1/25

- But: Gaussian filter is separable!

# Separable Gaussian Filter

- Separate 5x5 filter into 2 passes
- Perform 5x1 filter in  $u$
- Followed by 1x5 filter in  $v$



- Lookups can be performed via linear filtering
  - 5x1 filter with 3 lookups

# Separable Box Blur Shader

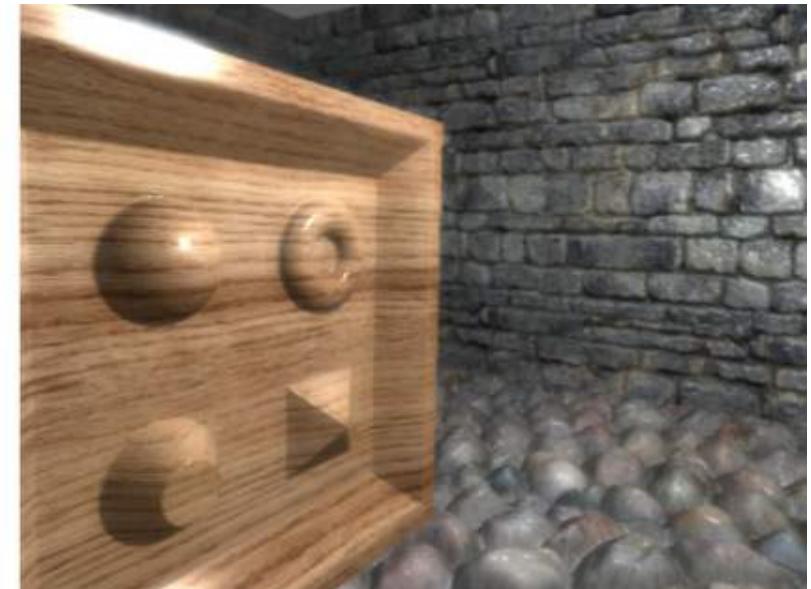
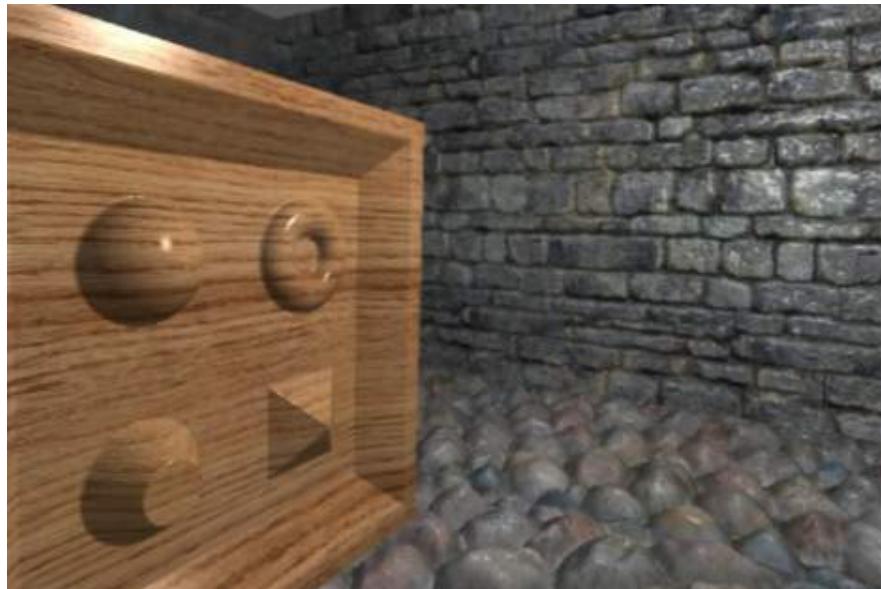
```

#version 330
uniform sampler2D color_buffer;
uniform float texel_size;
const int num_samples = 32;
#ifdef BLUR_Y
const vec2 dir = vec2(0.0f, 1.0f);
#else
const vec2 dir = vec2(1.0f, 0.0f);
#endif
in vec2 t; // incoming location
layout(location = 0) out vec4 color;
void main()
{
    vec4 c = vec4(0.0f, 0.0f, 0.0f, 0.0f);
    for (int i = 0; i < num_samples; ++i)
    {
        vec2 d = dir * texel_size * ((i - num_samples / 2) + 0.5f);
        c += texture(color_buffer, t + d) * (1.0f / num_samples);
    }
    color = c;
}

```

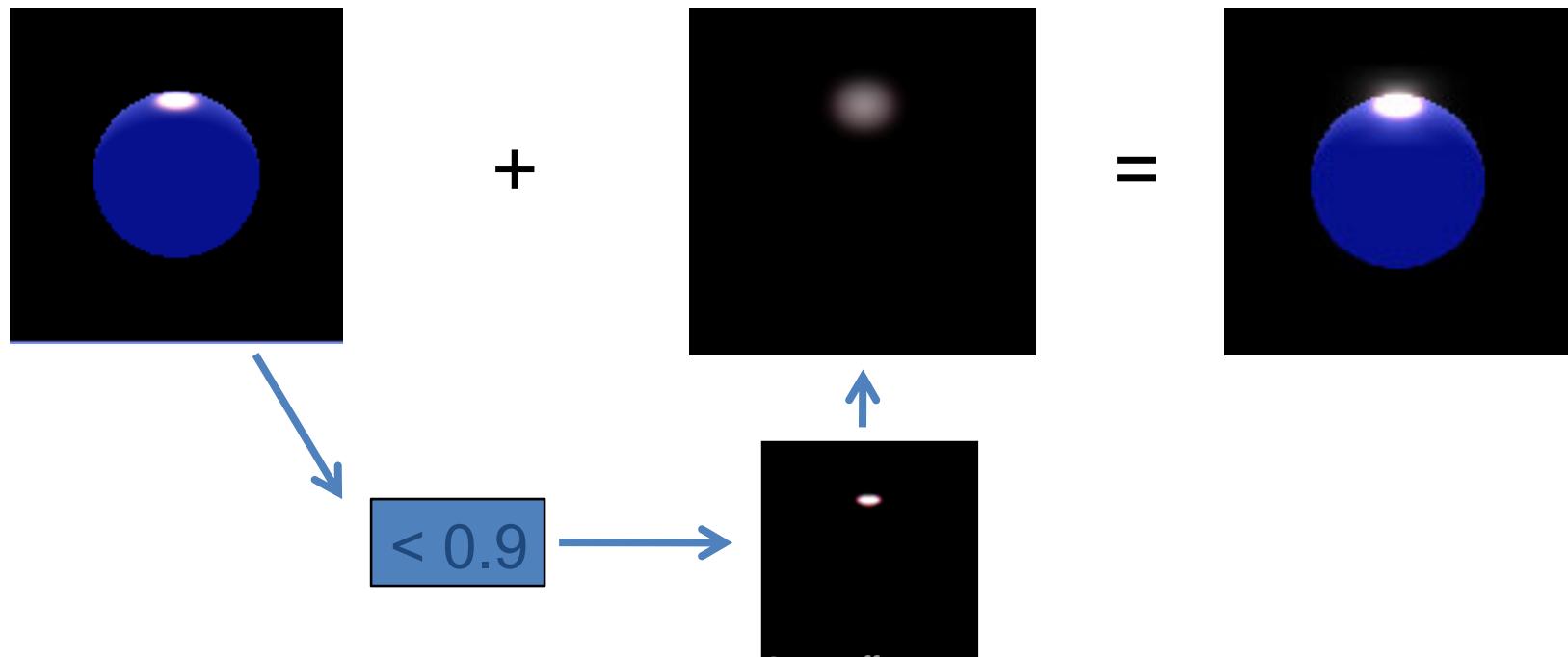
# Bloom

- Bright areas seem to “spill” light to vicinity streuen
- Apply blur to simulate this effect Umgebung



# Bloom

- Before Gaussian filtering
  - Modify rendered texture intensities
  - Clamp or glowing-object-only pass
  - Exponential weight
- Add filtered image to original image



# Bloom Discussion

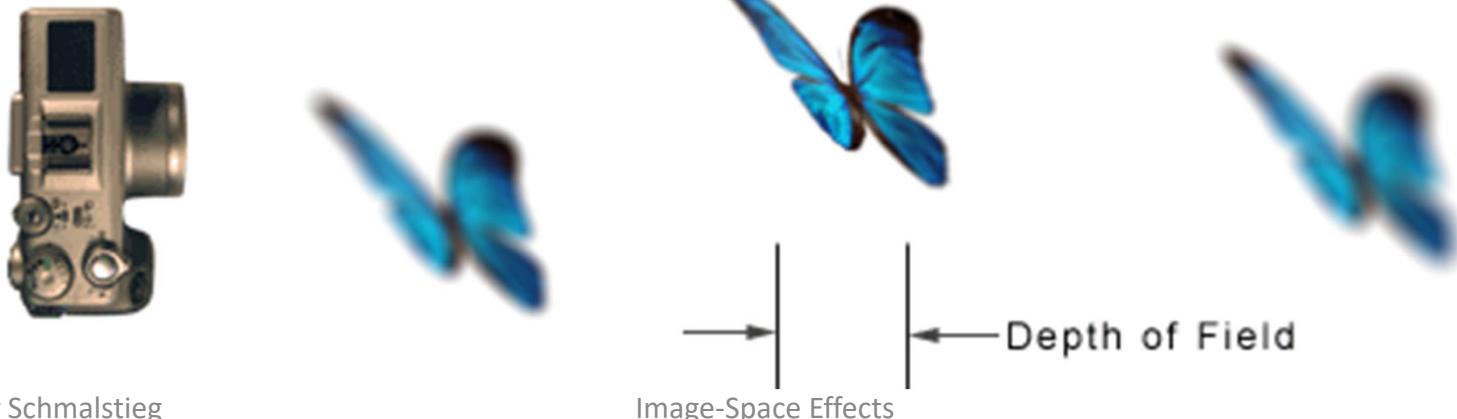
- Usually applied to downsampled render textures
  - 2x or 4x downsampled
  - Effectively increases kernel size
  - But: sharp highlights are lost
  - Combination of differently downsampled and filtered render textures possible
  - Allows good control of bloom effect
- Star effect 
  - Filter in  $u$  and  $v$  separately
  - Add  $u$  and  $v$  texture separately

# Bloom Remarks

- Disguises aliasing artifacts
- Works best for shiny materials and sun/sky
  - Only render sun and sky to blur pass
  - Only render specular term to blur pass
- A little bit overused these days
  - Use sparsely for most effect
- Can smudge out a scene too much
  - Contrast and sharp features are lost (fairytales look)

# Depth of Field

- DoF simulates camera property
  - Lens can only focus on one depth level
- Objects around that depth level appear sharp
- Rest is blurred, depending on distance to focal plane



# Depth of Field Example



# Depth of Field Application

- Guide the user's attention towards something



# Postprocessing Depth of Field

- For each pixel in fragment shader
  - Compute the circle of confusion (CoC) based on depth buffer
  - Blur image using convolution
  - Window size depends on the CoC
- Problem
  - Sharp foreground objects leak on blurry background
- Solution
  - Compare depth values, discard closer pixels
  - We need *depth-aware filtering*

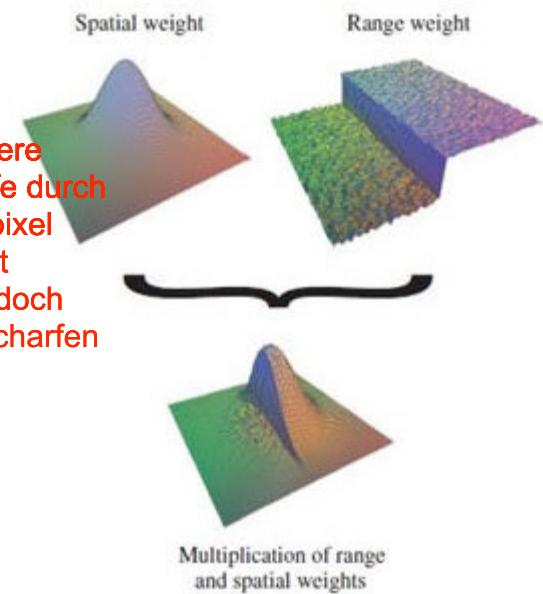


# Bilateral Filter



- Gaussian filter (or bilinear filter)
  - Only spatial weight
  - Always blurry
- Bilateral filter
  - Consider also colors (range weight)
  - Similar colors weighted stronger
  - Preserves color discontinuities
  - Color edges not blurred

Der Bilinear-Filter ist eine einfachere Methode, bei der die Tiefenschärfe durch Interpolation der Nachbarschaftspixel berechnet wird. Diese Methode ist schneller als der Biliteral-Filter, jedoch weniger präzise und kann zu unscharfen Ergebnissen führen.



Der Biliteral-Filter ist eine aufwendigere Methode, die die Tiefenschärfe durch die Verwendung von zwei Gauss-Glättungen berechnet. Diese Methode ermöglicht es, die Schärfe und Unschärfe in einer Szene präziser zu berechnen und erzeugt daher realistischere Ergebnisse als der Bilinear-Filter. Der Biliteral-Filter ist jedoch auch rechenintensiver und benötigt daher mehr Rechenleistung.

Joint Bilateral Filter ist eine Erweiterung des Bilateral Filters, der sowohl für die Schärfe als auch für die Farbe verwendet wird. Es ist eine Methode in der Computergrafik, die dazu verwendet wird, Bildrauschen zu reduzieren und gleichzeitig die Schärfe des Bildes zu erhalten.

# Joint Bilateral Filter

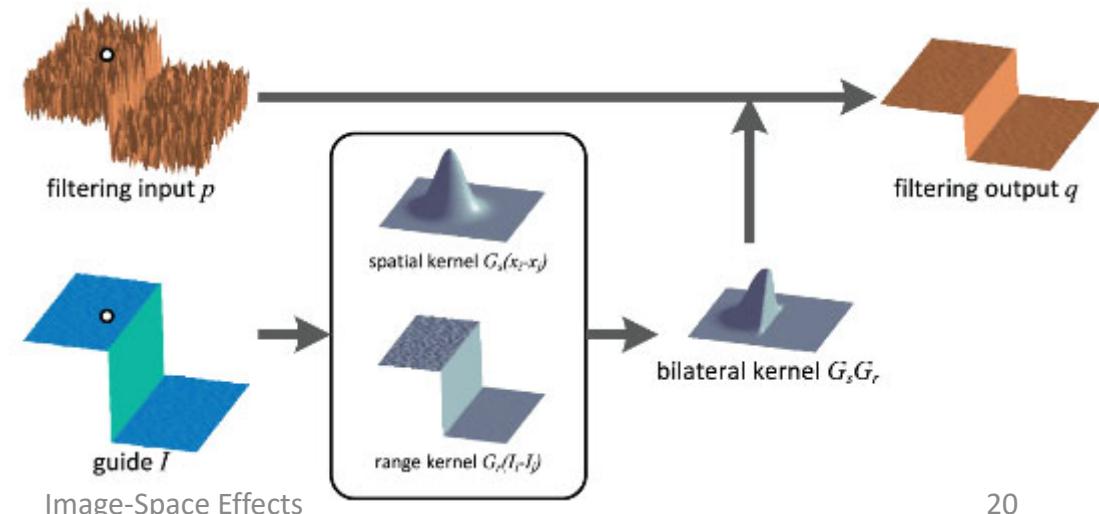


- Bilateral filter with range weights from secondary source channel
- For instance, depth buffer
- Joint bilateral filter avoids bleeding in depth of field

$$q_i = \sum_{j \in N(i)} W_{ij}(I) p_j$$

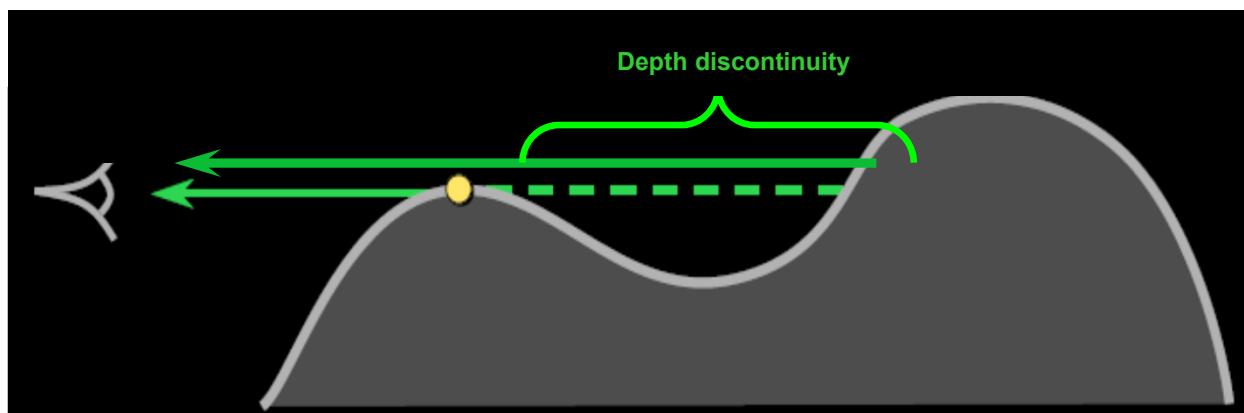
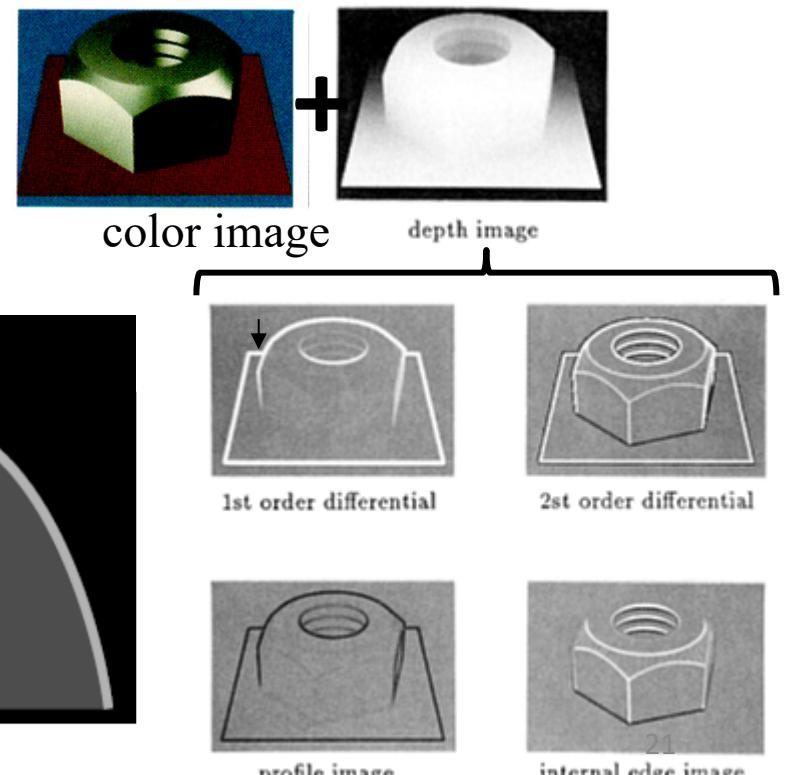
bilateral filter:  $I=p$

Der Joint Bilateral Filter berechnet die Schärfe indem er die räumliche und die Farbdifferenz zwischen Nachbarschaftspixeln verwendet. Es reduziert das Rauschen, indem es nur die Pixel berücksichtigt, die ähnliche Farben und ähnliche Tiefenwerte haben. Es ist auch bekannt als "Joint-Tiefe-Farbe-Filter" und "Cross-Bilateral-Filter".



# Edge Detection in Image Space

- Run filter on depth buffer
- Profile: 1<sup>st</sup> order differential
  - E.g., Sobel operator
- Internal: 2<sup>nd</sup> order differential
  - E.g., Laplace operator



Dieter Schmalstieg

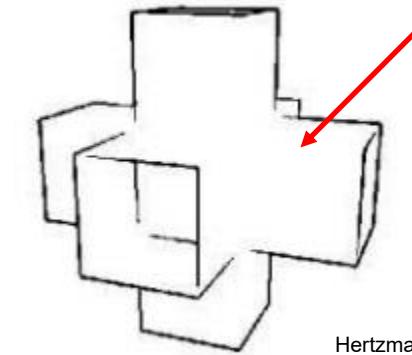
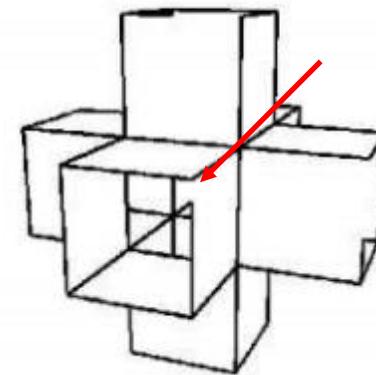
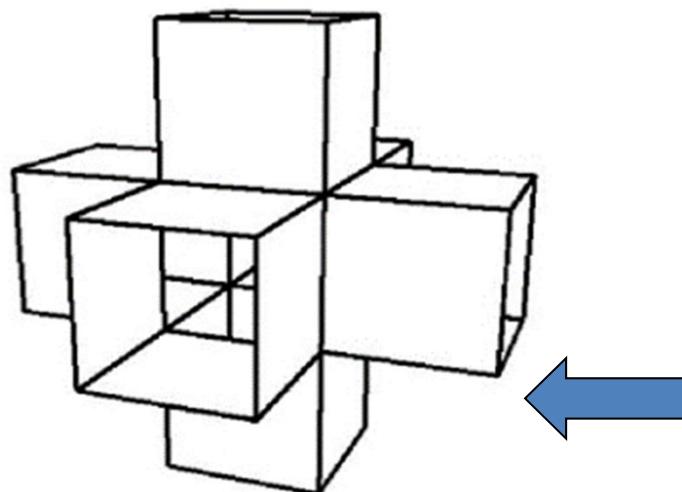
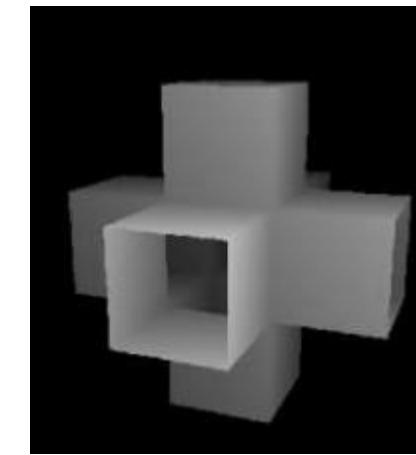
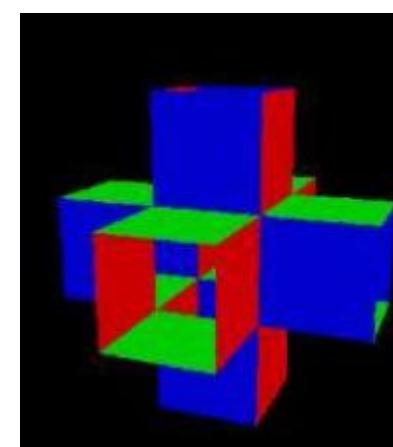
Image-Space Effects

Profile: Kanten  
eines Objekts

Internal: Zusätzliche  
Information (Form?)

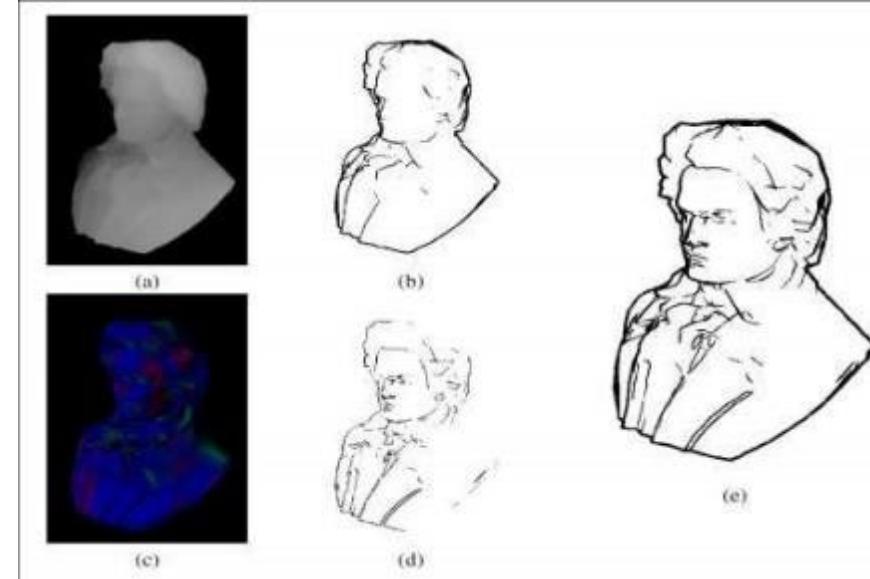
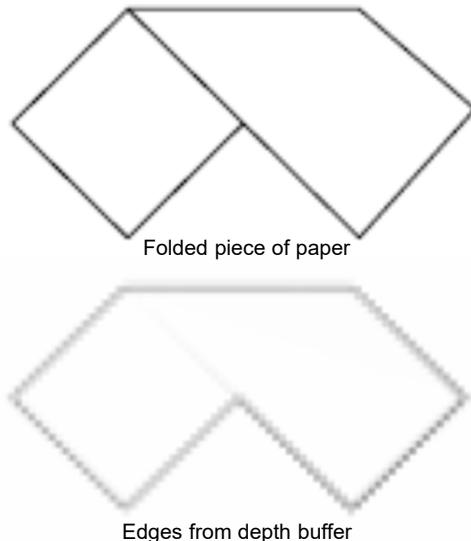
# Edge Detection on G-Buffer

- Make use of all available data
- Detect edges on depth buffer AND normal buffer



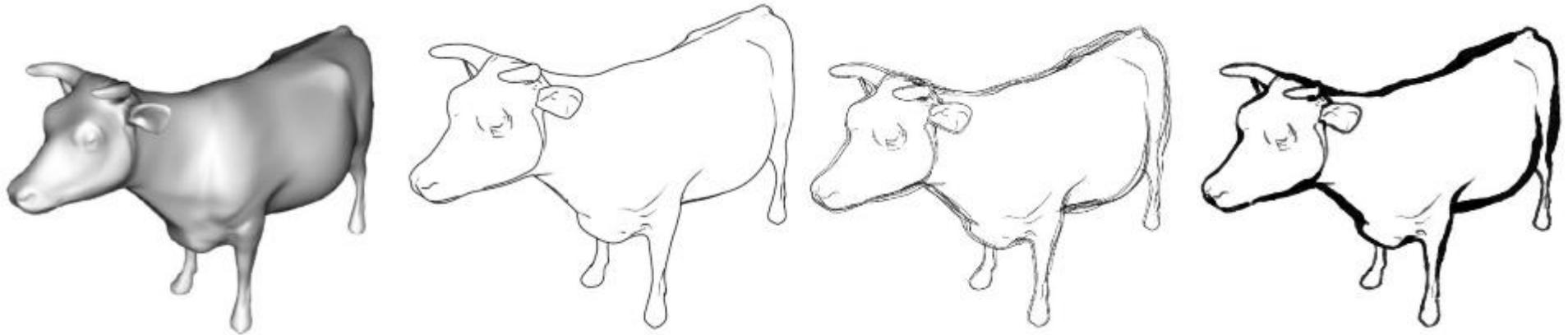
# Analysis

- Pro: Independent of polygon count
- Con: Dependent on image space resolution
  - Different results if zoomed in/out
- Con: Depth fighting can lead to noisy results



# Non-Photorealistic Rendering

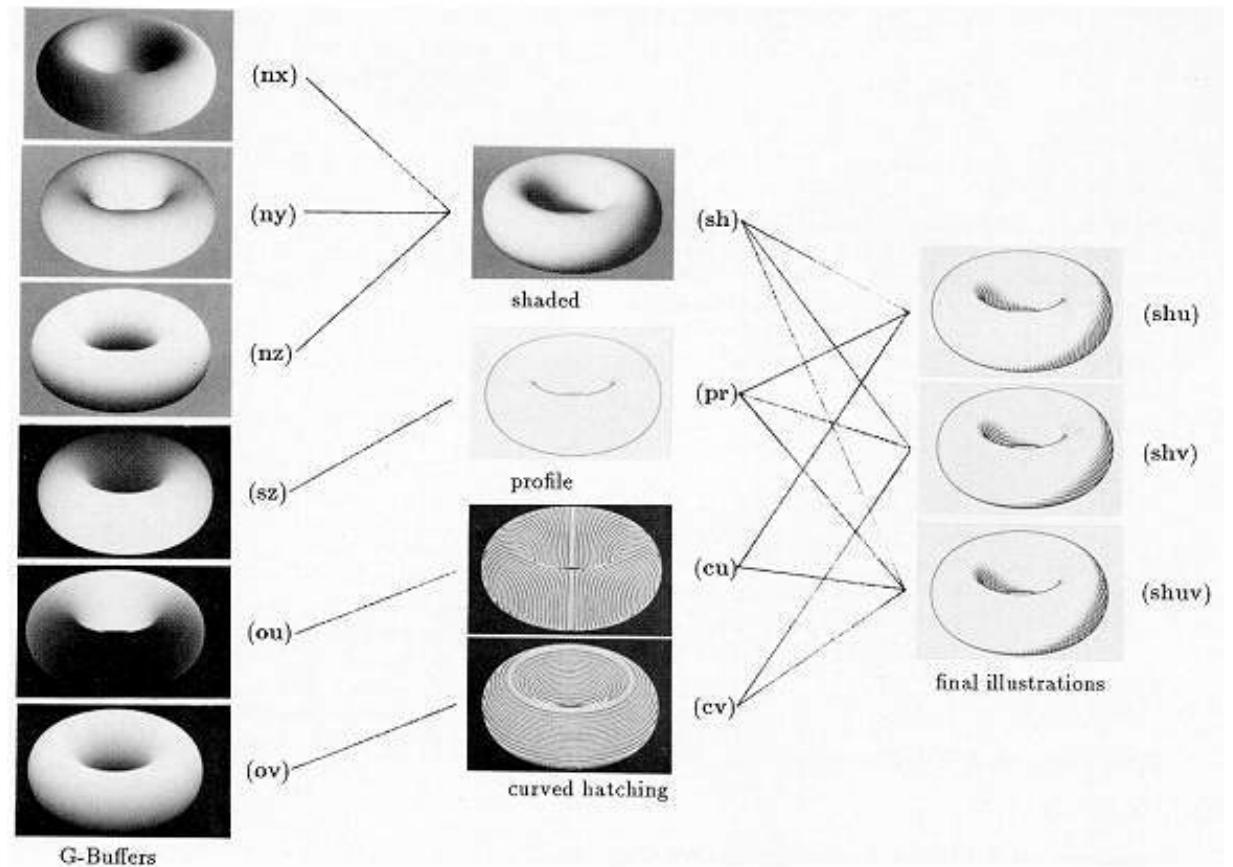
NPR often emphasizes object edges, silhouettes



# G-Buffer for Illustrations

- Hatching: create texture from brush strokes which follow the surface coordinates  $(u,v)$  of the object

- $nx$ : normal vector z
- $ny$ : normal vector y
- $nz$ : normal vector z
- $sz$ : screen coordinate
- $ou$ : coordinate  $u$   
(on curved surface)
- $ov$ : coordinate  $v$   
(on curved surface)



# Upsampling

- Naïve (bilinear) upsampling suppresses high frequencies
- Joint bilateral upsampling can help here as well



Original



Nearest neighbor



Bilinear filtering

# Joint Bilateral Upsampling on PS4

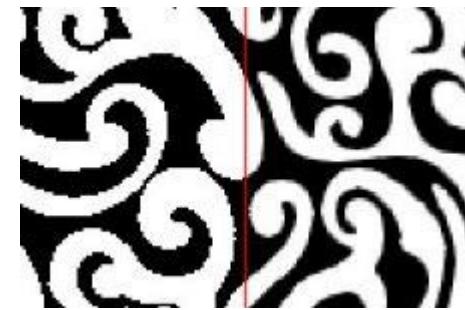
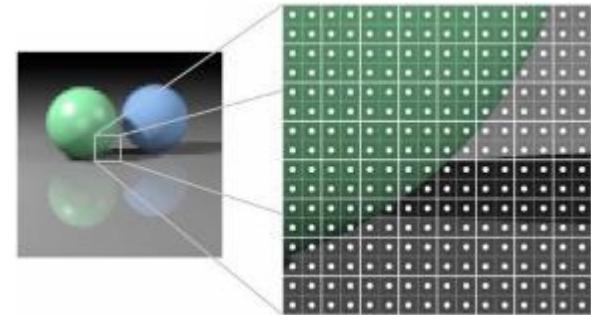
- PS4: 1080p resolution, PS4 Pro: 4K resolution
  - PS4 Pro generates 4x more pixels with only 2x more GPU cores
- Upsampling must work only in driver
  - Cannot assume access to game engine code
- Render color buffer in lower resolution (1/4)
- Render edge buffer in full resolution
  - Depth discontinuities between pixels
  - ID-buffer discontinuities between pixels
- Joint bilateral upsampling of color
  - Color buffer as primary channel
  - Edge buffer as secondary channel



# Anti-aliasing in Real-Time Graphics

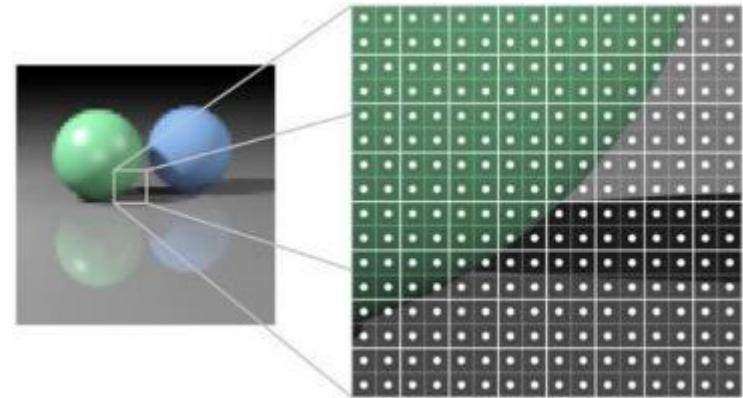
Anti-aliasing can mainly be categorized in

- 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



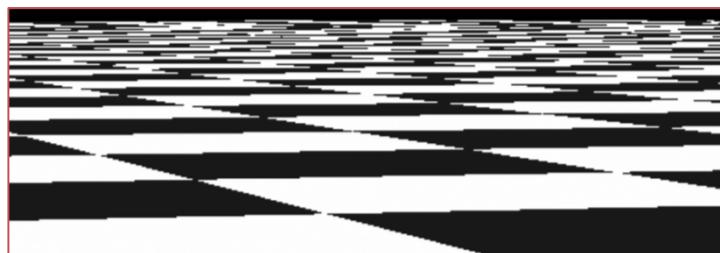
SSAA bekämpft Kantenbildung, indem es das Bild mit einer höheren Auflösung als die tatsächliche Anzeigeauflösung berechnet. Beispielsweise kann ein Bild mit einer Auflösung von 1920x1080 mit SSAA mit einer Auflösung von 3840x2160 berechnet werden. Dies erzeugt ein hochauflößtes Bild mit glatteren Kanten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

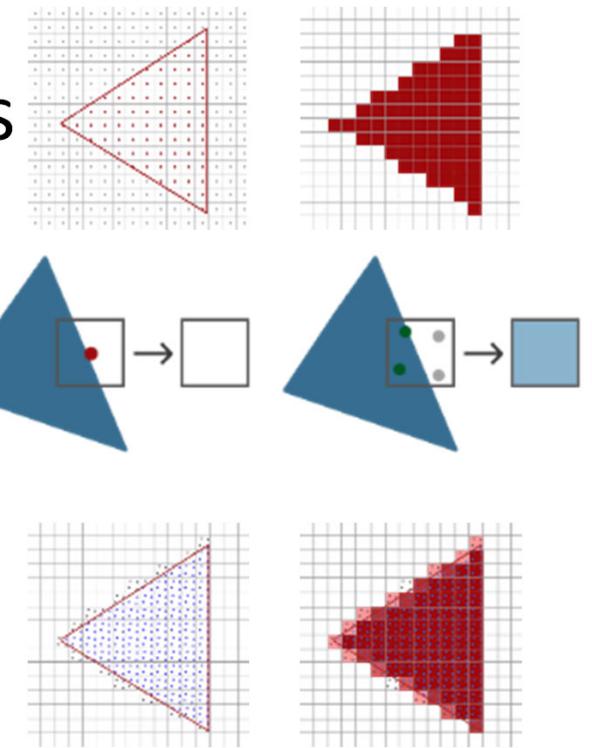


256 samples per pixel

# Multisampling (MSAA)

- 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 um den Endgültigen Wert für den Pixel zu bestimmen.



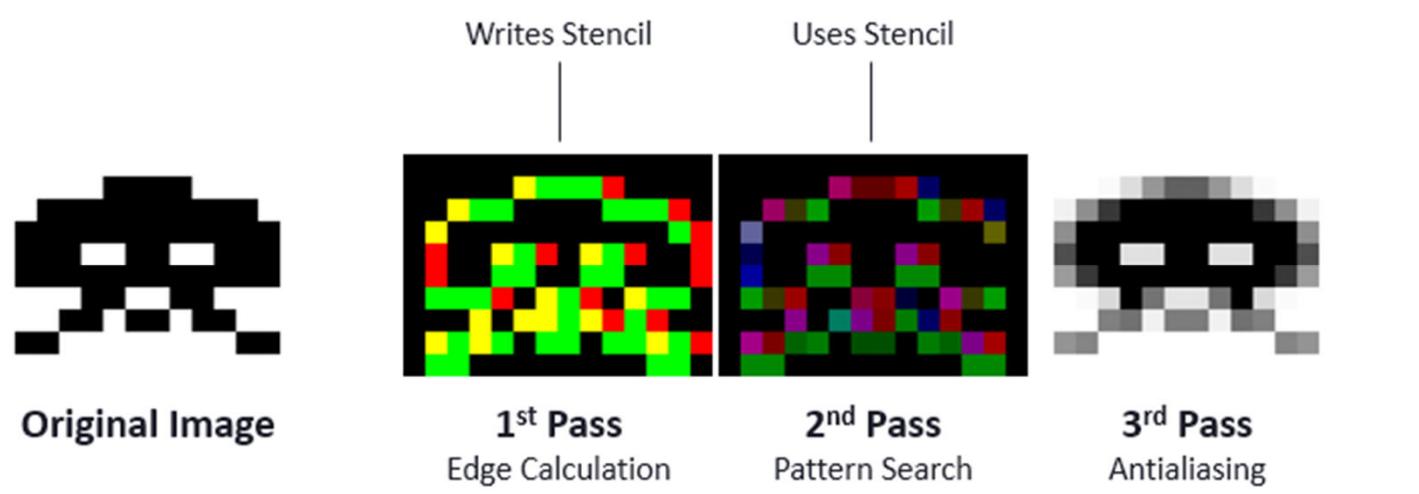
# MSAA in OpenGL



```
// Create multisample texture (used as attachment)
#define NUM_MSAA_SAMPLES 4
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, tex);
glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, NUM_MSAA_SAMPLES, GL_RGB, width, height, GL_TRUE);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, 0);
// Attach the multisample texture to our framebuffer as color attachment
glBindFramebuffer(GL_FRAMEBUFFER, multisampledFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D_MULTISAMPLE, tex, 0);
// Render
render();
// Resolve the multisample FBO by blitting into the default (display) framebuffer
glBindFramebuffer(GL_READ_FRAMEBUFFER, multisampledFBO);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glBlitFramebuffer(0, 0, width, height, 0, 0, width, height, GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

# Morphological Anti-Aliasing (MLAA)

1. Edge detection in input image
2. Compute blending texture based on distance to line end/crossing and line shape
3. Anti-Aliasing using blending map from step 2



# MLAA Example



# 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

Die Schritte von TAA sind folgendermaßen:

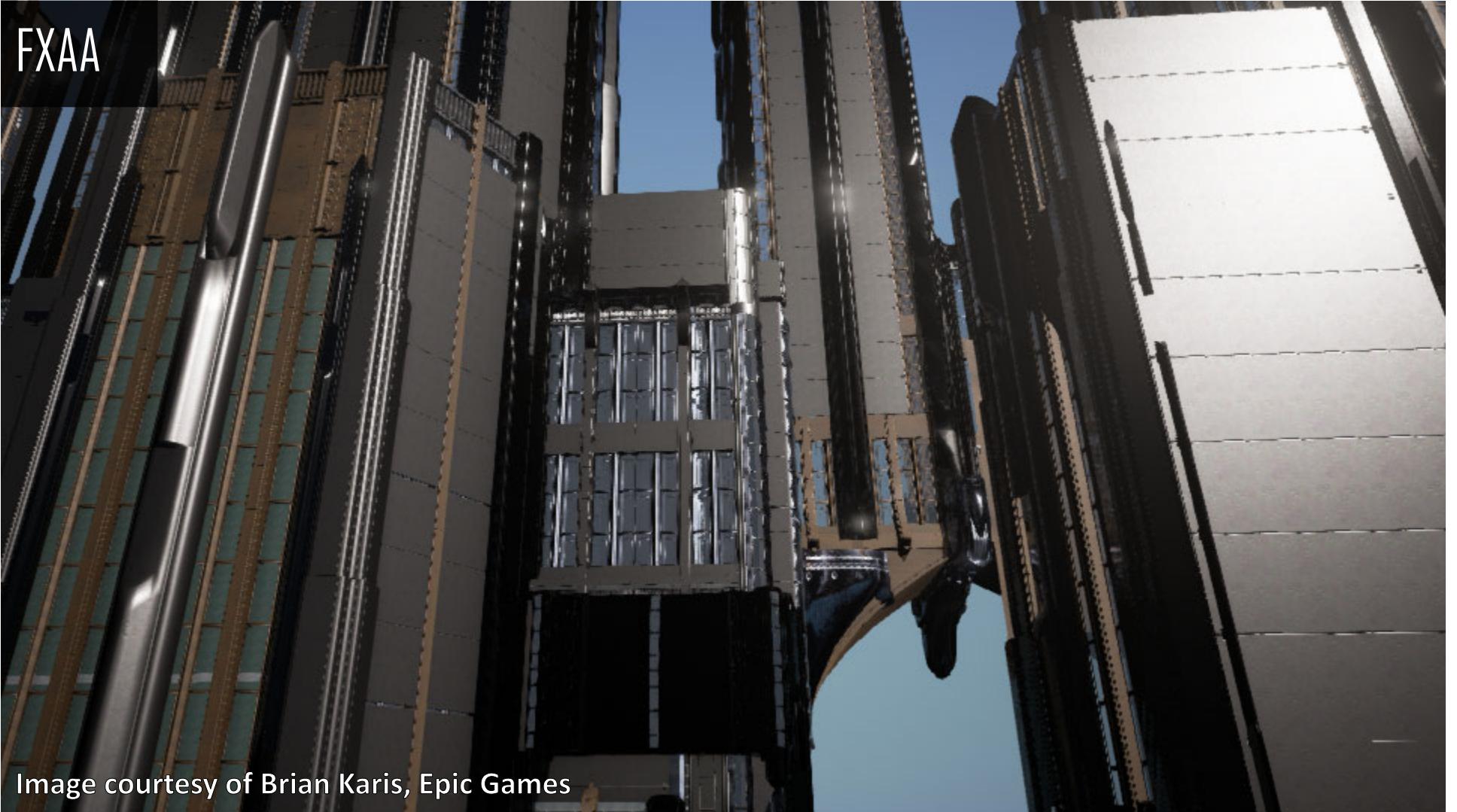
1. Motion-Vektor-Erkennung: Das TAA-System erkennt die Bewegungen von Objekten im Bild, indem es Motion-Vektoren berechnet. Diese Vektoren beschreiben die Bewegung jedes Pixels im Vergleich zum vorherigen Frame.
2. Überlagerung: Das TAA-System überlappt das aktuelle Frame mit dem vorherigen Frame, indem es die Motion-Vektoren verwendet, um die Position jedes Pixels im aktuellen Frame zu bestimmen.
3. Analyse: Das TAA-System analysiert die überlappten Bilder und bestimmt, welche Farben und Tiefenwerte am besten zum aktuellen Frame passen.
4. Zusammenfassung: Die besten Farben und Tiefenwerte werden verwendet, um das endgültige Bild zu erstellen, das auf dem Bildschirm angezeigt wird.

# Temporal Anti-Aliasing (TAA)

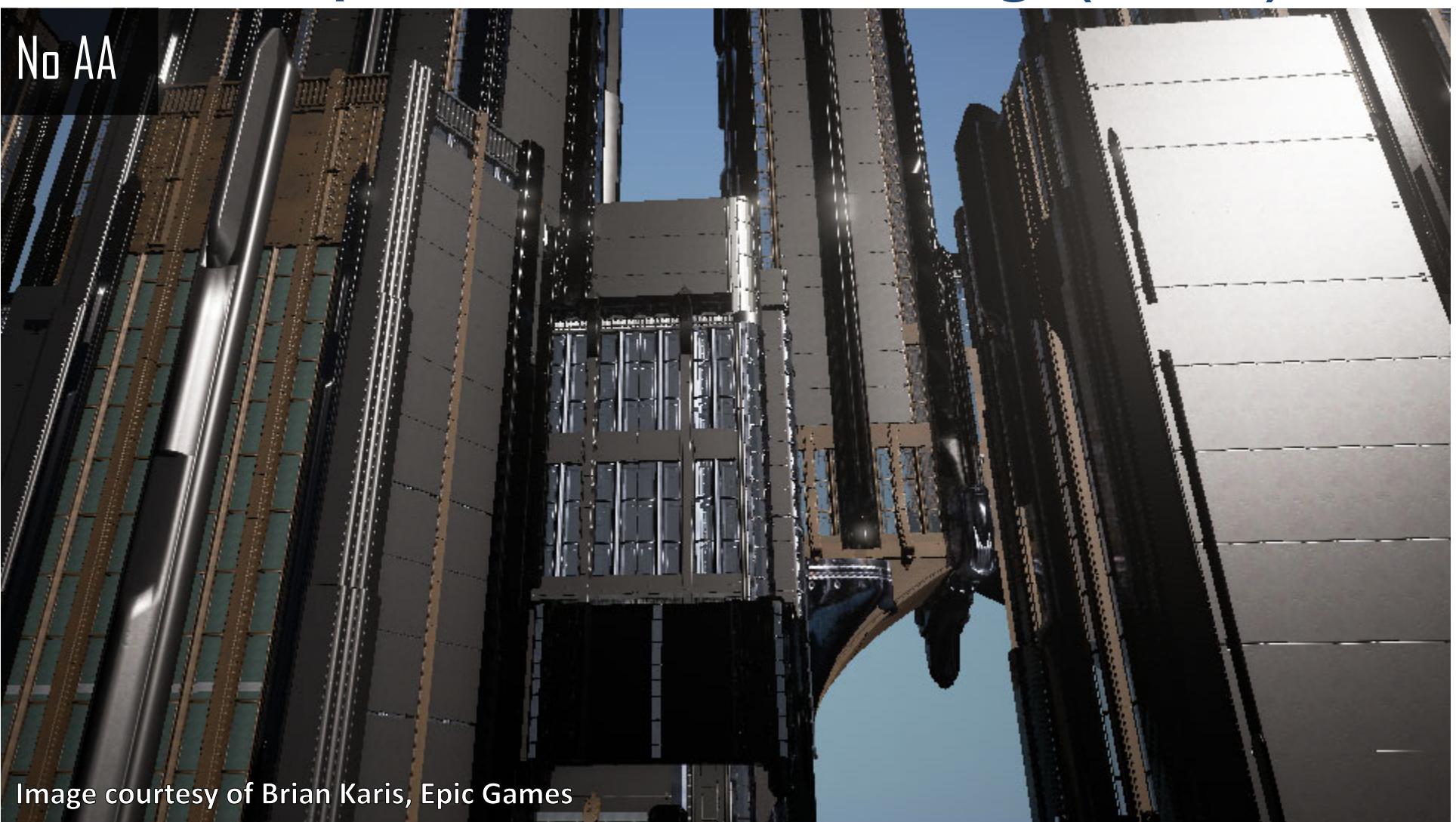


- Idea: Blend previous frames' **jittered shading result** with successive frames
  - Stochastic/subpixel supersampling
  - Works for **geometric and shading aliasing**
- Current gold standard for game engines
  - Works with any rendering architecture
- Requires more involved modifications to the renderer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

# Temporal Anti-Aliasing (TAA)



# Temporal Anti-Aliasing (TAA)



# Temporal Anti-Aliasing (TAA)

Temporal AA

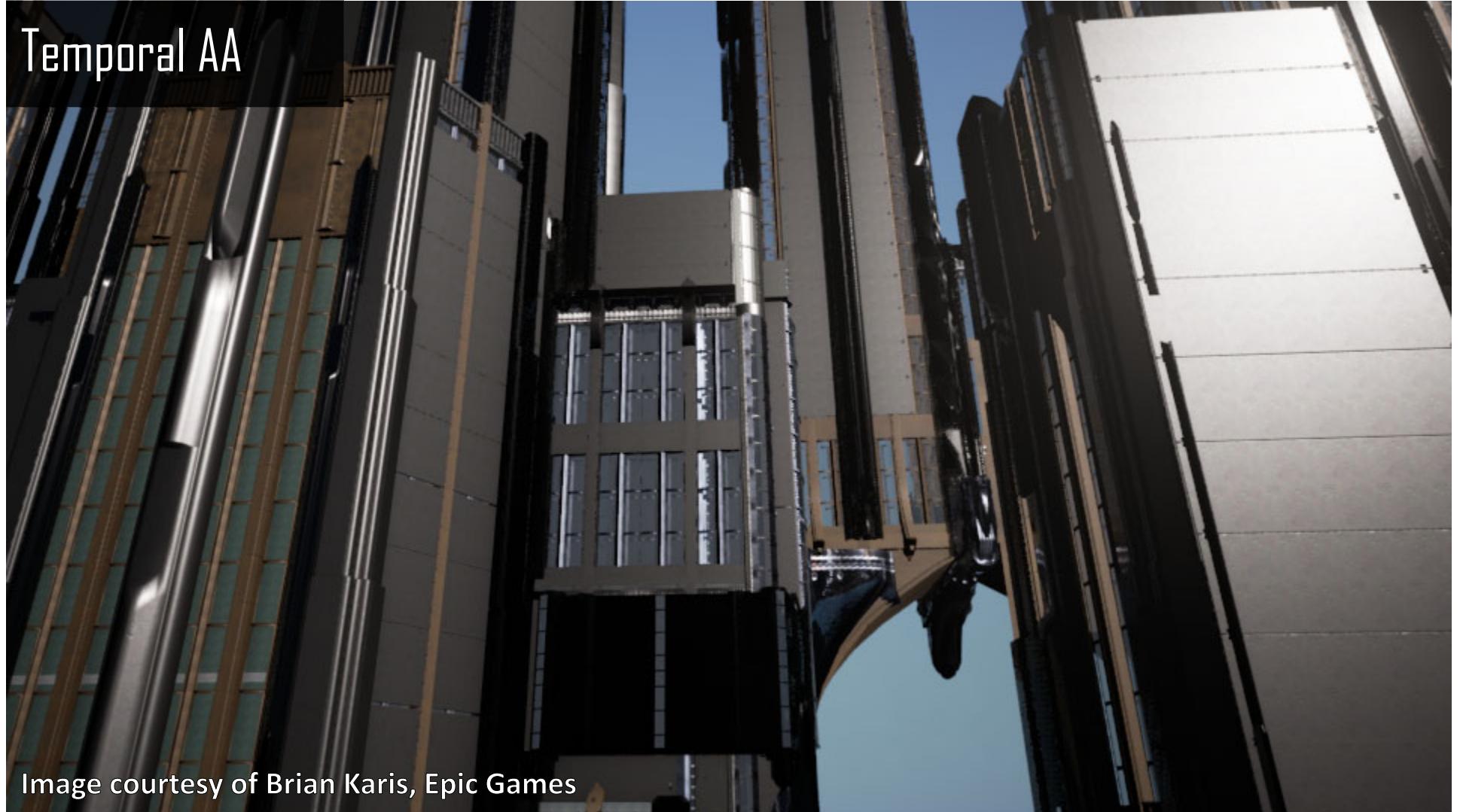
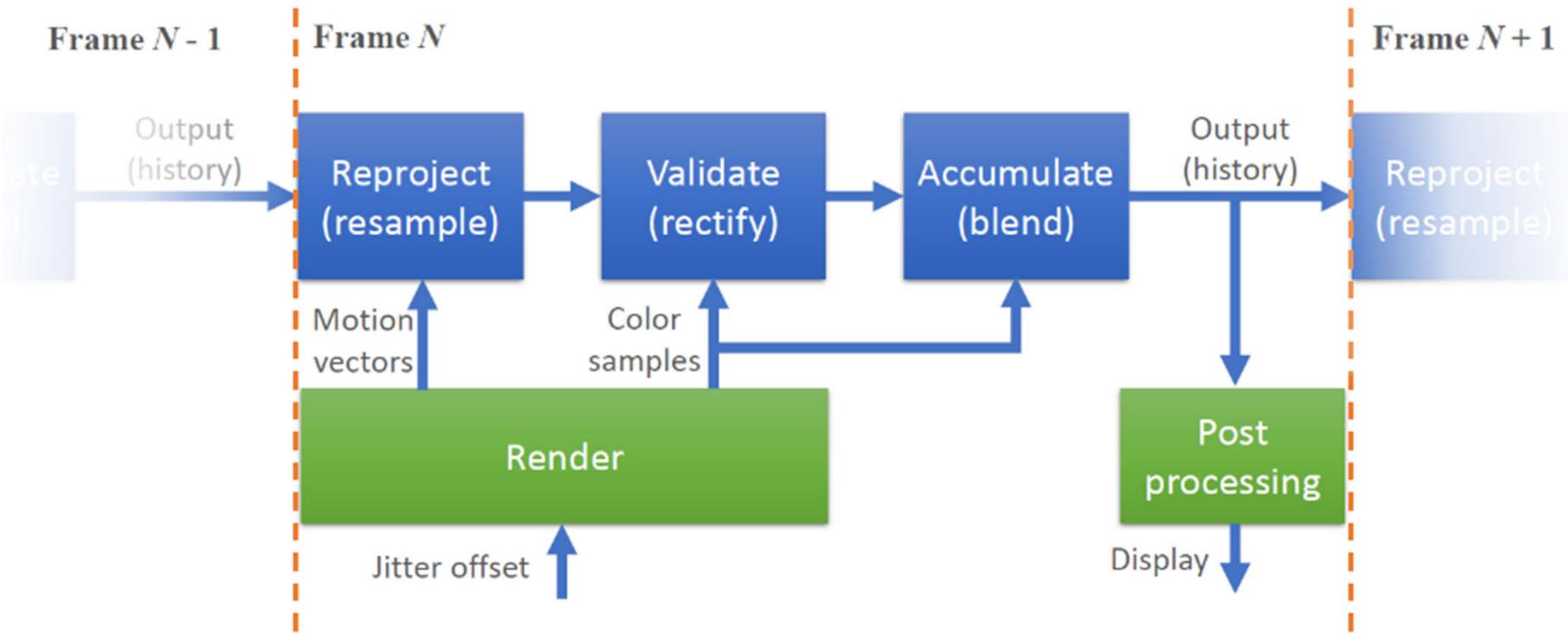


Image courtesy of Brian Karis, Epic Games

# Temporal Anti-Aliasing (TAA)



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>

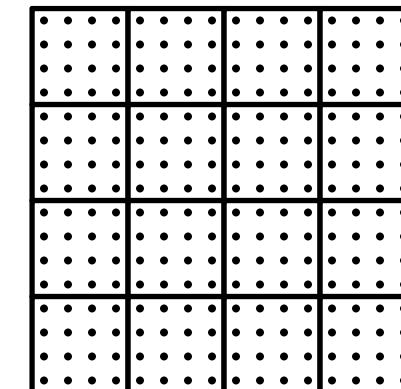
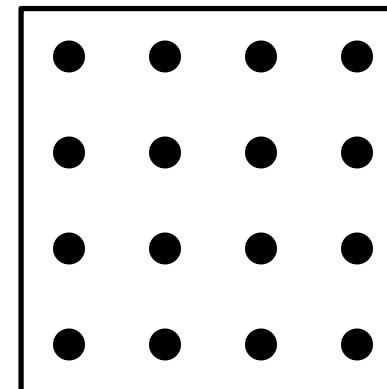
Jittering bezieht sich auf eine Technik, die verwendet wird, um die Genauigkeit von Temporal Anti-Aliasing (TAA) zu verbessern. Es handelt sich dabei um eine Methode, bei der das aktuelle Frame leicht versetzt wird, um die Motion-Vektoren (die die Bewegung jedes Pixels beschreiben) zu verbessern. Dies erhöht die Genauigkeit der Überlagerung von aktuellem Frame und dem vorherigen Frame und reduziert somit die sichtbaren Aliasing-Effekte.

# TAA: Jittering

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

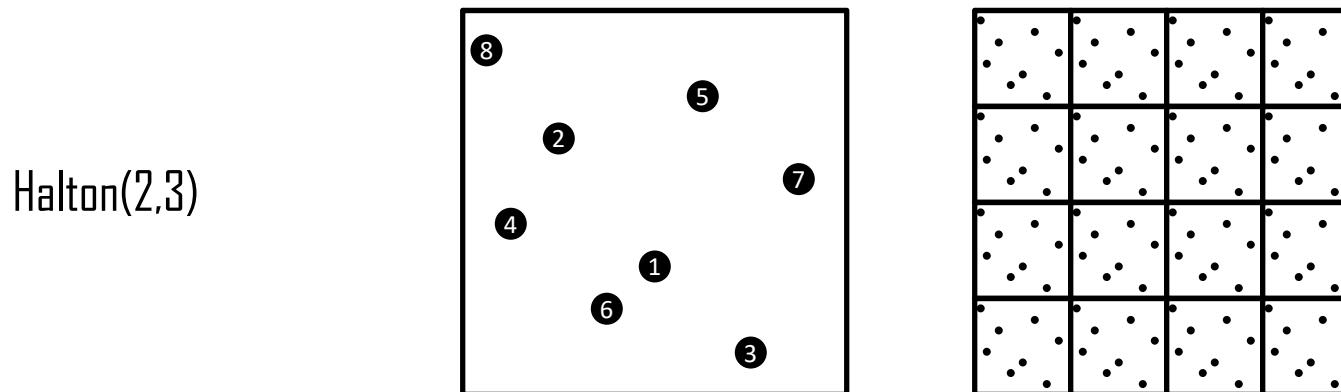
```
ProjMatrix[2][0] += ( SampleX * 2.0f - 1.0f ) / ViewRect.Width();  
ProjMatrix[2][1] += ( SampleY * 2.0f - 1.0f ) / ViewRect.Height();
```

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)



Es handelt sich dabei um eine Methode, bei der mehrere vergangene Frames gespeichert werden, um die Bewegung jedes Pixels im aktuellen Frame besser zu berechnen.

# TAA: History Buffering

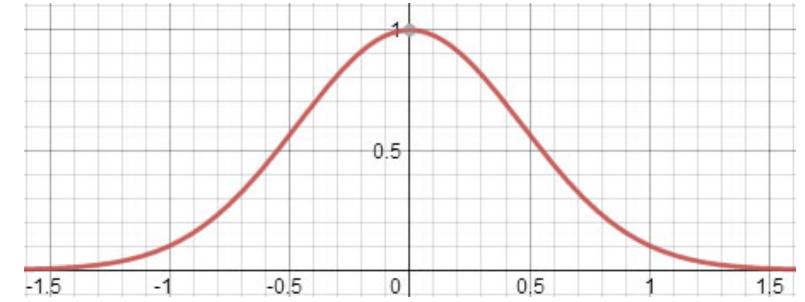
- Simple moving average is not feasible
  - Would require N times storage for N samples
- Exponential moving average
  - Nearly “infinite” number of samples with fixed storage
  - When  $\alpha$  is small, approaches a simple moving average
  - $\alpha = 0.1$  is common
- This can simply be stored in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

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

Es handelt sich dabei um eine Methode, bei der die Bewegung jedes Pixels im aktuellen Frame mit Hilfe von Motion-Vektoren berechnet wird und dann verwendet wird um das aktuelle Frame zu rekonstruieren.

# 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



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

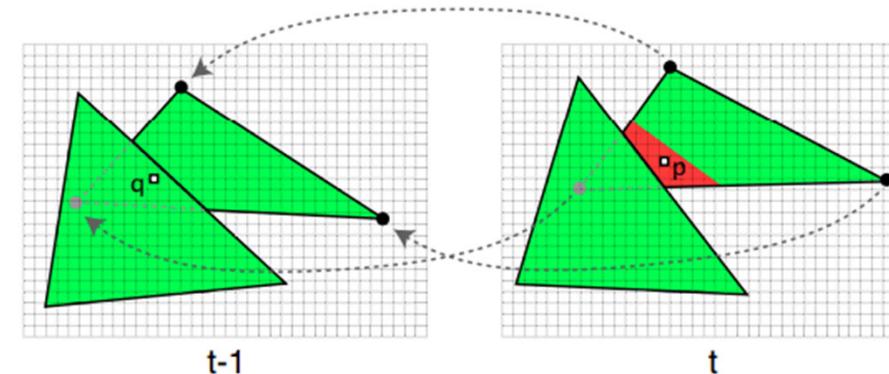
# TAA: Dynamic Scenes

- 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**

Es handelt sich dabei um eine Methode, bei der die Bewegung jedes Pixels im aktuellen Frame mit Hilfe von Motion-Vektoren berechnet wird und dann verwendet wird, um das aktuelle Frame auf die Position des letzten Frame zu projizieren.

# 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



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: Reprojection Ghosting

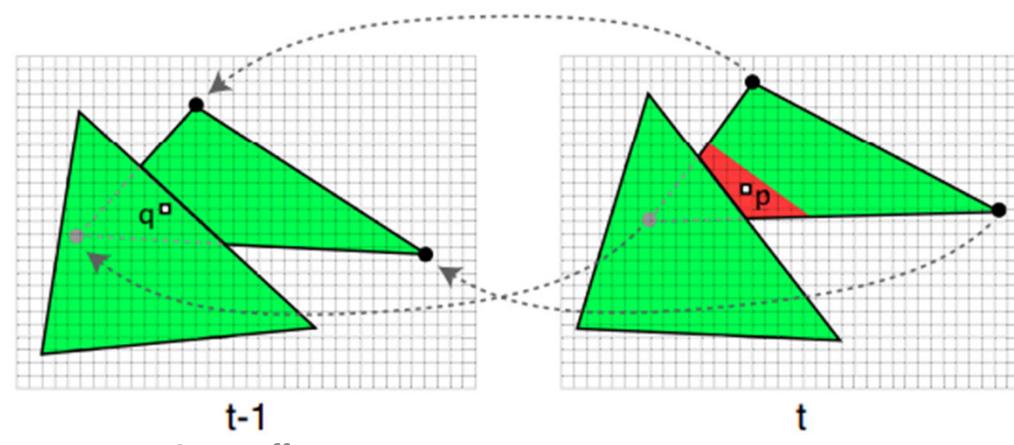


Image courtesy of Brian Karis, Epic Games

# TAA: History Rejection

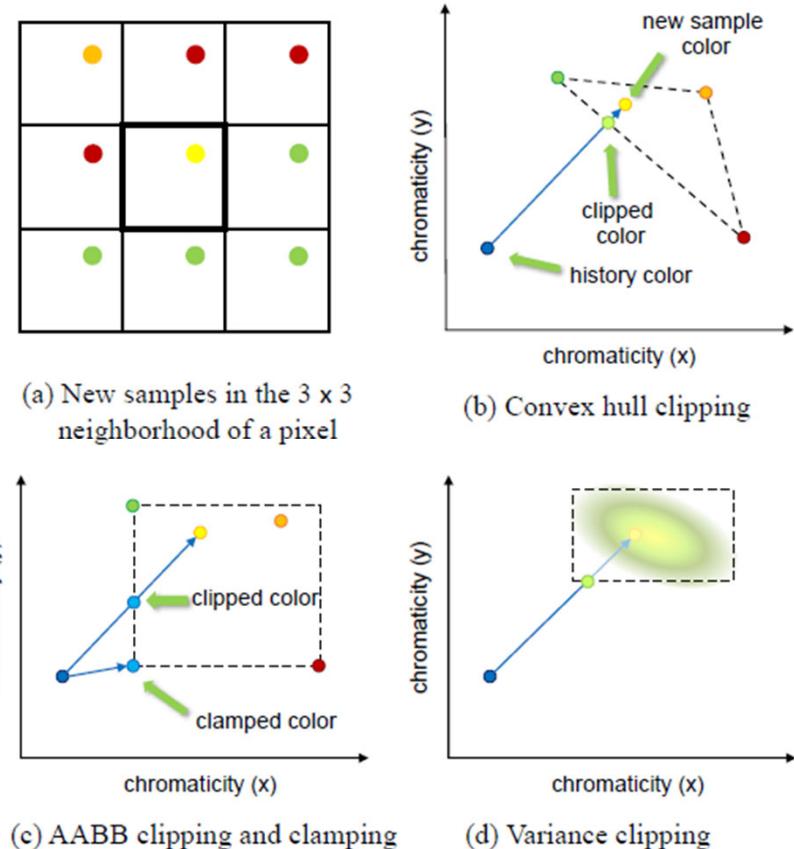
- When reprojecting, the history might be invalid
  - Occlusion/Disocclusion events
  - Shading changes (lights turning on/off...)
- Use **geometry data** (depth, normal, object ID, motion vector) or **color variance** to detect history confidence
- When history is invalid, simply clear the history buffer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

Temporal Anti-Aliasing (TAA) History Rejection ist eine Technik die verwendet wird, um die Bildqualität bei der Verwendung von TAA zu verbessern. Es handelt sich dabei um eine Methode, die das Aliasing durch die Verwendung von Informationen aus der Vergangenheit ablehnt. TAA History Rejection verwendet eine Kombination aus Temporal Reprojection und History Rejection, um das Aliasing zu reduzieren und eine höhere Bildqualität zu erreichen.



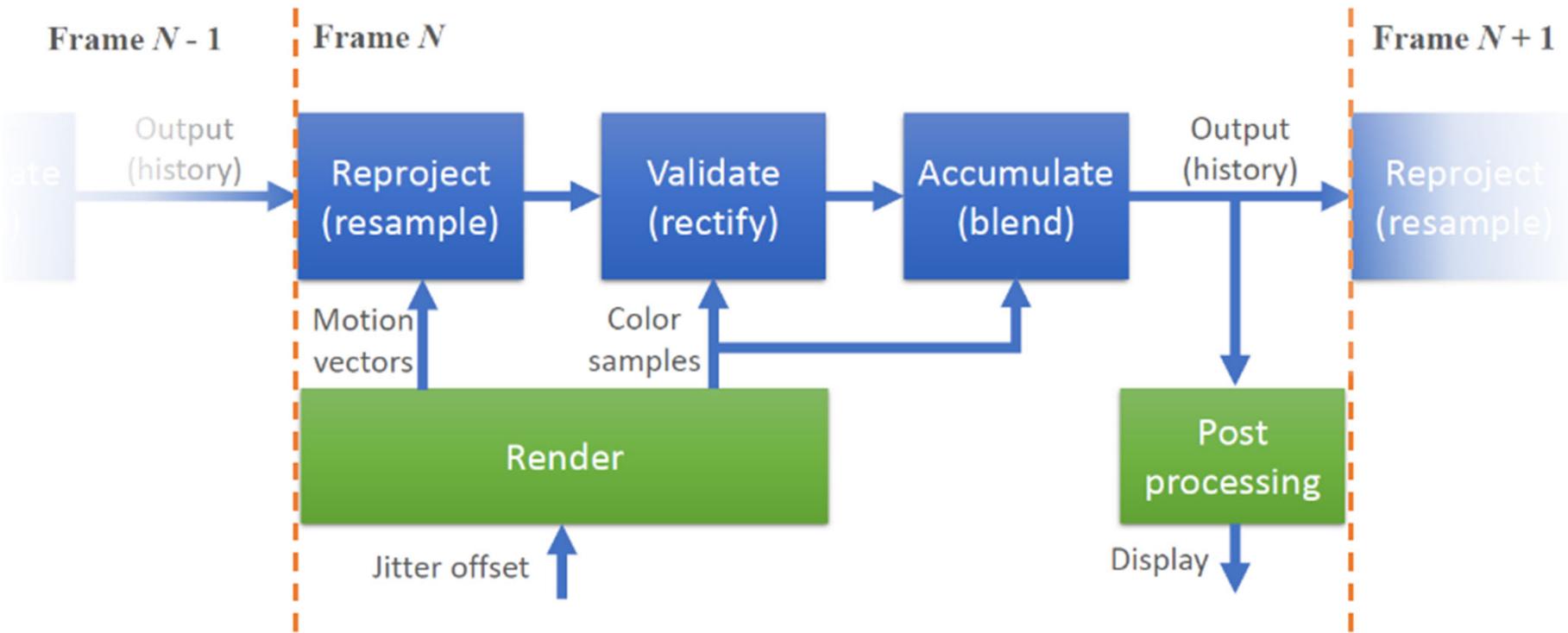
# TAA: History Rectification

- 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



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>

# TAA: Steps for Assignment 3

1. Implement a history buffer
2. Jitter the camera projection and accumulate/reconstruct samples
  - Test with a static camera at this point
  - This should already anti-alias your static image nicely
3. Generate motion vector texture
  - Since you need to extend shaders to have the previous and the current transform, it's ok to just do it for the camera for the exercise
4. Implement history rejection based on depth
  - At this point, you should be able to move the camera around with a lot fewer artifacts
5. Optional: implement further improvements

# 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



# Motion Blur Example



Picture: Crysis (Object Based Motion Blur)

# Motion Blur with Moving Camera

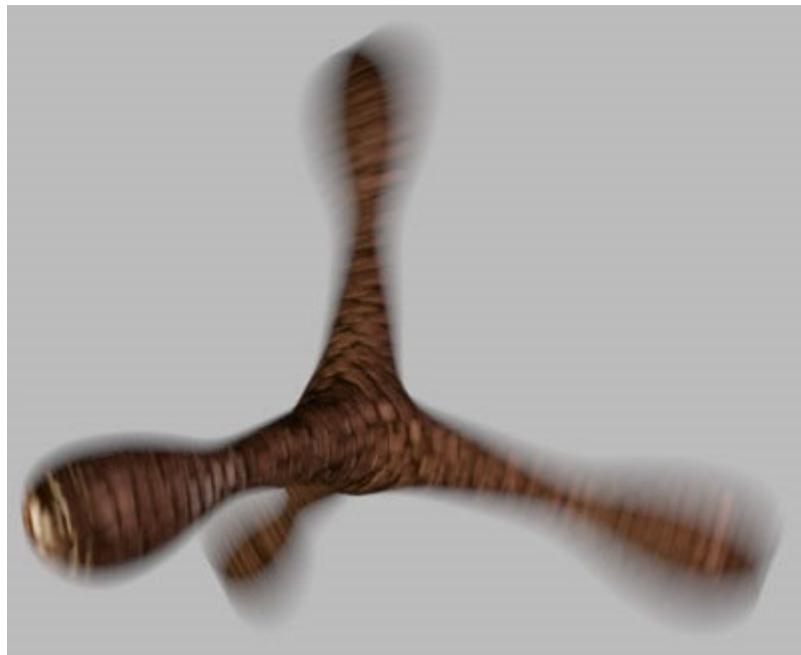
Blurry, moves fast relative to camera:



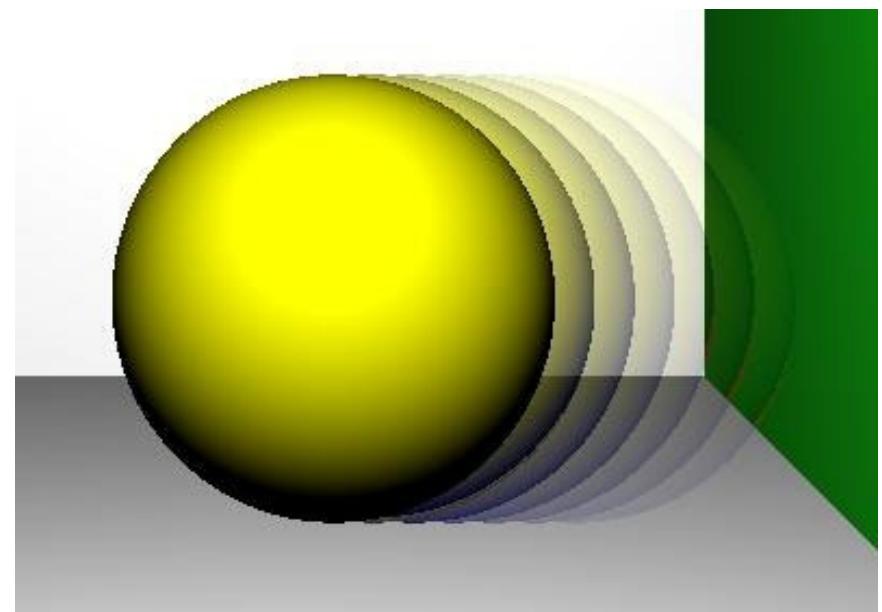
No blur, does not move relative to camera

# Continuous vs Discrete

Is a continuous effect



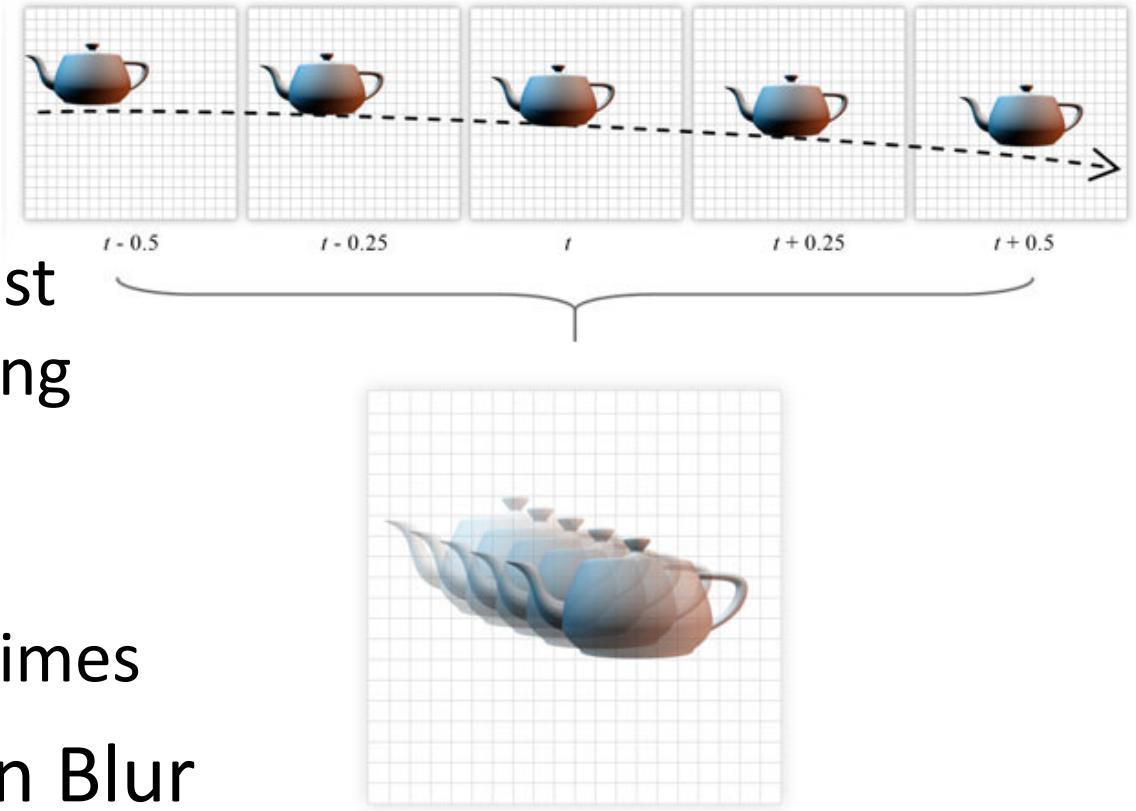
Correct, continuous MB



Approximated, discrete MB

# Discrete Methods

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

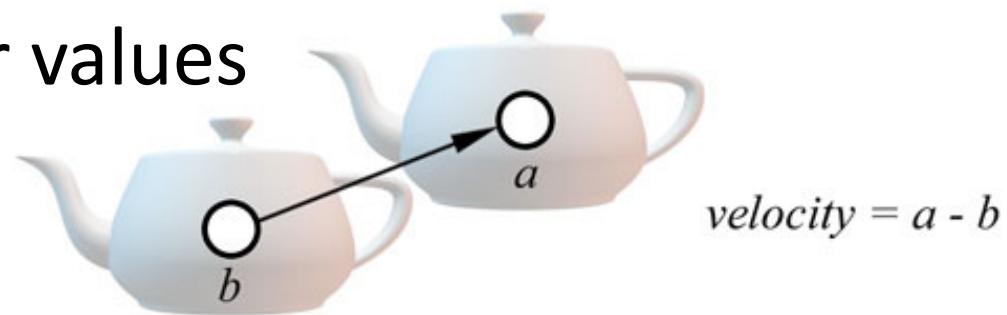


- Image Space Motion Blur
  - Render object to buffer
  - Copy buffer with varying transparency
  - More efficient

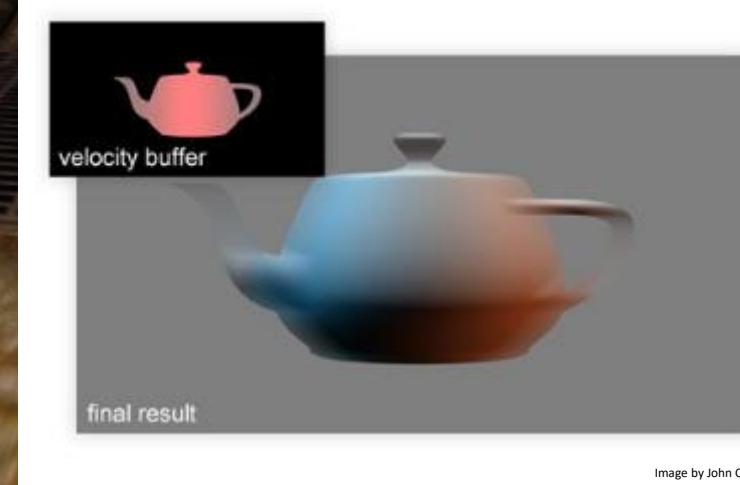
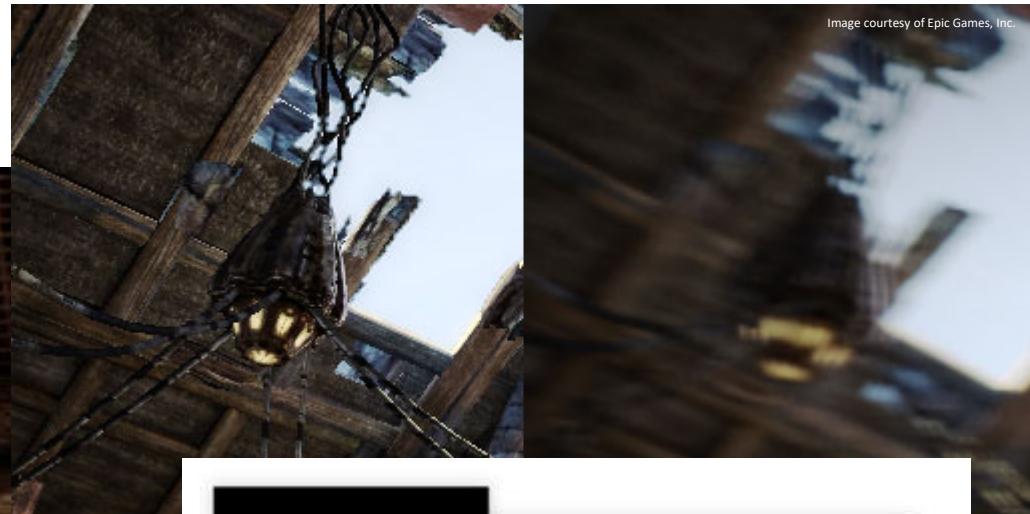
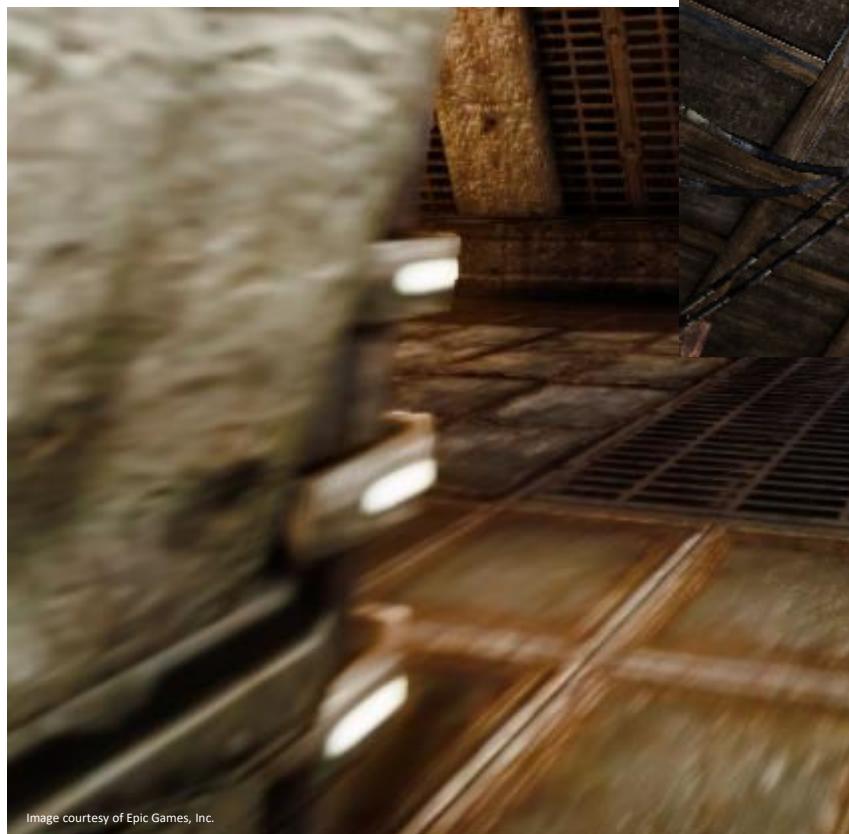
# Continuous Motion Blur

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



# Continuous Motion Blur – Examples 1



# Continuous Motion Blur – Examples 2



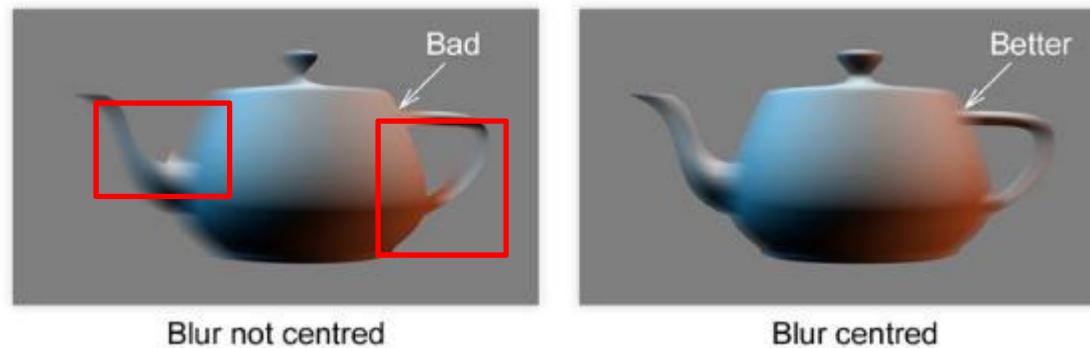
# Continuous Motion Blur – Artifacts

- Color bleeding
  - Slow foreground objects bleed into fast background objects



Images by John Chapman

- Discontinuities at silhouettes



# Image Compositing



- Image compositing idea
  - Use alpha channel to combine multiple buffers
  - Buffers contain partial rendering results
- Examples
  - Lens flare
  - Billboards
  - Particle systems

Simuliert den Effekt von Licht  
der direkt in die Kamera  
scheint

# Lens Flare

- A shortcoming of cameras that photographers try to avoid
- However: looks realistic and fancy
- Effect occurs inside lens system
  - Always on top
- Happens when light source inside image
- Star, ring or hexagonal shapes



# Lens Flare Example 1

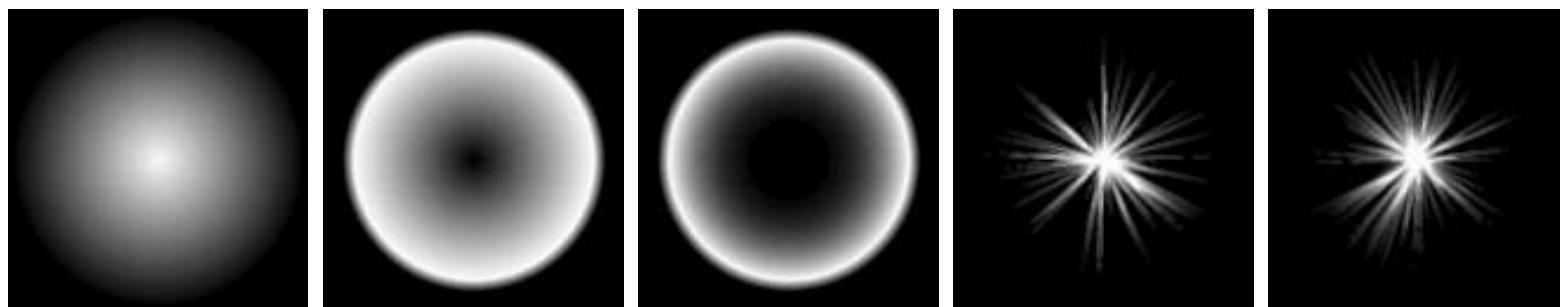


# Lens Flare Example 2



# Lens Flare Rendering

- Choose a lens flare texture
- All lens flares lie on the line between light source and image center
- Rendered with differently sized transparent billboards (alpha blending)

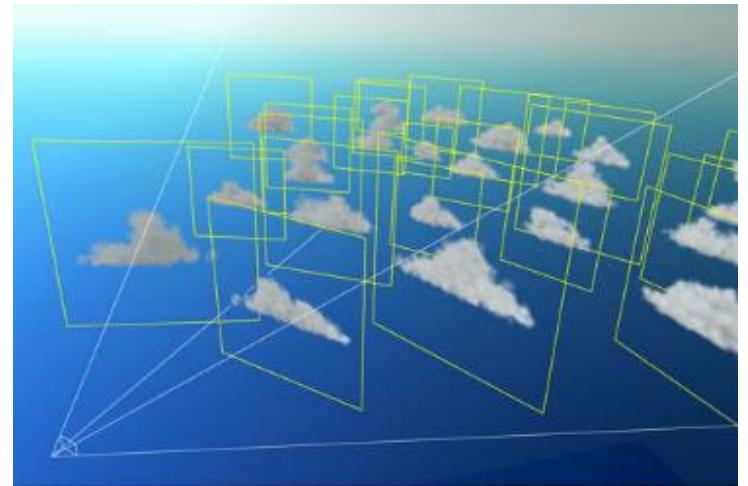


# Don't overdo it!



# Billboards

- Synonyms: impostors, sprites
- Textured rectangles which
  - Face the viewer, or
  - Are aligned with some axis
- Can be used in large quantities
  - Simple, only 2 textured triangles
- Low memory footprint
- Only look good at a distance or when small
- Example: clouds in a game



# Billboard needs to face camera

- How: modify the ModelView matrix  
(remove rotation)

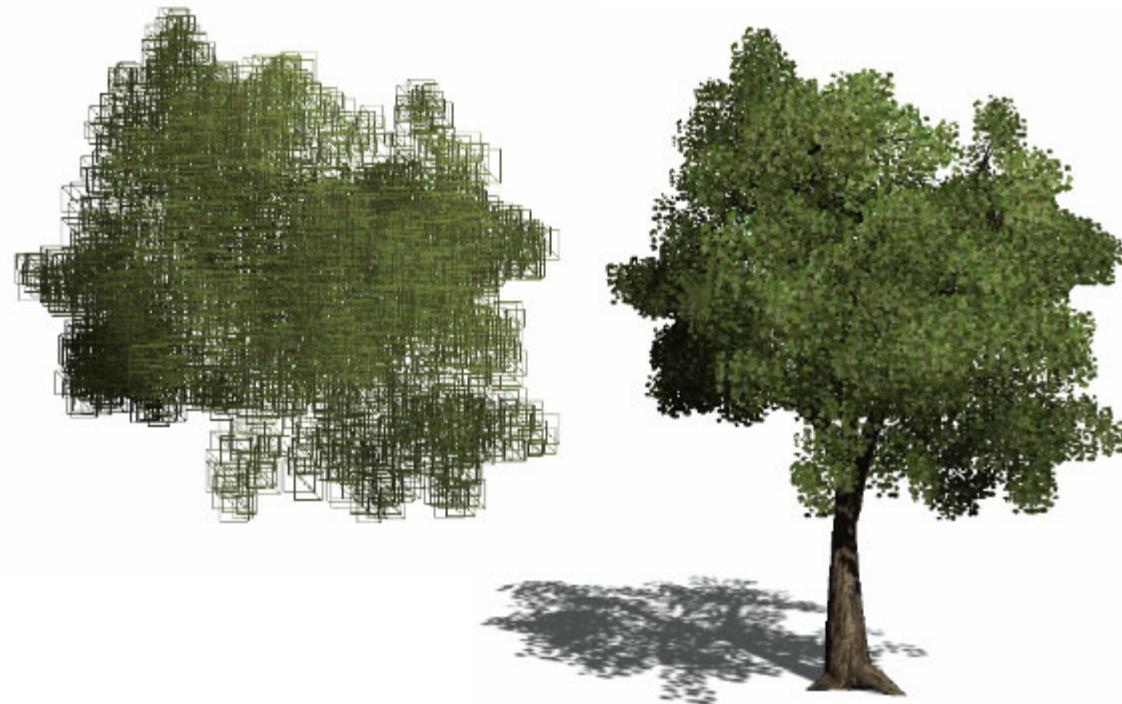
$$\begin{pmatrix}
 a0 & a4 & a8 & a12 \\
 a1 & a5 & a9 & a13 \\
 a2 & a6 & a10 & a14 \\
 a3 & a7 & a11 & a15
 \end{pmatrix} \rightarrow
 \begin{pmatrix}
 \quad & \quad & \quad & \quad \\
 \quad & \quad & \quad & \quad \\
 \quad & \quad & \quad & \quad \\
 a3 & a7 & a11 & a15
 \end{pmatrix}$$

↓

$$\begin{pmatrix}
 s0 & a12 \\
 s1 & a13 \\
 s2 & a14 \\
 a3 & a7 & a11 & a15
 \end{pmatrix}$$

- Maintain scale!
- Result: BB will appear at the right position and distance, but will face camera

# Billboard Clouds



- A set of billboards with different size/orientation
- Created procedurally (from 3D model or rule set)
- Can be animated by physical simulation

# 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/H_Ico7TSUIY)



<https://youtu.be/UnZMp17lqy4>



<https://youtu.be/RpIDGqjhOX8>

# History of Particle Systems 1

- 1982 Star Trek II: “Genesis Sequence”

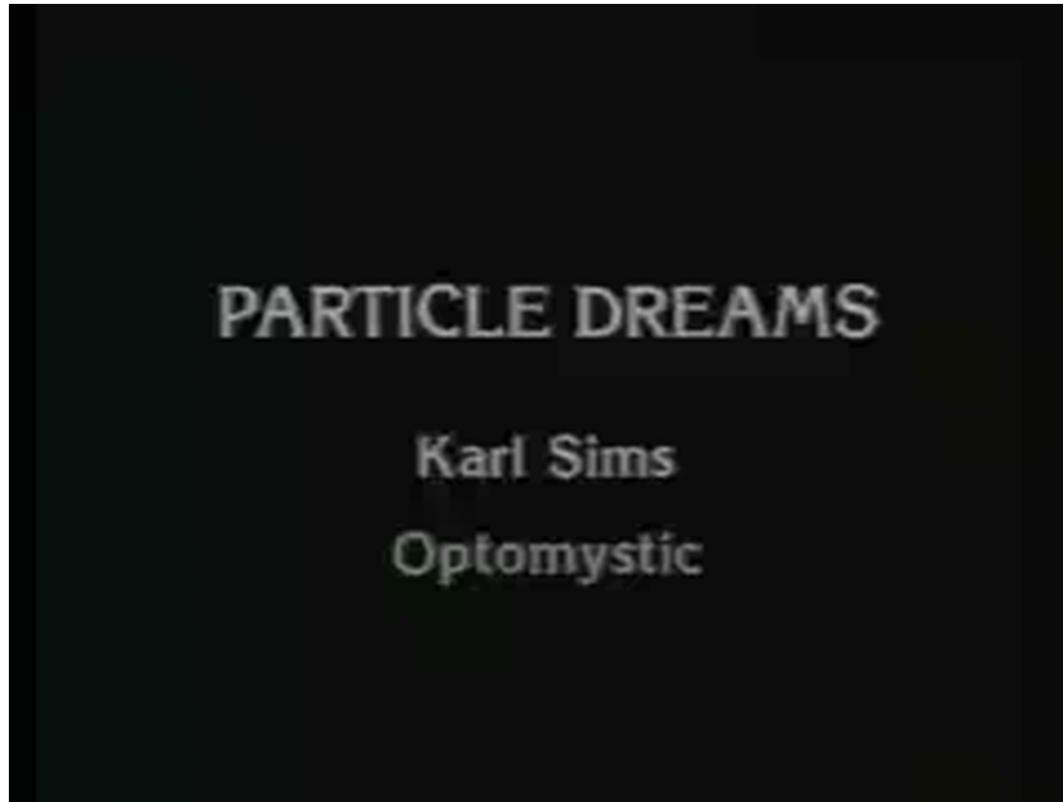


frame of animation (© Pixar 1982)

<https://youtu.be/fwTeO40hm6g>

# History of Particle Systems 2

- 1988: Carl Sims – Particle Dreams



<https://youtu.be/bfSGqLpuzNM>

# 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



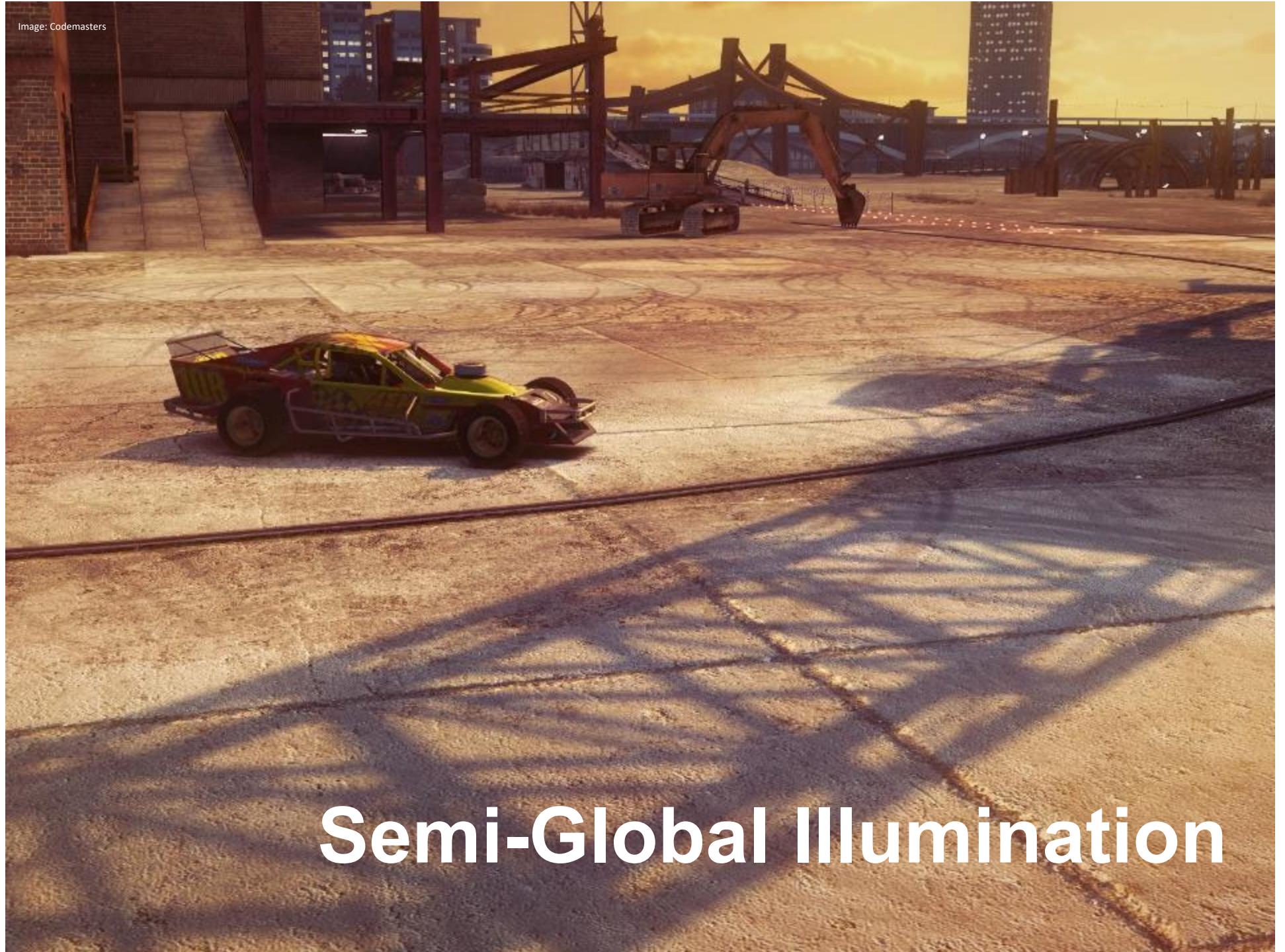
# Things to Consider

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

# Questions?



Image: Codemasters



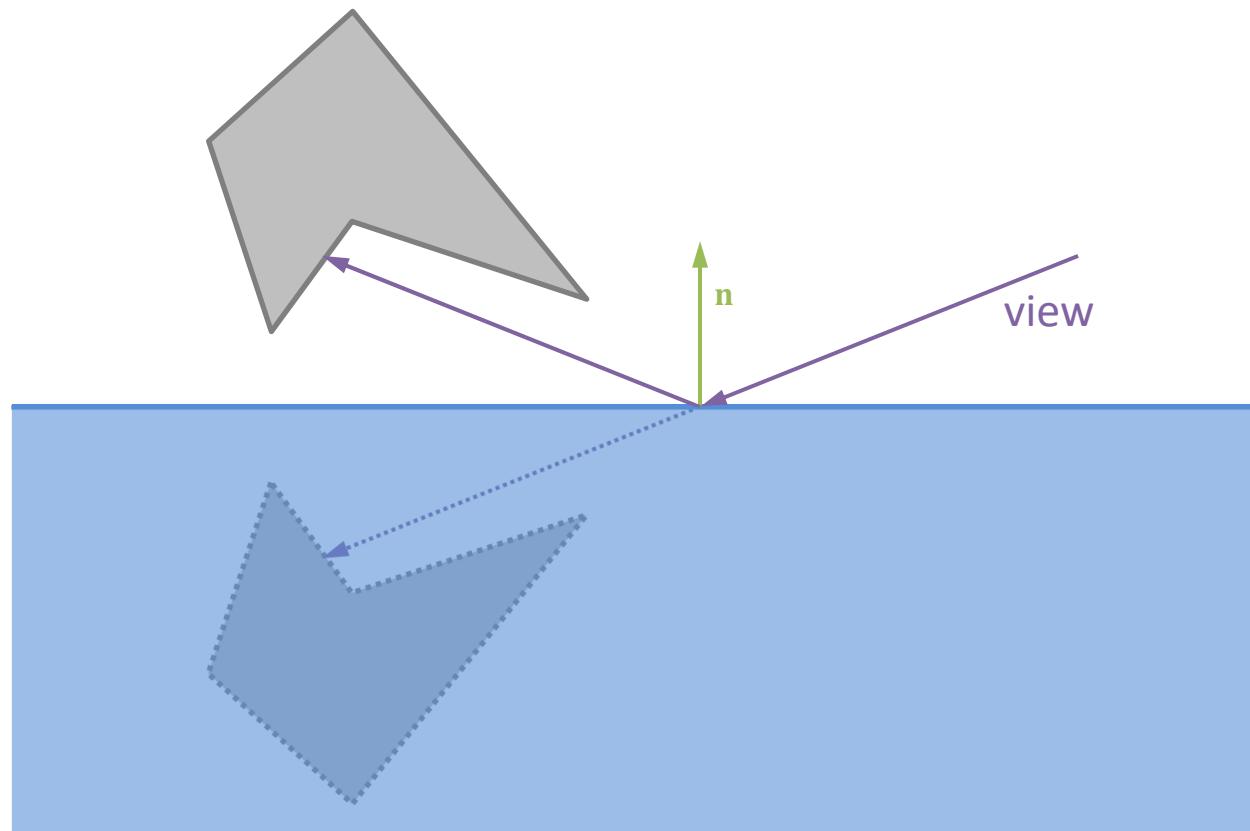
# Semi-Global Illumination

# Introduction

- Full global illumination is expensive
- Many subtle lighting effects may not be visible
- For real time rendering
  - Compute local illumination *everywhere*
  - Compute global illumination only *selectively*
    - Only certain types of light transport
    - Only for objects where it is visually important
- Important categories: reflections, transparency, shadows, ambient occlusion, precomputed radiance

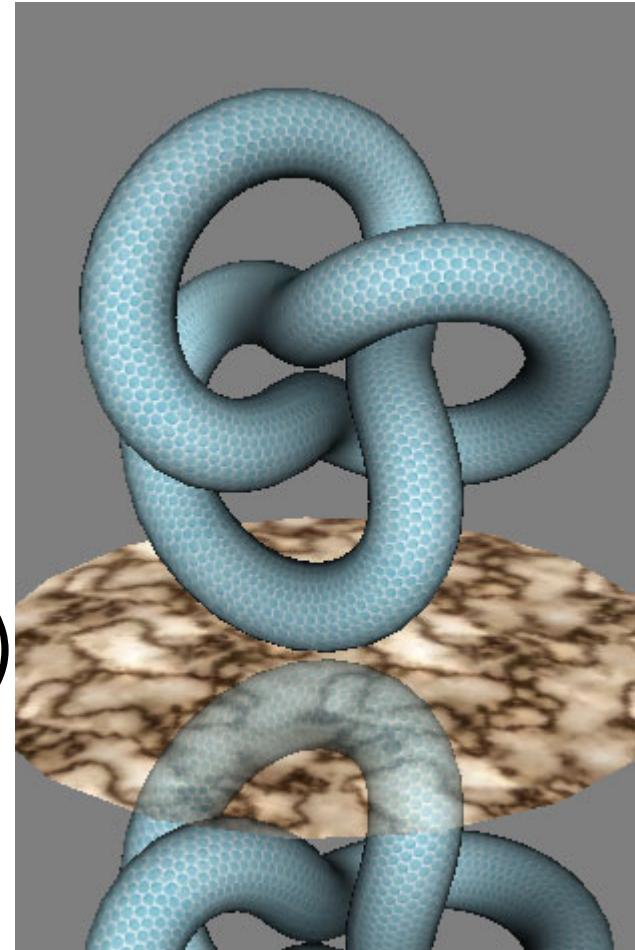
# Planar Reflections

Simulieren von Reflektieren auf  
glatte Oberflächen wie Spiegel  
oder Wasseroberflächen

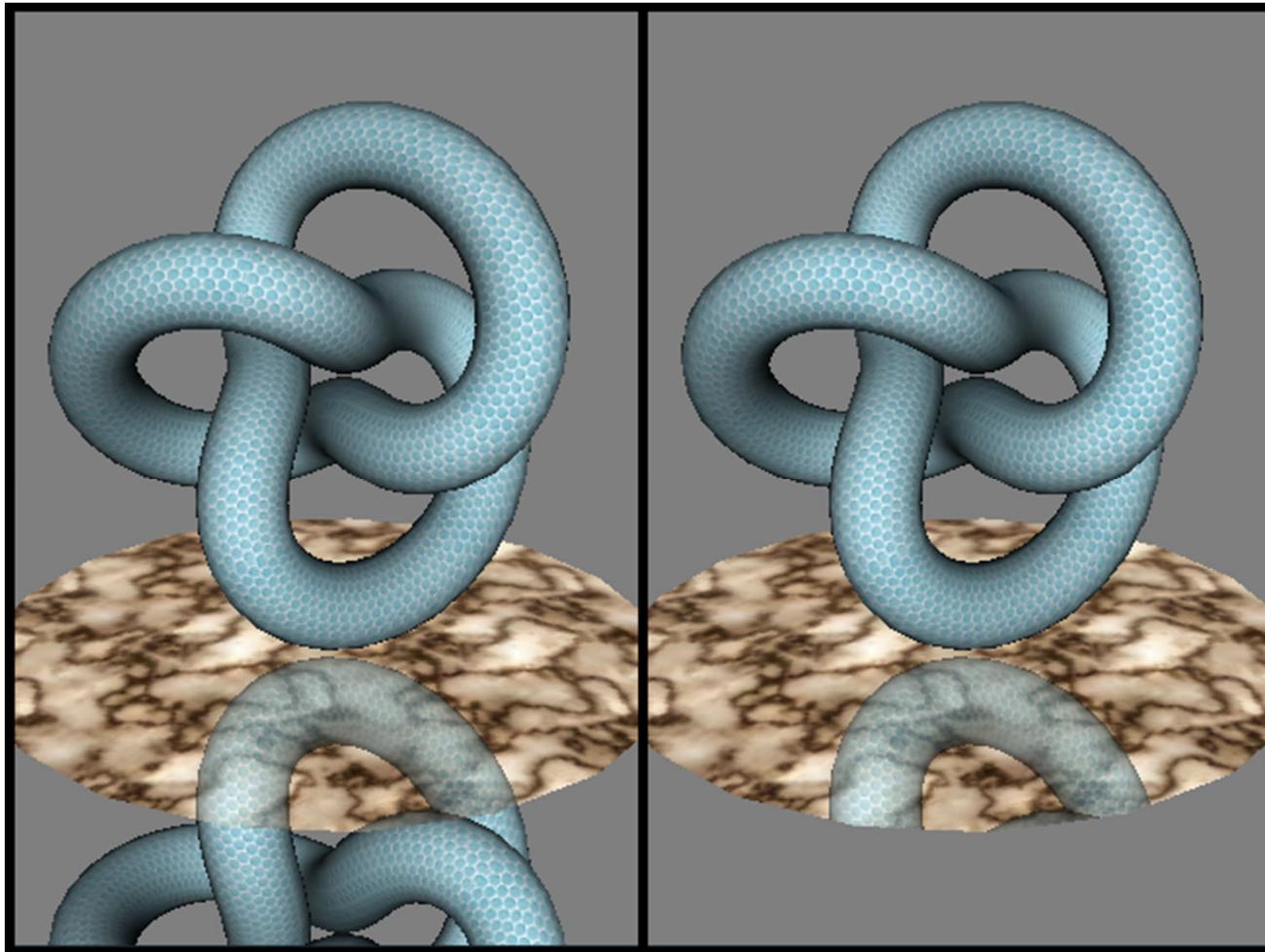


# Planar Reflections in OpenGL

- Define a plane, e.g., z-plane
  - Scale(1, 1, -1)
- Adapt culling to mirroring
  - Cull front faces
  - Do *not* cull back faces
- Draw the object (mirrored)
- Draw the mirror (transparent)
- Draw the object (normal)



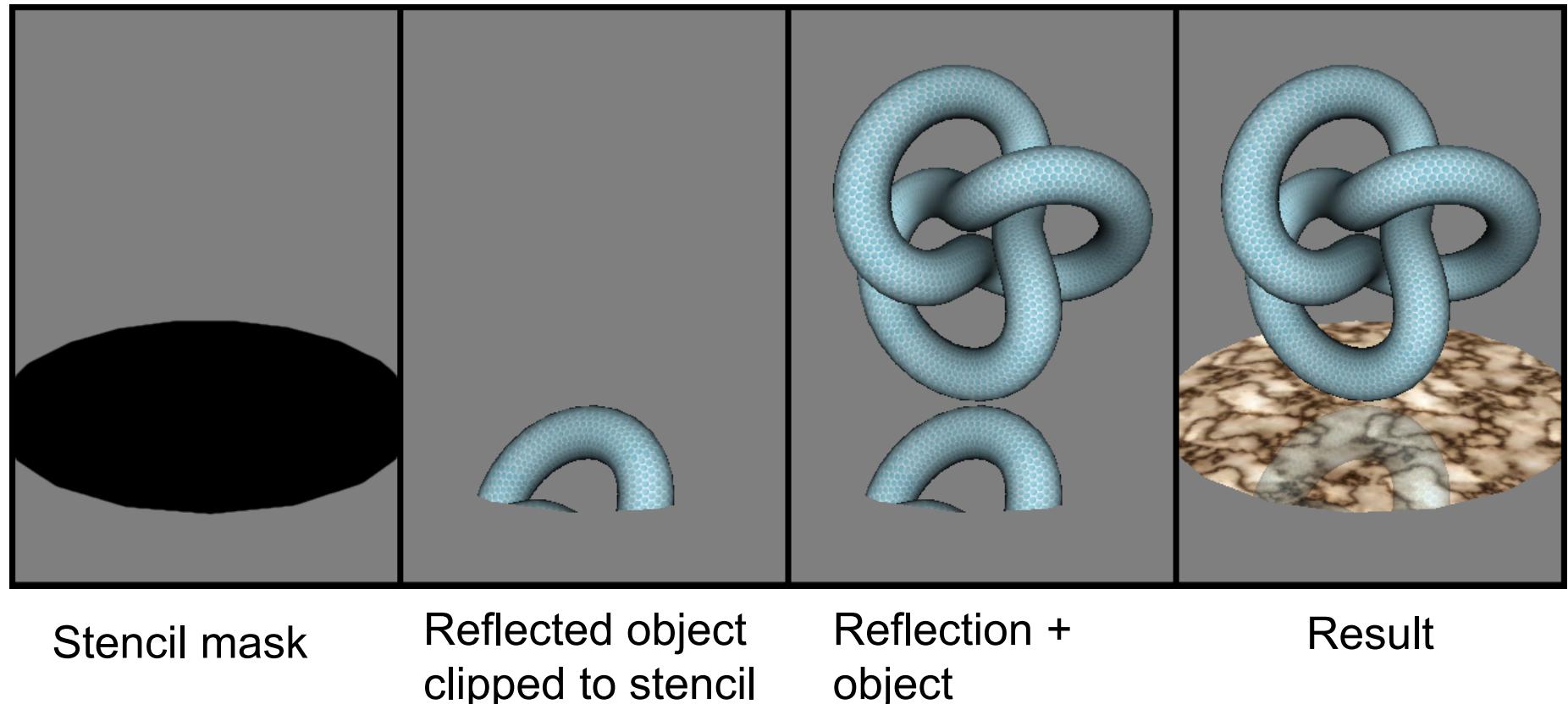
# Problem



Object not restricted to mirror

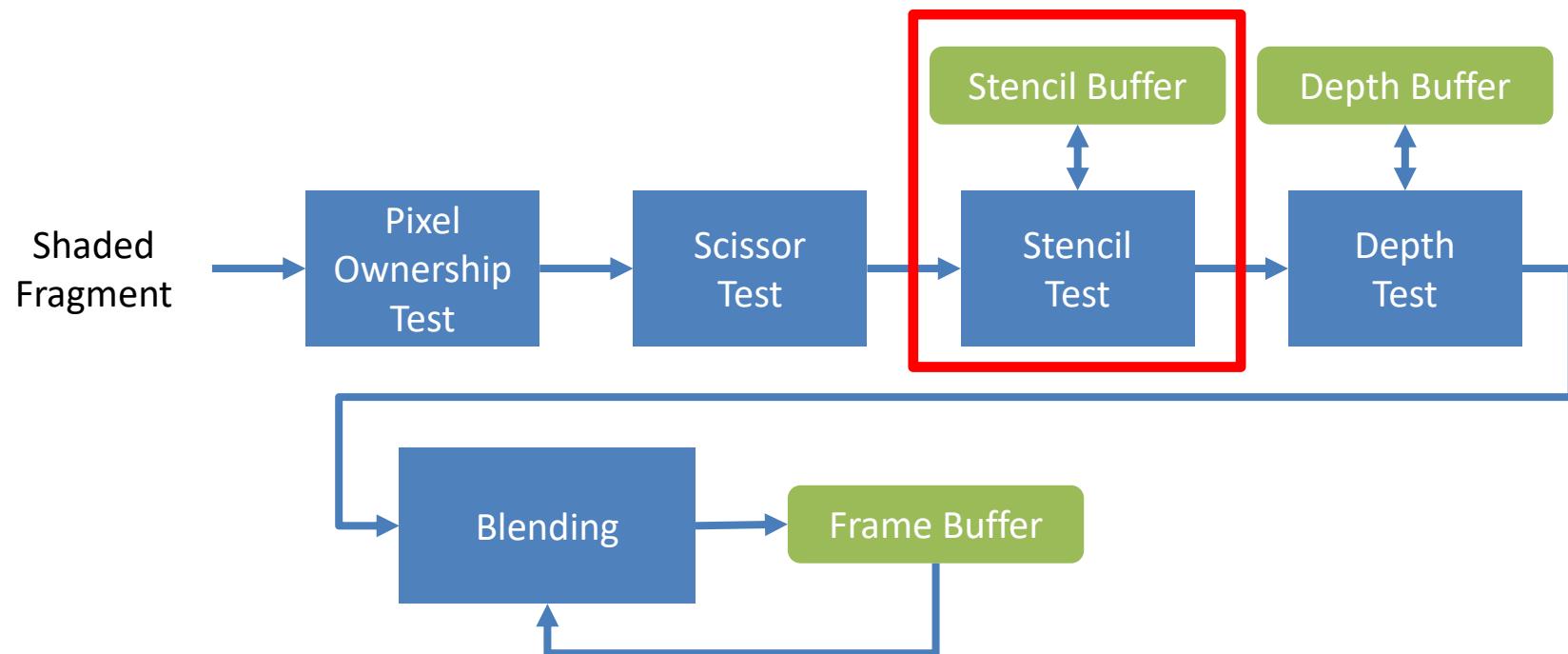
Solution: Stencil buffer

# Reflection with Stencil Buffering



# Fixed-Function Raster Operations

- When does the stencil test happen?



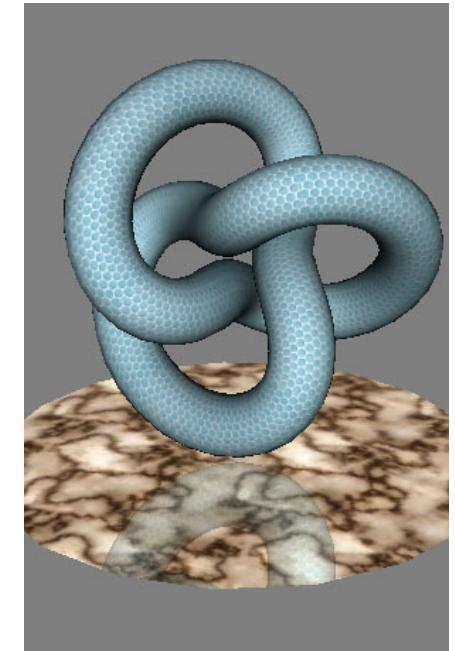
# Stencil Buffer

- Allows to mask parts of the framebuffer
- 1 Bit Stencil Buffer ( $\in \{0,1\}$ )
- 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



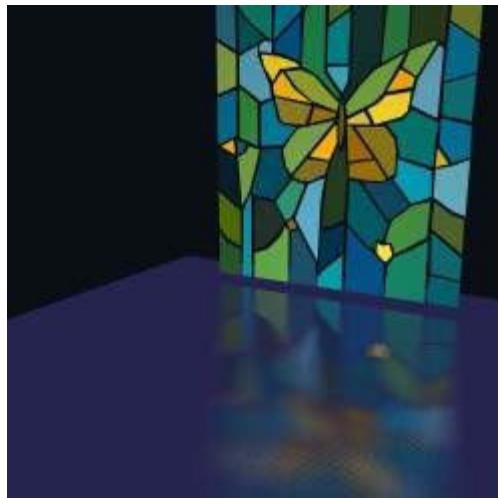
# Stencil Buffer Reflections

- 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



# Advanced Reflections with Shaders

- Fade object color with increasing distance to reflector
- Fuzzy reflections: compute distorted texture coordinates



# Caveats

handhabung

- Reflection changes handedness
  - Adjust winding order of front faces when rendering reflected geometry
- If scene intersects mirror plane
  - Need to clip against plane to avoid artifacts

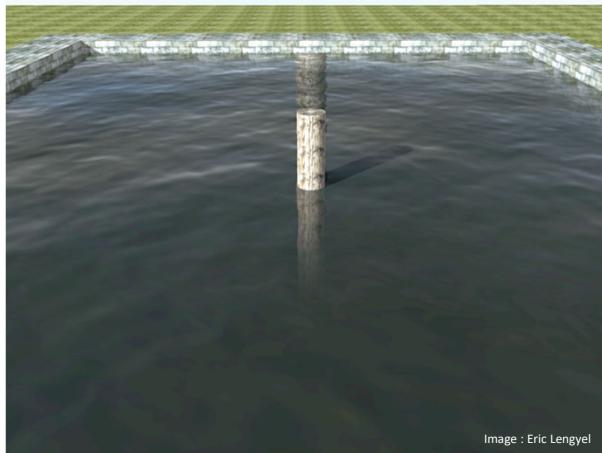


Image : Eric Lengyel

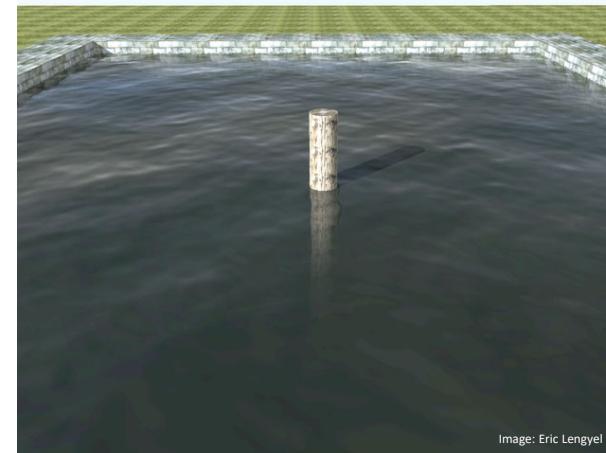
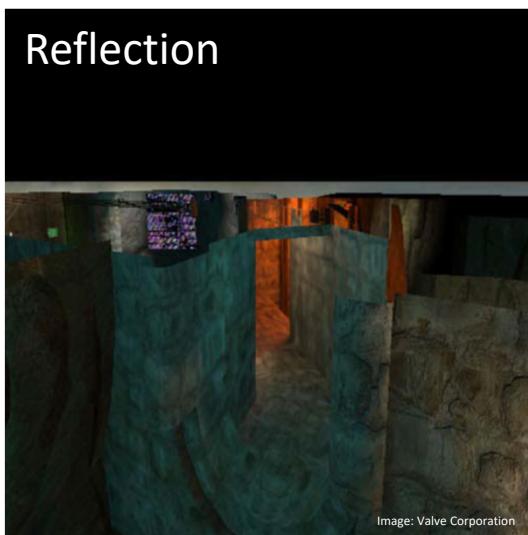


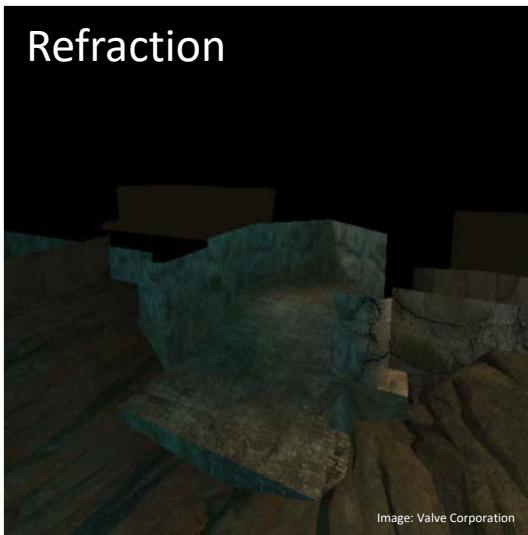
Image: Eric Lengyel

# Shader-Based Approach

Reflection



Refraction



Dieter Schmalstieg

- Can also do refractions using shaders
- Render parts of the scene below the plane into additional texture
- Use fog to simulate scattering in water

Result



Semi-Global Illumination

# Screen Space Reflections

Picture: Killing Floor 2 (Tripwire Interactive)



# Screen Space Reflections (SSR)

- 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

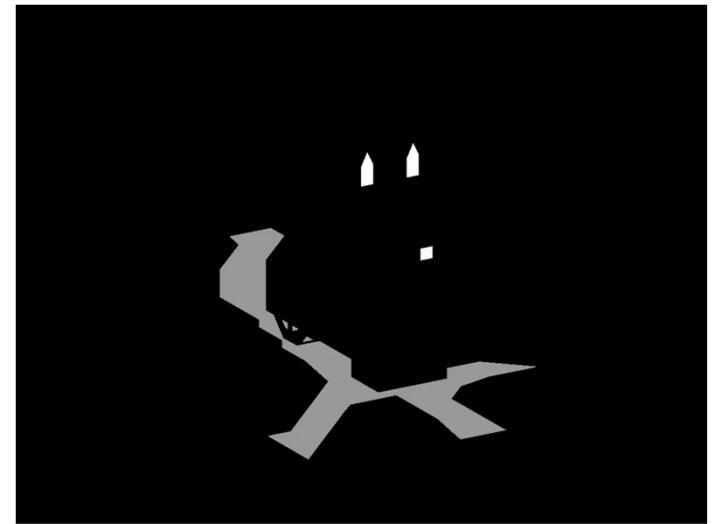
Screen Space Reflections (SSR) sind eine Technik in der Computergrafik, die es ermöglicht, Spiegelungen auf Oberflächen in Echtzeit darzustellen. SSR berechnet die Reflexionen auf der Grundlage der Informationen, die auf dem Bildschirm sichtbar sind, anstatt die Reflexionen in der 3D-Szene selbst zu 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>

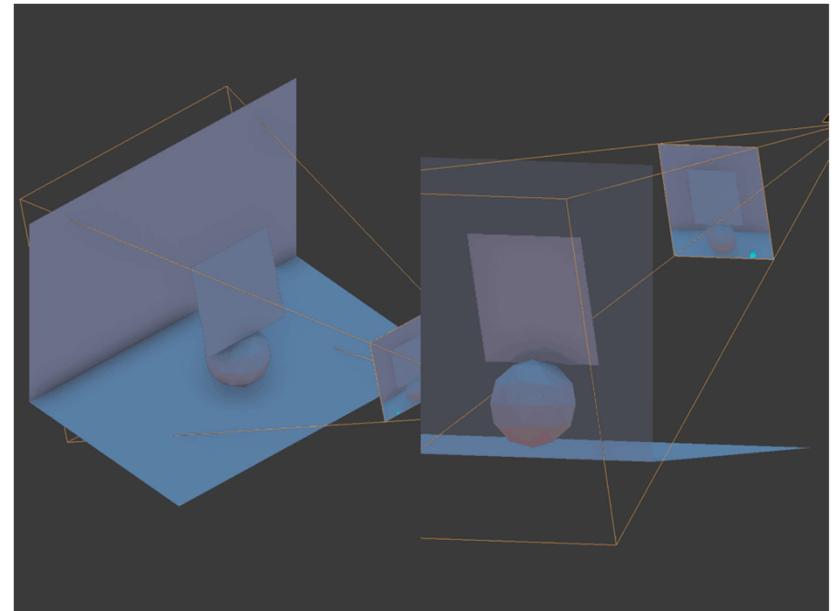
# SSR: Reflection Map

- 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: Ray Marching Code

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

```
#define MAX_REFLECTION_RAY_MARCH_STEP 0.02f
#define NUM_RAY_MARCH_SAMPLES 16

bool GetReflection(
    vec3 ScreenSpaceReflectionVec,
    vec3 ScreenSpacePos,
    out vec3 ReflectionColor)
{
    // Raymarch in the direction of the ScreenSpaceReflectionVec until
    // you get an intersection with your z buffer
    for (int RayStepIdx = 0; RayStepIdx<NUM_RAY_MARCH_SAMPLES; RayStepIdx++)
    {
        vec3 RaySample = (RayStepIdx * ReflectionRayMarchStep)
            * ScreenSpaceReflectionVec + ScreenSpacePos;

        float ZBufferVal = texture(DepthSampler, RaySample.xy).r;
        if (RaySample.z > ZBufferVal)
        {
            ReflectionColor = texture(SceneColorSampler, RaySample.xy).rgb;
            return true;
        }
    }
    return false;
}
```

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

# SSR: Intersection Code

- Fixed step size can lead to inaccurate results
- Use binary search between points just before and after depth intersection

```
#define NUM_BINARY_SEARCH_SAMPLES 6
// PrevRaySample: Sample just before the depth buffer intersection
// RaySample: Sample just after the depth buffer intersection
// DepthSampler: sampler2D to access the rendered depth texture

if (bFoundIntersection)
{
    vec4 MinRaySample = PrevRaySample;
    vec4 MaxRaySample = RaySample;
    vec4 MidRaySample;

    for (int i = 0; i < NUM_BINARY_SEARCH_SAMPLES; i++)
    {
        MidRaySample = mix(MinRaySample, MaxRaySample, 0.5);
        float ZBufferVal = texture(DepthSampler, MidRaySample.xy).r;

        if (MidRaySample.z > ZBufferVal)
            MaxRaySample = MidRaySample;
        else
            MinRaySample = MidRaySample;
    }
}
```

# SSR: Limitations 1

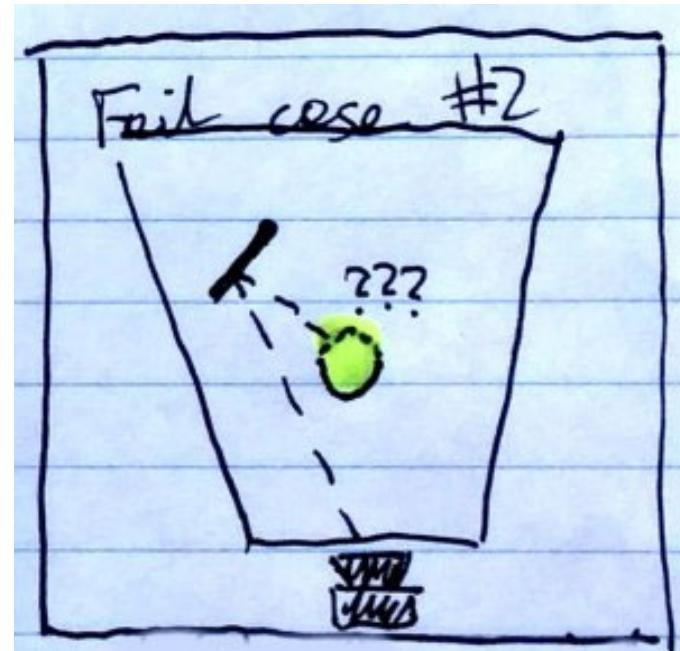
- Viewer facing reflections
- Reflections outside viewport



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

# SSR: Limitations 2

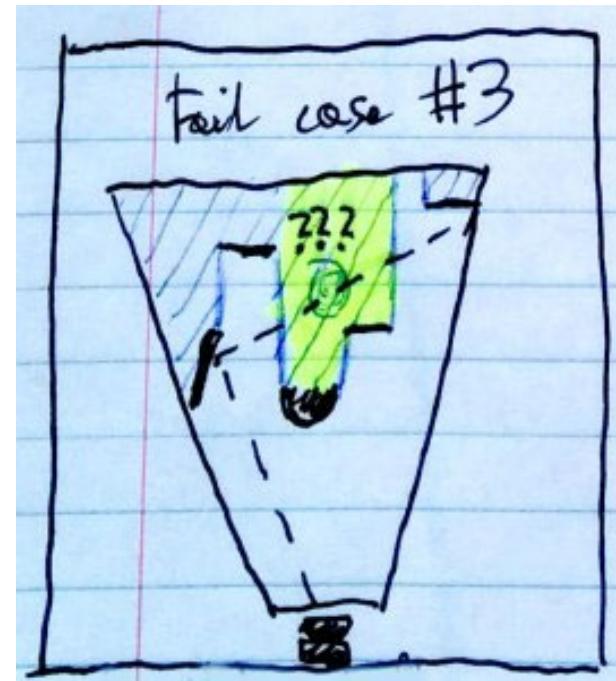
- Reflections of back faces



[https://bartwronski.com/2014/01/25/the  
future-of-screenspace-reflections/](https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/)

# SSR: Limitations 3

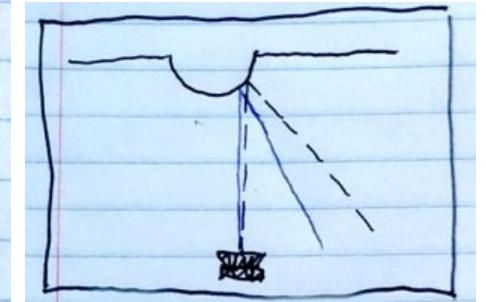
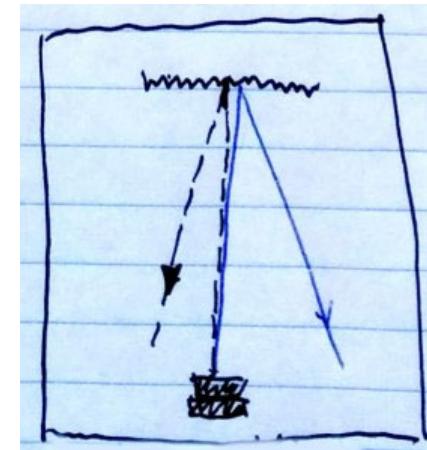
- Depth complexity



<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

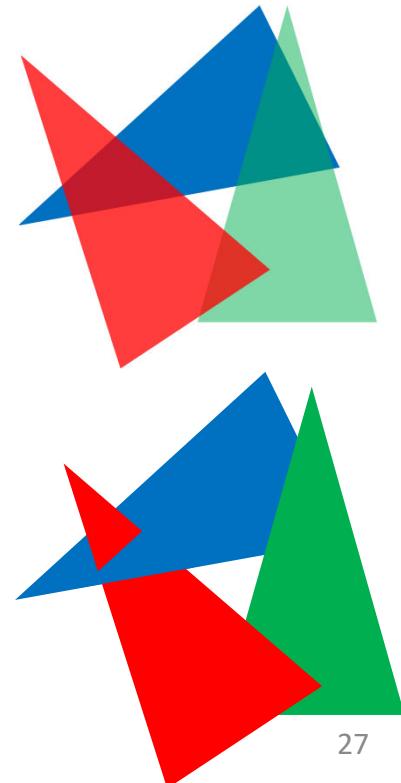


# Methods for Sorting Fragments

- 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)

# Explicit Depth Sorting

- Explicit depth sorting into draw buffer
- (+) Rendering with standard GPU primitive order
- (-) Writing sorted order to memory is expensive
- (-) Not always good enough
  - Cyclic overlap
  - Intersecting triangles
  - Can either split triangles (ugly, brittle), or
  - Can sort per fragments instead

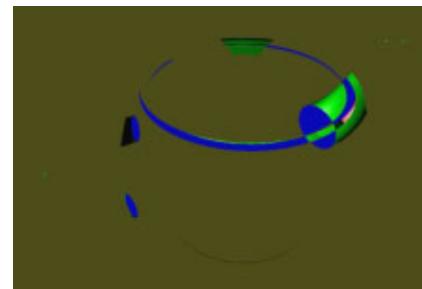
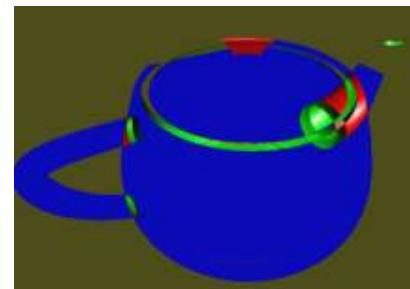
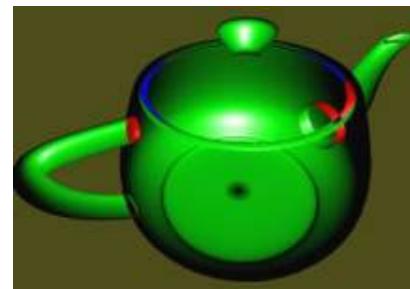


Depth Peeling ist eine Technik in der Computergrafik, die es ermöglicht, Transparenz- und Alphablitzeffekte in Echtzeit darzustellen. Dies erreicht indem es Schichten von transparenten Polygonen voneinander trennt und jede Schicht einzeln berechnet und darstellt. Dies ist wichtig, weil Standard-Transparenztechniken nicht in der Lage sind, korrekt die richtige Reihenfolge der transparenten Polygone zu bestimmen und dadurch falsche Ergebnisse liefern, wenn transparente Objekte sich überlappen. Depth Peeling ermöglicht es, die richtige Reihenfolge der Transparenz korrekt darzustellen, was die Qualität der dargestellten Bilder verbessert.

# Depth Peeling

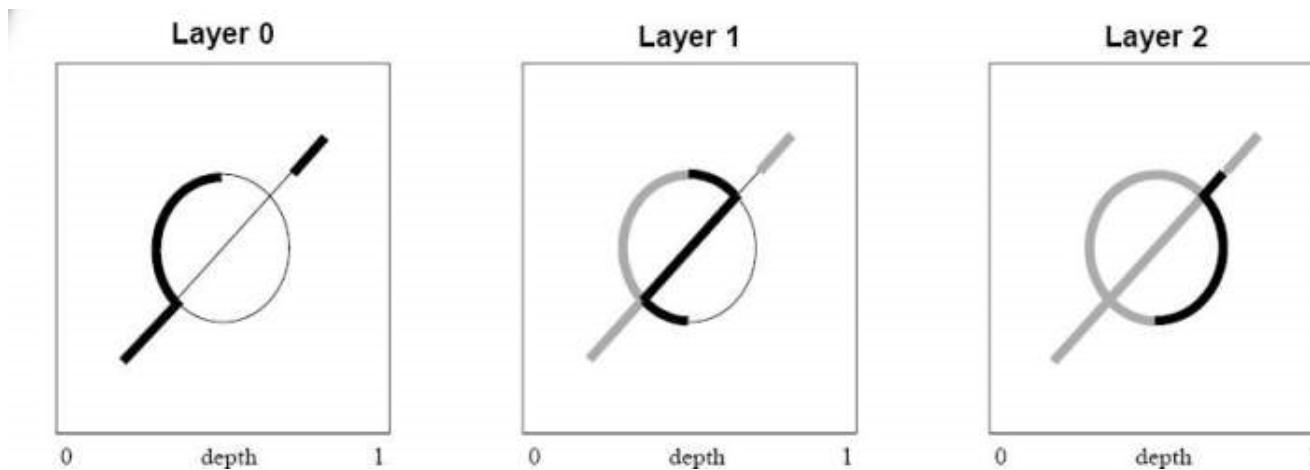
Interactive order independent transparency [Everitt'01]

- 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
- Save the results in texture using render-to-texture/FBO
- Each pass forms a layer
- Starting from the second pass, use a shader for comparison with previous layer
- Compose  $n$  layers



Dual Depth Peeling ist eine Weiterentwicklung der Depth Peeling Technik, die die Leistung und Qualität der Darstellung von Transparenz und Alphablending Effekten in Echtzeit verbessert. Es erreicht dies indem es zwei Schichten von transparenten Polygonen gleichzeitig berechnet und darstellt. Im Gegensatz zu der einfachen Depth Peeling Technik, die nur eine Schicht berechnet und darstellt, ermöglicht Dual Depth Peeling es, die Informationen von zwei Schichten gleichzeitig zu berücksichtigen, was die Qualität der Darstellung verbessert und die Leistungsaufnahme verringert.



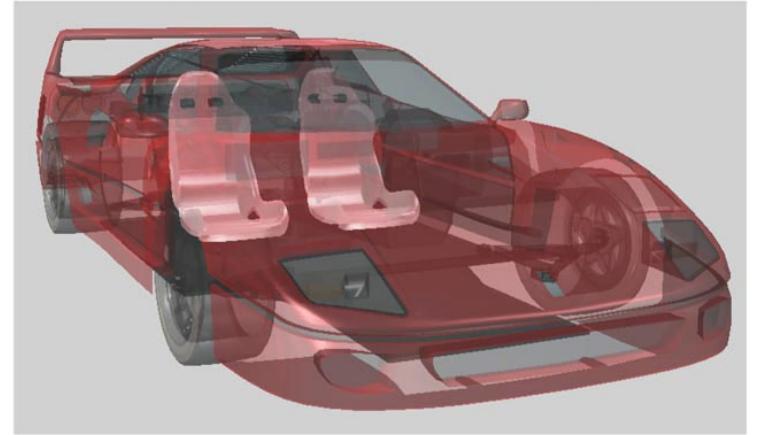
# Dual Depth Peeling

- Peel front and back in one pass
- Keep track of min/max depth in two textures
- Compare current depth with min/max values
  - If  $<$  min write to front layer
  - If  $>$  max write to back layer
- Use previous front/back layer depth values in current pass
- Decreases number of passes from  $n$  to  $n/2+1$

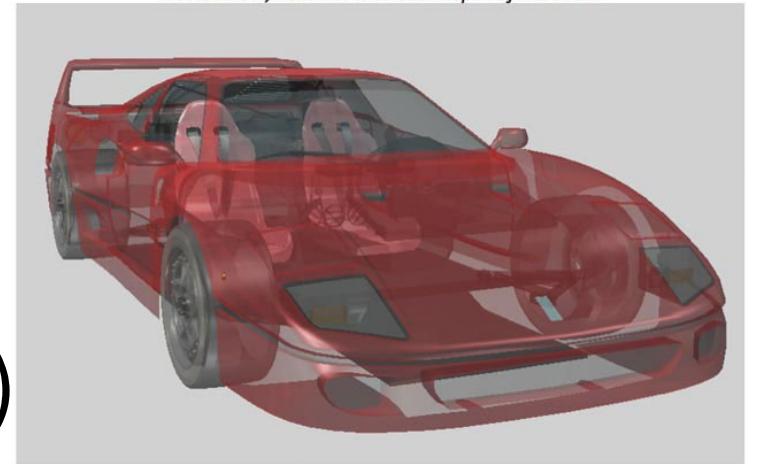
In dem Kontext von Dual Depth Peeling werden Deep Framebuffers verwendet, um die Informationen der transparenten Schichten zu speichern und zu verwalten. Ein Deep Framebuffer ist ein spezielles Framebuffer, das nicht nur die Farbinformationen eines Pixels speichert, sondern auch die Tiefeninformationen. Dies ermöglicht es, die Tiefeninformationen von transparenten Polygonen zu speichern und zu verwalten, was erforderlich ist, um die richtige Reihenfolge der Transparenz korrekt darzustellen.

# 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)*



Without OIT, note the incorrect depth of the seats



With OIT applied

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

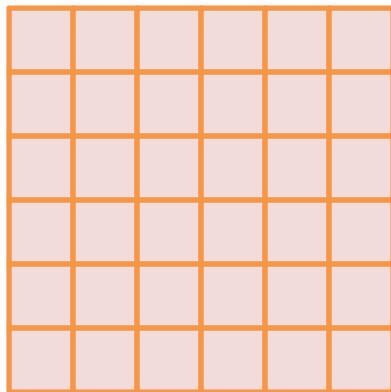
- Algorithm
  - Render scene and generate linked list per pixel
  - Sort list and compose fragments
- Requires read/write buffer
- Need two additional buffers
  - Fragment and link buffer
  - Start offset buffer

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.

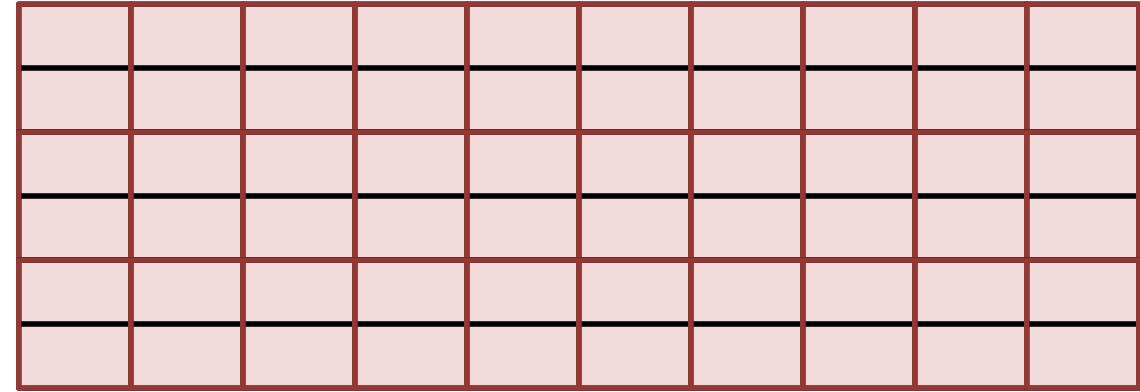
# Fragment and Link Buffer

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

Viewport



Fragment and Link Buffer



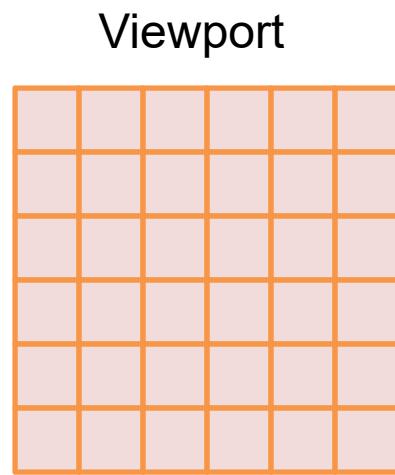
# Start Offset Buffer

- 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



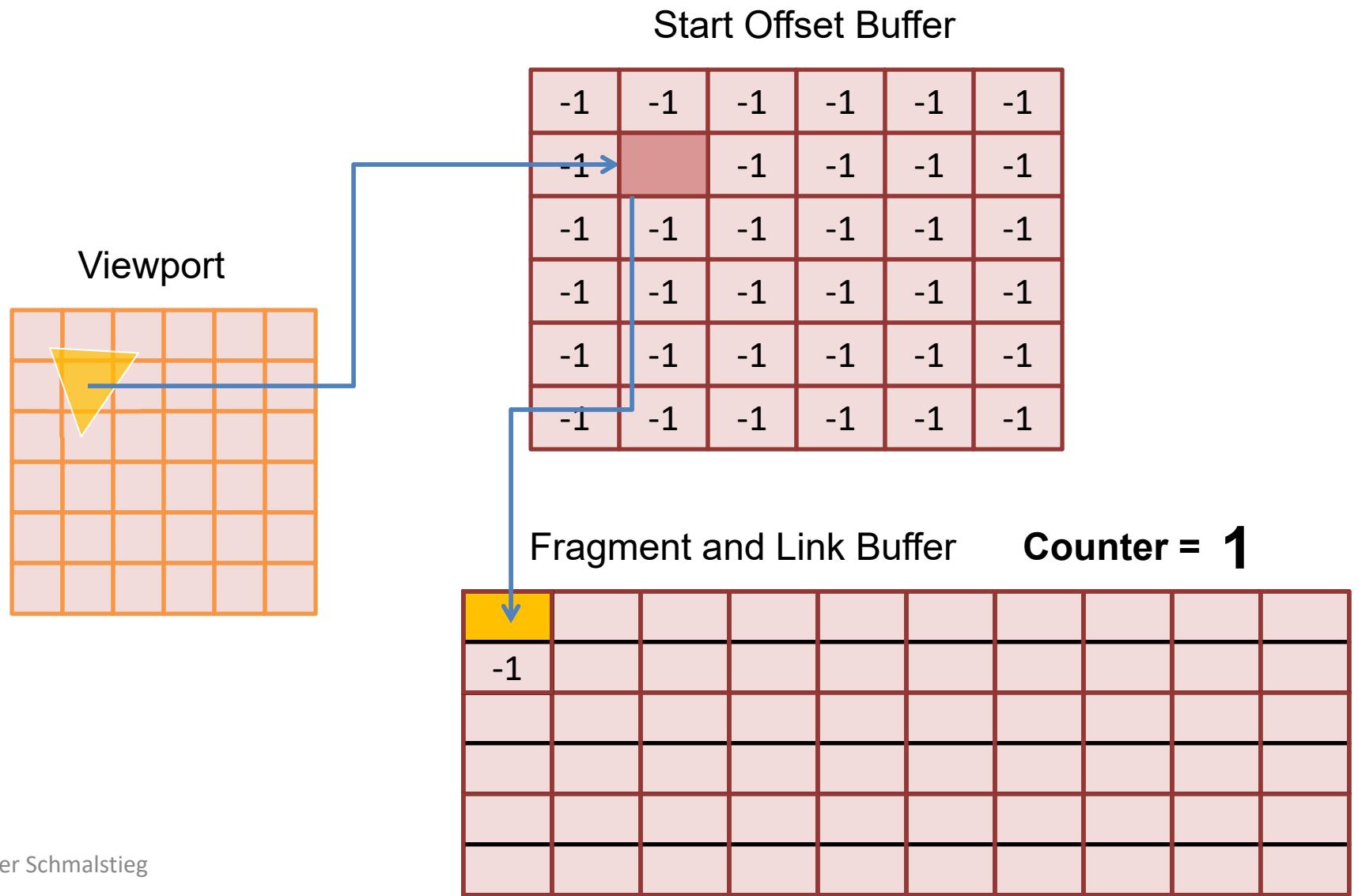
Start Offset Buffer

-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

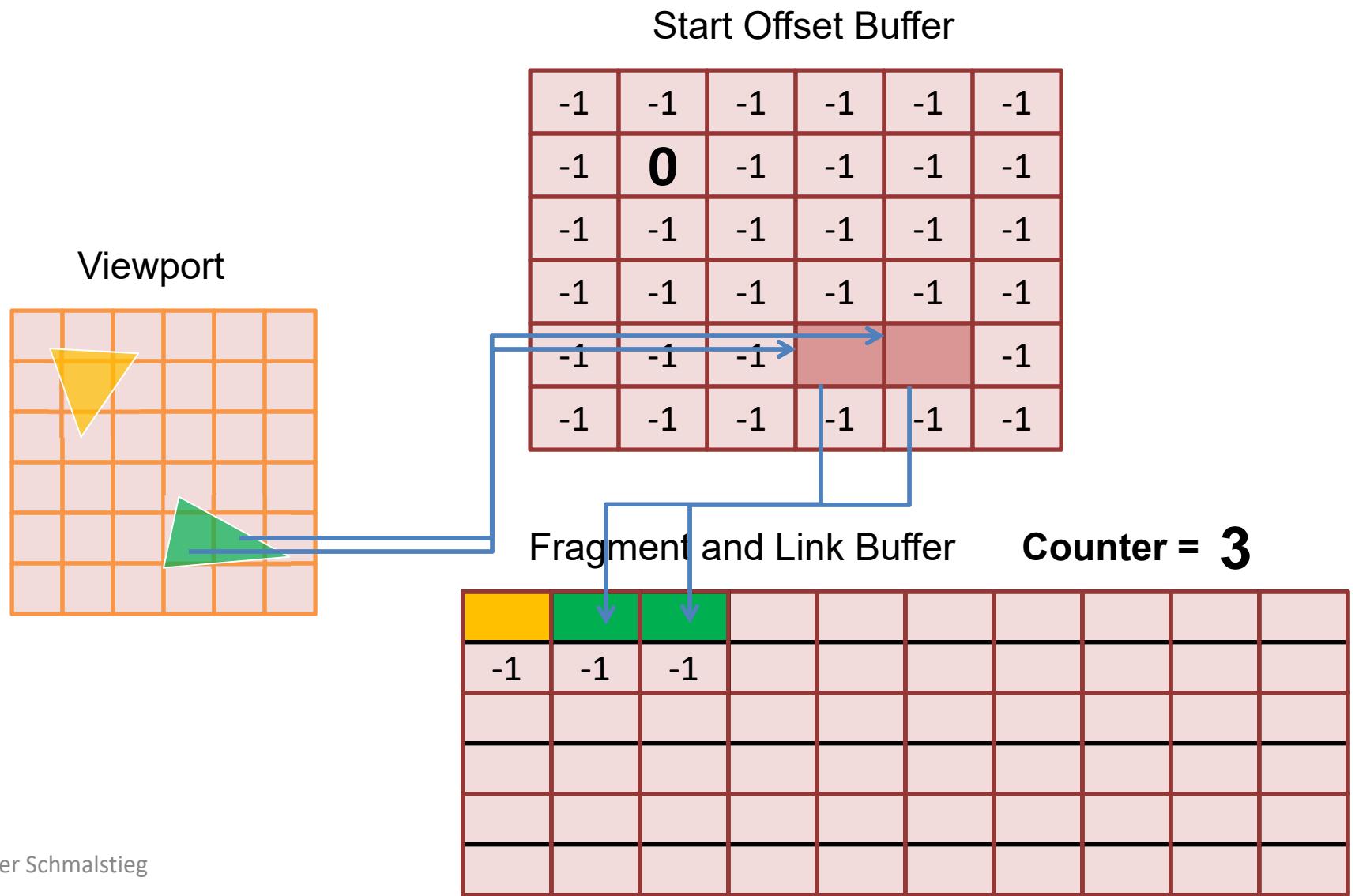
Fragment and Link Buffer

**Counter = 1**

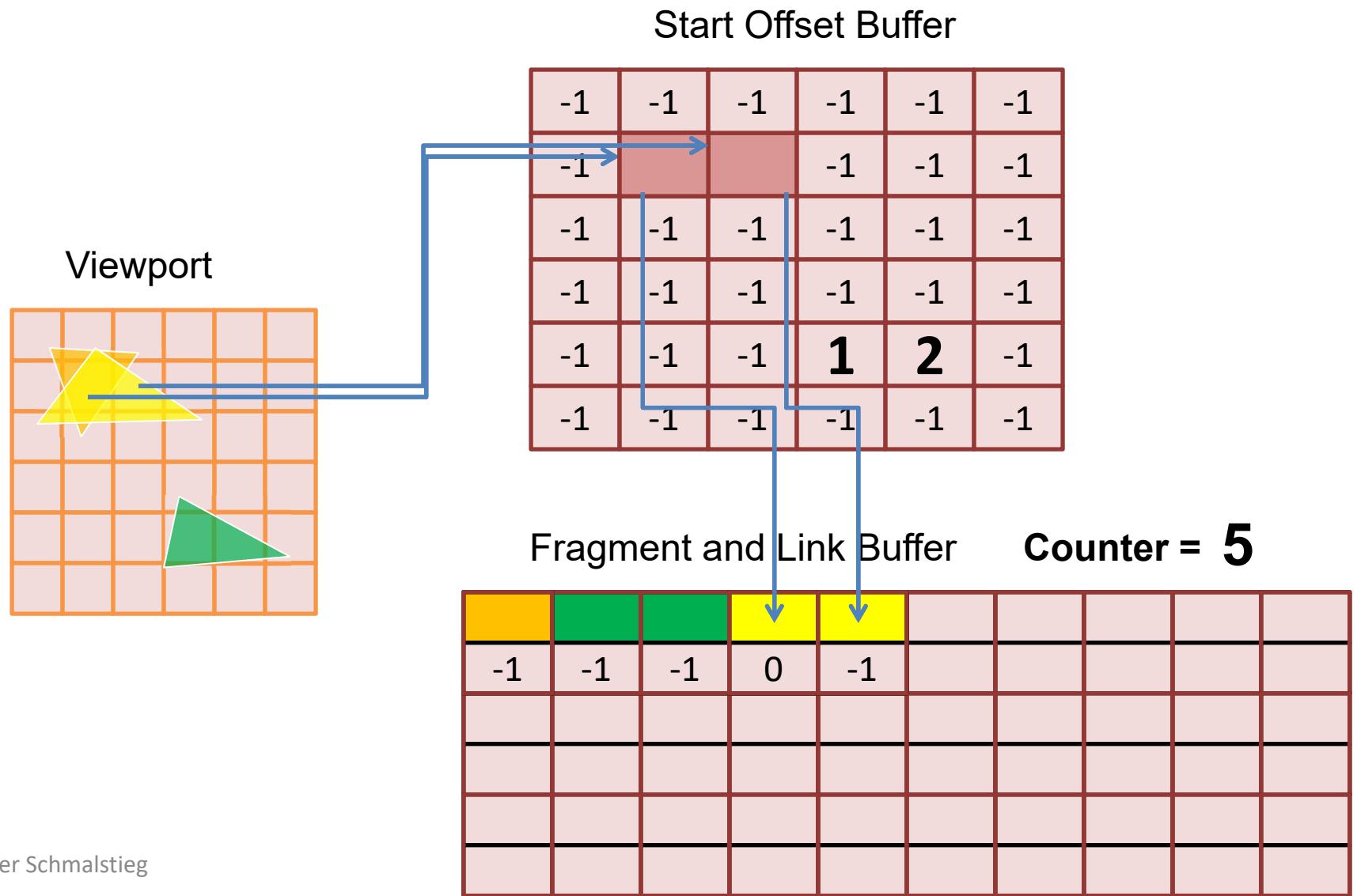

# Linked List Creation 2



# Linked List Creation 3



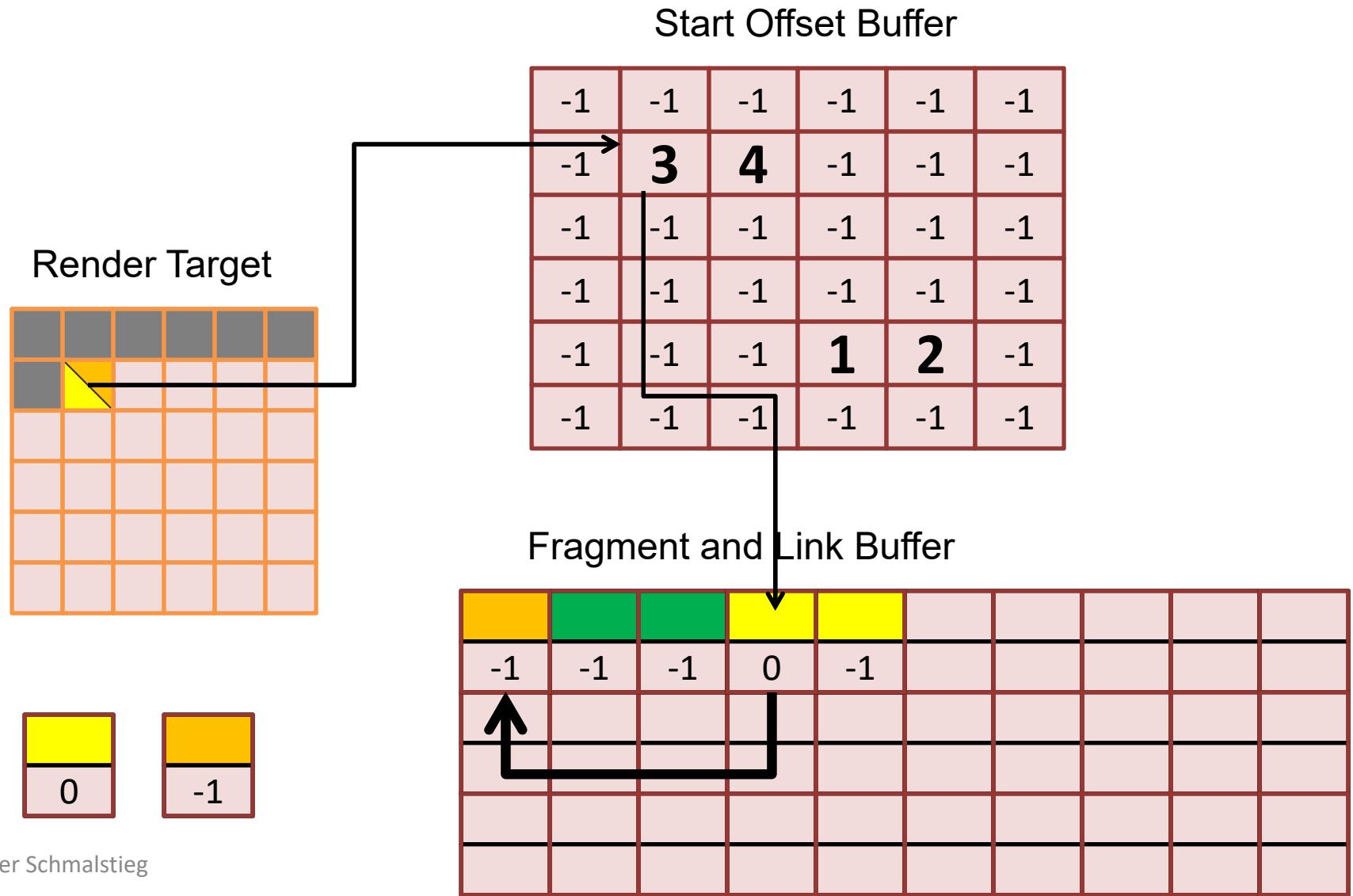
# Linked List Creation 4



# Linked List Traversal

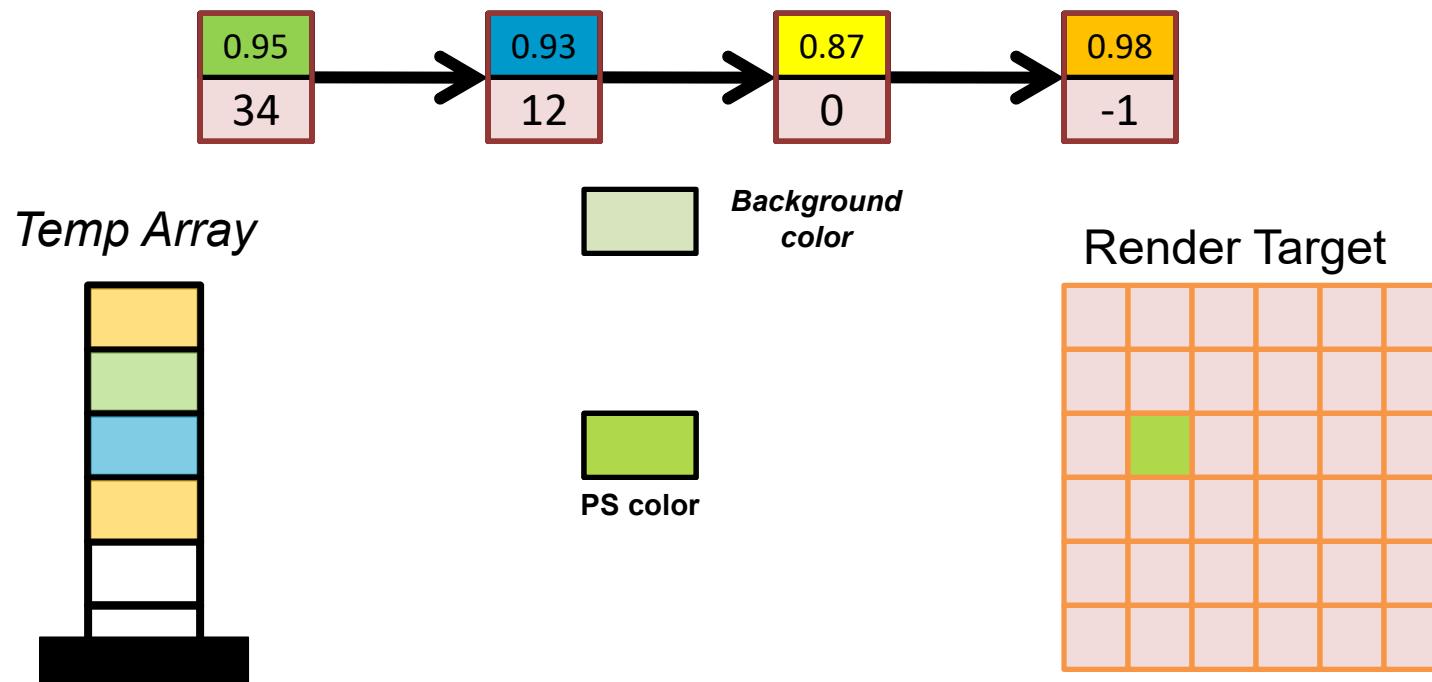
- Render a fullscreen quad (or compute shader)
- For each pixel
  - Parse the linked list
  - Retrieve fragments for this screen position
- Process list of fragments (Sort and Blend)

# Rendering from Linked List



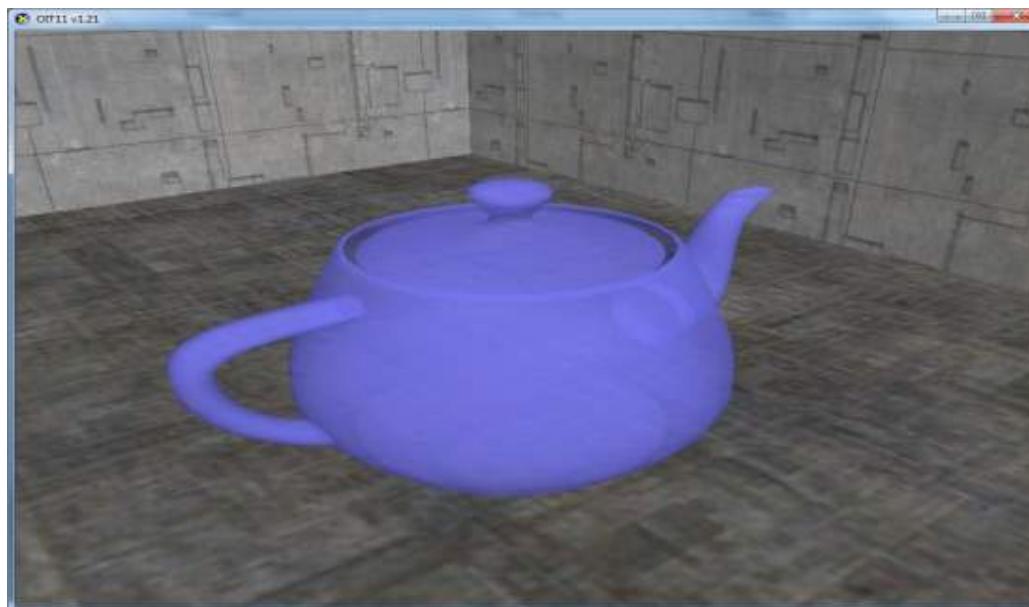
# Sorting and Blending

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



# Performance Comparison

	Teapot	Dragon
Linked List	743 fps	338 fps
Depth Peeling	579 fps	45 fps
Dual Depth Peeling	---	94 fps



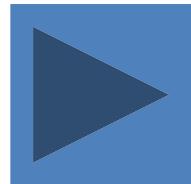
Dieter Schmalstieg



Semi-Global Illumination

# Result

- 602K scene triangles VIDEO
  - 254K transparent triangles

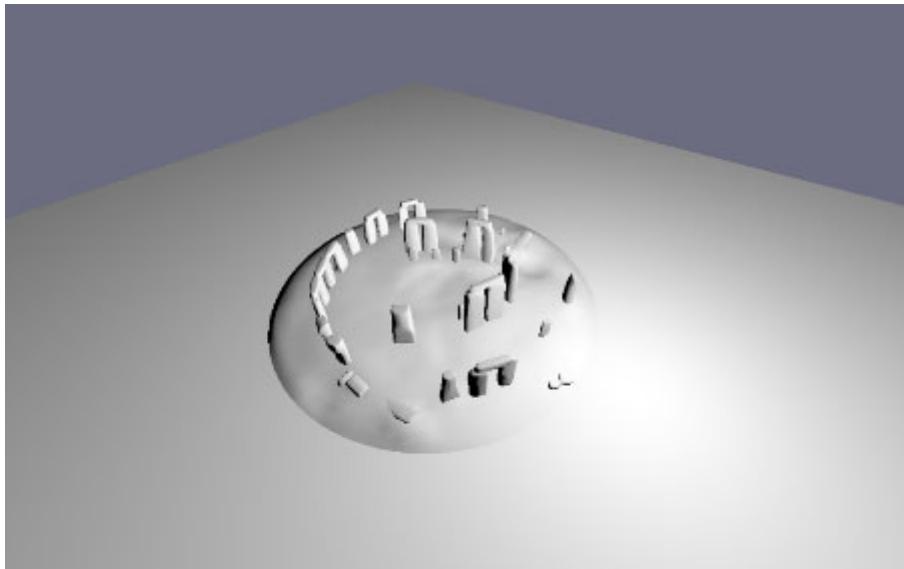


<https://youtu.be/SYrHi4jF4dU>

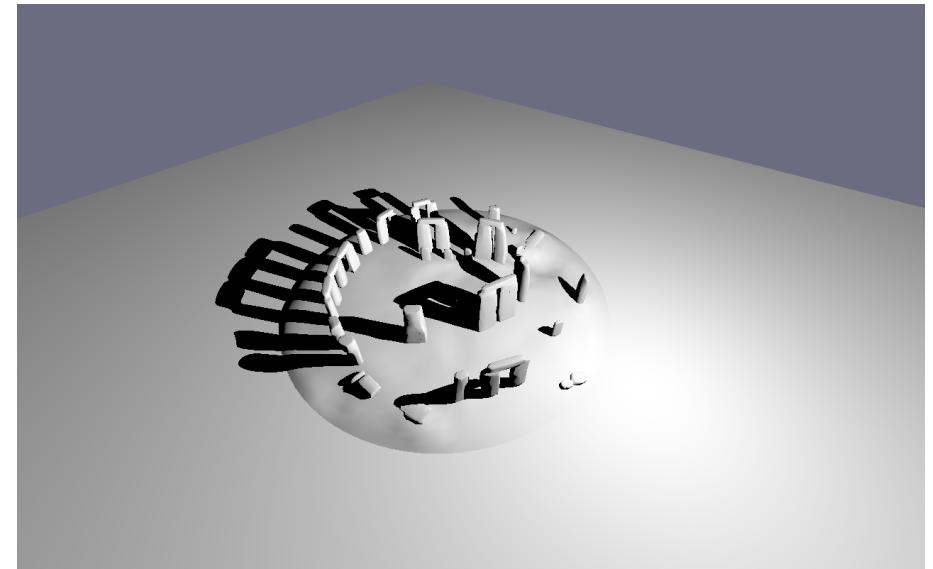


# Shadows

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



Dieter Schmalstieg



Semi-Global Illumination

# Shadows for Visual Realism

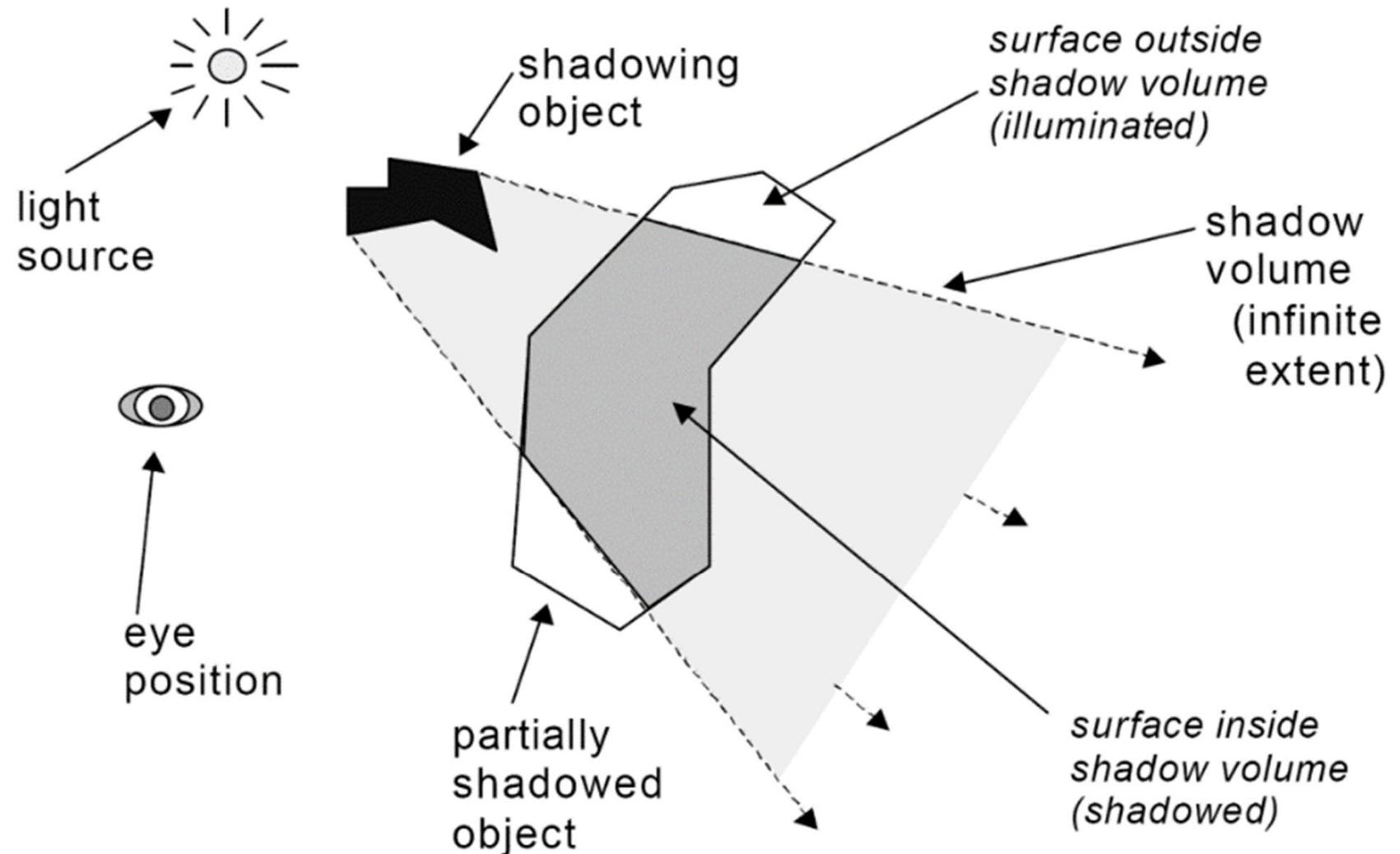


Objects look like they are “floating” → Shadows can fix that!

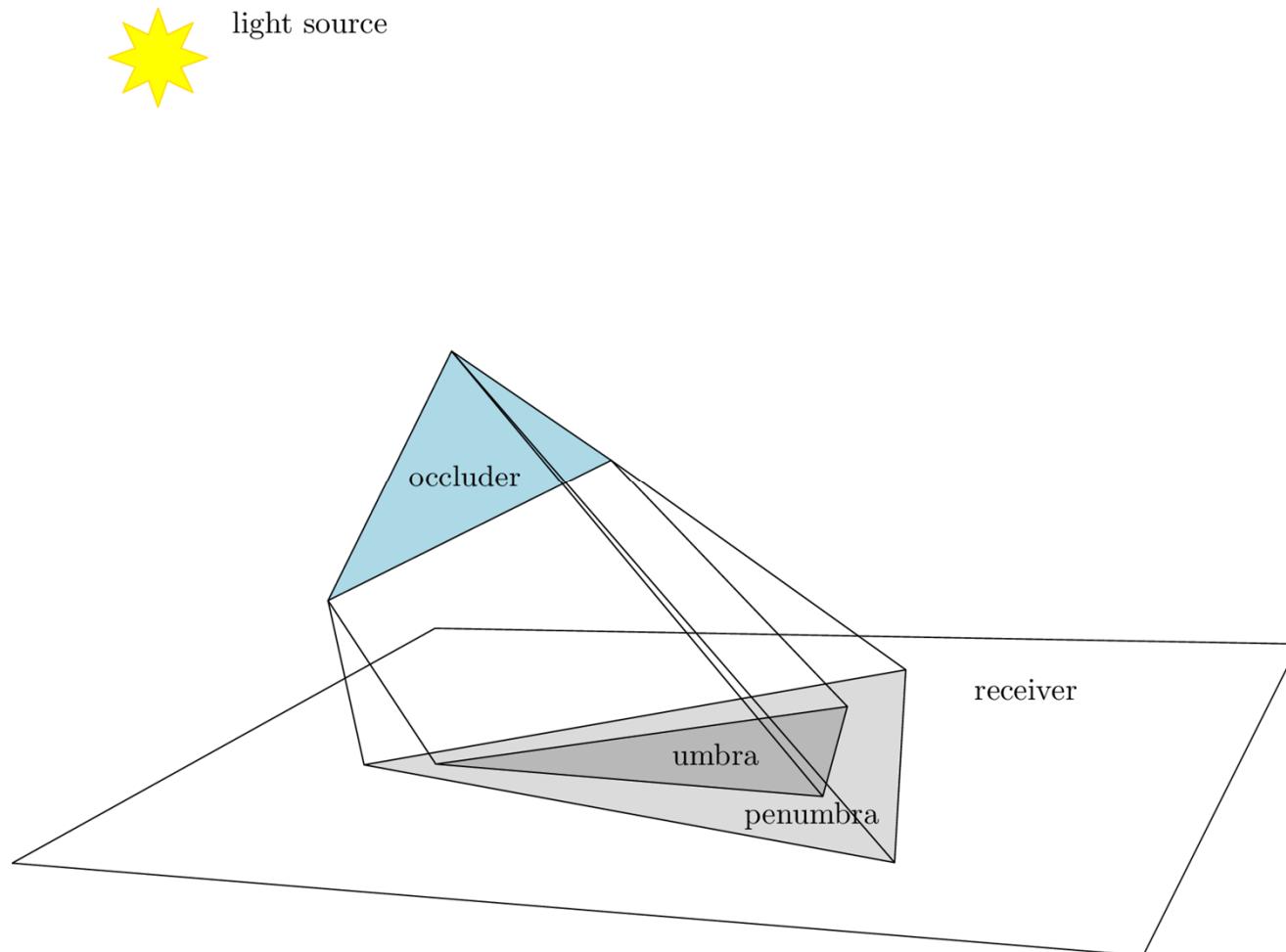
# Shadows are Non-Local

- Shadows are a non-local effect between
  - Light source
  - Object currently being rendered (“receiver”)
  - Object blocking light from reaching rendered object (“blocker”, “occluder”, “caster”)
- Shading can be seen as pseudo self-shadowing

# Shadow Geometry



# Umbra and Penumbra



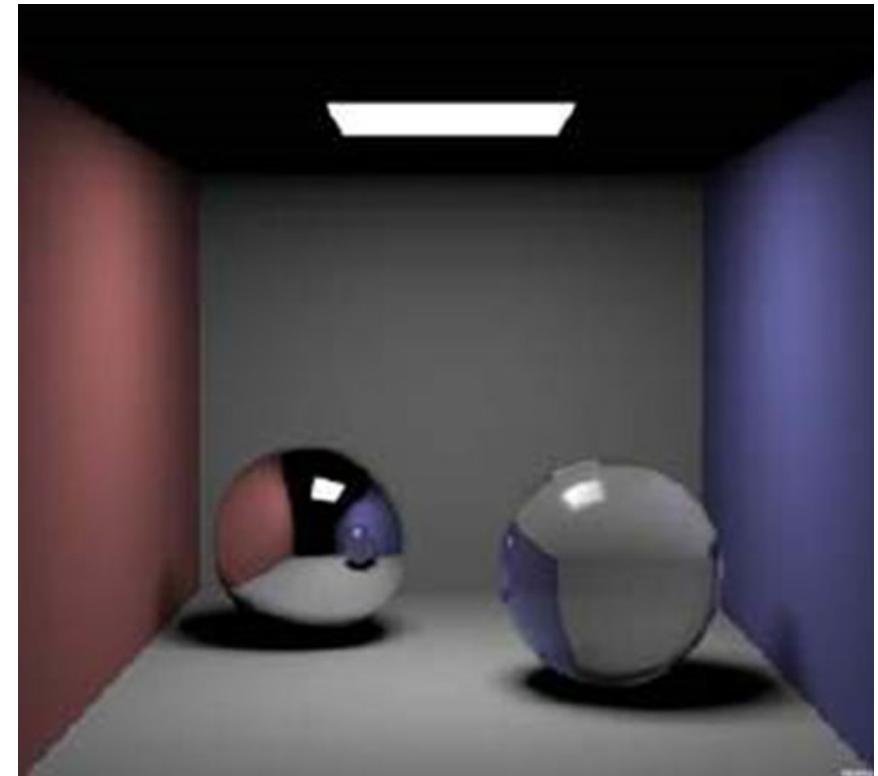
# Global Illumination

Shadows as byproduct of global light transport



Radiosity

Dieter Schmalstieg



Raytracing

Semi-Global Illumination

49

# The Problem

- Shadows are a *global* illumination (GI) problem
- GI is hard, especially in online rendering
  - Global information not available during rendering
- Address problem with two-pass techniques
  1. Compute and store (global) shadow information
  2. Render scene using that information

# Real-Time Shadows

- Static shadows
  - Light maps - see also previous lectures
  - Updating in real-time is problematic
- Approximate shadows
- Planar projected shadows
- Shadow Maps
- Stencil Shadows

# Static Shadows

- Glue to surface whatever we want
- Idea: incorporate shadows into light maps
  - For each texel, cast ray to each light source
- Bake soft shadows in light maps
  - Not by texture filtering alone
  - Must also sample area light sources

# Static Soft Shadow Example

1 sample

no filtering



filtering

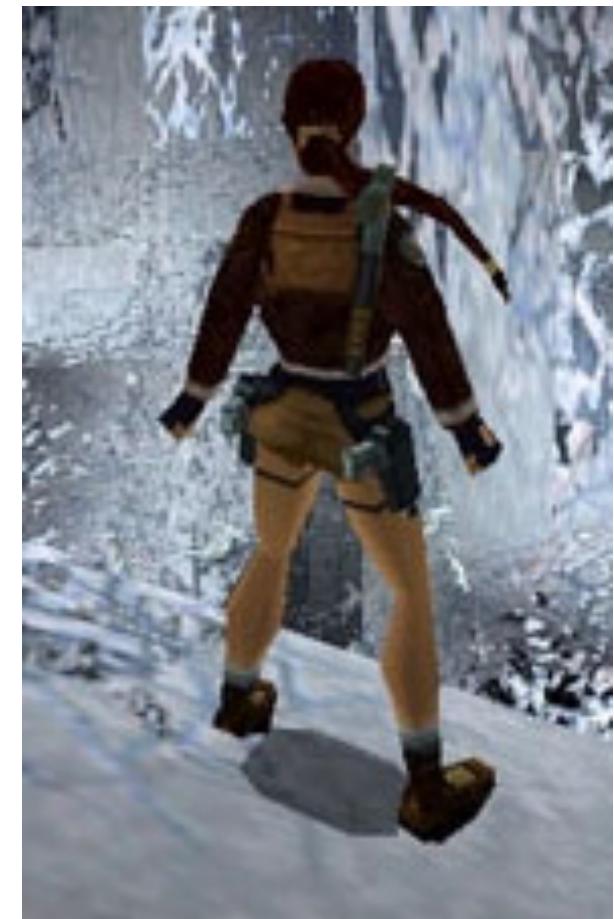


$n$  samples



# Approximate Shadows 1

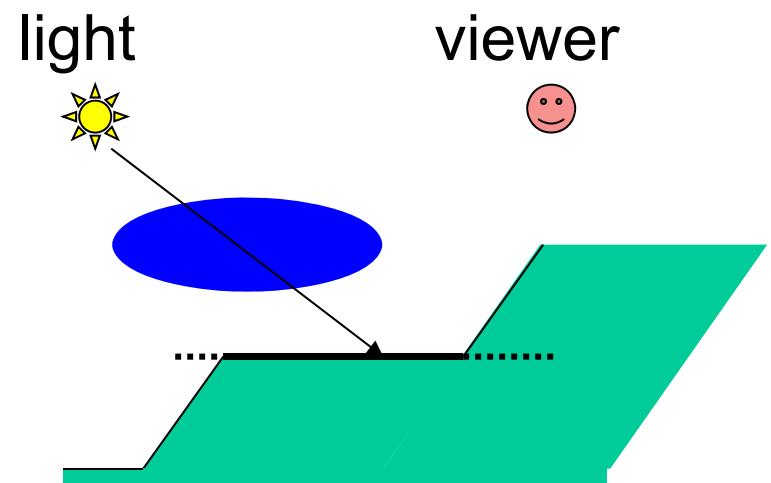
- Hand-drawn approximate geometry
  - Perceptual studies suggest:  
shape not so important
  - Minimal cost



You can tell how young Lara was by counting the vertices of her shadow...

# Approximate Shadows 2

- Dark polygon (maybe with texture)
  - Cast ray from light source through object center
  - Blend polygon into frame buffer at location of hit
  - May apply additional rotation/scale/translation to incorporate distance and receiver orientation
- Problem with z-quantization



Blend at hit polygon + z-test equal  $\rightarrow$  z-buffer quantization errors!

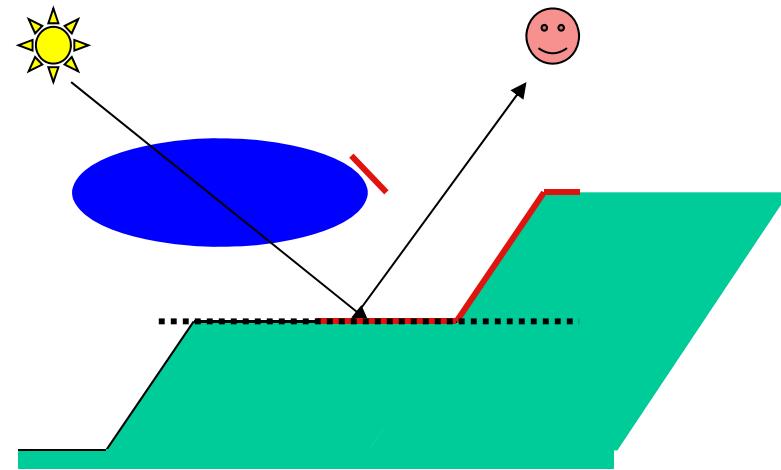
# Approximate Shadows 3



light ☼

viewer ☺

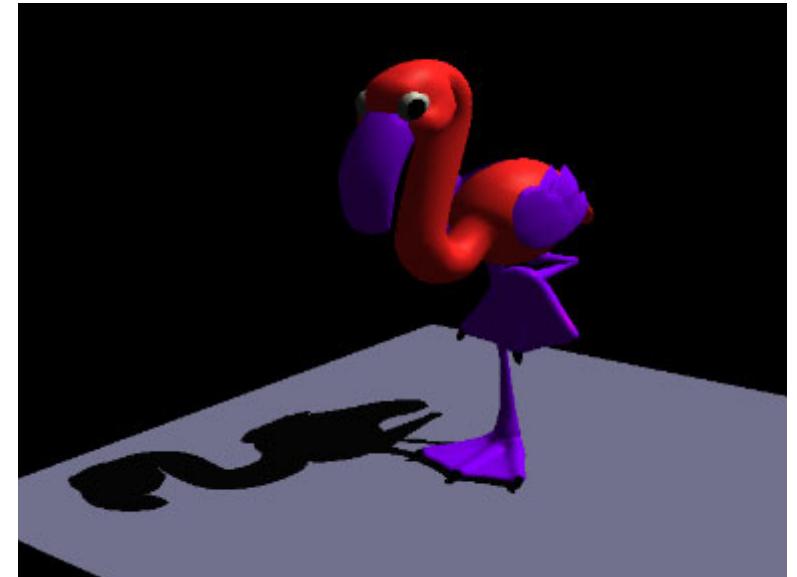
Elevate above hit polygon + Z-test less or equal  
→ shadow too big → may appear floating



No z-test, only one eye ray → shadow too big, maybe in wrong place

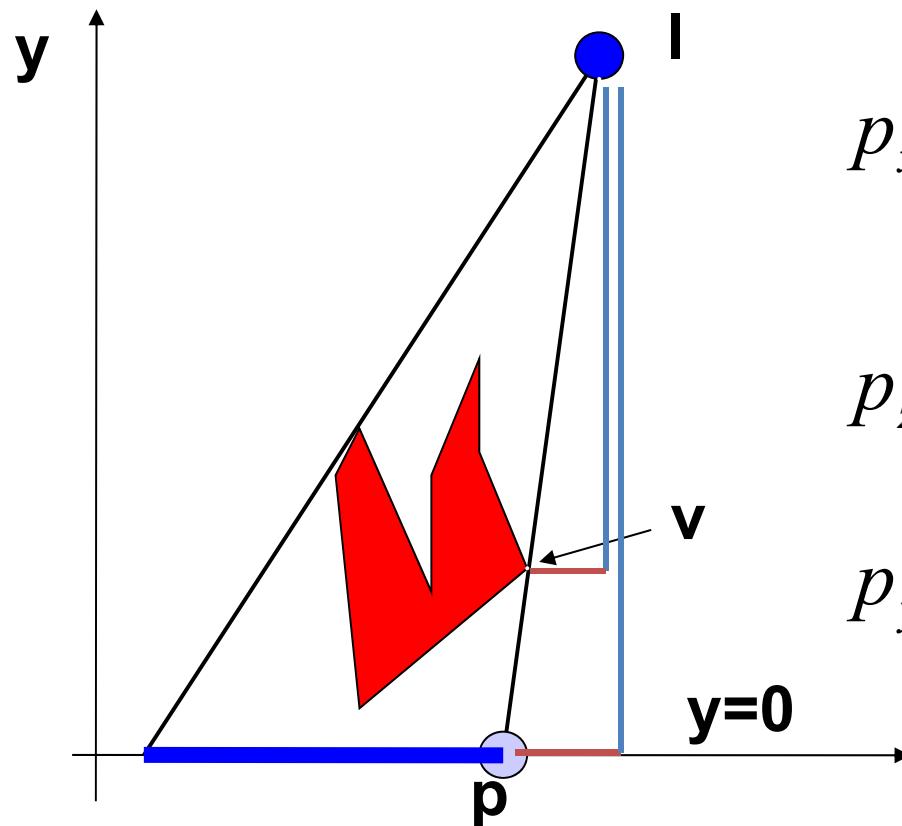
# Planar Projected Shadows

- Very simple trick [Jim Blinn, 1988]
- Supports arbitrary shadow caster
- But only planar shadow receiver
- Method
  - ModelView matrix transforms object into plane
  - Object is drawn “flat”
  - Darken surface underneath using framebuffer blend



# Projection Onto XZ-Plane

- $\mathbf{p} = \mathbf{M} \mathbf{v}$
- Similar triangles



$$\frac{p_x - l_x}{v_x - l_x} = \frac{l_y}{l_y - v_y}$$

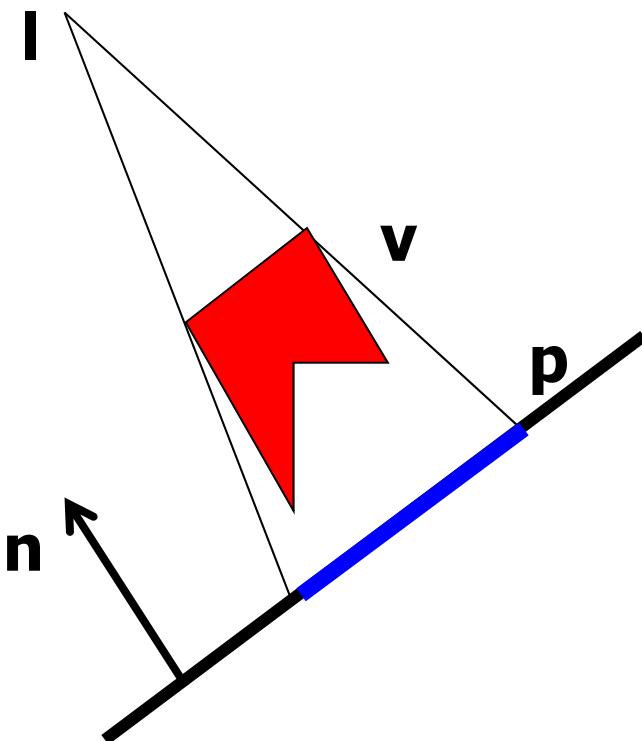
$$p_x = \frac{l_y v_x - l_x v_y}{l_y - v_y}$$

$$p_z = \frac{l_y v_z - l_z v_y}{l_y - v_y}$$

$$p_y = 0$$

$$\mathbf{M} = \begin{pmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{pmatrix}$$

# Projection Onto General Plane



$$\text{ray: } \mathbf{p} = \mathbf{l} + \alpha(\mathbf{v} - \mathbf{l}) \quad \dots \dots \alpha = (p - l) / (v - l)$$

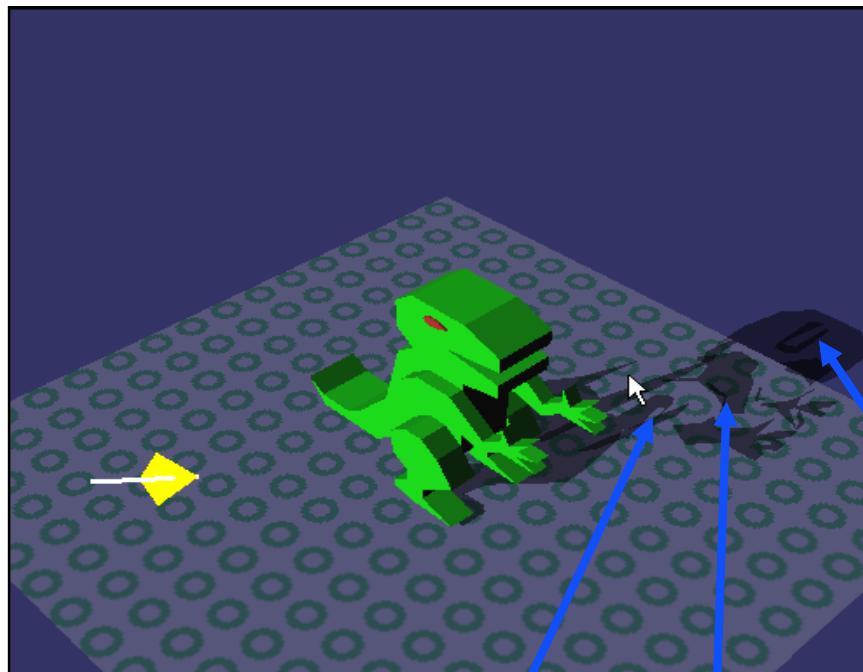
$$\text{plane: } \mathbf{n} \cdot \mathbf{p} + d = 0 \quad \dots \dots \begin{aligned} p &= -d/n \\ \alpha &= (-d/n - l) / (v - l) = -d/(n(v - l)) - l/(v - l) \end{aligned}$$

$$\mathbf{p} = \mathbf{l} - \frac{d + \mathbf{n} \cdot \mathbf{l}}{\mathbf{n} \cdot (\mathbf{v} - \mathbf{l})} (\mathbf{v} - \mathbf{l})$$

$$\mathbf{p} = \mathbf{M} \mathbf{v} \rightarrow \mathbf{M} = \begin{pmatrix} \mathbf{n} \cdot \mathbf{l} + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & \mathbf{n} \cdot \mathbf{l} + d - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & \mathbf{n} \cdot \mathbf{l} + d - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & \mathbf{n} \cdot \mathbf{l} \end{pmatrix}$$

# Projected Shadows - Artifacts

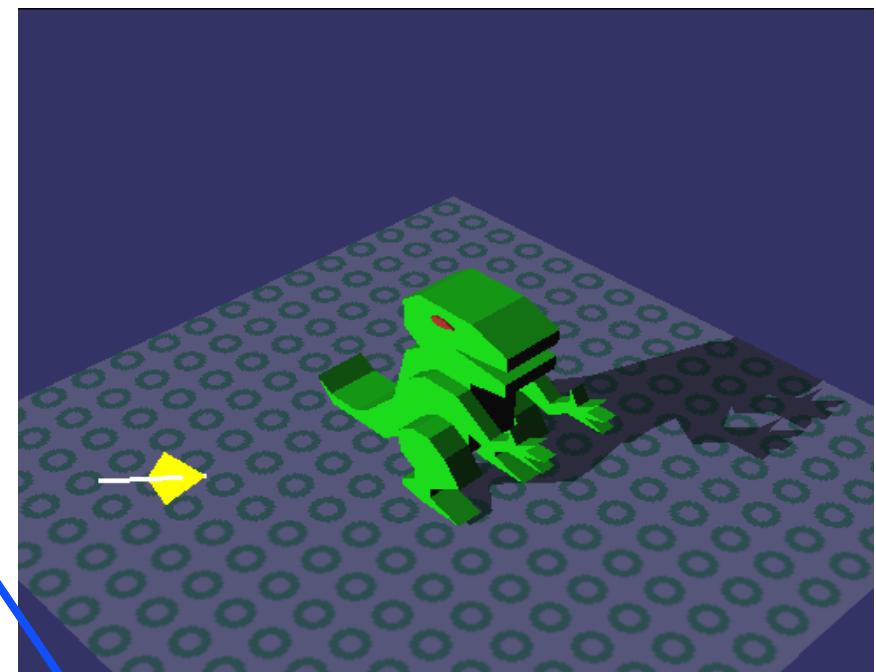
Bad



Z-fighting

double blending

Good



extends off ground

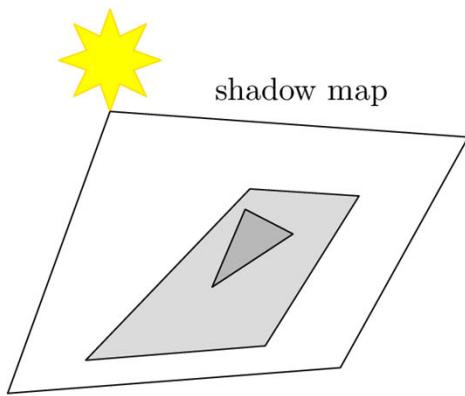
# Projected Shadows with Stencil

- Clear stencil buffer to 0
- Draw receiver plane
  - Write 1 to stencil buffer
- Draw 3D object
- Draw shadow
  - Where stencil is 1
  - Write 0 to stencil buffer
  - No z-fighting problem

# Projected Shadows - Summary

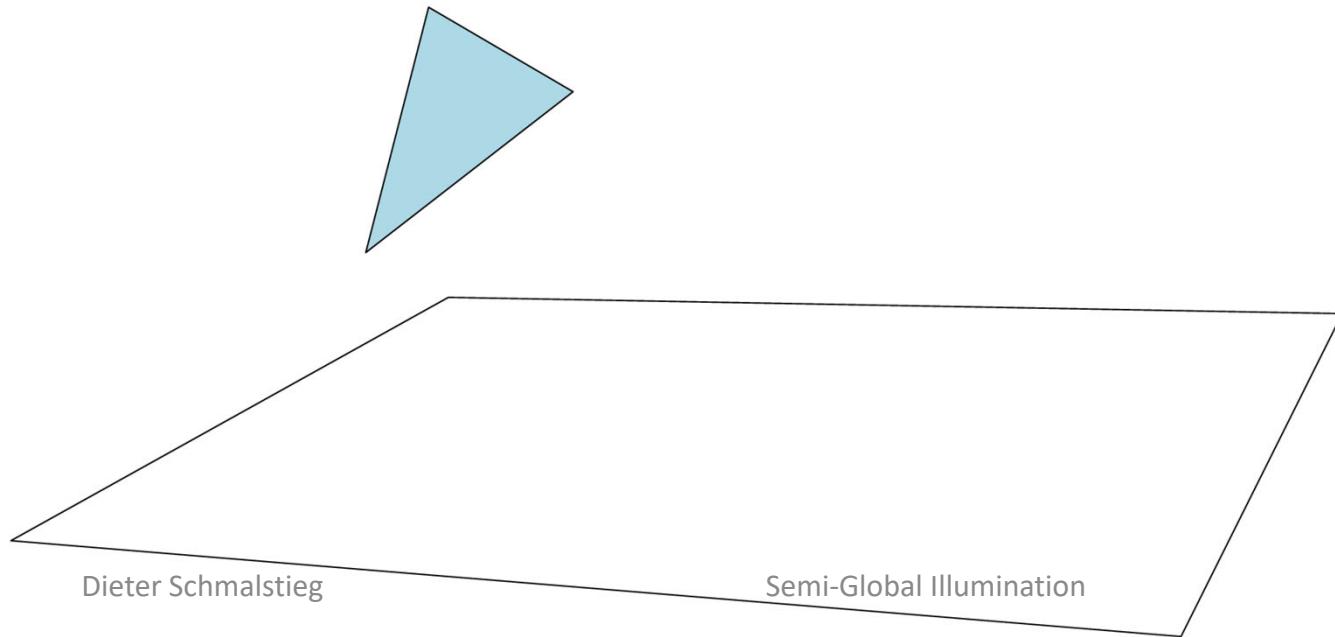
- Easy to implement
  - GLQuake first game to implement it
- Only practical for very few, large receivers
- No self-shadowing
- Possible remaining artifacts: wrong shadows
  - Objects behind light source
  - Objects behind receiver

# Shadow Mapping Algorithm 1

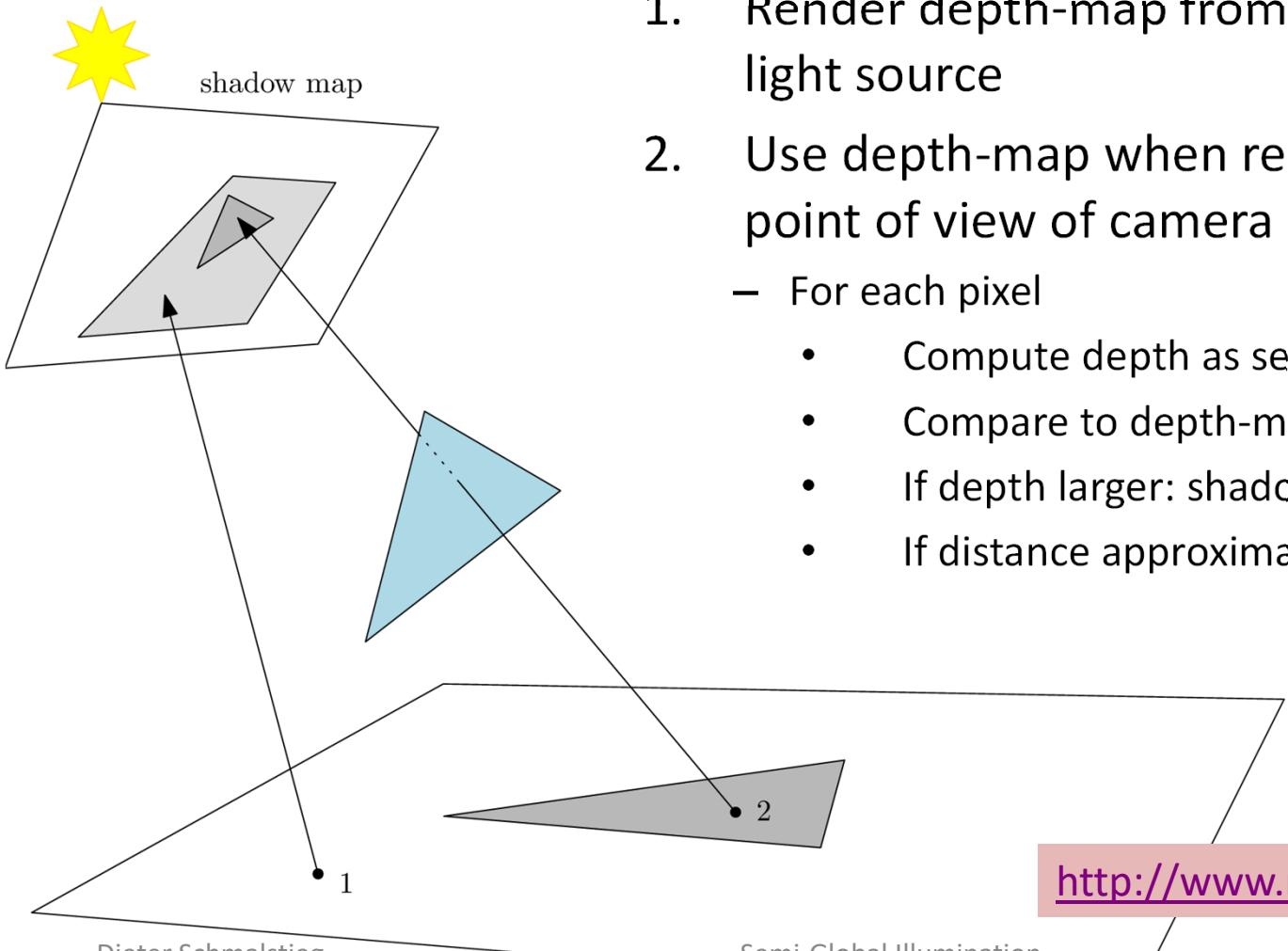


Two-step algorithm:

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



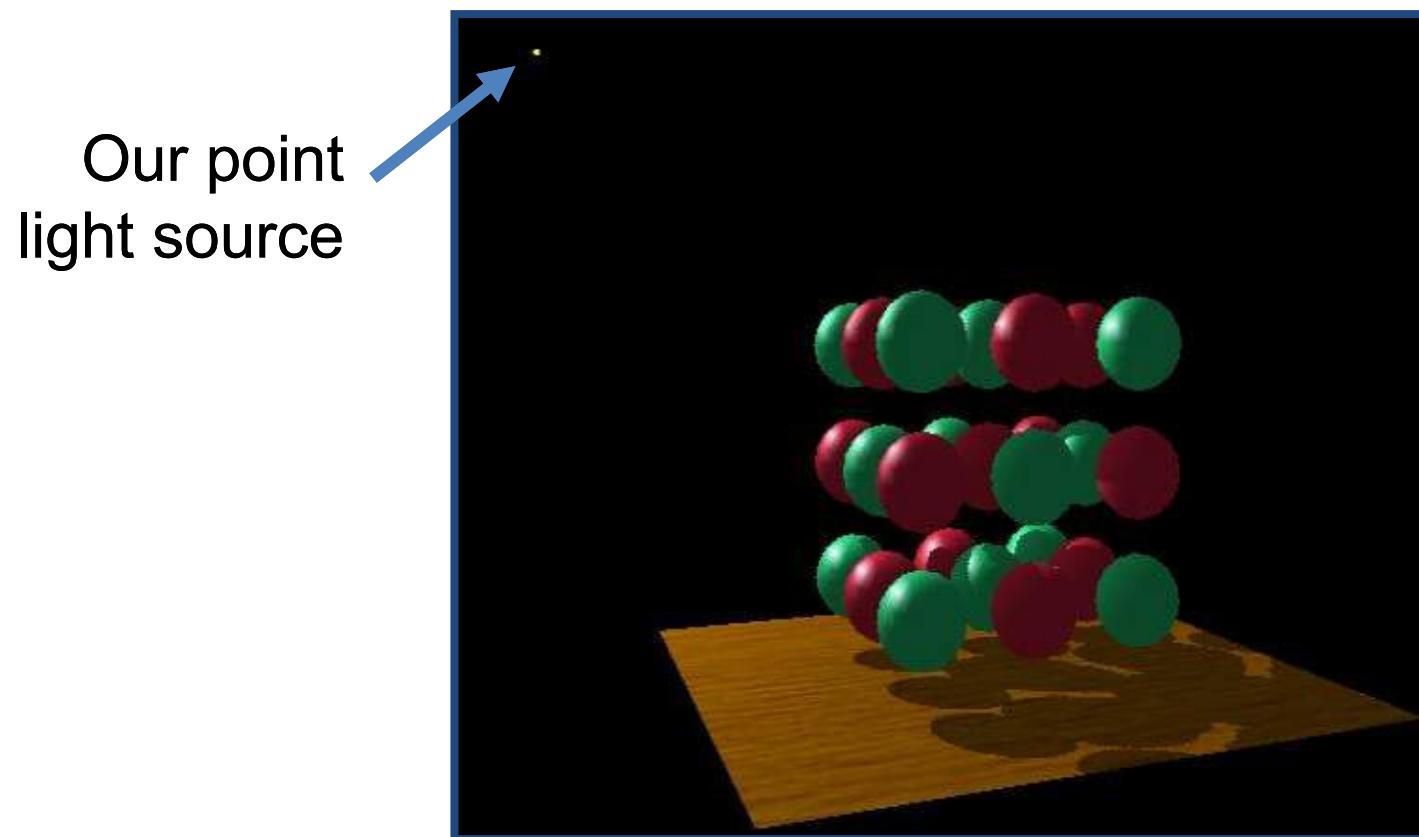
# Shadow Mapping Algorithm 2



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

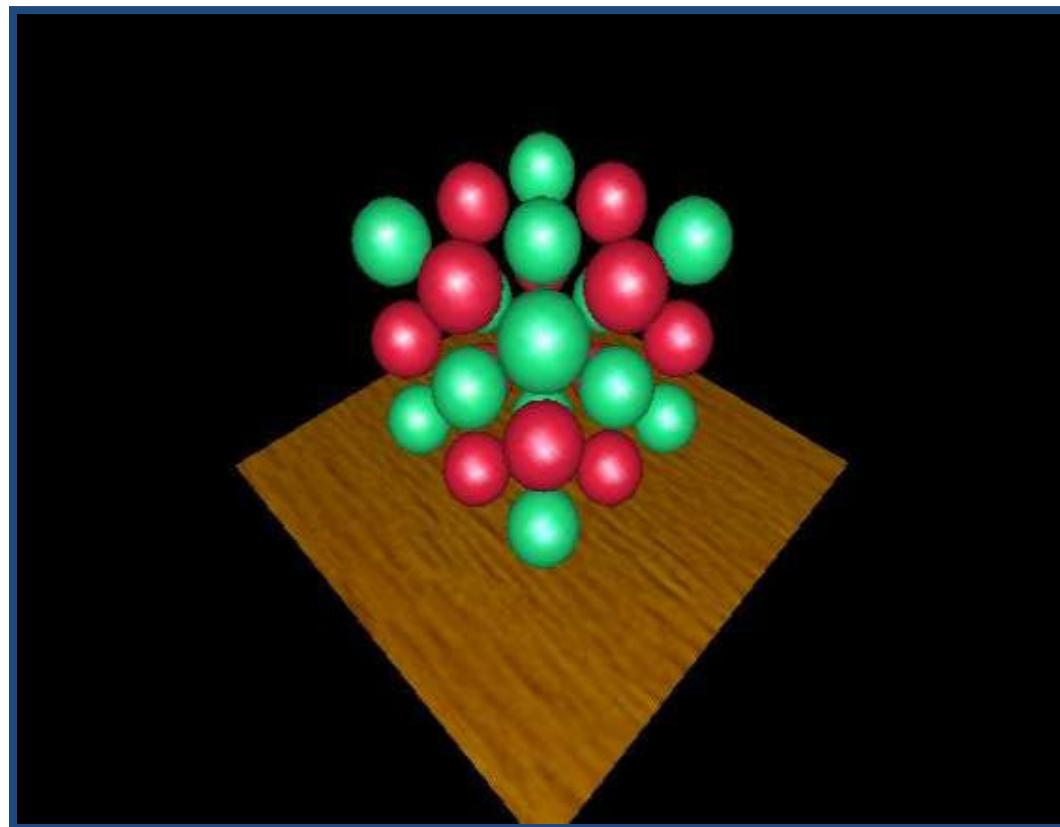
<http://www.nutty.ca/webgl/shadows/>

# Shadow Mapping - Point Light



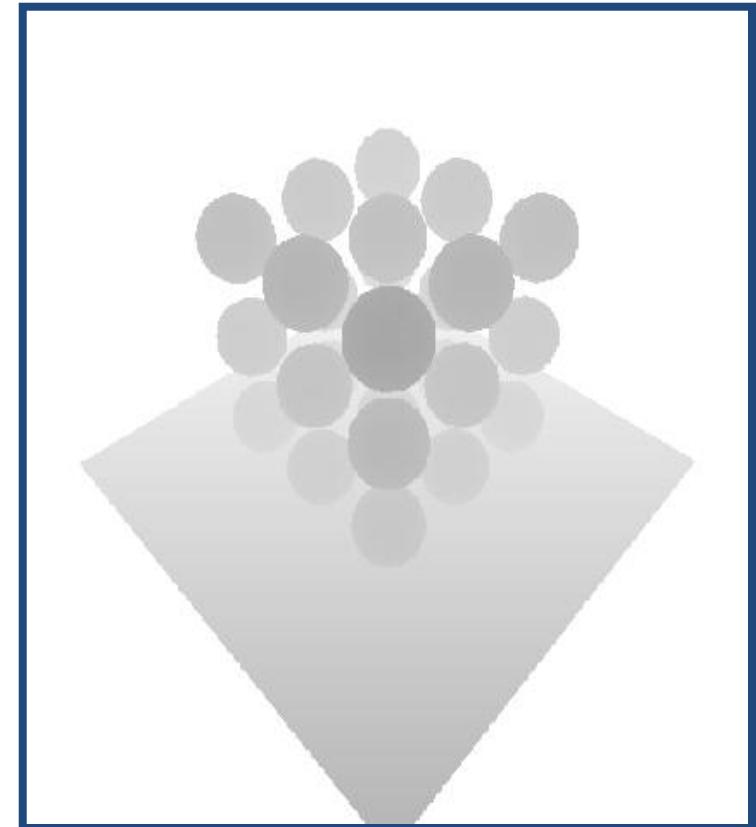
# Shadow Map as Depth Map

- Rendering the Depth-Map



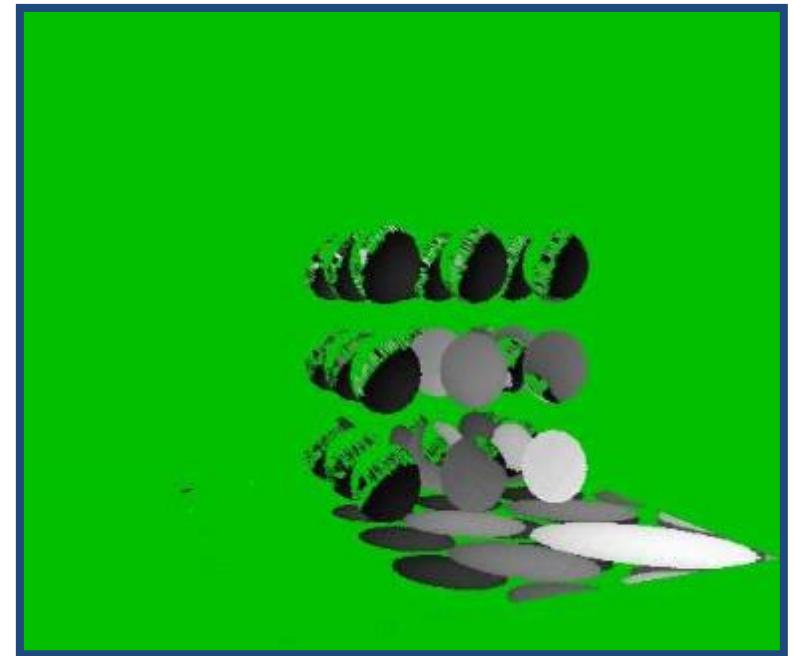
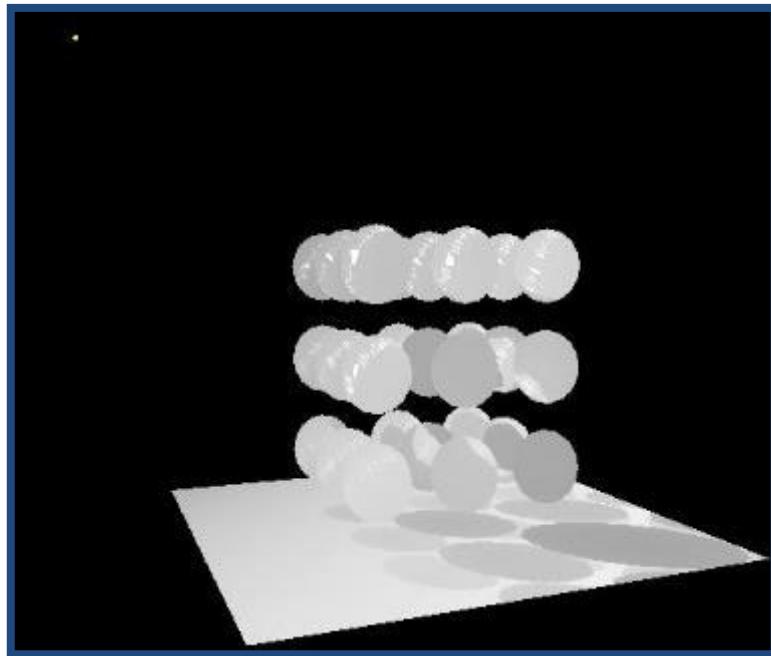
Dieter Schmalstieg

Semi-Global Illumination



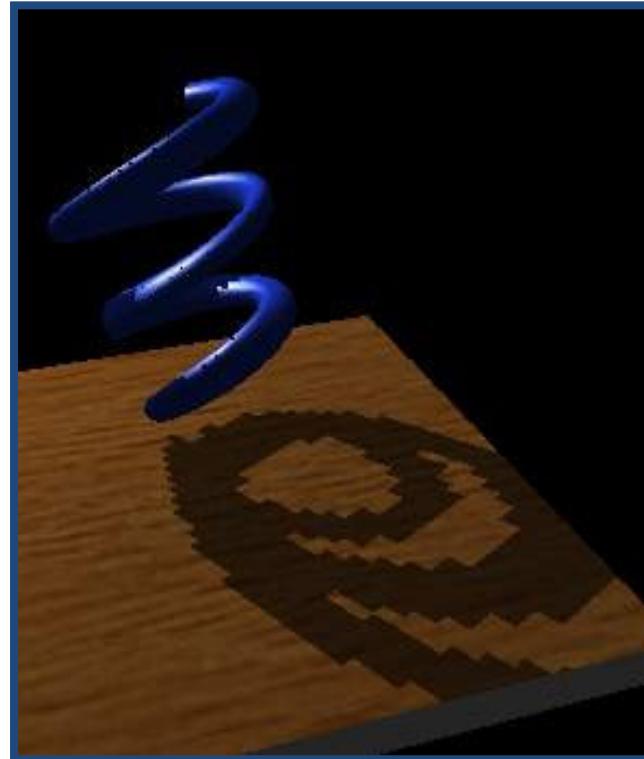
# Shadow Mapping - Depth Test

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

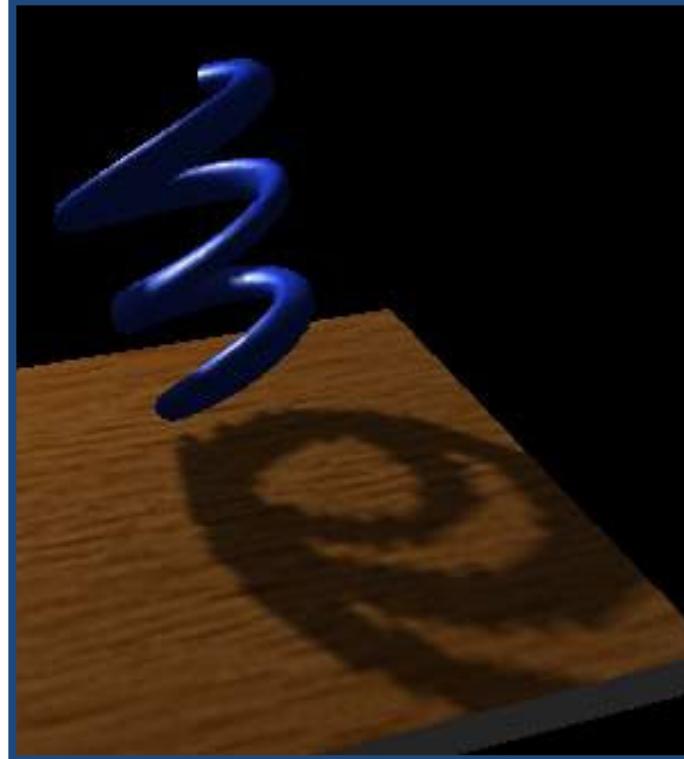


# Aliasing

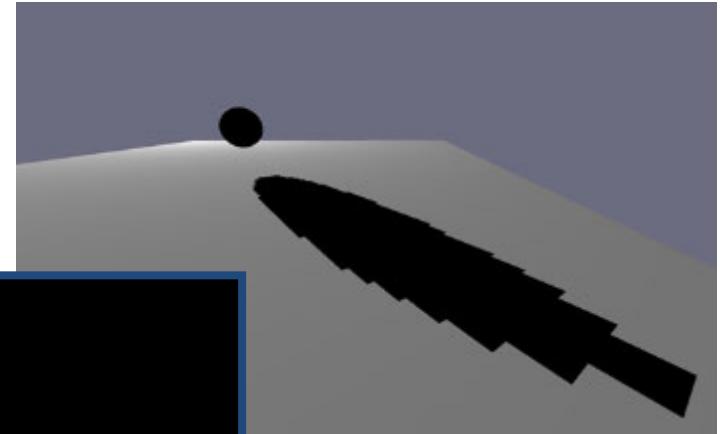
- Two components
  - Perspective aliasing
  - Projection aliasing



Dieter Schmalstieg

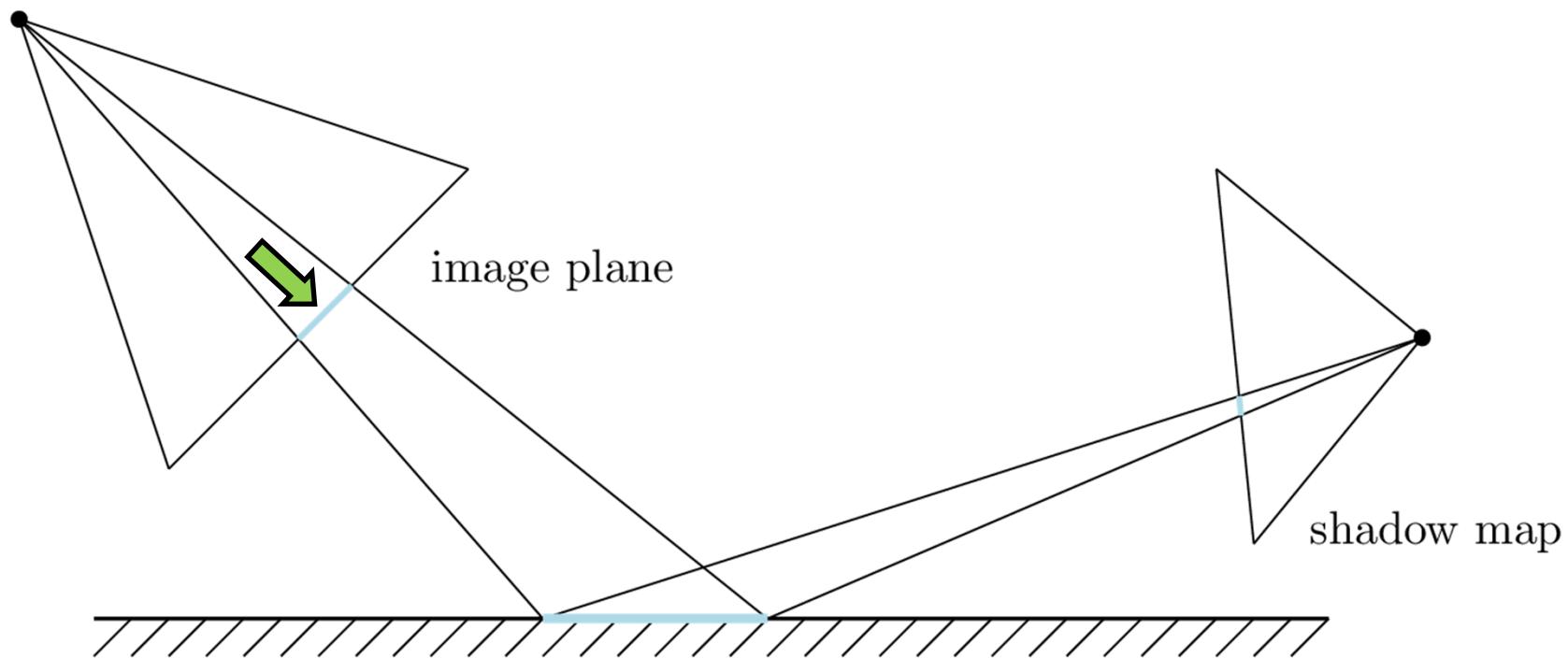


Semi-Global Illumination



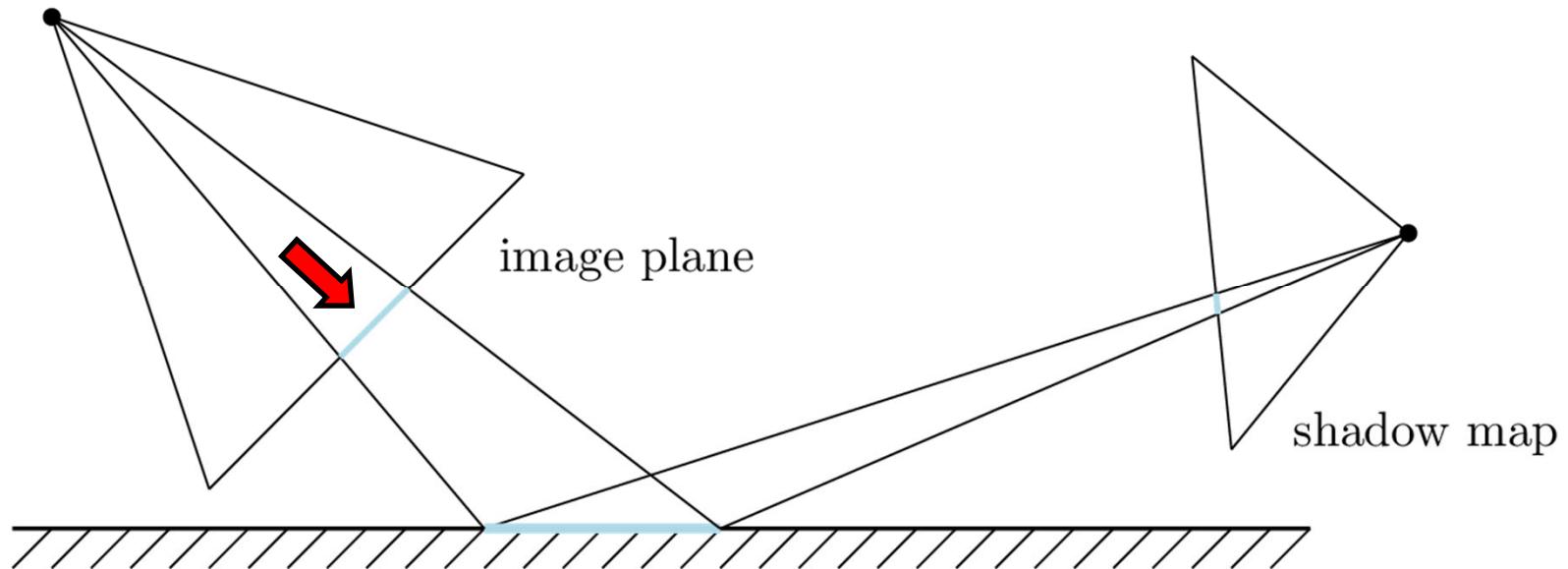
# Perspective Aliasing 1

- Texel resolution in image plane is okay



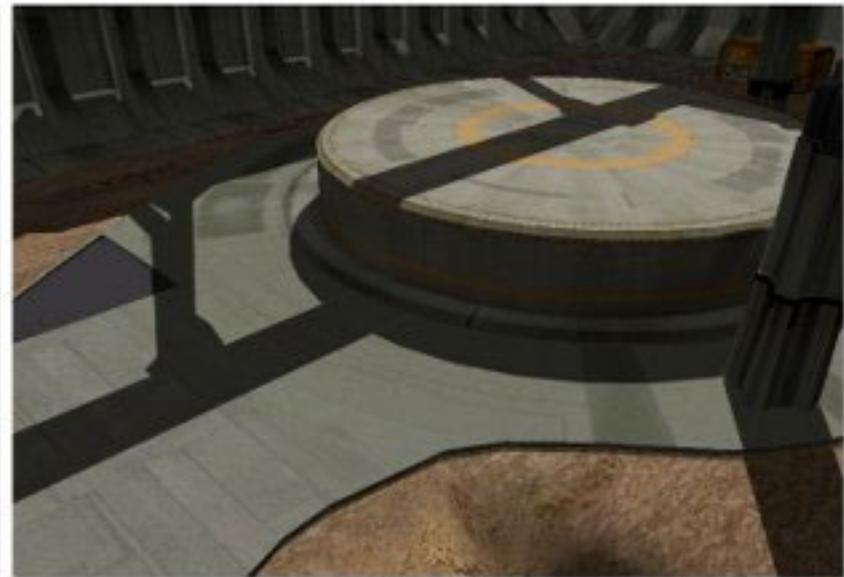
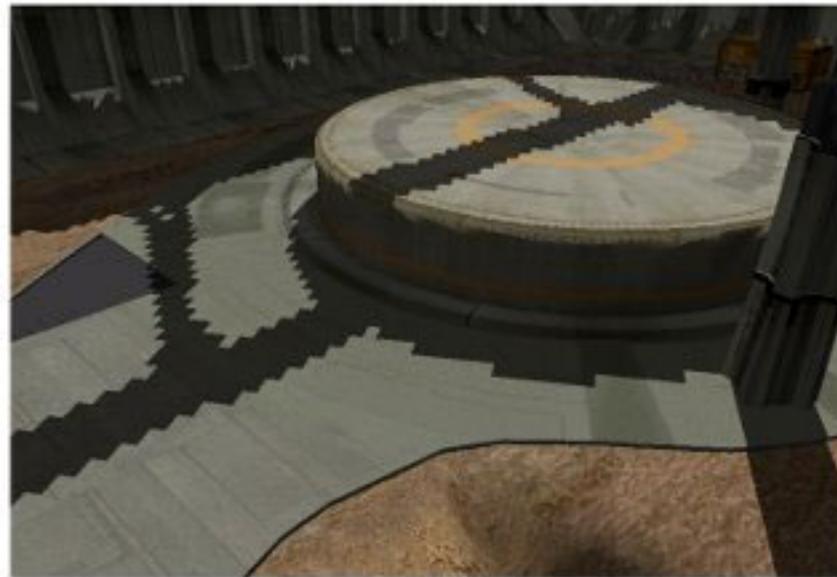
# Perspective Aliasing 2

- If camera moves closer,  
texel resolution in image space gets worse



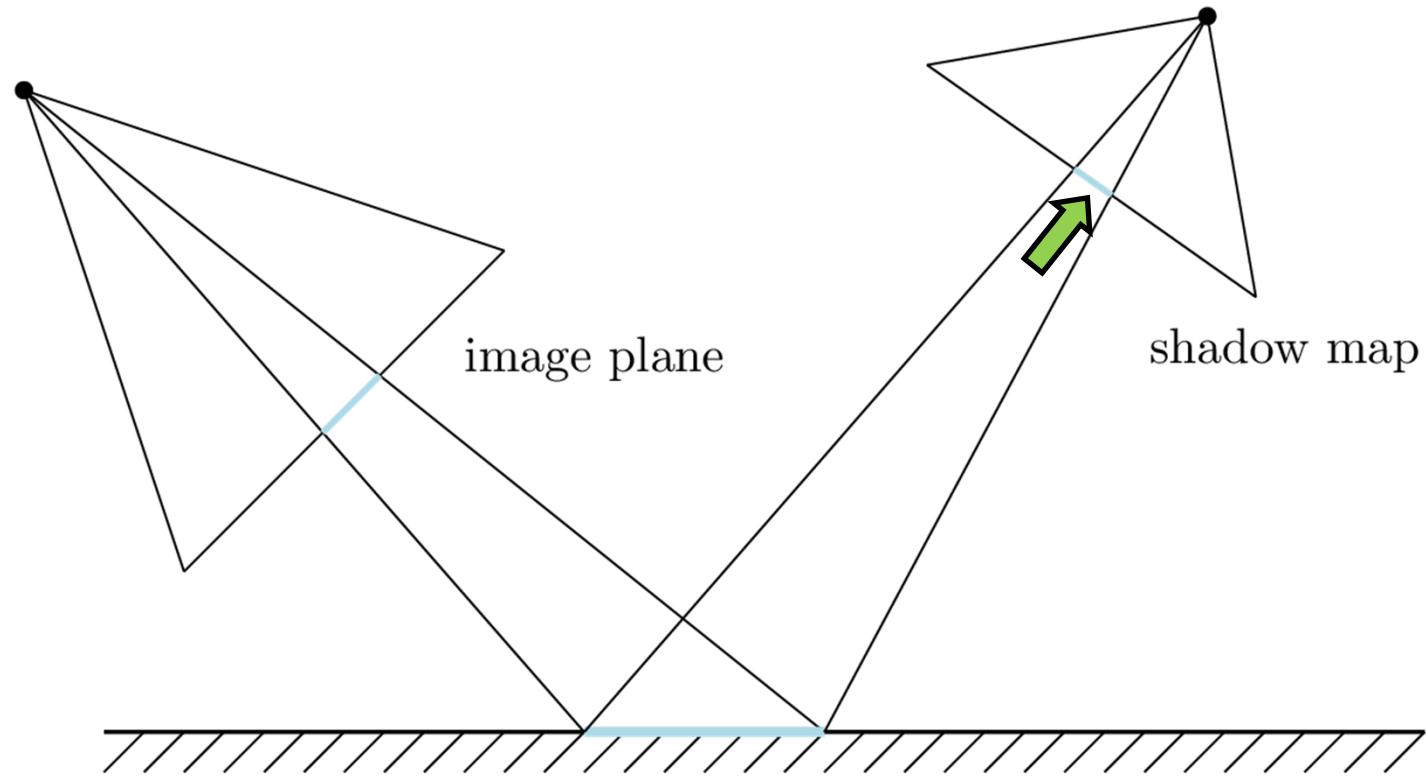
# Example: Perspective Aliasing

Shadow texels large and blocky



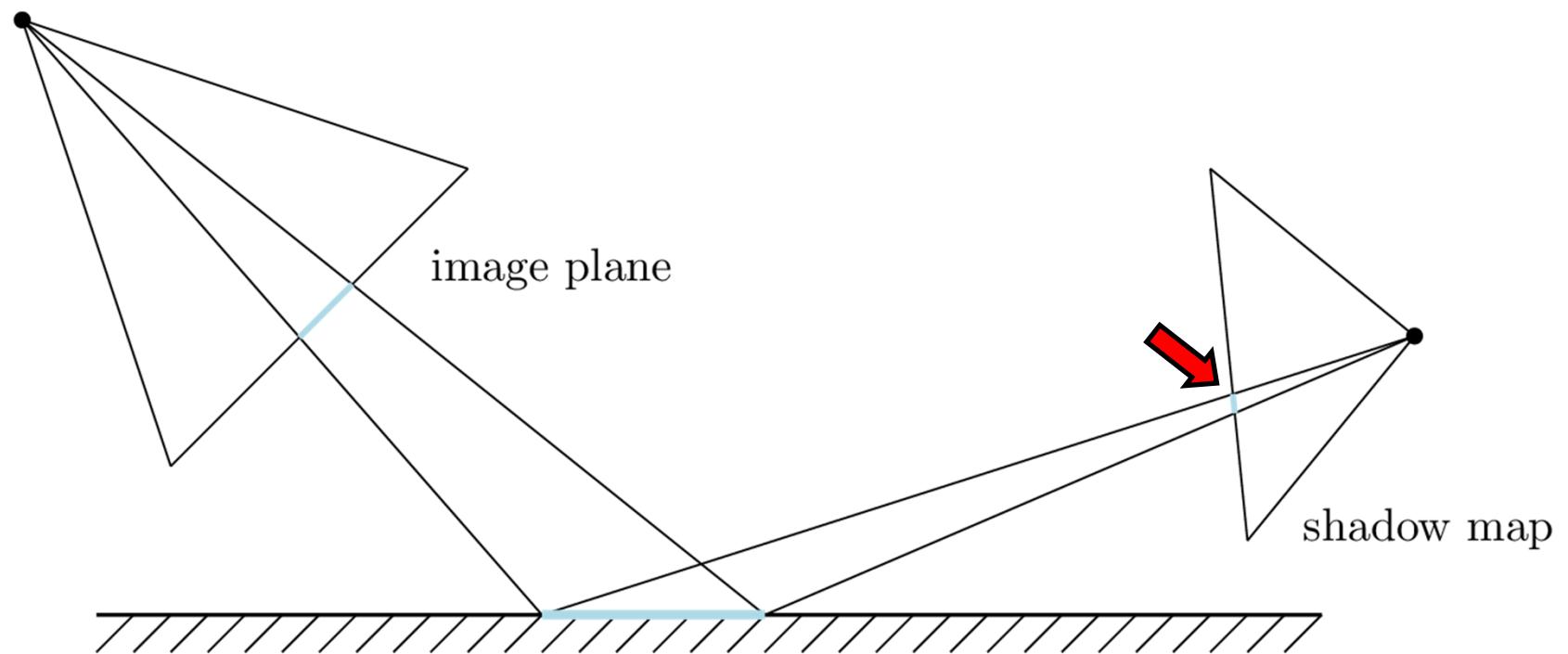
# Projection Aliasing 1

- Shadow map resolution okay



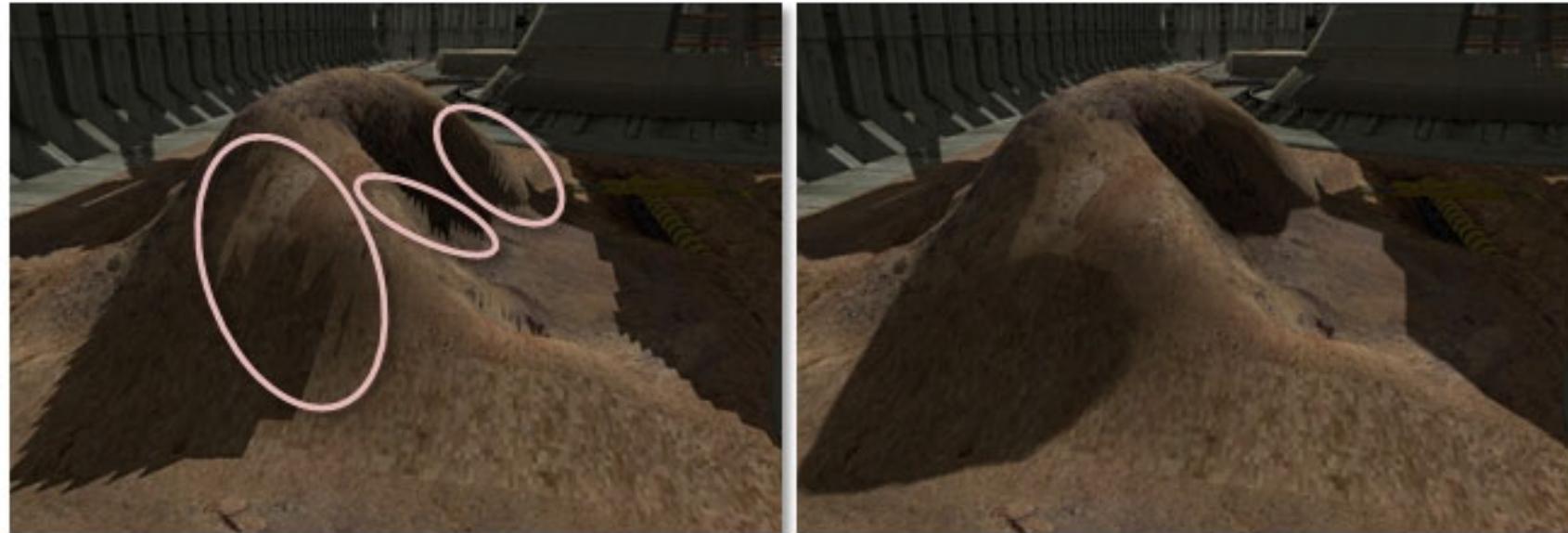
# Projection Aliasing 2

- If angle to light source gets larger, shadow map resolution gets worse



# Example: Projection Aliasing

Shadow texels stretched

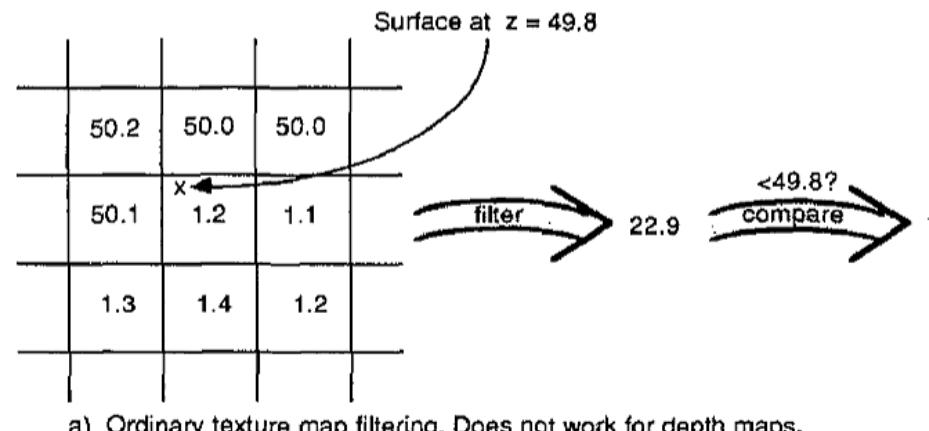


# How to Reduce Aliasing?

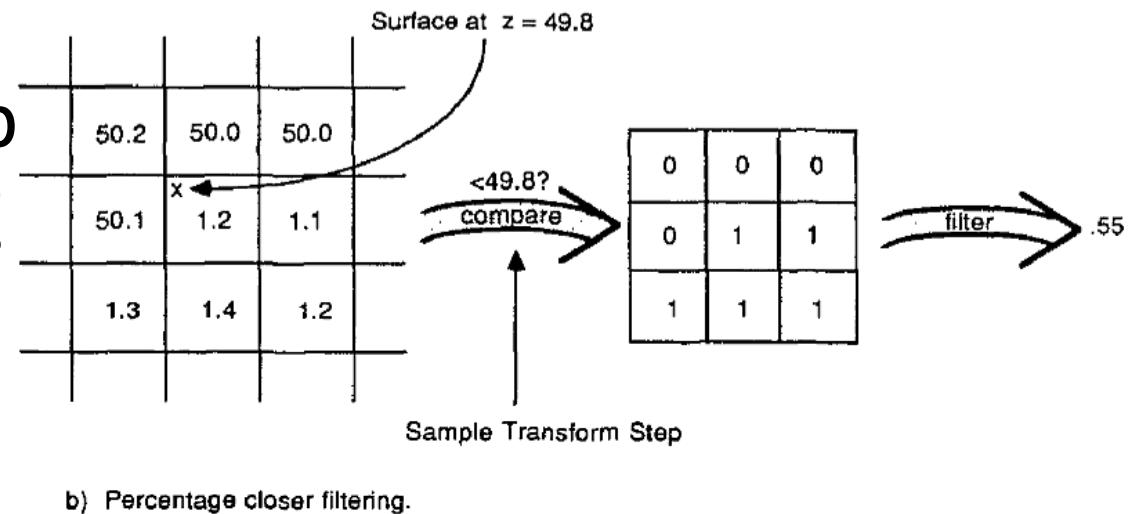
- Increase shadow map resolution
  - Not practical in general
- Practical approaches
  - Anti-aliasing by filtering
  - Optimize sample distribution in shadow map

# Percentage Closer Filtering

- Normal bilinear filtering cannot be used on depth

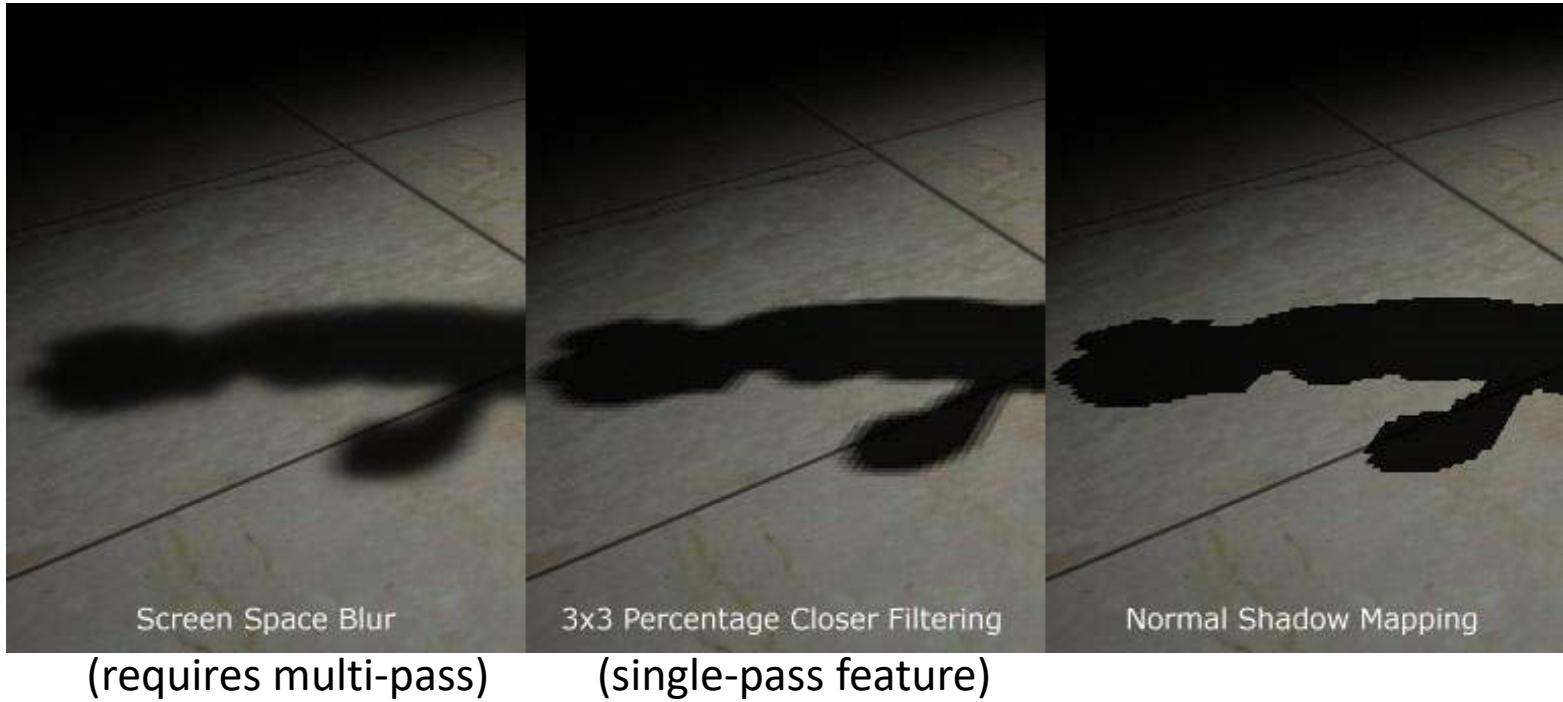


- Must filter lookup result, not depth!



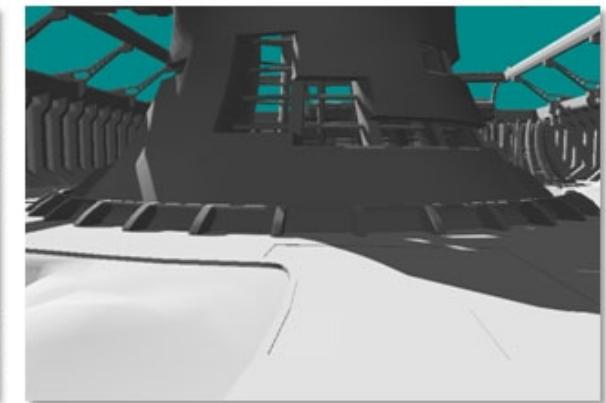
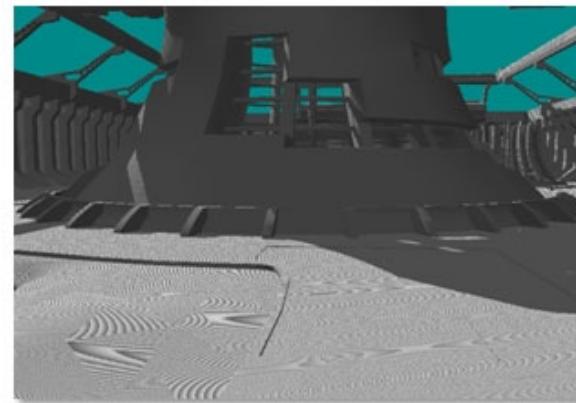
# PCF Example

- <https://www.youtube.com/watch?v=qA920lufMxU>



# Depth Buffer Precision Problems

- Shadow Acne
  - Depth offset too small

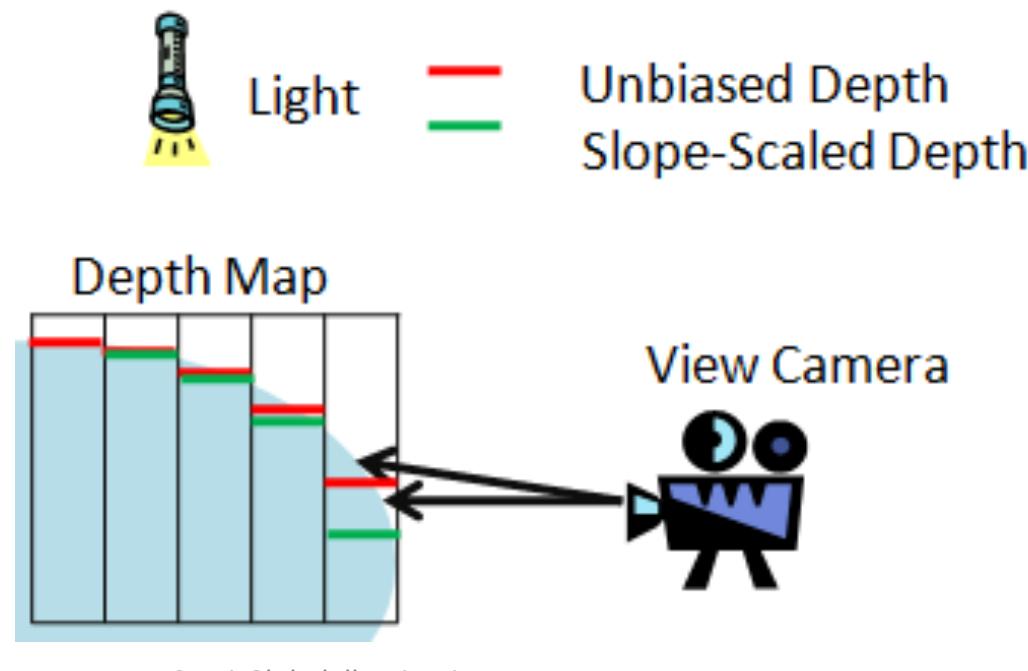


- Peter Panning
  - Shadow offset too large



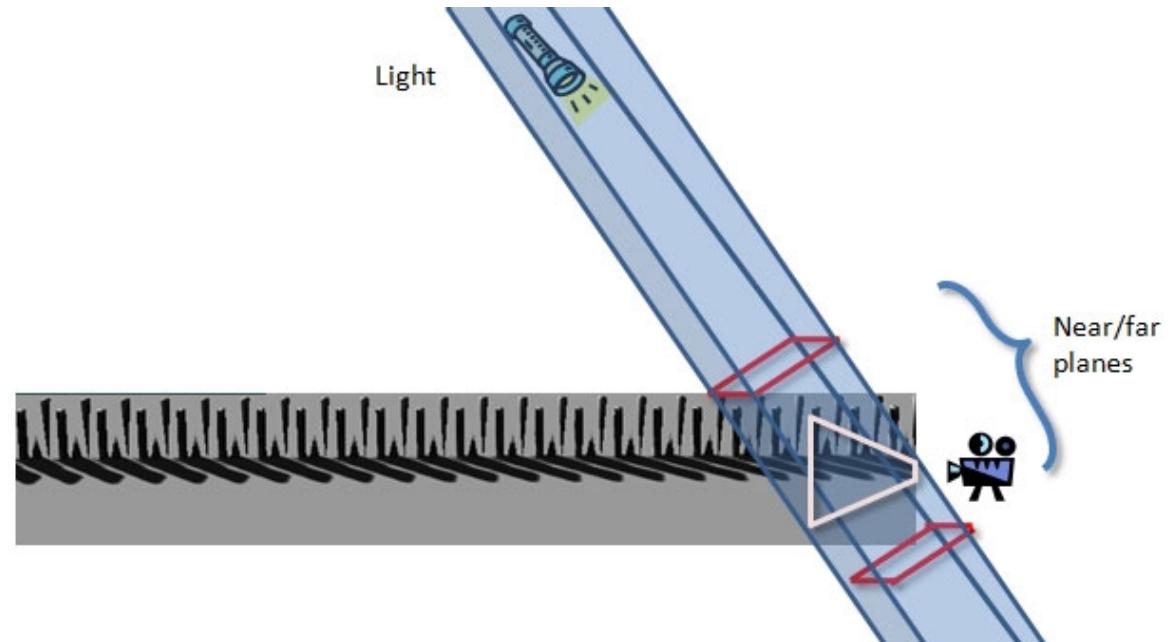
# Slope Scale Bias

- Depth offset is modified by adding a bias
- Bias a polygon based on its slope with respect to the view direction

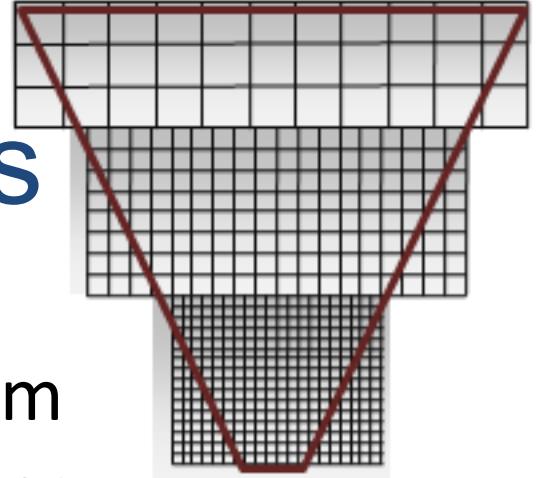


# Optimize Shadow Frustum

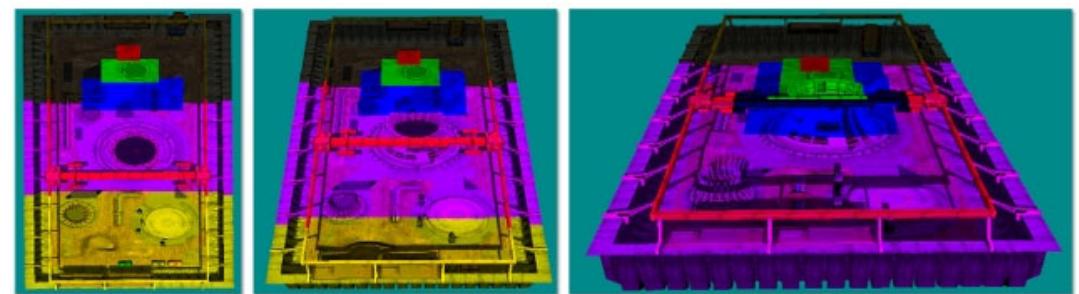
- Optimal use of depth precision: far-near → min
- Choose near/far planes based on intersection of light frustum and scene bbox



# Cascaded Shadow Maps

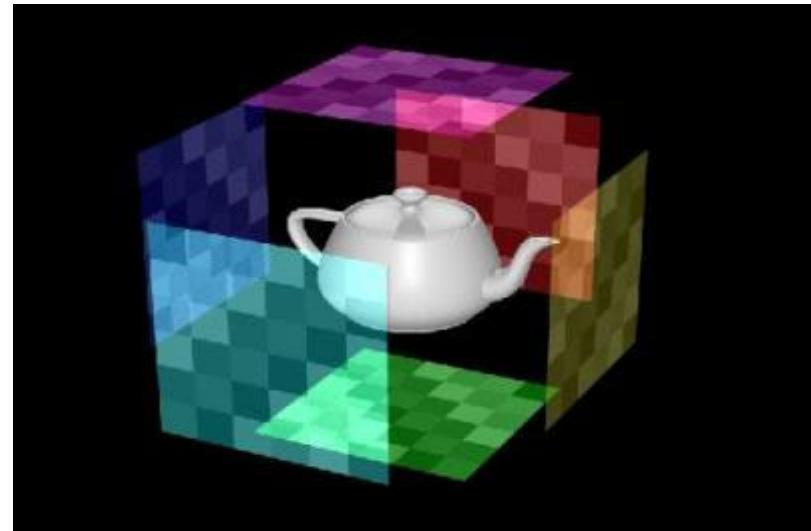


- Partition view frustum
- Generate one cascade per sub-frustum
- Simultaneous multi-viewport rendering
  - One viewport created per cascade
  - Each viewport renders to separate portion of the depth buffer
- Cascades must overlap (for blending)
- Cascade overlap increases as light direction becomes parallel with camera direction

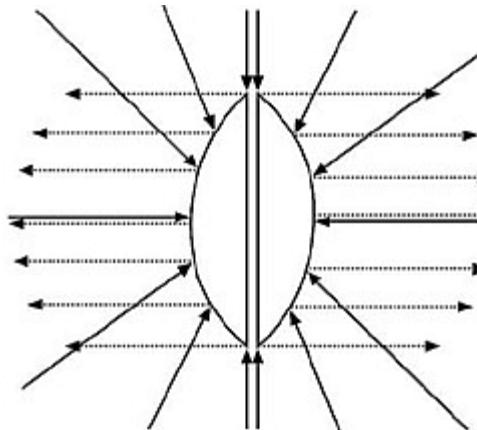
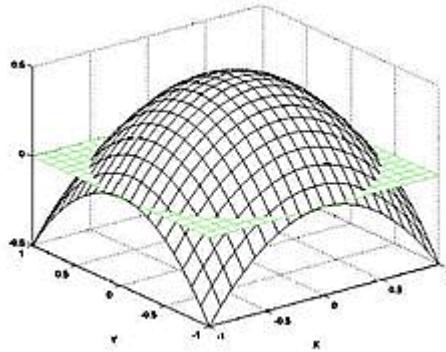


# Omni-directional Shadow Maps

- Shadow maps limited to spot-light
- Cannot have a 360° “viewing frustum”
- Use six shadow maps for omni-directional light source
- Expensive!



# Dual Paraboloid Shadow Map



- Paraboloid maps directions inside one half-sphere to a disc
- Arrange two such discs inside a shadow map (or environment map)
- Supported as native texture coordinate mode
- Can be generated in two passes (instead of six)

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

# Shadow Maps Summary

- Advantages
  - Fast – only one additional pass
  - Independent of scene complexity  
(no additional shadow polygons!)
  - Self shadowing (but beware bias)
  - Can sometimes reuse depth map
- Disadvantages
  - Problematic for omnidirectional lights
  - Biasing tweak (light leaks, surface acne)
  - Jagged edges (aliasing)

# Shadow Volumes

[Frank Crow, 1977]

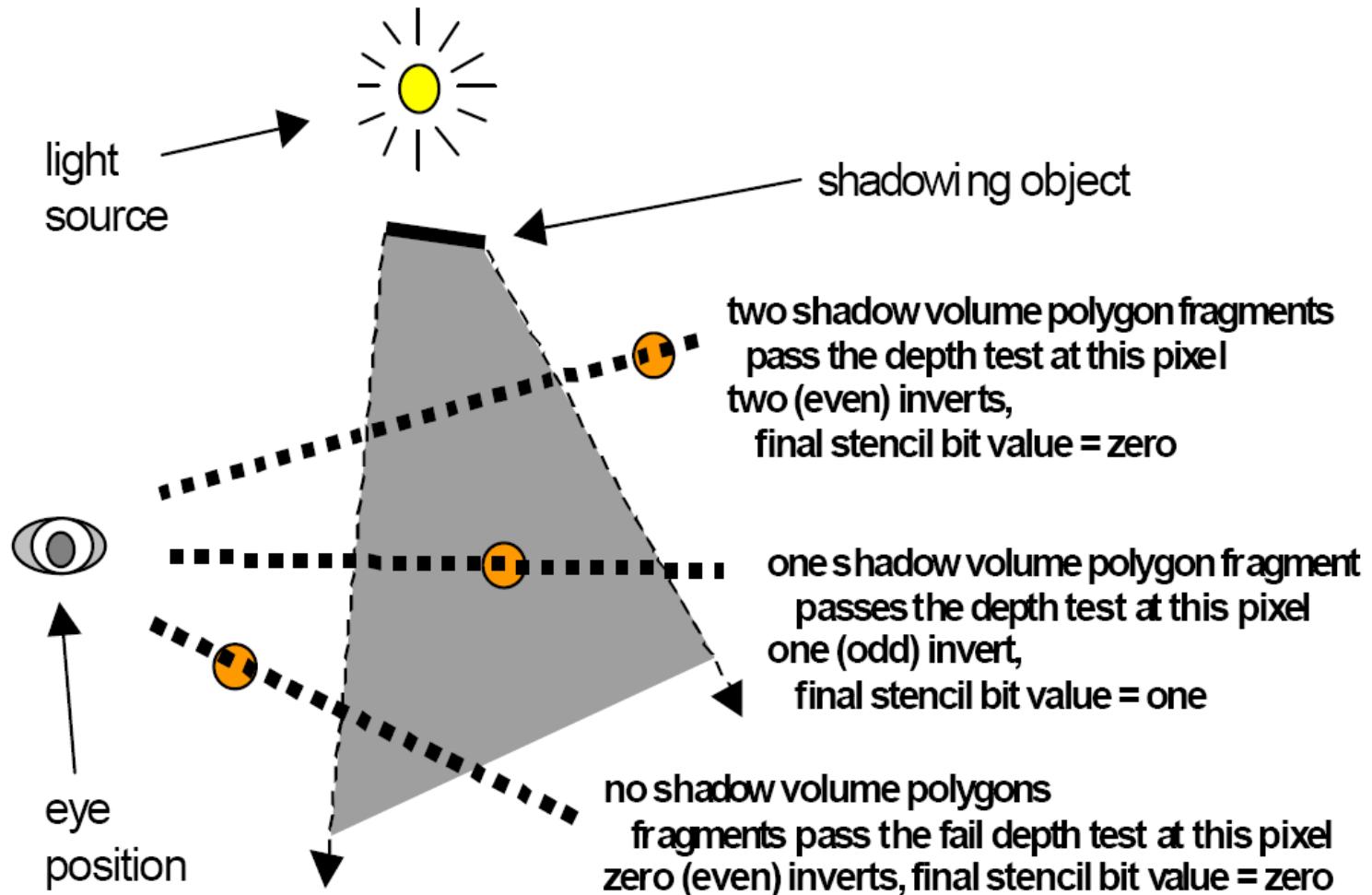
- Complex, but can be implemented efficiently using stencil buffer
- No aliasing
- Method
  - Intersect view rays with shadow volume
  - Count number of intersections, until receiver is hit



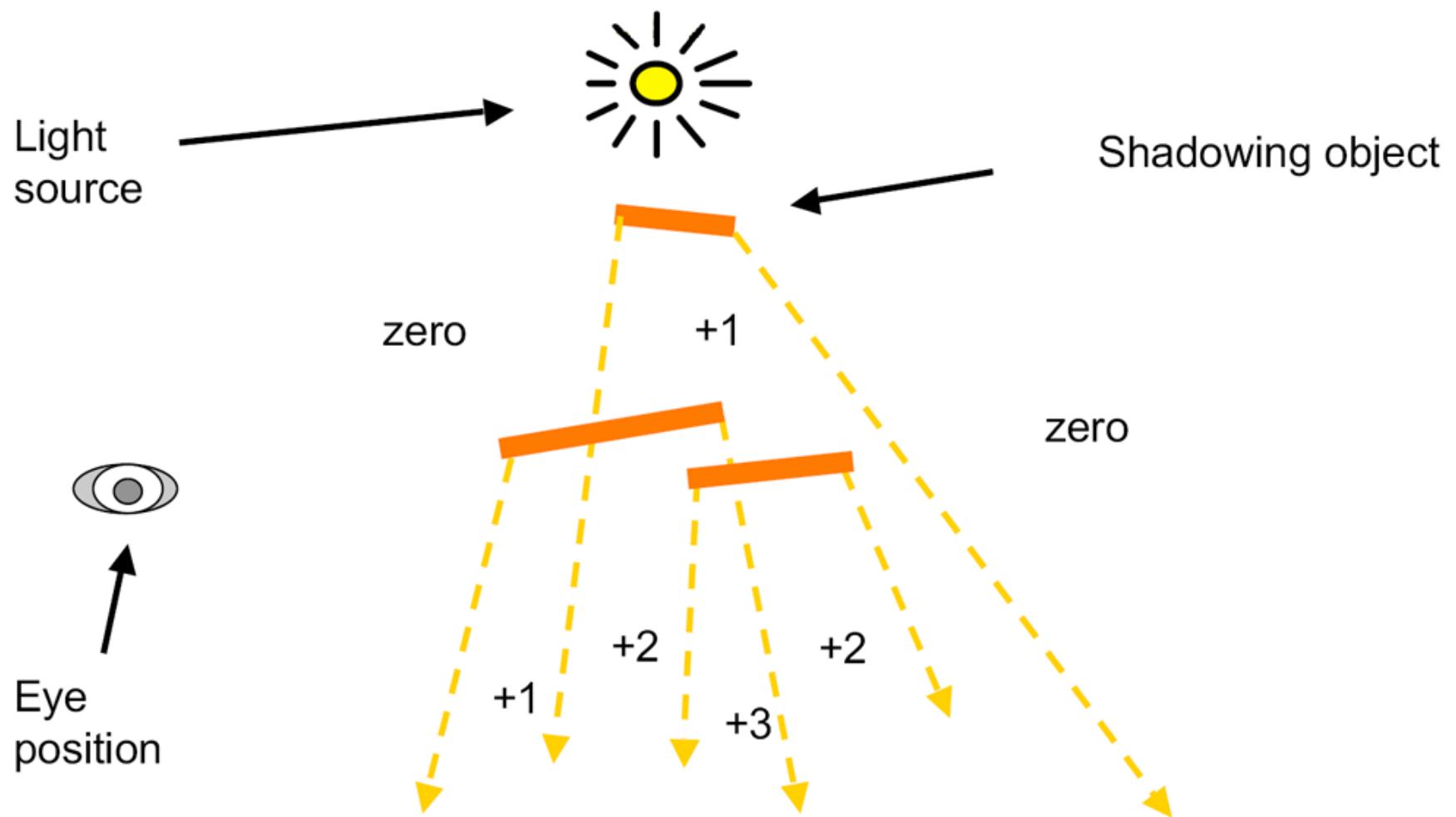
# Example: Doom 3



# Shadow Volumes

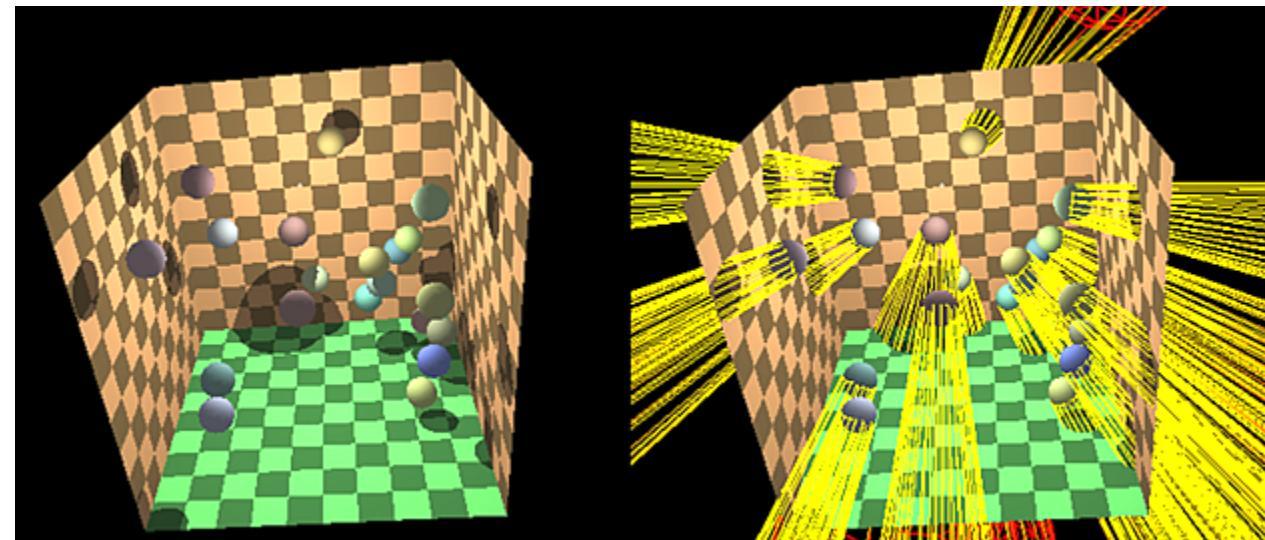


# Counting Scheme



# 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

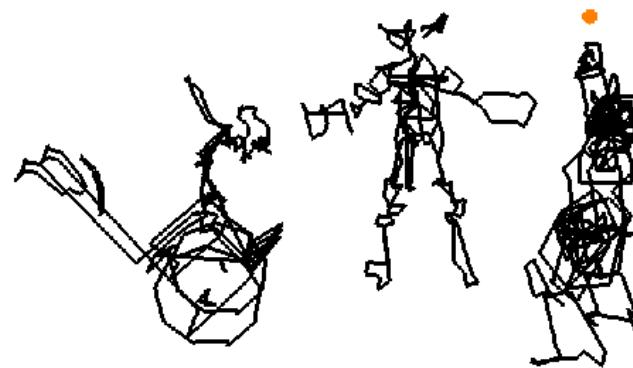


# Shadow Volume Geometry

- 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?

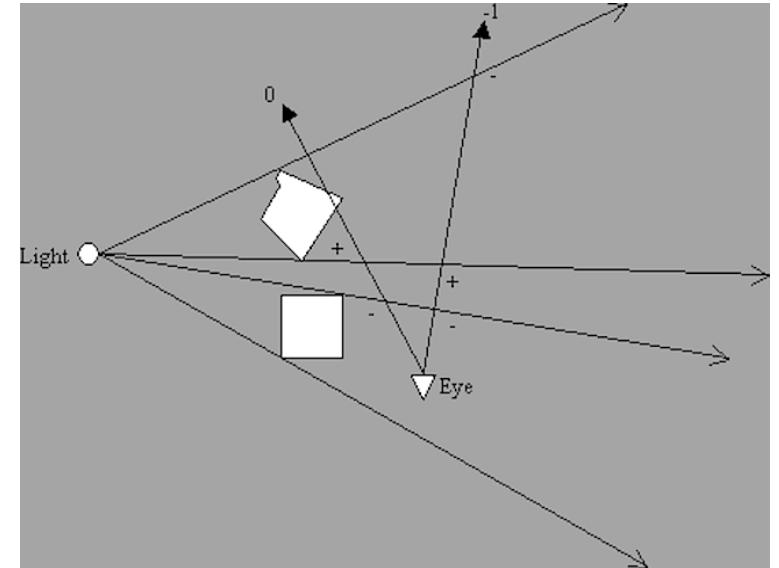
# Silhouette Detection

- 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!



# Shadow Algorithms Compared

## Shadow Volumes

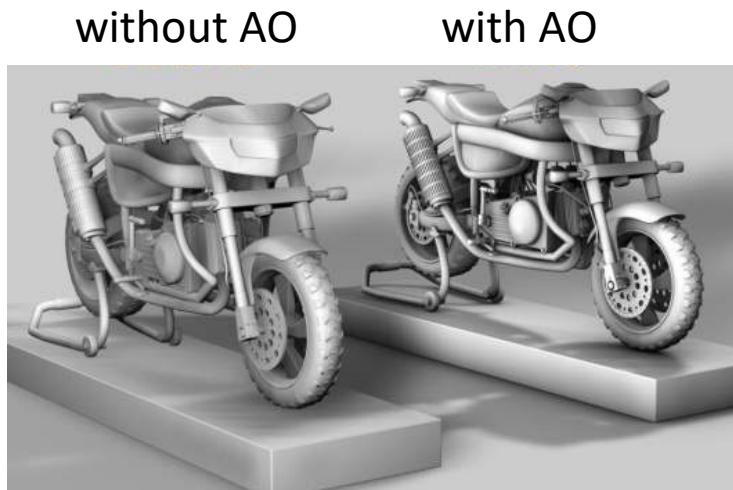
- Needs geometry
  - Closed manifolds ☺
- Cost hard to predict
  - Geometric complexity of shadow volume ☹
- Shadow rendering is bandwidth intensive ☹
  - Stencil buffer techniques
  - Additional pass
- No sampling artifacts ☺

## Shadow Mapping

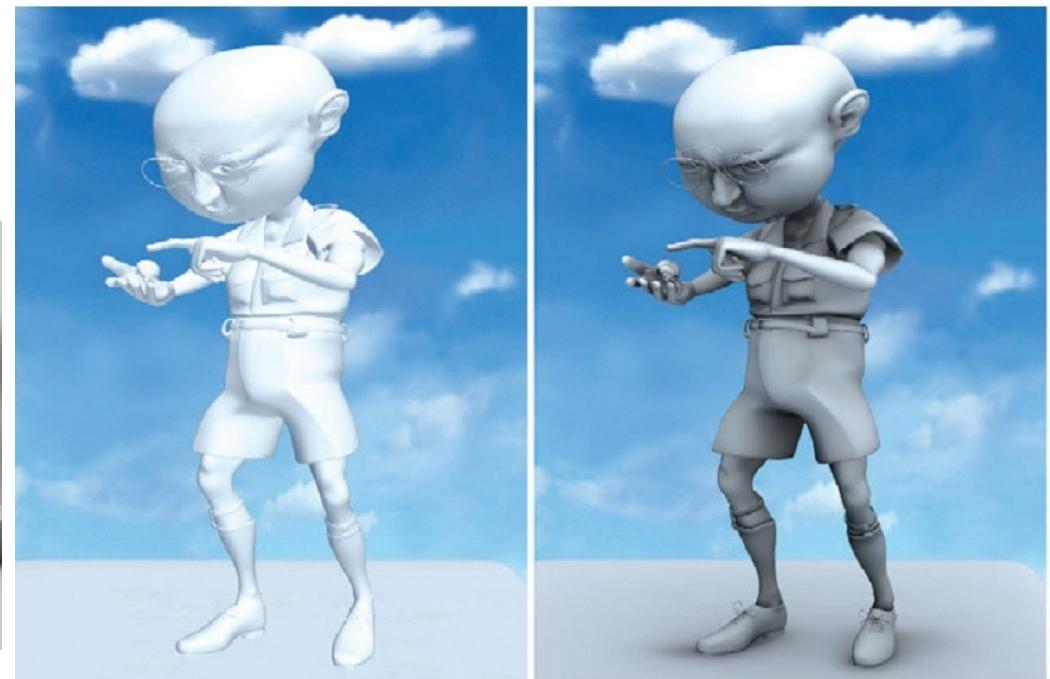
- Only depth needed
  - Everything one can render can cast a shadow ☺
- Predictable cost
  - Rendering the scene once ☺
- Shadow rendering is simple texture lookup ☺
  - Hardware Support
- Sampling artifacts ☹

# Shadowing with Environmental Lighting

- So far: shading with known light position
- Now: shadowing by ambient light
- Independent of light/object orientation
- Darken surfaces which are partially visible in the environment
- Adds depth and contrast



Dieter Schmalstieg

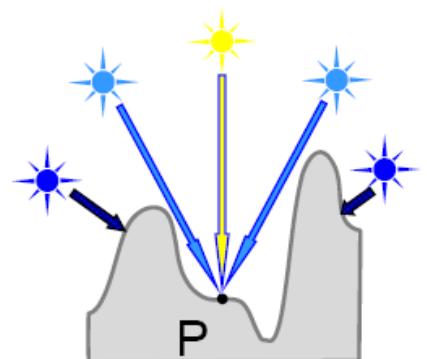


Semi-Global Illumination

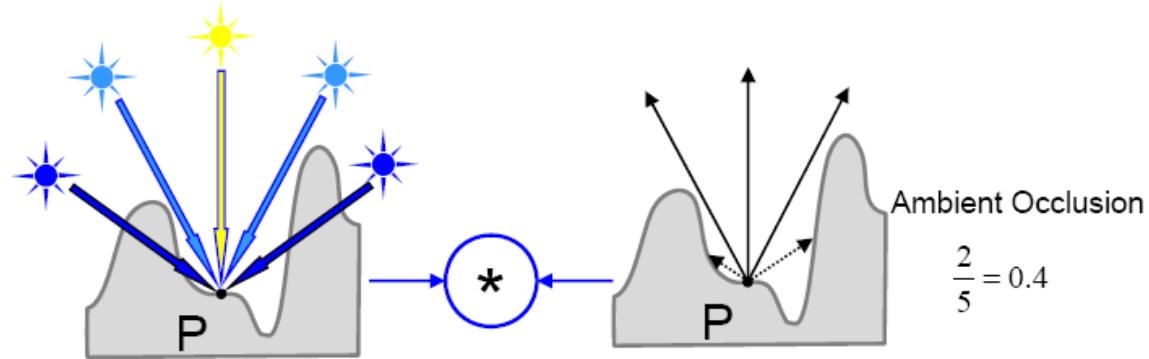
# Ambient Occlusion

- 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

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



Dieter Schmalstieg

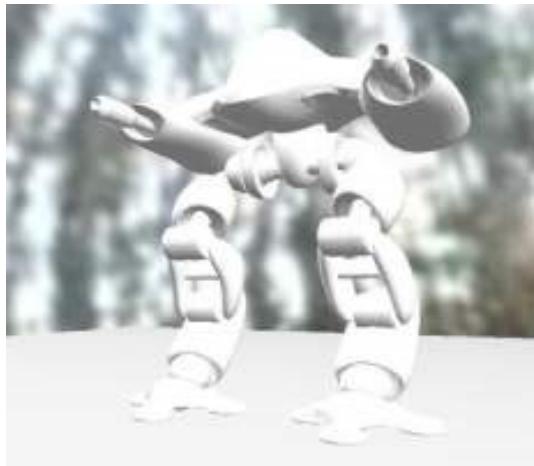


Semi-Global Illumination

# Mean Visibility

- 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

# Ambient Occlusion



(a) Gouraud shading



(b) Ambient Occlusion

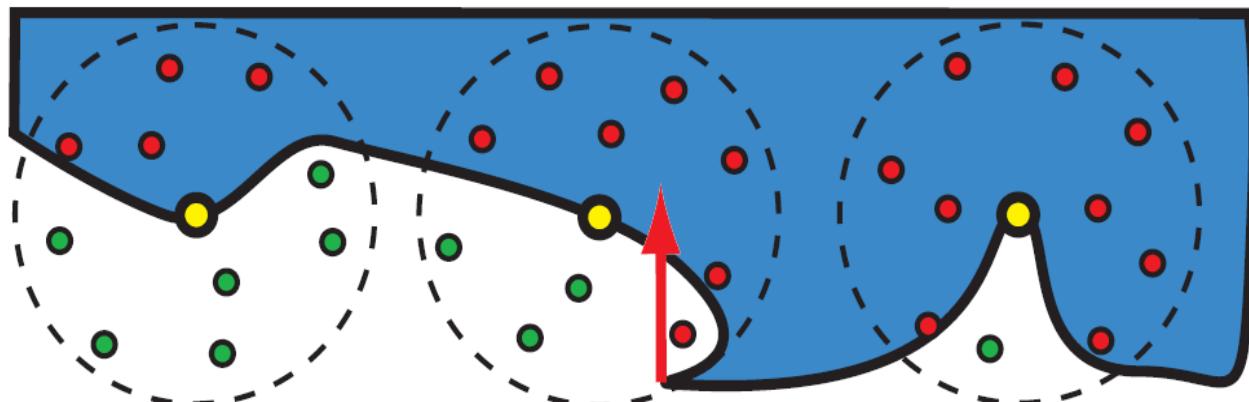


(c) Ground truth

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

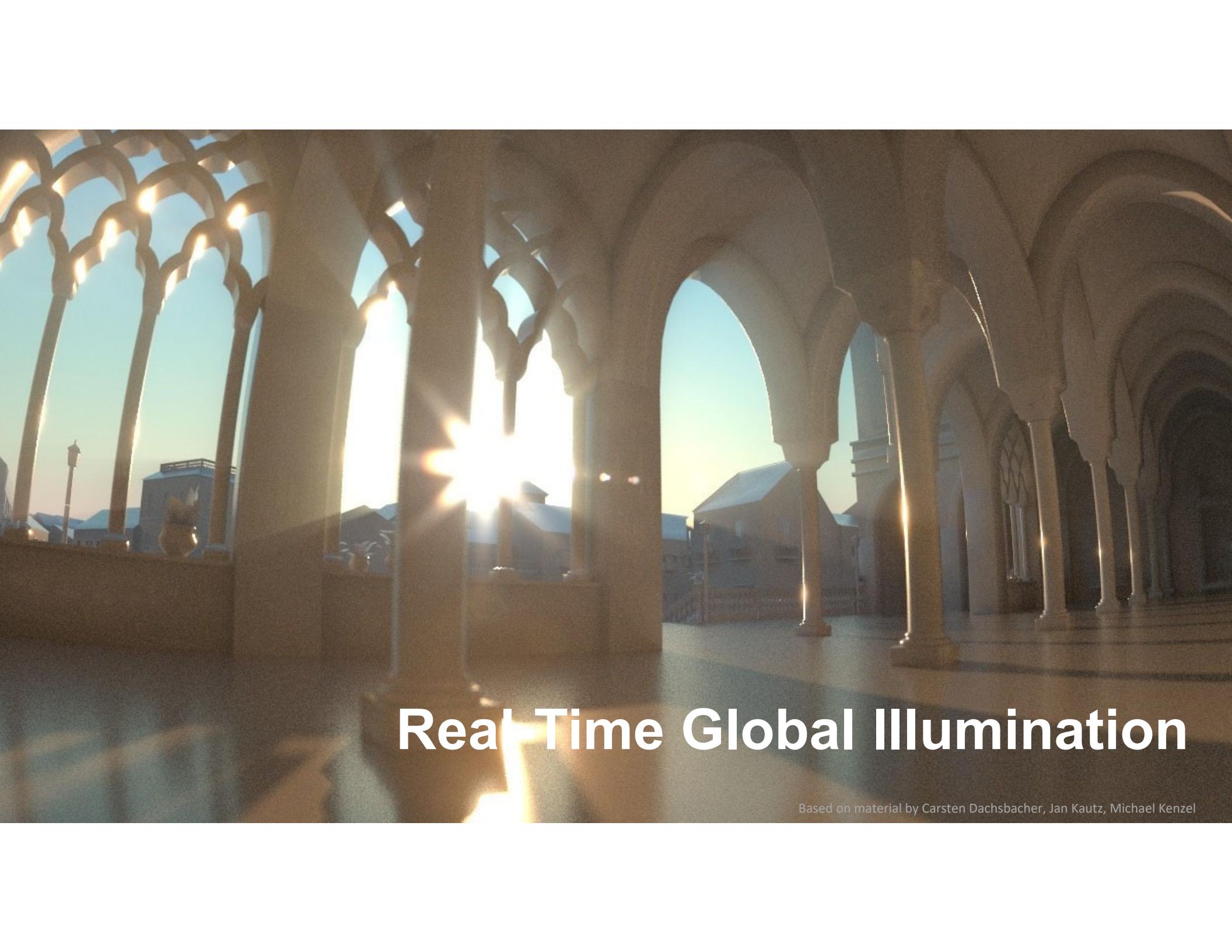
# Screen Space Ambient Occlusion

- 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
  - $\omega_i$  = incoming (=light source) direction
  - $\theta_i$  = angle to normal



# Questions?





# Real-Time Global Illumination

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

# Two classes of Rendering Algorithms

- Offline rendering
  - Complete scene information available at all times
- Online rendering
  - Generates image while streaming scene data
  - E.g., surface fragments only available during rasterization

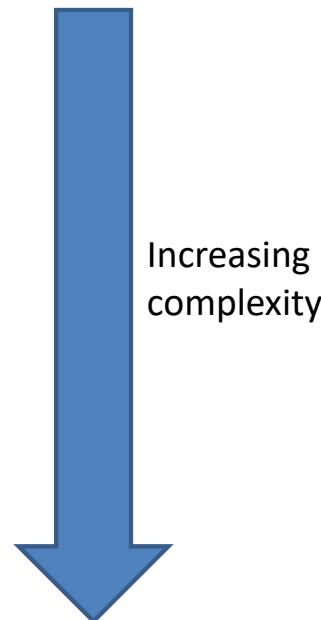


You are here (usually)

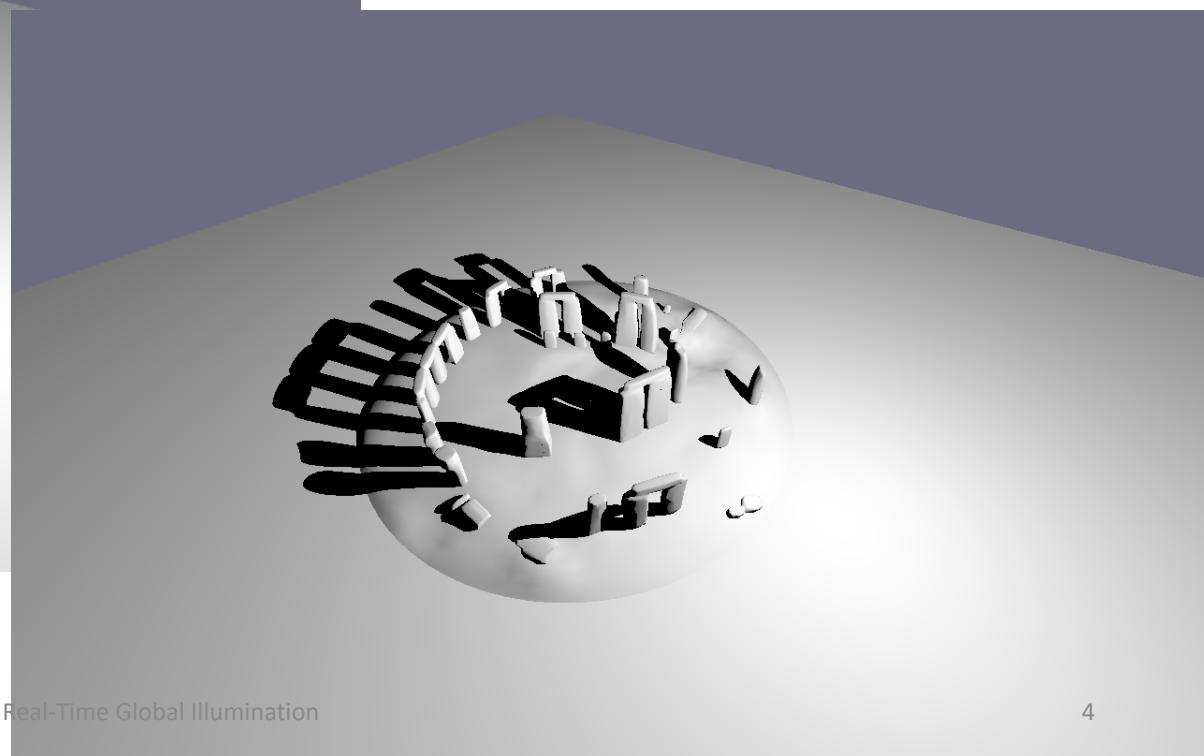
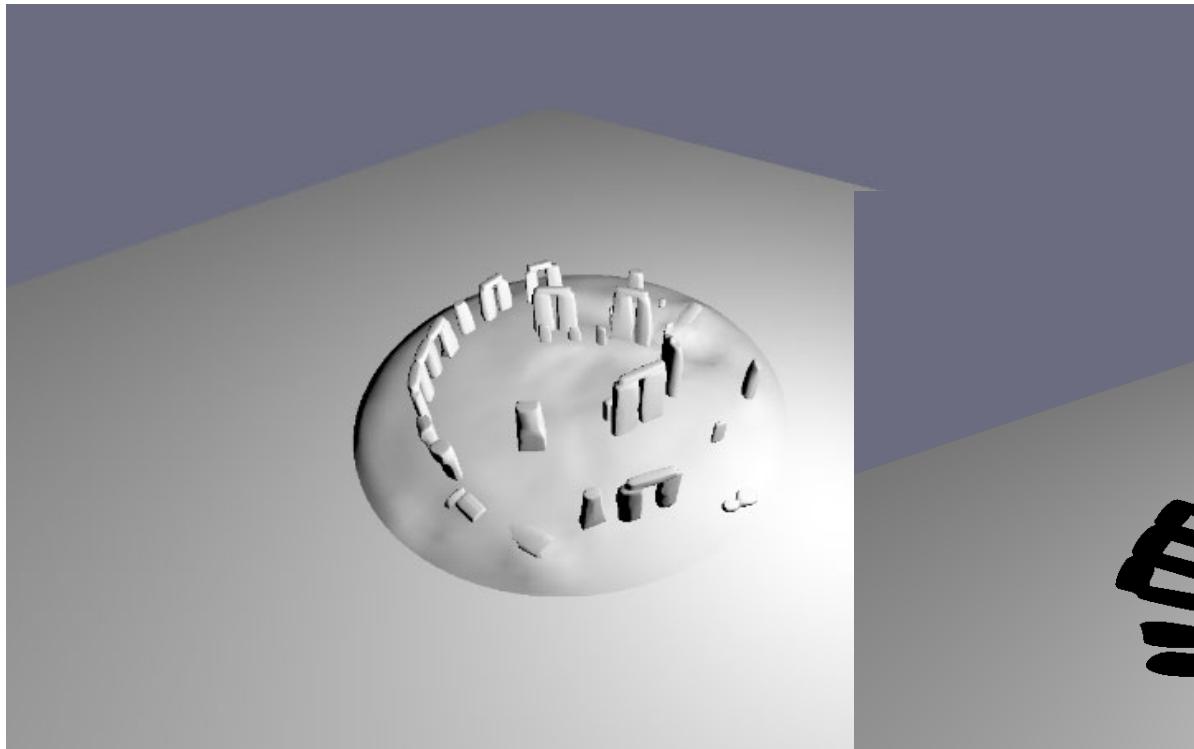
# What is Global Illumination?

Objects influence each other's appearance

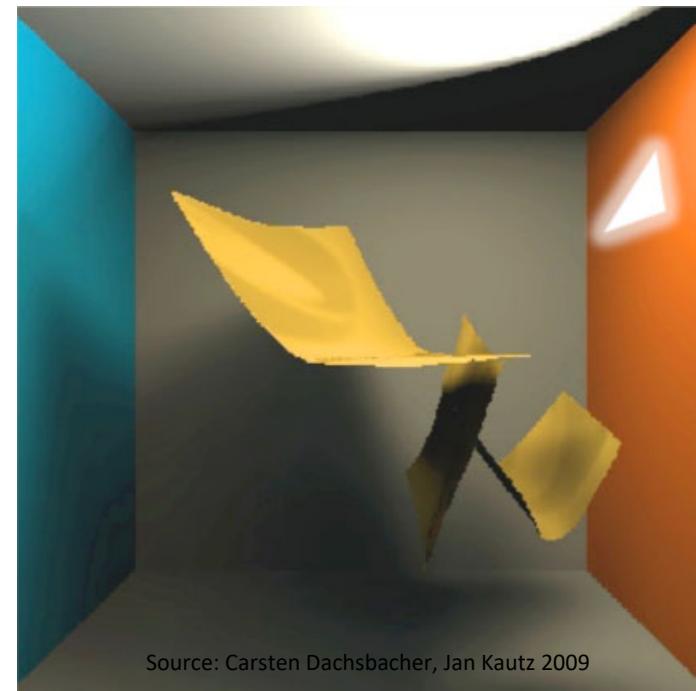
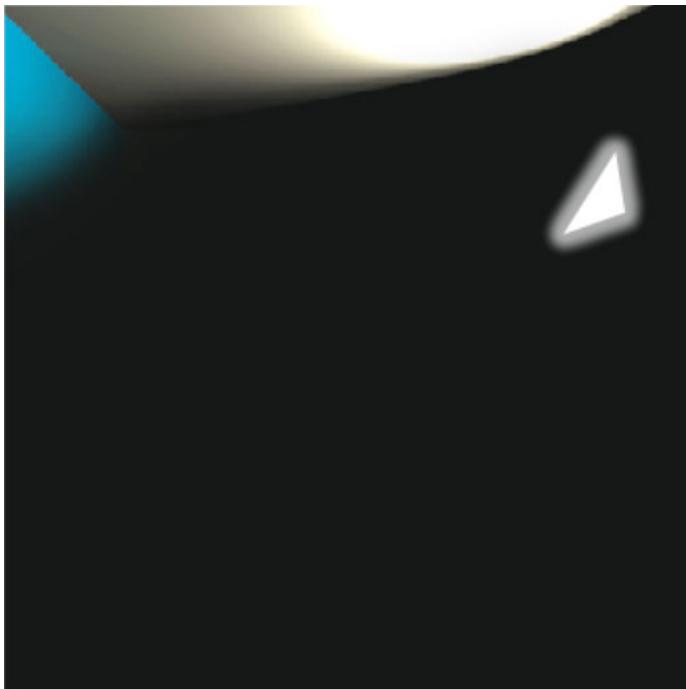
- Hard shadows
- Reflections
- Refractions
- Indirect Illumination
- Ambient Occlusion
- Caustics...



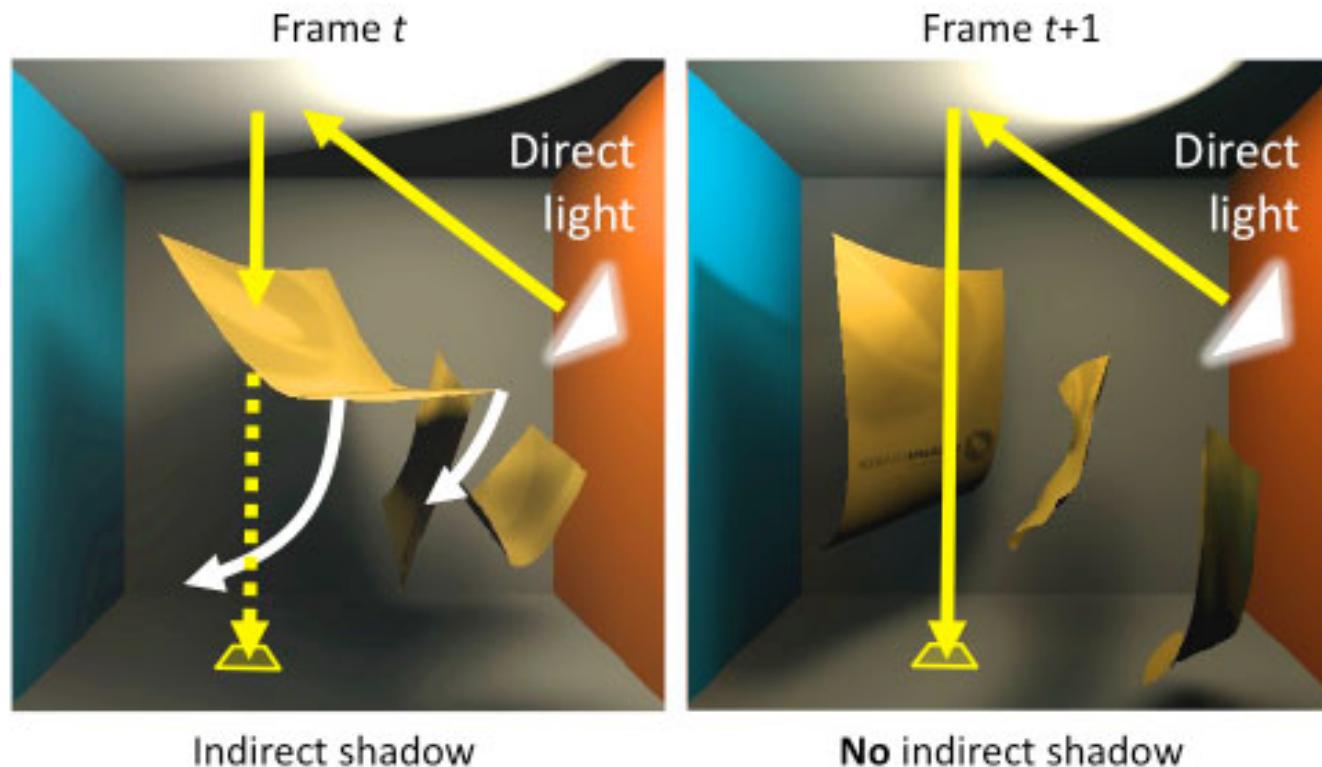
# Adding Cast Shadows



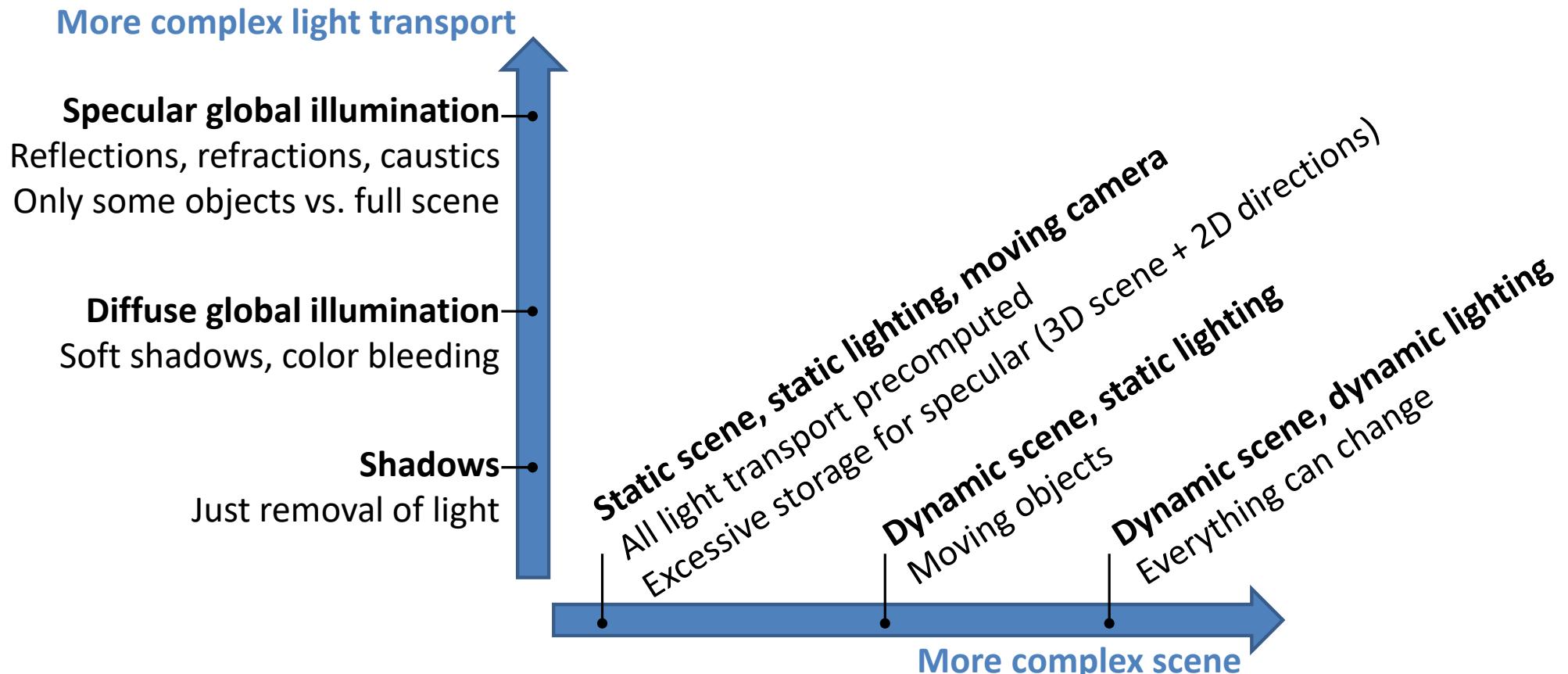
# Adding Indirect Illumination



# Shadows from Indirect Illumination



# Two Dimensions of Complexity



# Recap: Radiometry

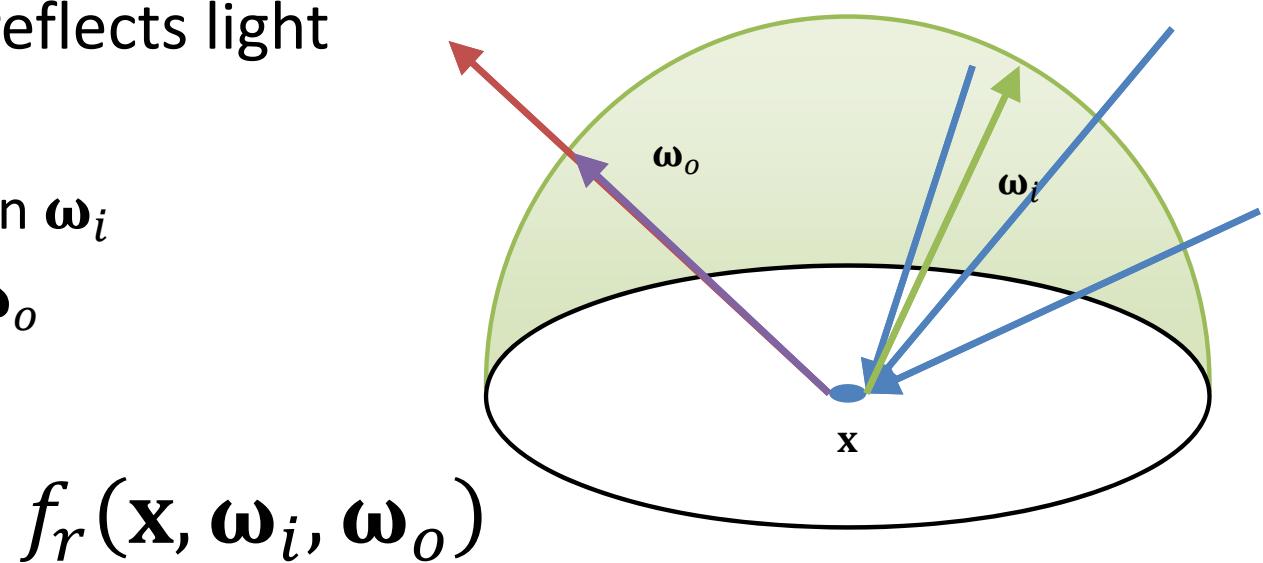
- What are we dealing with?
  - Light
  - Electromagnetic wave
- Light Field
  - Described by *plenoptic function*  $L(\mathbf{x}, \omega)$
  - Amount of light passing through
    - Each point in space  $\mathbf{x}$
    - In each direction  $\omega$
    - For every wavelength (usually just R, G, B)
    - For every moment (in practice, compute every frame separately)
    - ...

# Rendering Equation

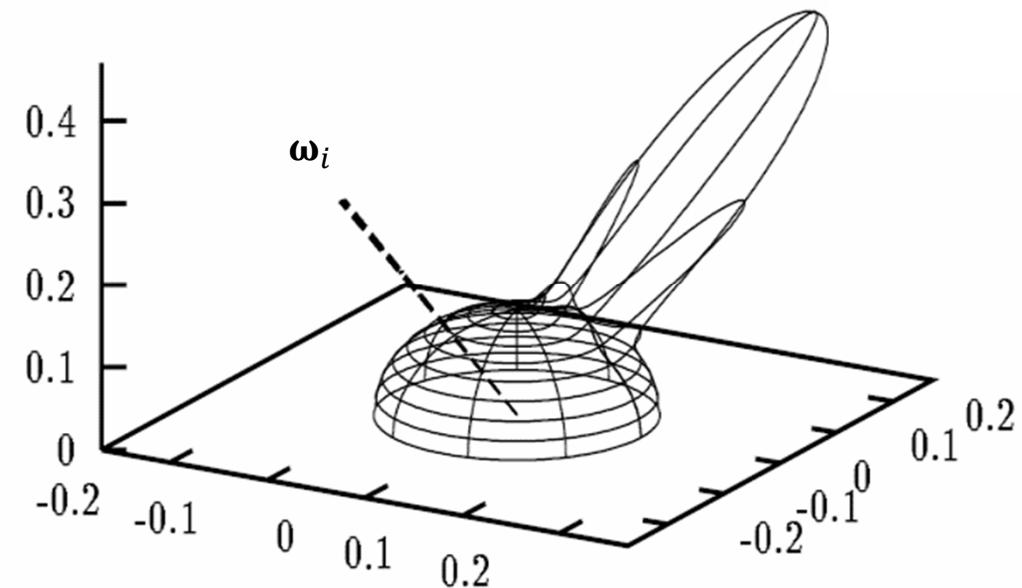
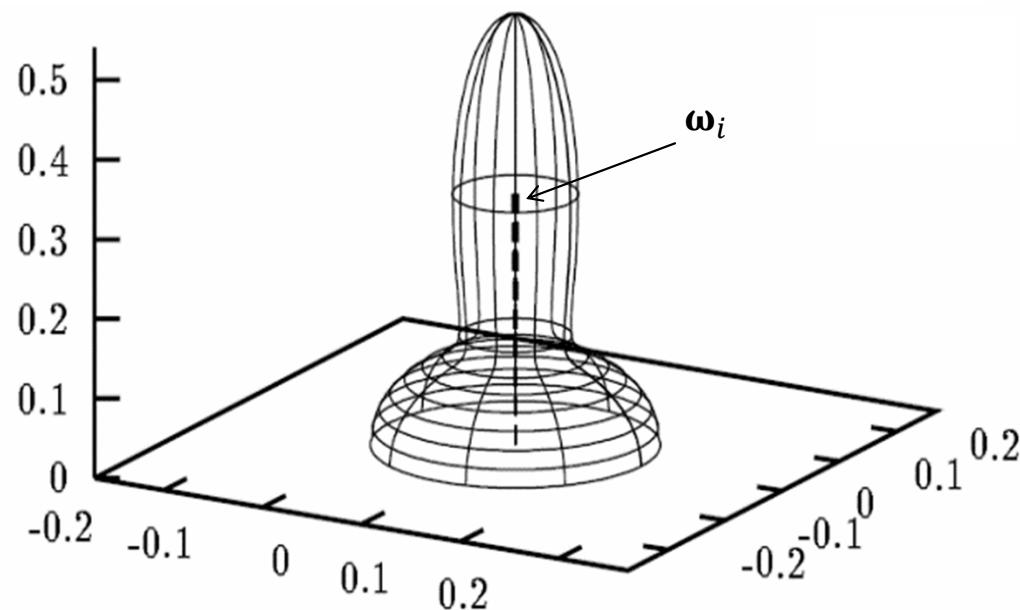
The diagram illustrates the rendering equation. It shows a scene volume represented by a large oval. Inside the volume, a point  $\mathbf{x}$  is highlighted. From this point, a blue arrow points outward, labeled "emitted light". Another blue arrow points inward, labeled "reflected light". The reflected light path is depicted as a curved arrow originating from point  $\mathbf{x}$  and terminating at another point  $\mathbf{y}$ . A label "BRDF" with a blue arrow points to the curved part of the reflected light path.

# BRDF

- Bidirectional Reflectance Distribution Function
- Describe how surface reflects light
  - At location  $\mathbf{x}$
  - From incoming direction  $\omega_i$
  - To outgoing direction  $\omega_o$



# Example: Phong BRDF

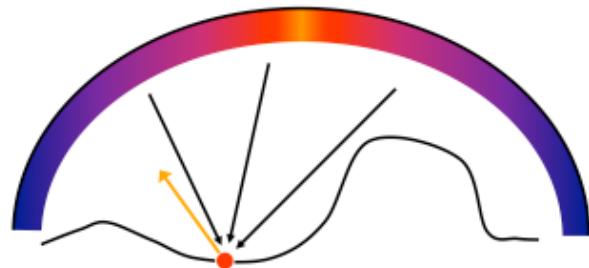


# BRDF Properties

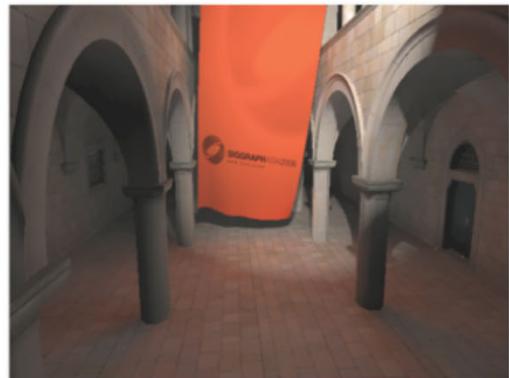
- Reciprocity
  - $f_r(\mathbf{x}, \omega_1, \omega_2) = f_r(\mathbf{x}, \omega_2, \omega_1) \quad \forall \omega_1, \omega_2$
- Energy conservation
  - $\int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o) d\omega_i \leq 1 \quad \forall \omega_o$
- Positivity
  - $f_r(\mathbf{x}, \omega_1, \omega_2) \geq 0$

# Neumann Expansion of Indirect Illumination

Outgoing radiance as infinite series:  $L(\mathbf{x}, \omega_o) = L_0(\mathbf{x}, \omega_o) + L_1(\mathbf{x}, \omega_o) + \dots$

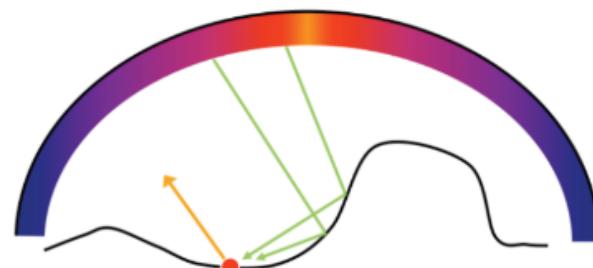


$L_0$  Direct lighting arriving at point  $\mathbf{x}$



Direct + Indirect

Dieter Schmalstieg  $L(\mathbf{x}, \omega_o) = L_0(\mathbf{x}, \omega_o) + L_1(\mathbf{x}, \omega_o)$



$L_1$  All paths from source that take 1 bounce



Indirect only

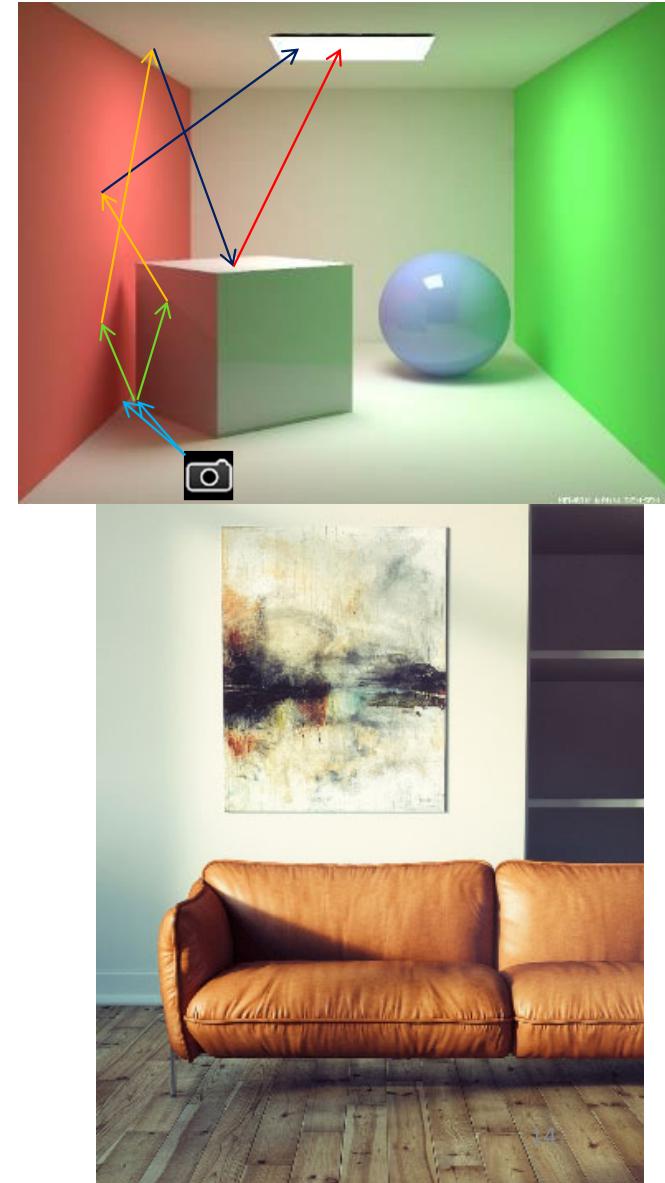
$L_1(\mathbf{x}, \omega_o)$

Direct only

$L_0(\mathbf{x}, \omega_o)$

# Path Tracing

- Reference solution
- Solve rendering equation numerically
  - Follow all possible ray paths (specular+diffuse)
  - Monte Carlo approach → noise
- No fundamental limits concerning realism
- Not really efficient
  - Hard to find a path that connects camera and light source with strong (specular) transport



# Two-Pass Global Illumination

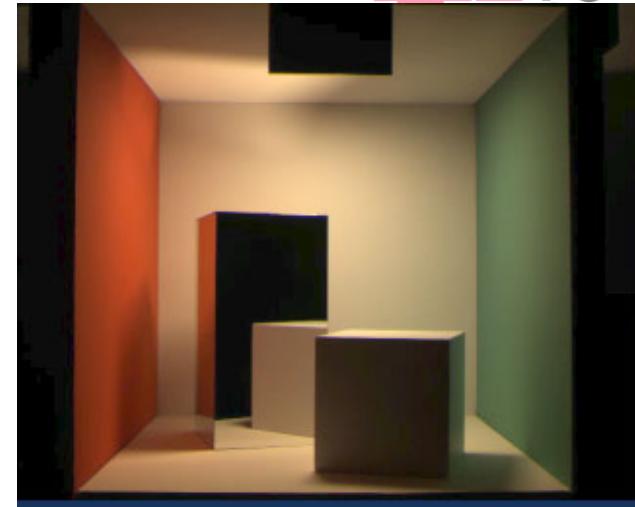
- 1<sup>st</sup> pass computes light transport in the scene
- 2<sup>nd</sup> pass collects lighting information to generate final image
- Every pass chooses independently
  - Type of light transfer (diffuse *or* specular)
  - Rendering method (ray tracing *or* rasterization)
  - Update rate (once *or* sometimes *or* every frame)

# Examples of Two-Pass Methods

Method	Transport	1 <sup>st</sup> pass result	2 <sup>nd</sup> pass
Radiosity	Diffuse	Light maps	Render using texture maps
Photon mapping	Diffuse + some specular	Photon map	Raytracing
...			

# Radiosity

- Pass 1: Create lightmaps
  - Finite elements approximation
  - Discretize scene into patches
  - Assume everything is perfectly diffuse
  - Light transport → linear equation system
  - Solve equation system
- Pass 2: Render using texture mapping



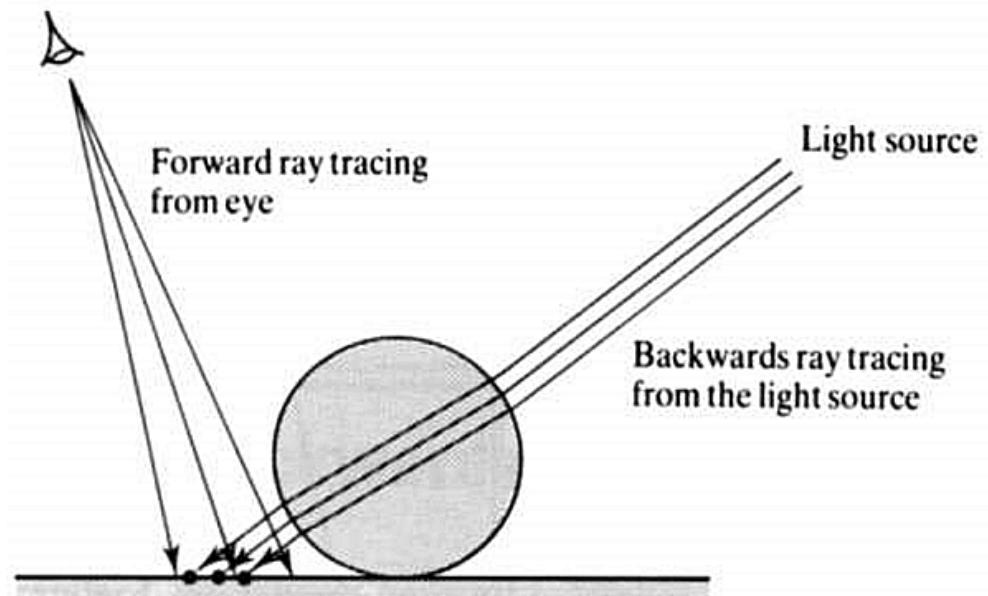
# Photon Mapping

Two passes

## 1. Photon map creation

- Shoot photons from light source
- Store particles hitting diffuse surfaces in data structure

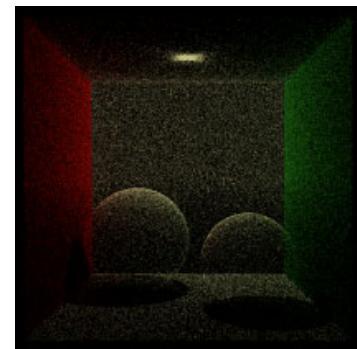
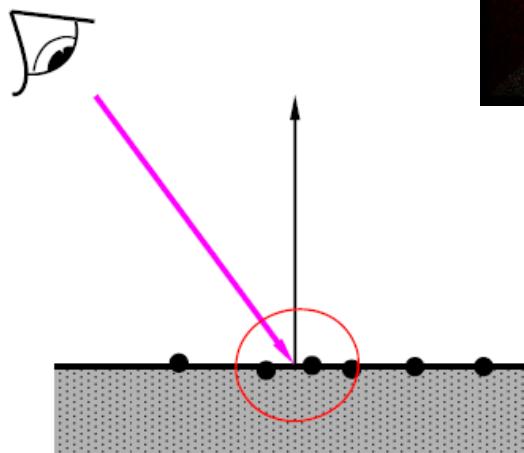
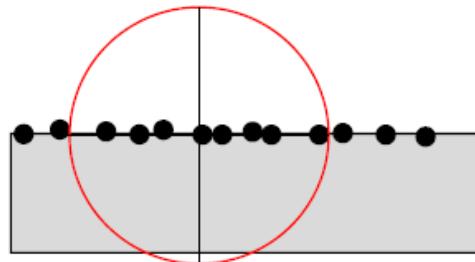
## 2. Rendering using photon map



aus: Watt, Watt: Advanced animation and rendering techniques

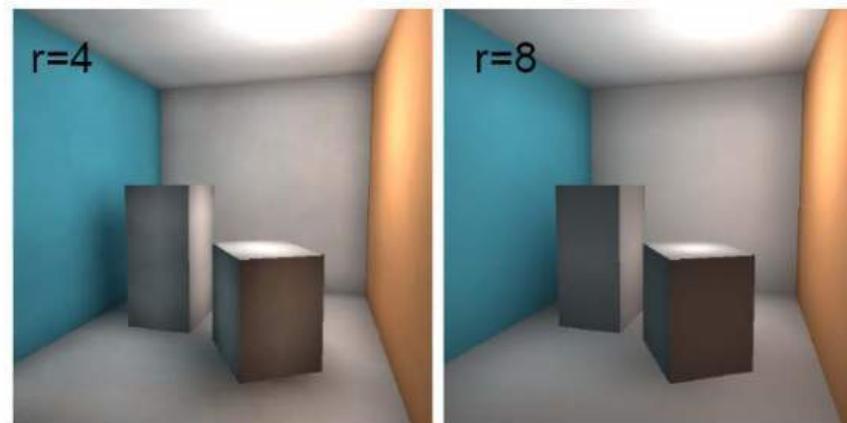
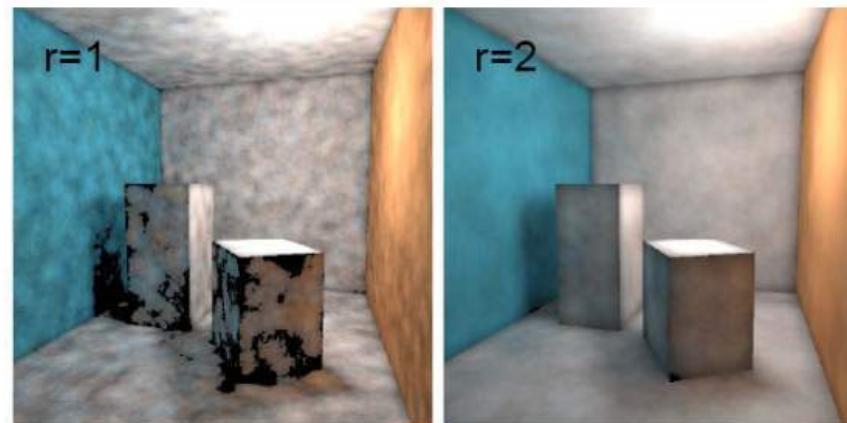
# Photon Mapping - Creation

- At point  $x$ : collect photons contained in sphere of radius  $r$
- Number of photons = intensity at point  $x$
- Can simulate complex light phenomena (e.g., caustics)
- Requires fast spatial search (kd-tree)



# Photon Mapping - Rendering

Small radius  
→ noise



Large radius  
→ loss of detail

# Render Cache

- Data structure connecting the passes
- What data to store
  - Radiance cache (outgoing illumination for every point), or
  - Irradiance cache (incoming illumination for every point)
- How to index
  - Points only on surfaces (needs a surface index, like kd-tree)
  - Points everywhere in space (needs a voxel grid or octree)

# Render Cache Organization

- Organized only by position
  - Photon maps: sparse kd-tree
  - Classic Radiosity: radiance on surface points or patches
- Organized only by orientation
  - Environment map: cubic radiance map
- Organized in projective space
  - 3D positions in a 2D depth map
  - Shadow mapping, instant radiosity
- Organized by both position and orientation
  - Position as main index
  - Orientation as sub-indexed, often compression using Spherical Harmonics (SH)
  - Irradiance volumes *or* light propagation (=irradiance) volumes

# How to Compute Illumination in Real Time

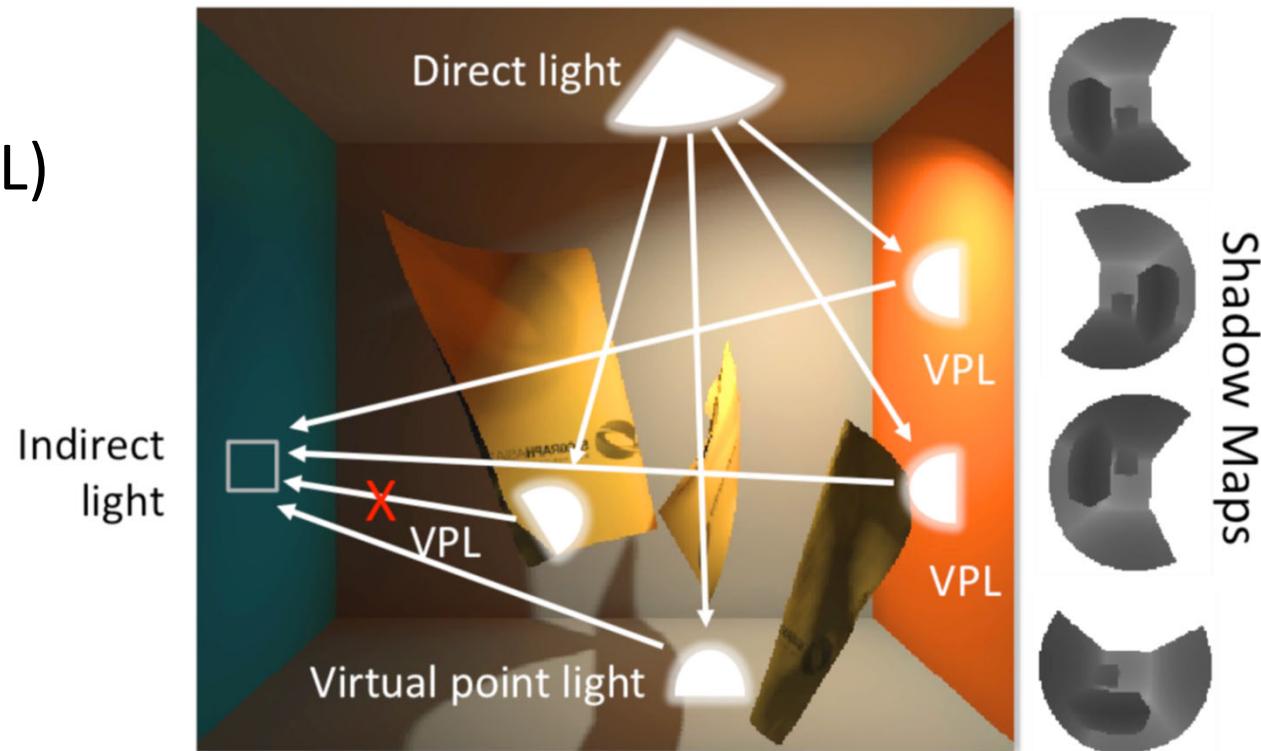
- General global illumination
  - Extremely high complexity
  - Global transport does not naturally fit to online rendering
- Where can we approximate?
  - Semi-global: global effects only for some objects
  - Low-order: allow only 1-2 bounces of light
  - Ignore specular light transport
  - Static scene
  - Static light sources
- Rasterization is more efficient than raytracing

# Real-Time Two Pass Methods

Method	Transport	1 <sup>st</sup> pass result	2 <sup>nd</sup> pass
Radiosity	Diffuse	Light maps	Render using texture maps
Photon mapping	Diffuse + some specular	Photon map	Raytracing
Instant Radiosity	Diffuse	Shadow map (rasterization)	Deferred rendering (rasterization)

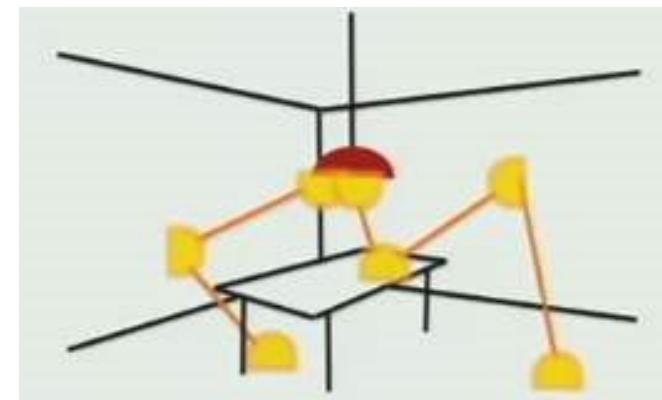
# Instant Radiosity

- Idea: model the light bounces as light sources
- Virtual point lights (VPL)
- Convert indirect into direct light transport



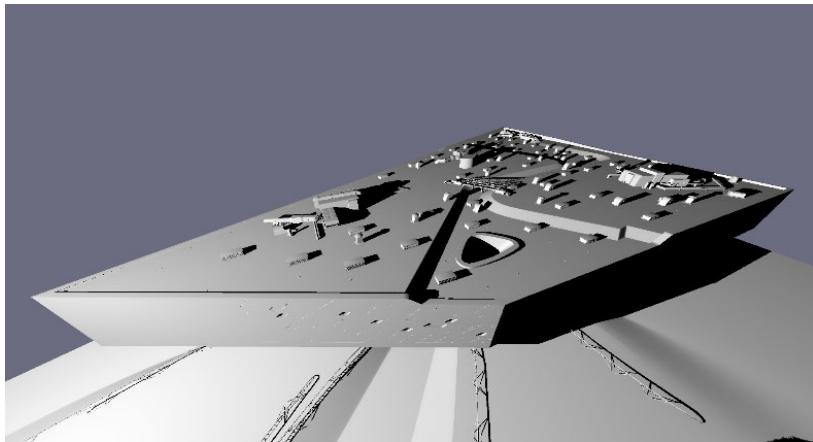
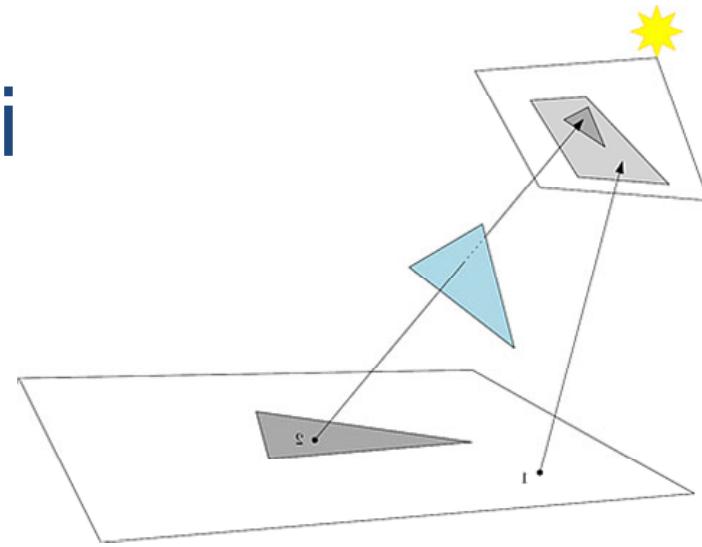
# Virtual Point Light Sources

- 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



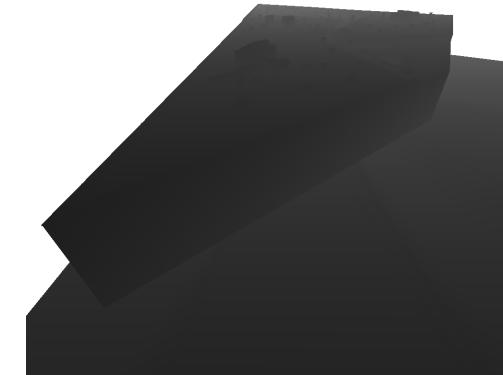
# Recap: Shadow Mappi

- Shadows are a visibility problem
  - Point in shadow  $\leftrightarrow$  invisible for light source
  - Shadow maps use depth buffering



Dieter Schmalstieg

Real-Time Global Illumination



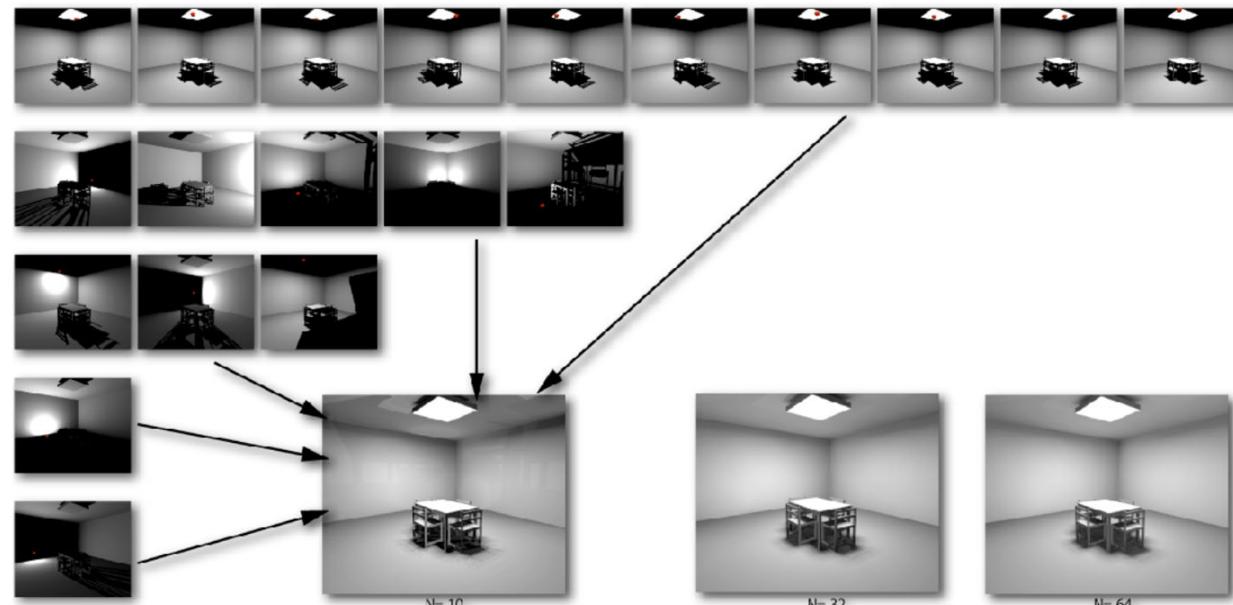
# Recap: Deferred Shading

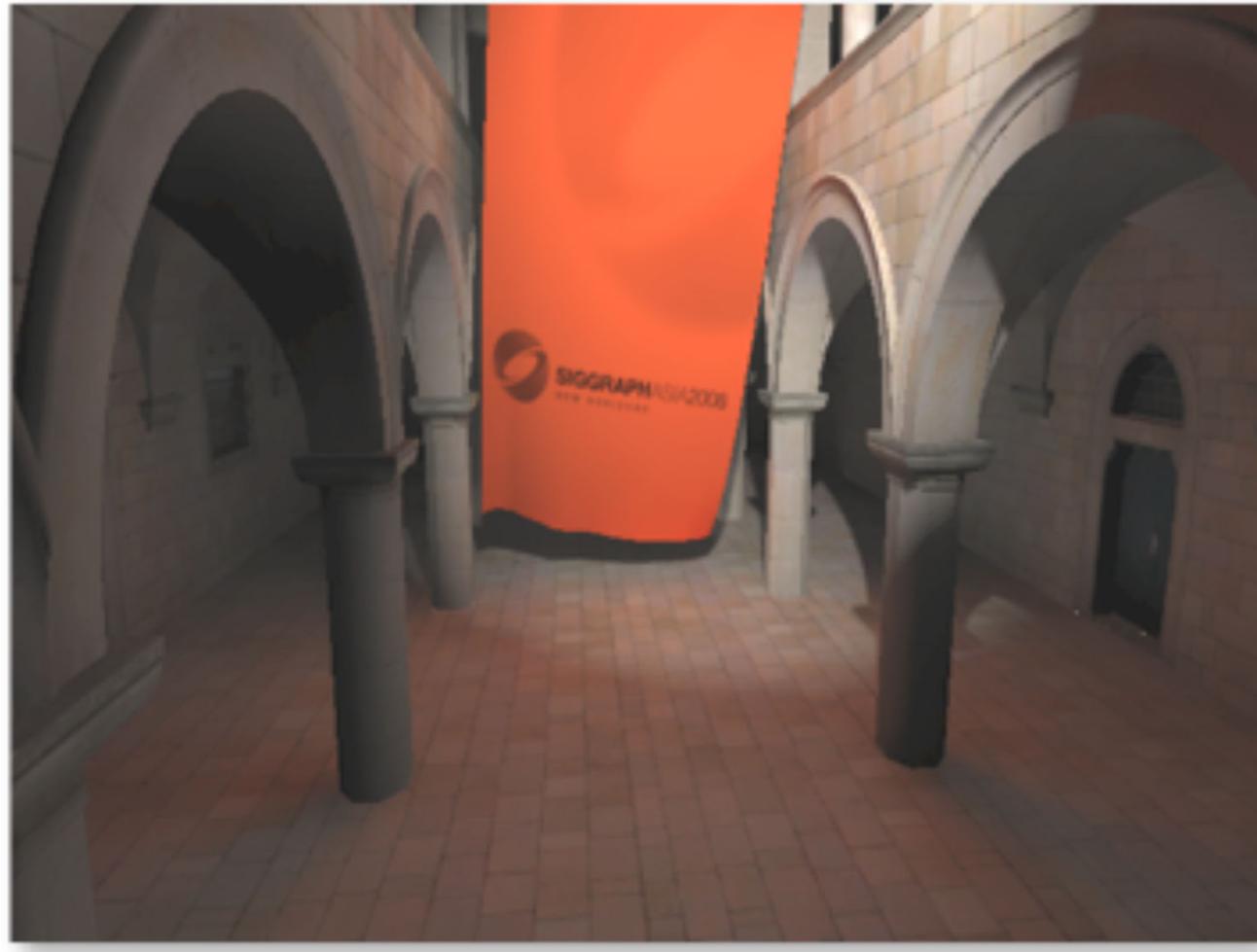
- Traditional shading requires many passes for many light sources
- Deferred shading rasterizes only once and stores in G-buffer
- Compute shading in G-buffer in separate task
- Decouples geometric complexity from lighting complexity



# VPL Distribution

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

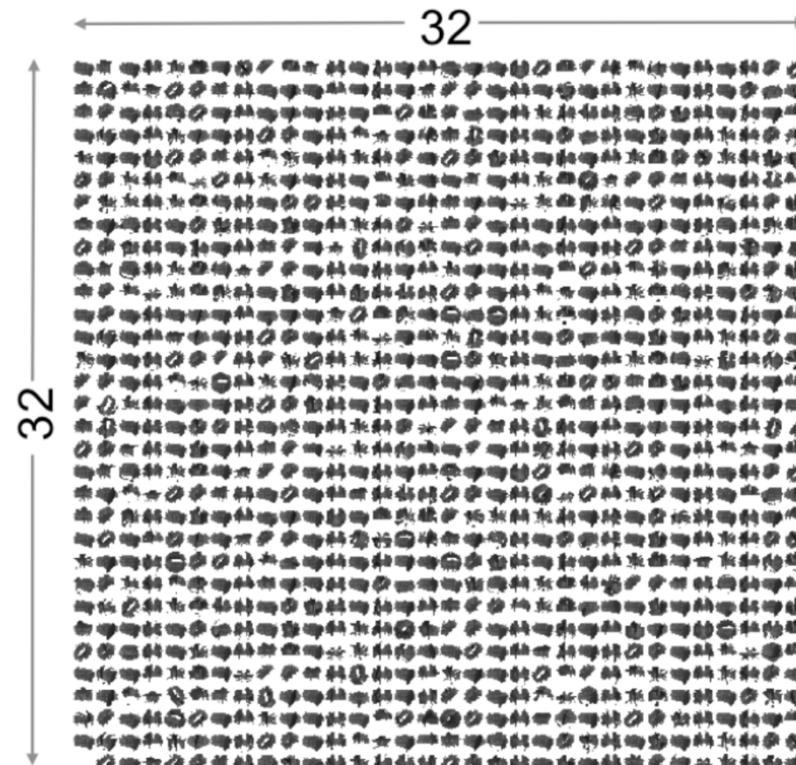




# Instant Radiosity

- Scene: 100K triangles
- Illumination: 1024 VPLs

→ Drawing 100M triangles is  
too much...



# Incremental Instant Radiosity

- Optimization assumes a semi-static scene
- Exploit frame-to-frame coherency
- Reuse VPLs from previous frames
  - Only a couple of new shadow maps per frame

# Imperfect Shadow Maps

- VPL approaches do not require accurate geometry



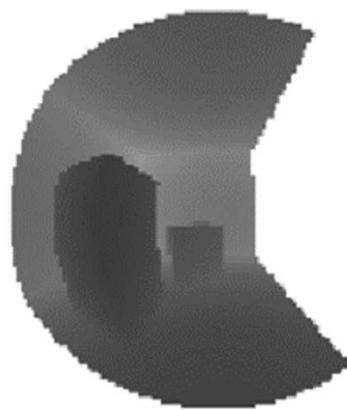
High-Quality Depth



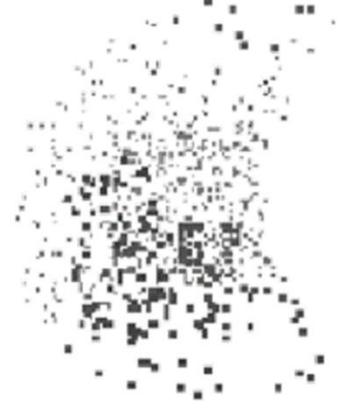
Low-Quality Depth  
(20% corrupted)

# Creating Imperfect Shadow Maps

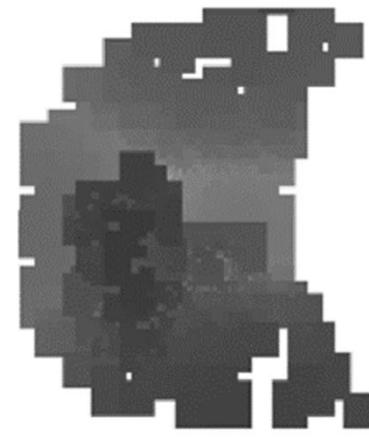
- Parabolic (omnidirectional) shadow map
- Coarse approximation of scene as point cloud
- Limited number of points



Classic  
Triangles

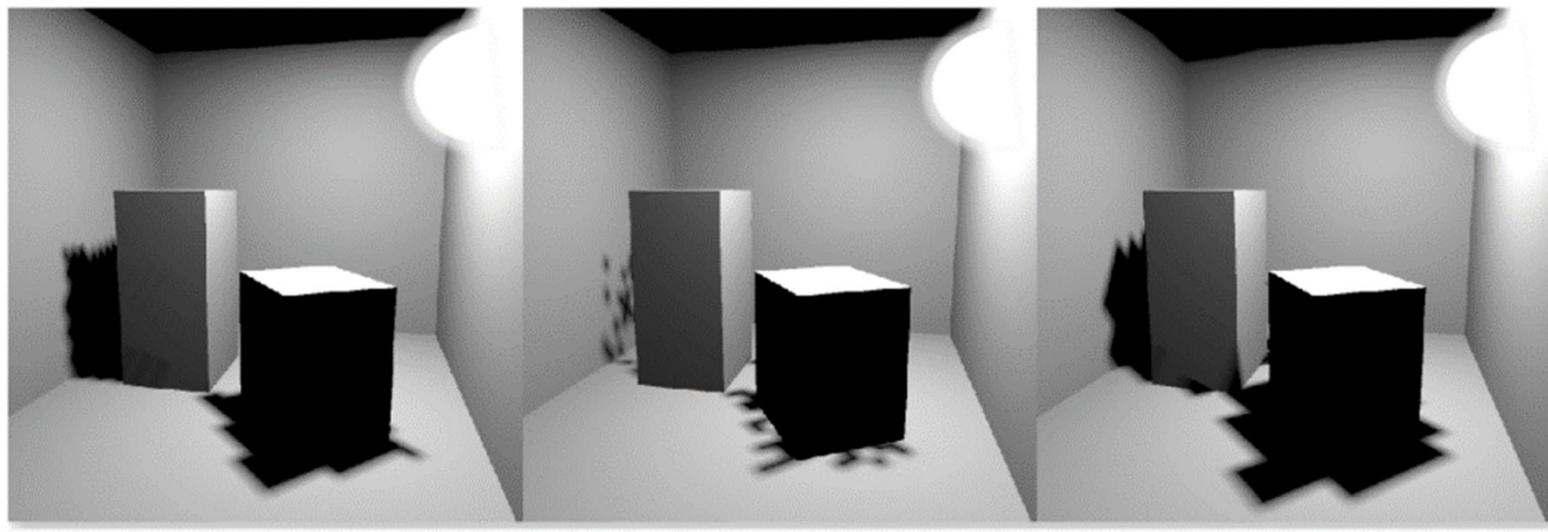


Imperfect  
Smaller points  
Real-time Global Illumination



Imperfect  
Pull-Push  
Hole Filling

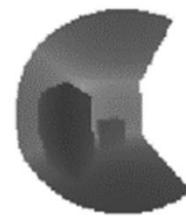
# Dense Geometry with Pull-Push Interpolation



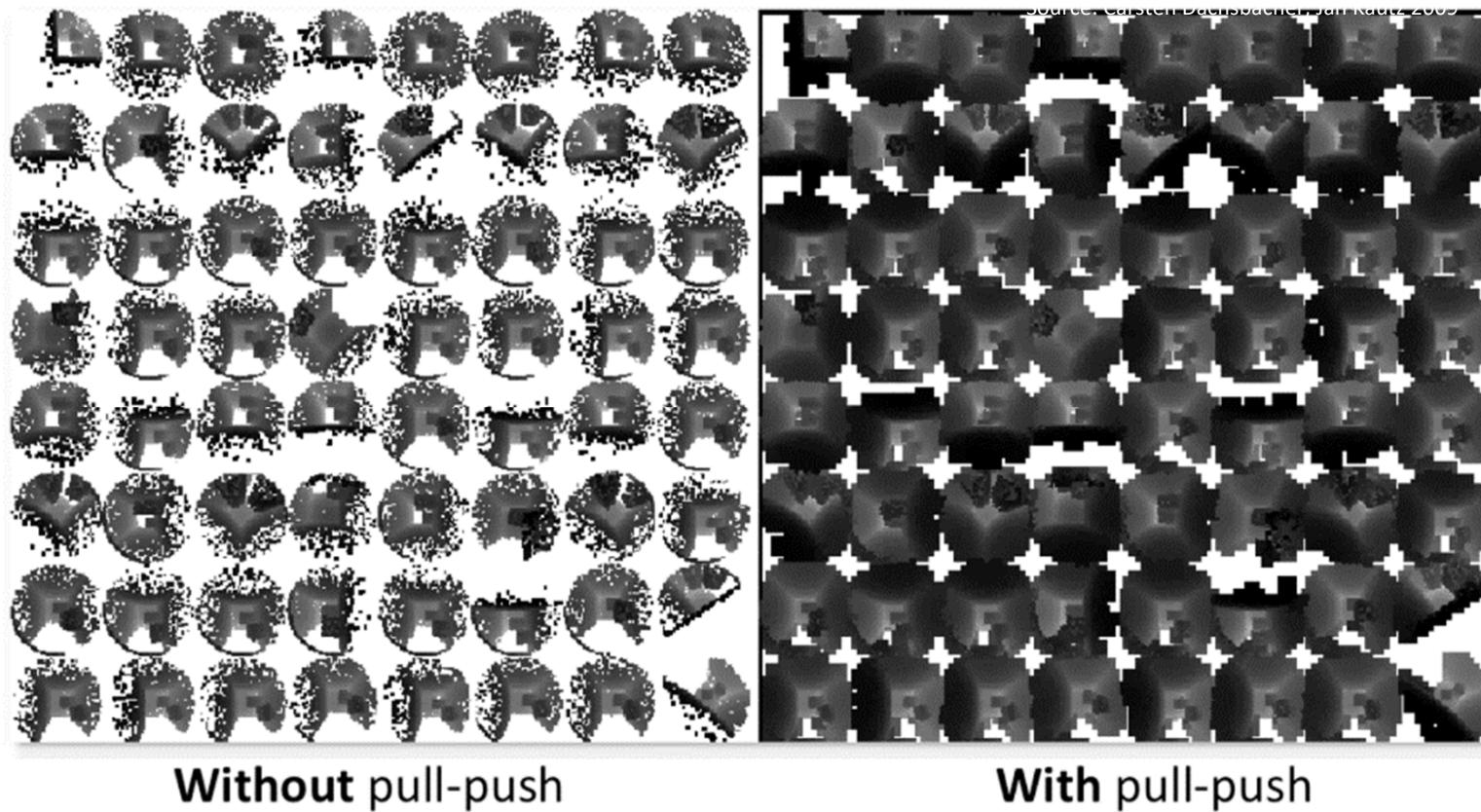
Classic

Without pull-push

With pull-push

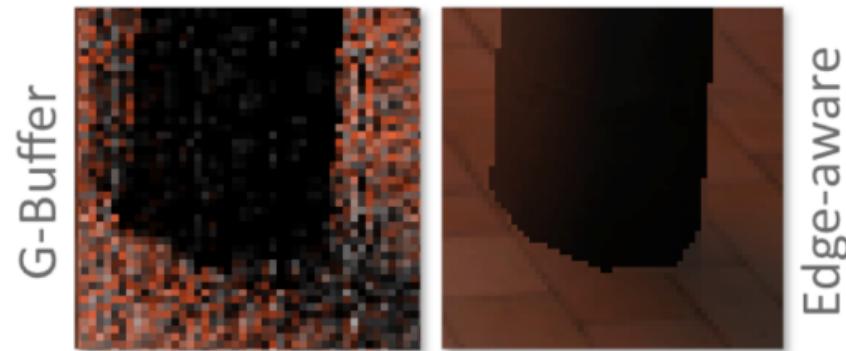
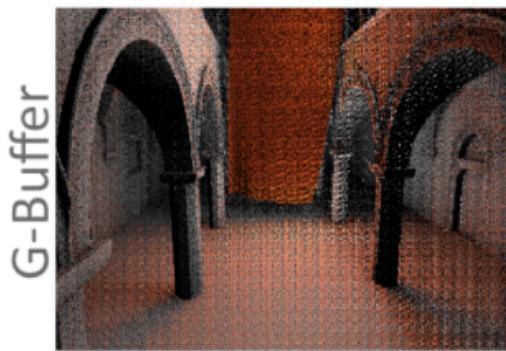


# Comparison Pull-Push



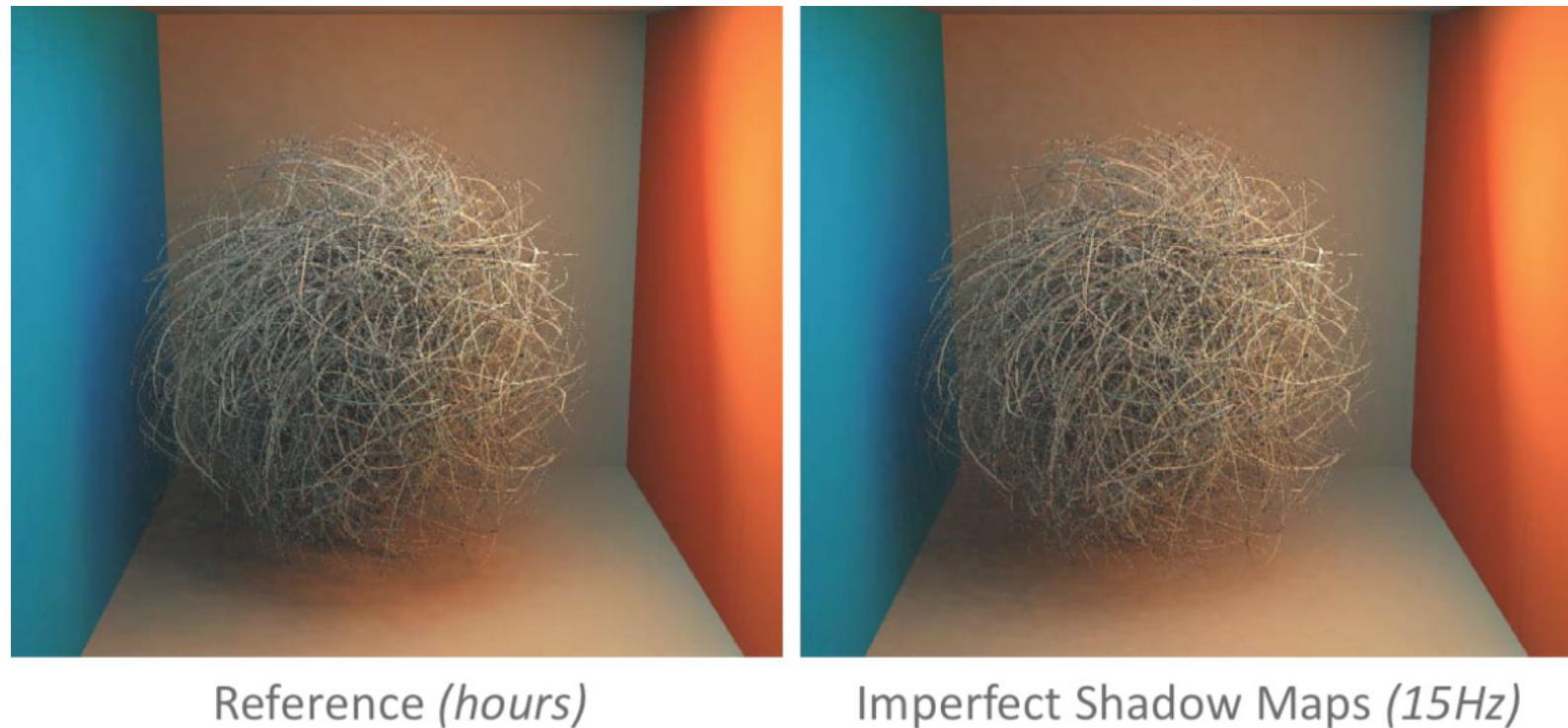
# Interleaved Sampling

- Sample light sources randomly
  - Only few light sources per pixel
- Average results
  - Using edge aware (bilateral) filter

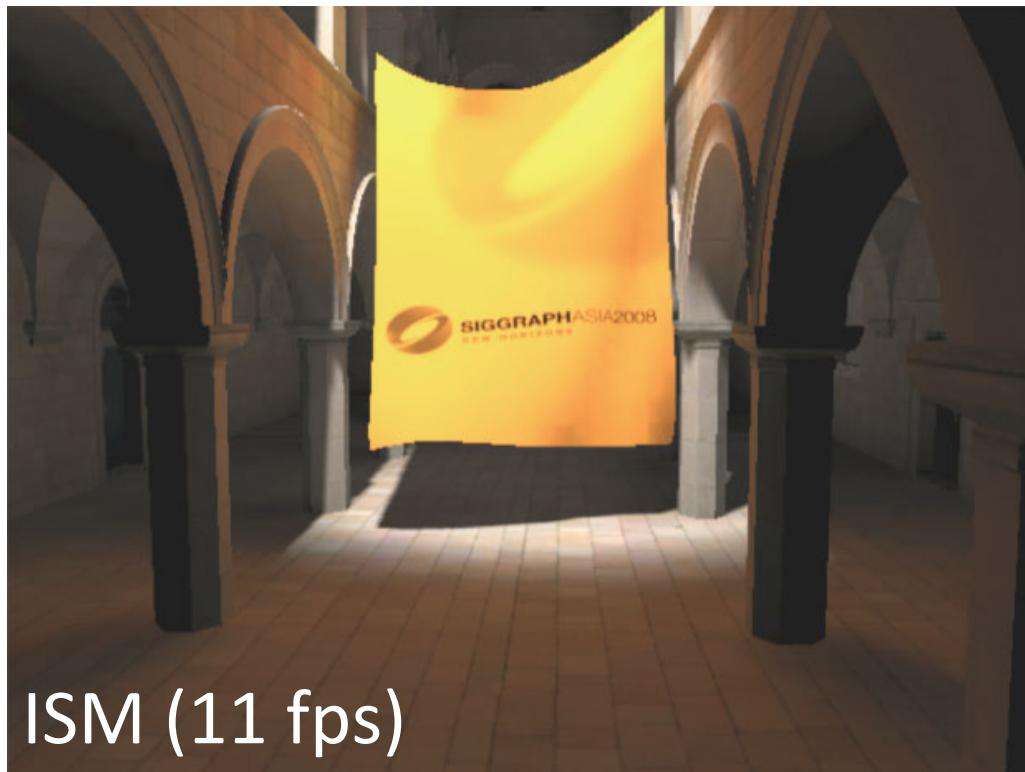


Edge-aware

# Imperfect Shadow Maps Results 1



# Imperfect Shadow Maps Results 2



ISM (11 fps)



Reference (hours)

# More Results and Effects

Cornell box  
horse



Christo's Sponza

Multiple  
bounces



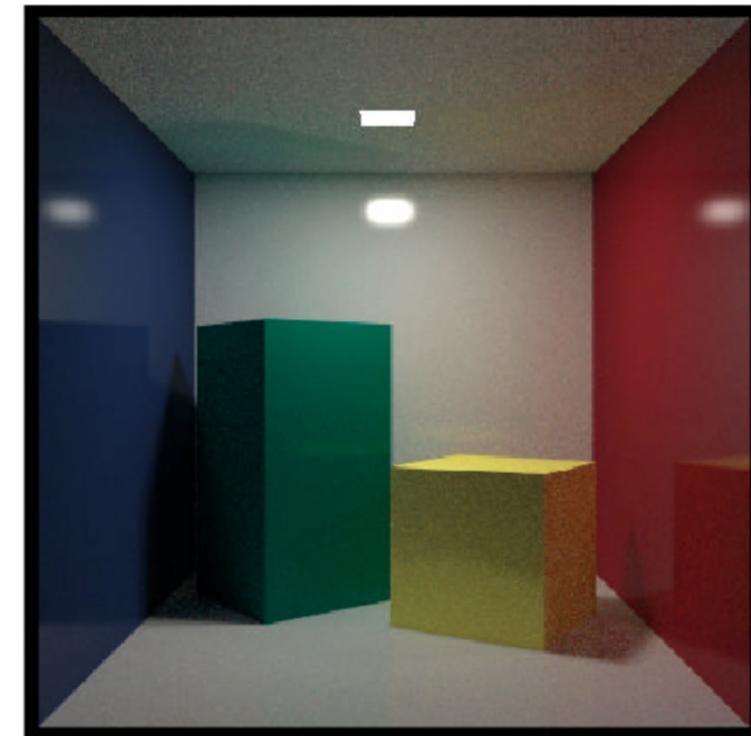
Caustics

# Problems of Instant Radiosity

- Scalability
- Performance
  - Can reach interactive framerates
  - But still not fast enough for games



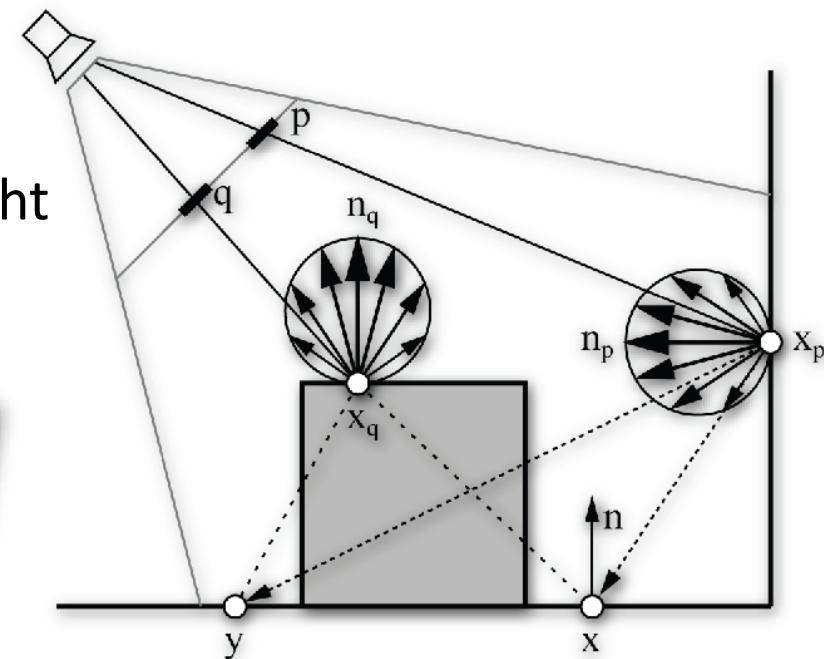
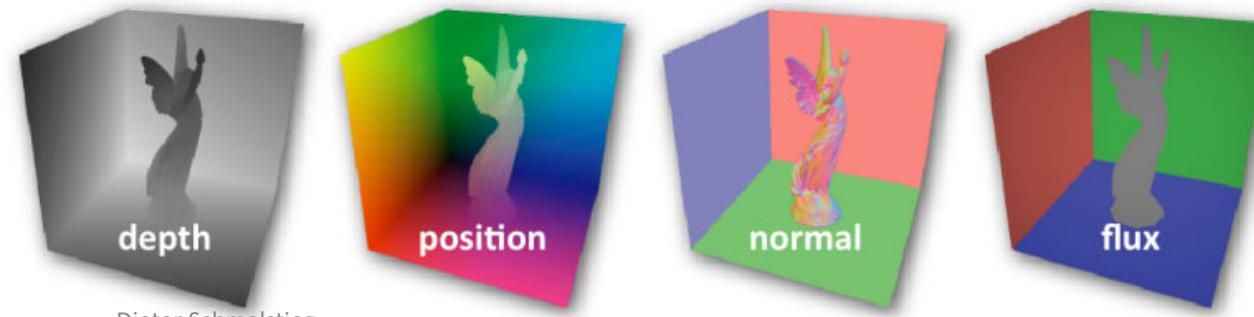
Instant Radiosity



Path Tracing

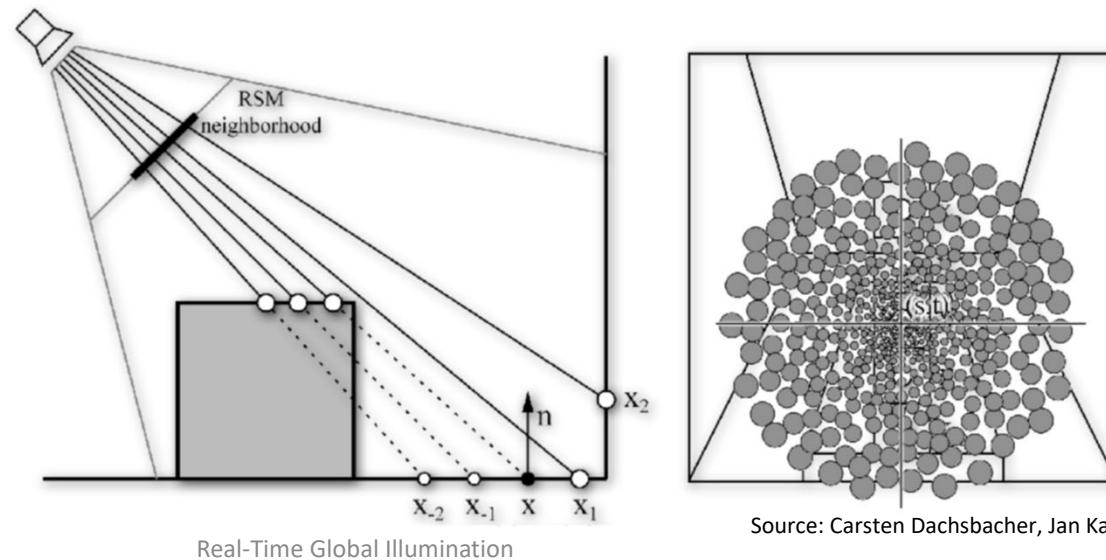
# Reflective Shadow Maps

- Restrict to one bounce
- Single bounce illumination from surfaces seen by light source
- Each pixel is small light source
- During rendering of shadow map
  - Store additional information on received light
  - Called a *reflective shadow map* (RSM)

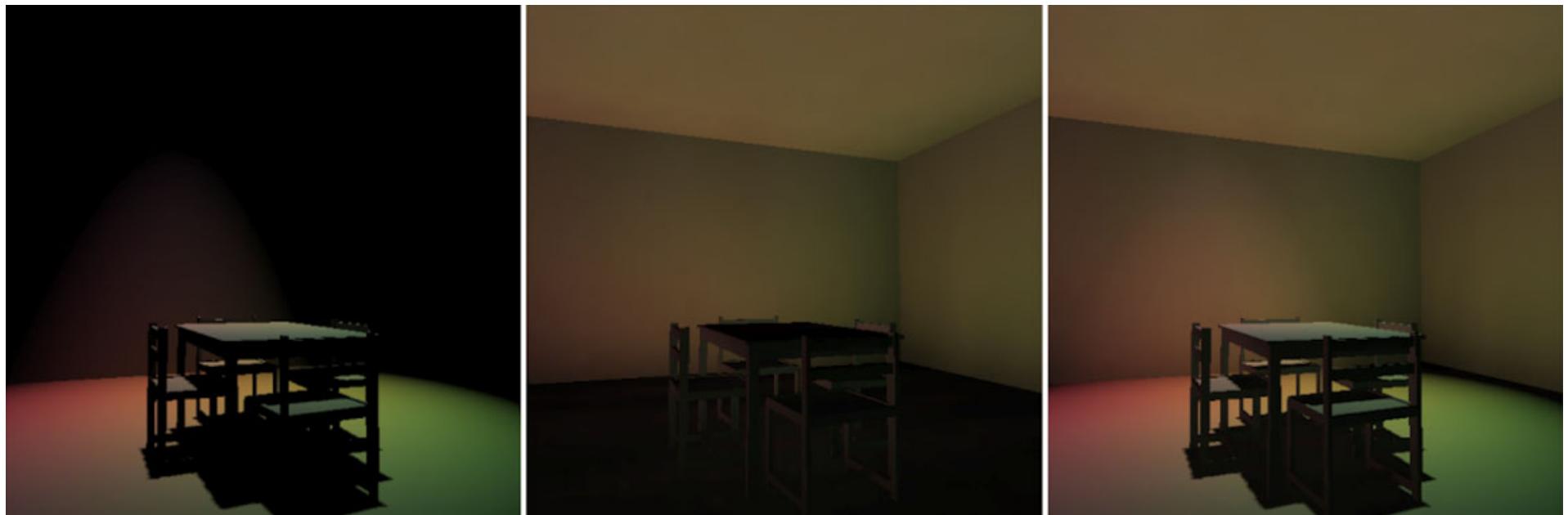


# Rendering with Reflective Shadow Maps

- Gather illumination from RSM
- Too many pixel lights
- Restrict to samples close to current location



# Reflective Shadow Maps Result



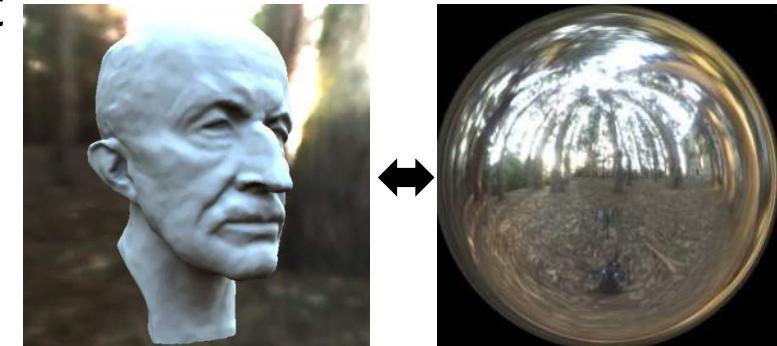
# Reflective Shadow Maps

- Crude approximation
  - Diffuse reflectors
  - No indirect shadows
  - ...



# 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

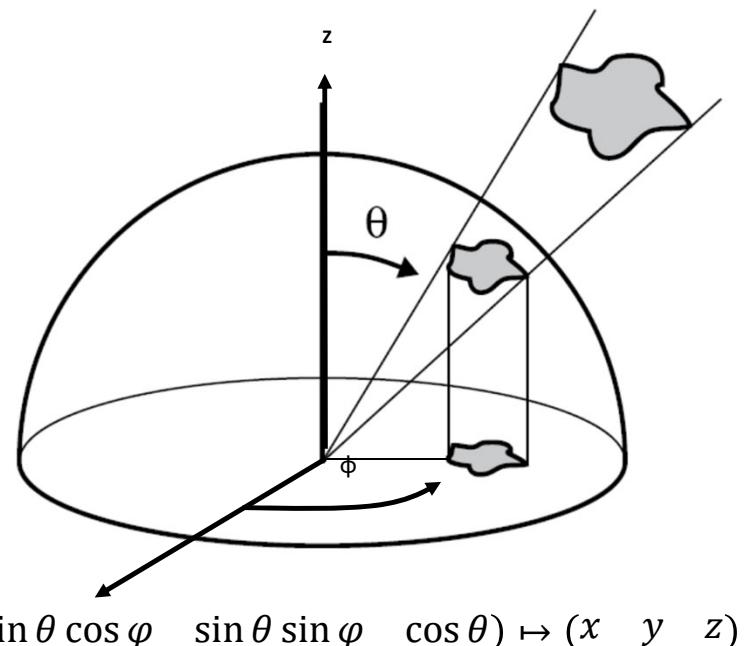
- During rendering, in real-time

Spherical harmonics sind ein Werkzeug in der Computergrafik zur Beschreibung von Lichtverteilungen auf einer Kugelfläche. Sie können verwendet werden, um die Beleuchtung von 3D-Modellen realistisch darzustellen und die Beleuchtung von Umgebungen in Computerspielen und virtuellen Umgebungen zu simulieren. Sie basieren auf der Mathematik der harmonischen Analysis und ermöglichen es, die Beleuchtungsinformationen in eine begrenzte Anzahl von Koeffizienten zu komprimieren, was die Berechnungszeit und den Speicherbedarf reduziert.



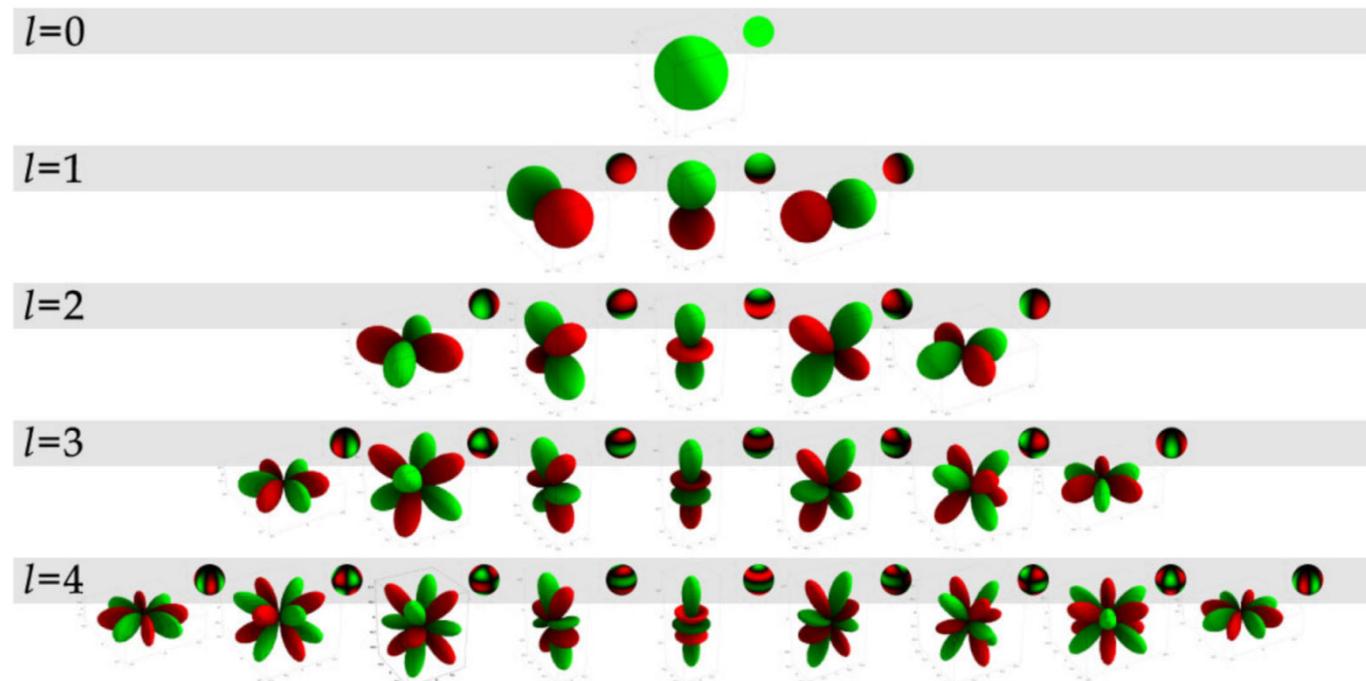
# Spherical Harmonics

Base functions for representing functions with a spherical domain

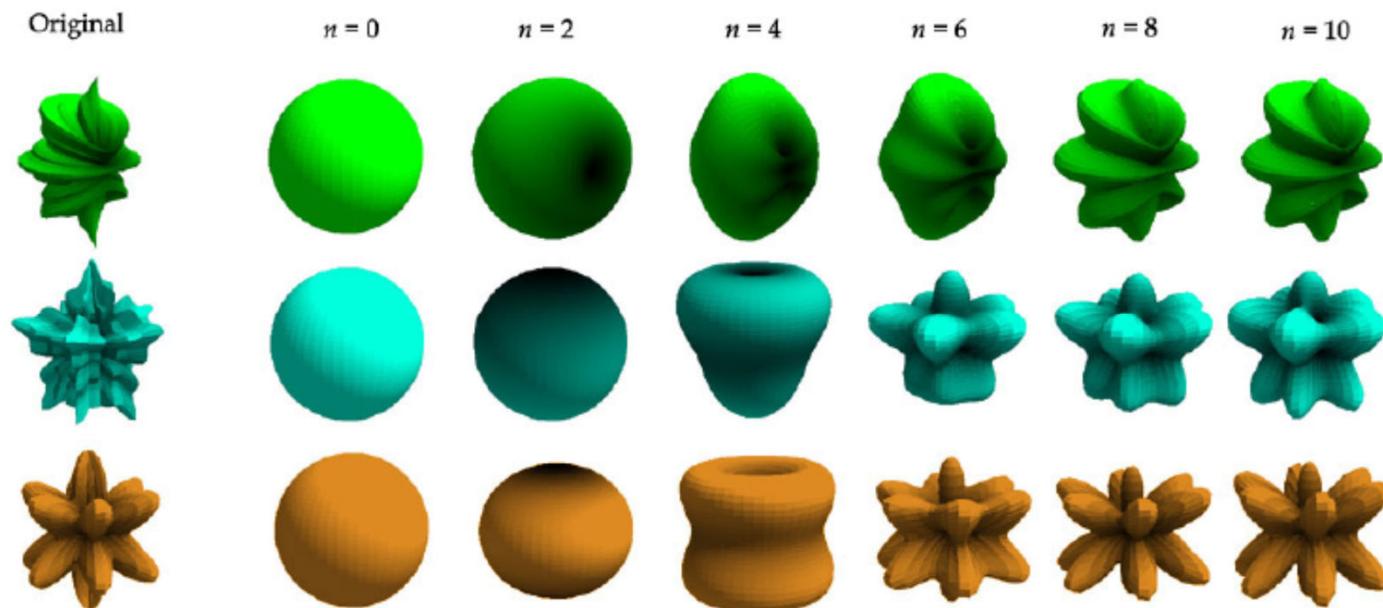


# Spherical Harmonics Bands

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

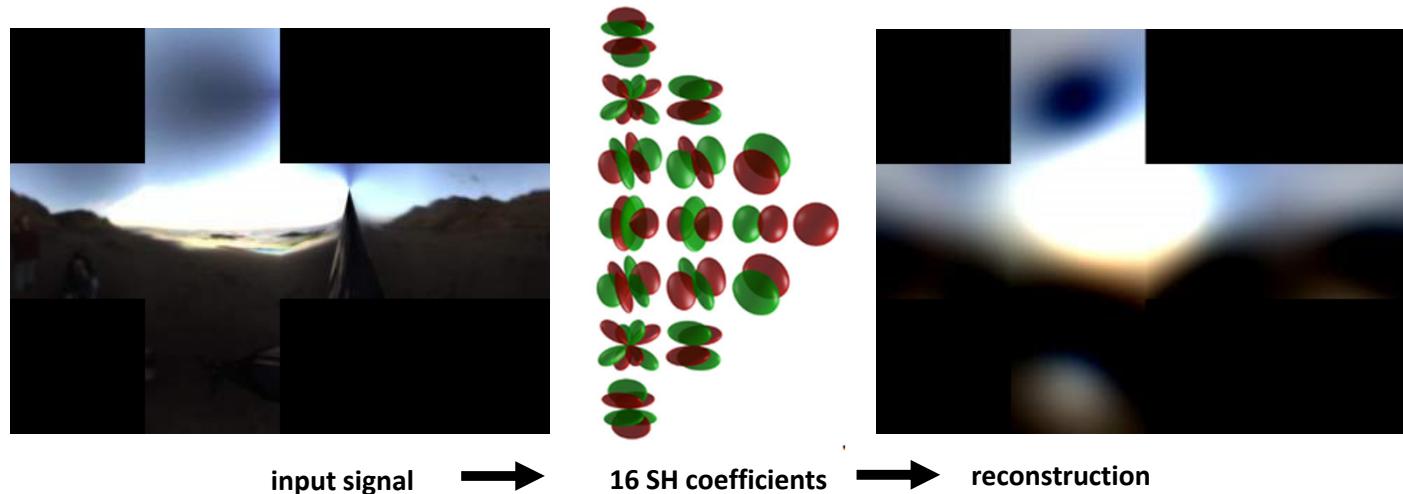


# Examples for SH-Reconstruction



# SH Representation for Environment Maps

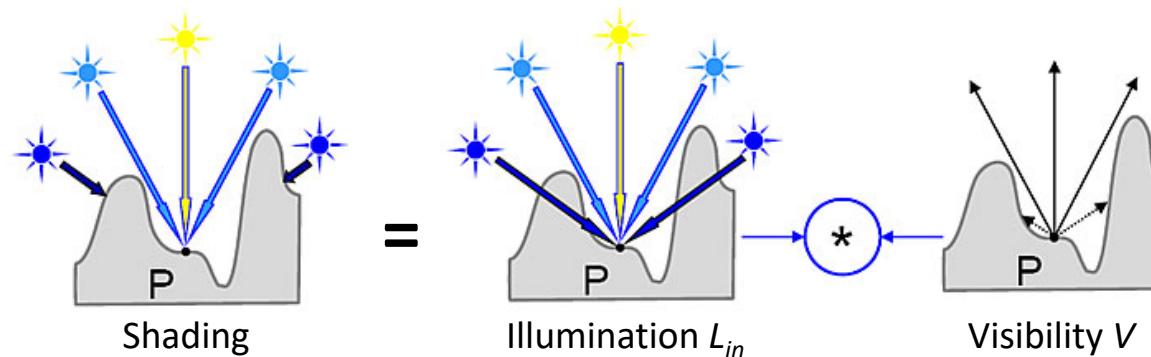
- HDR environment map approximated by SH
- 4 bands = 16 coefficients of SH
- Computation of coefficients by projection



# SH Representation for Light Transfer

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

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

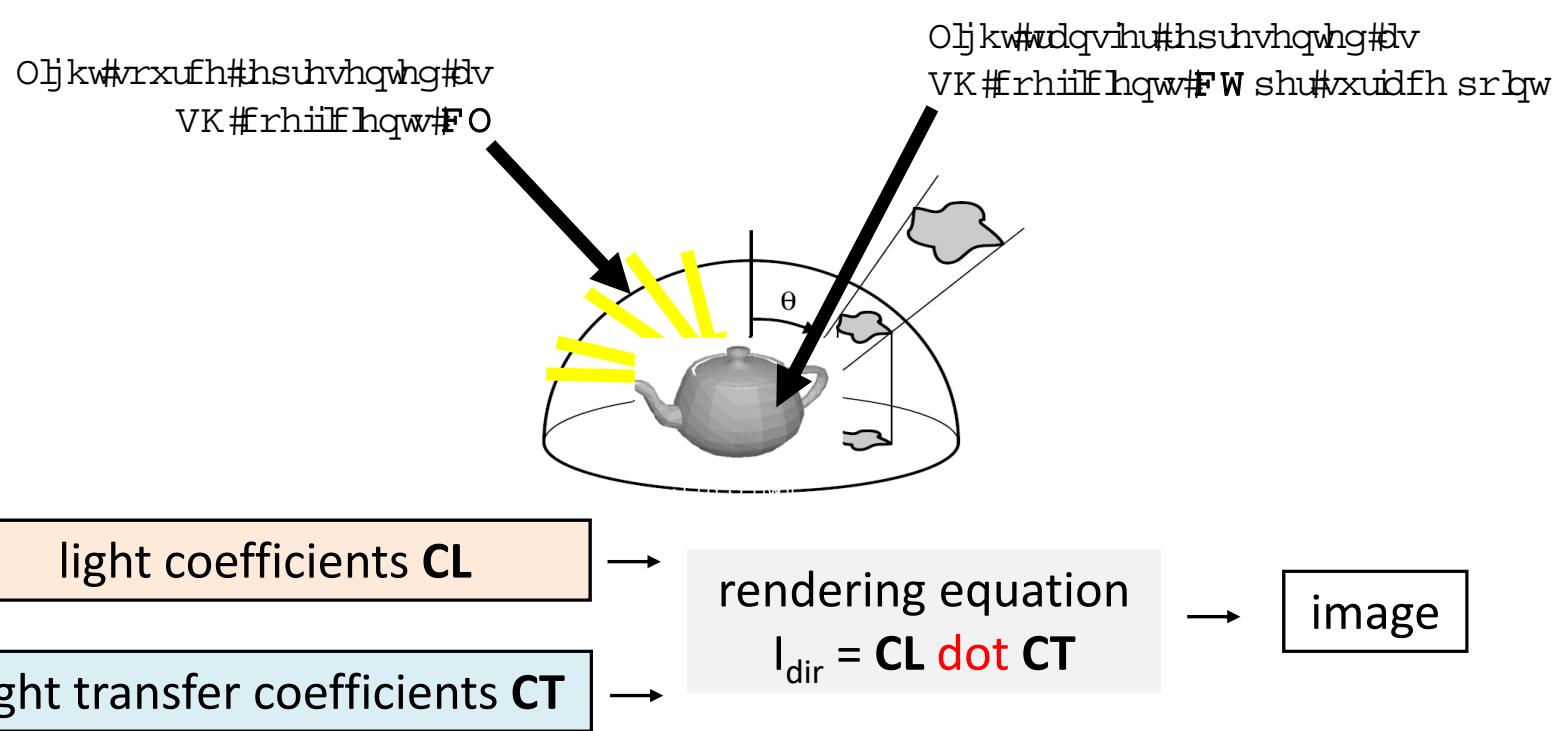


PRT (Precomputed Radiance Transfer) ist eine Technologie in der Computergrafik, die verwendet wird, um die Beleuchtung von 3D-Modellen realistisch darzustellen. Es handelt sich hierbei um eine Art von Global Illumination (GI) Technik.

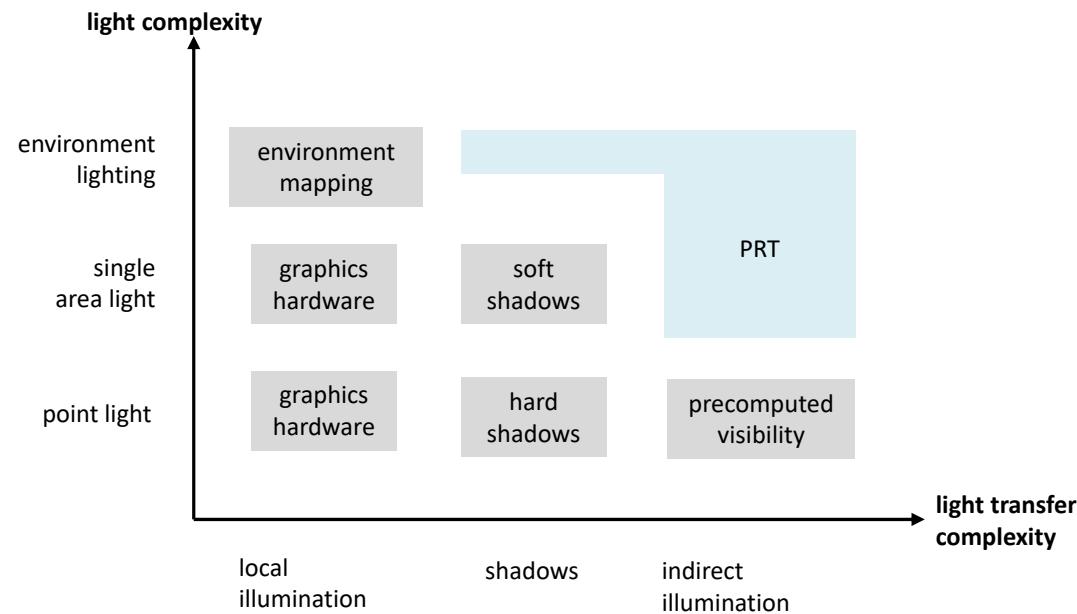
PRT basiert auf dem Konzept des vorab berechneten Lichtübertragung. Es ermöglicht es, die Beleuchtungsinformationen auf ein 3D-Modell vorab zu berechnen und zu speichern, anstatt die Berechnungen in Echtzeit durchzuführen. Dadurch können hochwertige Beleuchtungs- und Schatteneffekte erzielt werden, ohne dass es zu einem großen Leistungsverlust kommt.



## PRT Rendering

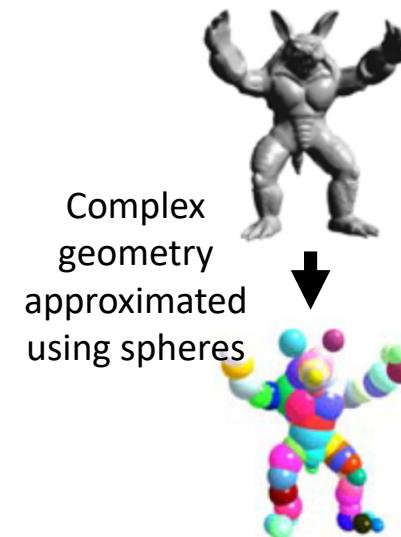
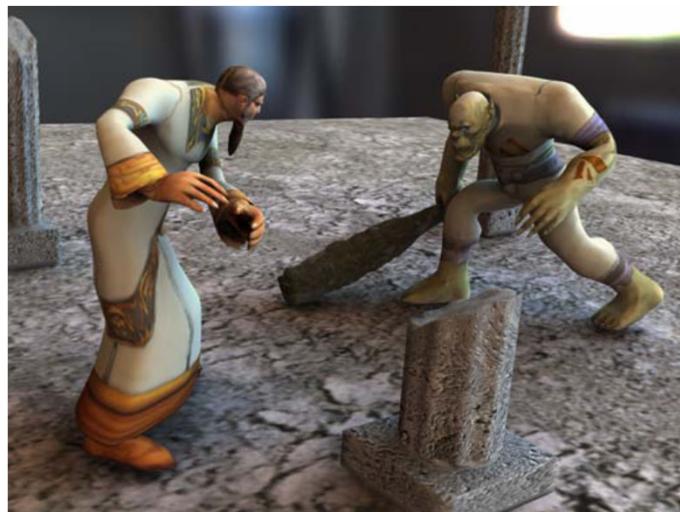


# PRT – Applications



# SH Lighting for Dynamic Geometry

- Dynamic geometry



# 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)

# Questions?



# GPU Raytracing

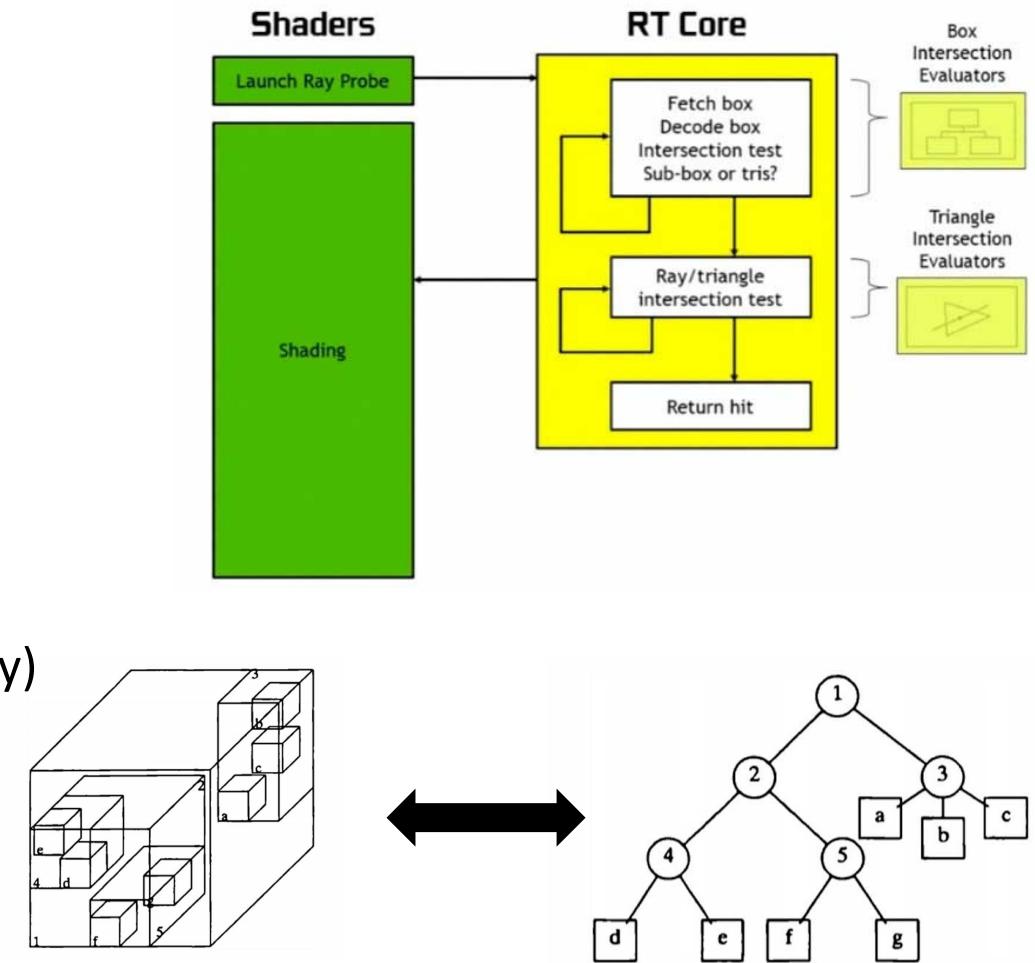
---

Dieter Schmalstieg



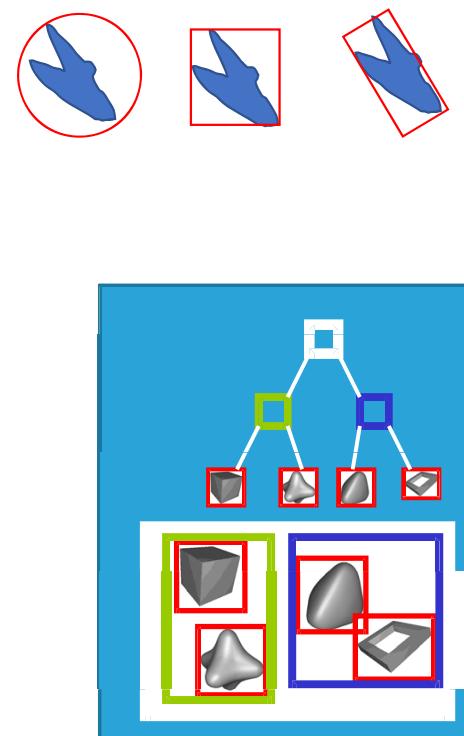
# GPU Hardware

- Shader cores
  - General purpose code
- Fixed function units
  - Rasterizer
  - Texture sampler
  - Tensor core (matrix-matrix multiply)
  - ...
- RT cores (**new**)
  - Intersection ray/box
  - Intersection ray/triangle



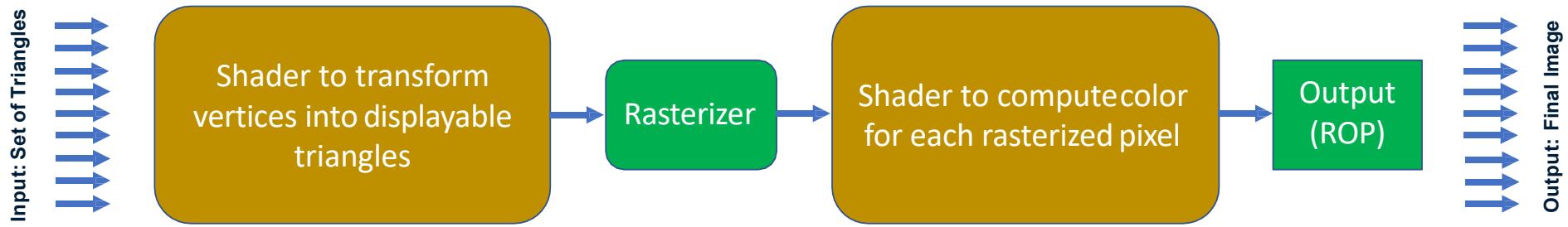
# Bounding Volume Hierarchy

- 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

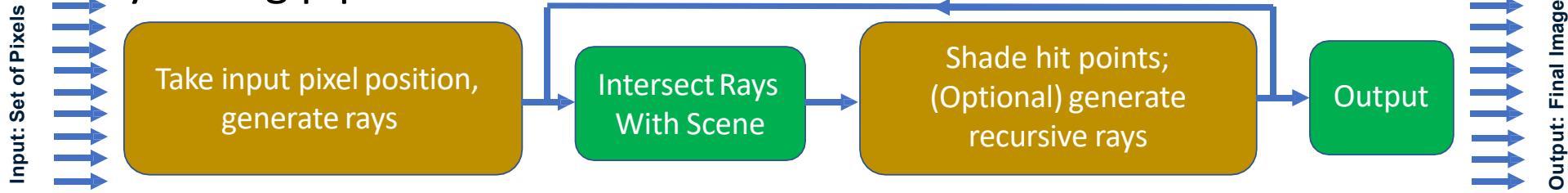


# Rasterization vs Raytracing Pipeline

## Rasterization pipeline

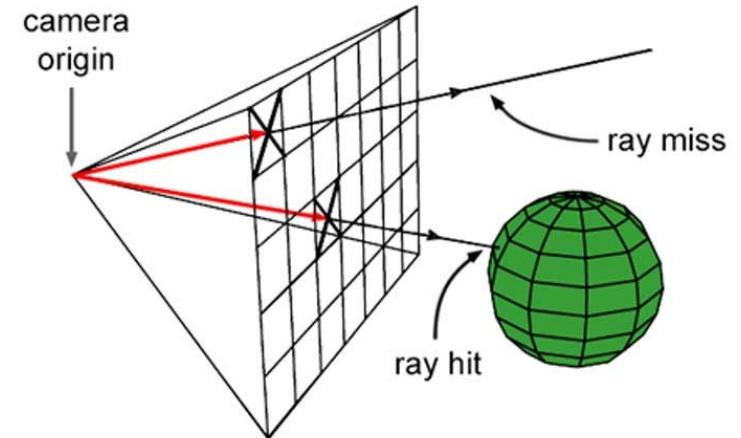
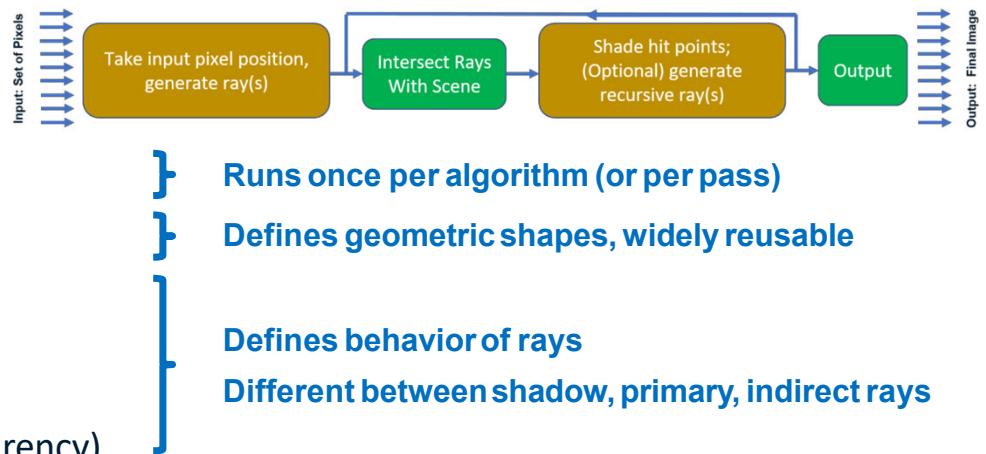


## Raytracing pipeline



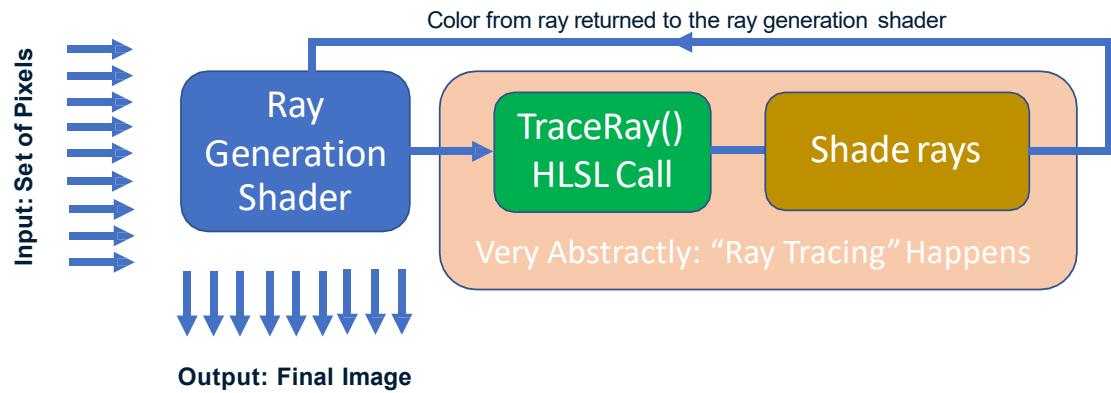
# DirectX Ray Tracing Pipeline

- 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)
  - **Any-hit shader** run once per hit (e.g., to determine transparency)



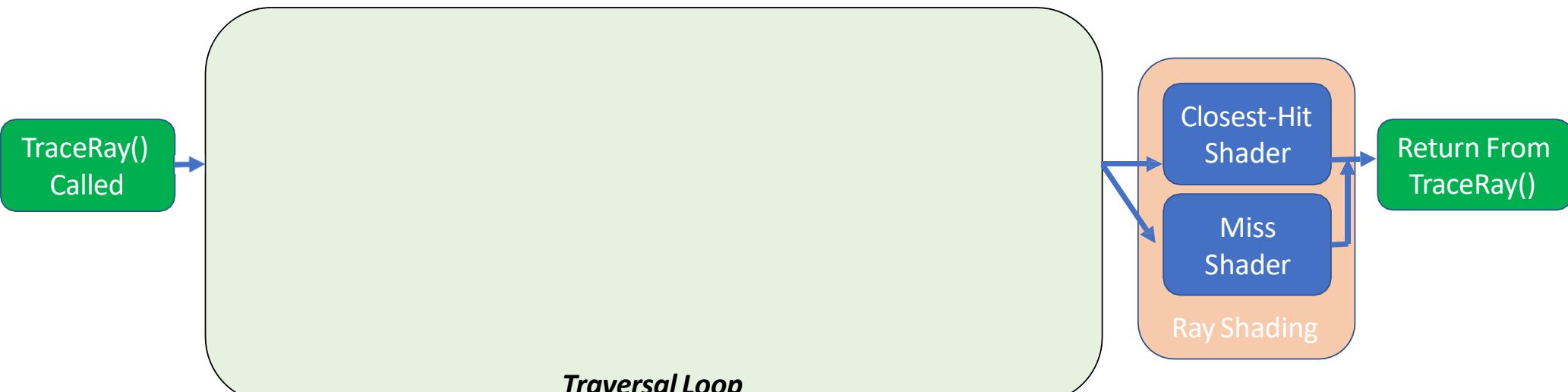
# Ray Generation Shader

- Specify what rays to trace for each pixel
  - Launch rays by calling new HLSL TraceRay() intrinsic
  - Accumulate ray color into image after ray tracing finishes



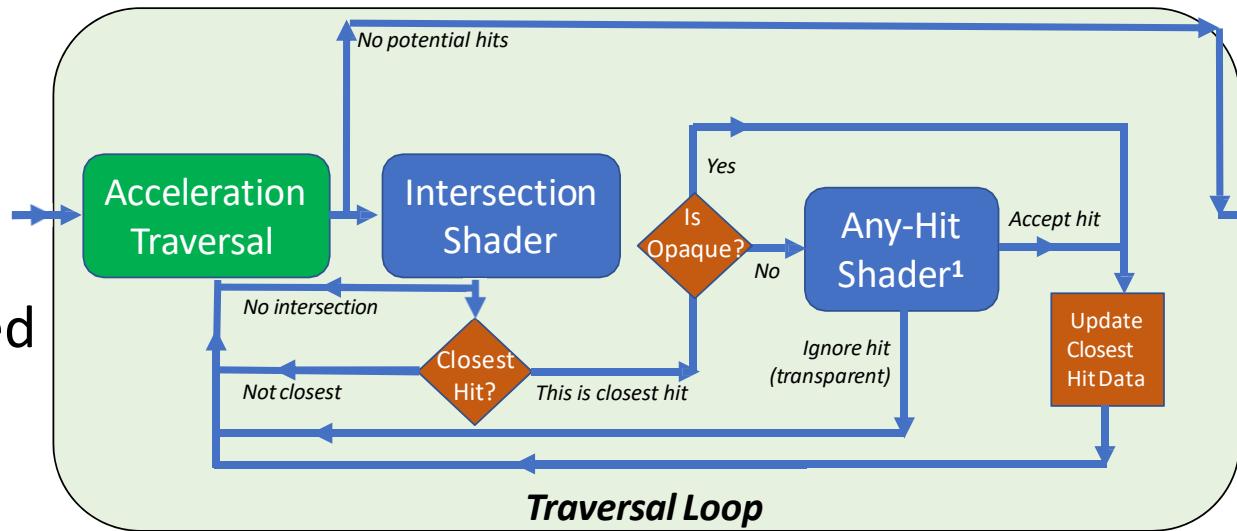
# Tracing A Ray

- 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



# 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



# Starting a DXR Shader

- As any program, need an entry point where execution starts
  - Think `main()` in C/C++
- Shader entry points can be arbitrarily named
- Type specified by HLSL attribute: `[shader("shader-type")]`
  - Remember the **ray generation shader** is where ray tracing starts
- Starting other shader types look like this:
  - **RayPayload** is a user-defined (and arbitrarily named structure)
  - **IntersectAttrs** has data reported on hits (by intersection shader)

```
[shader("raygeneration")]
void PinholeCameraRayGen()
{ ... <Place code here> ... }
```

```
[shader("intersection")]
void PrimitiveIntersection ()
{ ... <Place code here> ... }
```

```
[shader("miss")]
void RayMiss(inout RayPayload data) // User-defined struct
{ ... <Place code here> ... }
```

```
[shader("anyhit")]
void RayAnyHit(inout RayPayload data,
               IntersectAttrs attrs)
{ ... <Place code here> ... }
```

```
[shader("closesthit")]
void RayClosestHit(inout RayPayload data,
                   IntersectAttrs attrs)
{ ... <Place code here> ... }
```

# Ray Payload

- Ray payload is arbitrary user-defined struct
  - Contains intermediate data needed during ray tracing
  - Note: Keep ray payload as small as possible
    - Large payloads will reduce performance
    - Spills registers into memory
- A simple ray might look like this:
  - Sets color to blue when the ray misses
  - Sets color to red when the ray hits an object

```
struct SimpleRayPayload
{
    float3 rayColor;
};

[shader("miss")]
void RayMiss(inout SimpleRayPayload data)
{
    data.rayColor = float3( 0, 0, 1 ); // blue
}

[shader("closesthit")]
void RayClosestHit(inout SimpleRayPayload data,
                    IntersectAttribs attribs)
{
    data.rayColor = float3( 1, 0, 0 ); // red
}
```

# Intersection Attributes

- Communications intersection information needed for shading
  - E.g., how do you look up textures for your primitive?
- Specific to each intersection type
  - One structure for triangles, one for spheres, one for Bezier patches
  - DirectX provides a built-in for the fixed function triangle intersector
  - Could imagine custom intersection attribute structures
- Limited attribute structure size: max 32 bytes

```
struct BuiltinIntersectionAttrs {
    // Barycentric coordinates of hit in
    float2 barycentrics; // the triangle are: (1-x-y, x, y)
}

struct PossibleSphereAttrs {
    // Giving (theta,phi) of the hit on
    float2 thetaPhi; // the sphere (thetaPhi.x, thetaPhi.y)
}

struct PossibleVolumeAttrs {
    // Doing volumetric ray marching? Maybe
    float3 vox; // return voxel coord: (vox.x, vox.y, vox.z)
}
```

# Data Needed for GPU Raytracing

Besides our shader, what data is needed on GPU to shoot rays?

- We need somewhere to write our **output**
- Where are we looking? – **camera data**
- Need to know about our **scene geometry**
- Also need information how to shade scene
  - Depends on material format
  - Depends on shading models

```
RWTexture<float4> outTex;
```

```
// HLSL “constant buffer”, populated by host C++ code
cbuffer RayGenData {
    float3 wsCamPos; // World space camera position
    float3 wsCamU, wsCamV, wsCamW; // right/up/forward
};
```

```
// Our scene’s ray acceleration structure, setup via the C++ DirectX API
RaytracingAccelerationStructure sceneAccelStruct;
```

# Ray Generation Shader Code

```
[shader("raygeneration")]
void PinholeCamera() {
    uint2 curPixel      = DispatchRaysIndex().xy;           What pixel are we currently computing?
    uint2 totalPixels   = DispatchRaysDimensions().xy;        How many rays, in total, are we generating?
    float2 pixelCenter  = (curPixel+float2(0.5,0.5))/totalPixels; Find pixel center in [0..1] x [0..1]
    float2 ndc          = float2(2,-2)*pixelCenter+float2(-1,1); Compute normalized device coordinate (as in raster)
    float3 pixelRayDir  = ndc.x*wsCamU+ndc.y*wsCamV+wsCamZ; Convert NDC into pixel's ray direction (using camera)
    RayDesc ray; Setup our ray
    ray.Origin          = wsCamPos;
    ray.Direction        = normalize( pixelRayDir );
    ray.TMin             = 0;
    ray.Tmax             = 999999999;
    SimpleRayPayload payload = { float3(0, 0, 0) }; Setup our ray's payload
    TraceRay( new intrinsic function in HLSL, can be called from ray generation, miss, and closest-hit
              sceneAccelStruct, Our scene acceleration structure
              RAY_FLAG_NONE, Special traversal behavior for this ray? (Here: no)
              0xFF, mask 0xFF = test all geometry (could ignore some geometry)
              HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER, Which intersection, any-hit, closest-hit, and miss shaders to use?
              ray, What ray are we shooting?
              payload ); What is the ray payload? Stores intermediate, per-ray data
    outTex[curPixel] = float4( payload.color, 1.0f ); Write ray query result into our output texture
}
```

**RayDesc** is a new HLSL built-in type:

```
struct RayDesc {
    float3 Origin; // Where the ray starts
    float TMin; // Min distance for a valid hit
    float3 Direction; // Direction the ray goes
    float TMax; // Max distance for a valid hit
};
```

# Hit and Miss Shader Code

- Returns red if rays hit geometry

```
[shader("closesthit")]
void RayClosestHit(inout SimpleRayPayload data,
                     BuiltinIntersectionAttribs attrs)
{
    data.color = float3( 1, 0, 0 ); // red
}
```

- Returns blue on background

```
[shader("miss")]
void RayMiss(inout SimpleRayPayload data)
{
    data.color = float3( 0, 0, 1 ); // blue
}
```

- This is a complete DirectX Raytracing shader

- Both intersection shader and any-hit shader are optional
- Shoots rays from app-specified camera

# Performance optimization

- Exploit ray coherence
  - Trace rays with similar direction at once
  - Coherent rays performing similar operations and memory access
  - Arbitrary reflection/refraction rays are bad
- Adaptive raytracing
  - Trace only where necessary
  - Shoot rays on demand
  - Filter/denoise the results

# Hybrid Rasterization + Raytracing

- Pure raytracing is usually too expensive
- Game engines use a mixture
  - Main rendering done with rasterization
  - Special effects added with raytracing
- DirectX offers easy interoperability between raster and raytracing
  - Raytracing and rasterization shaders can share code and data types
  - E.g., HLSL material shaders - directly usable for a hybrid raytracing
  - Raytrace primary rays to fill G-buffer in deferred rendering
  - Raytracing of shadows, ambient occlusion, reflections

# Raytraced Shadows

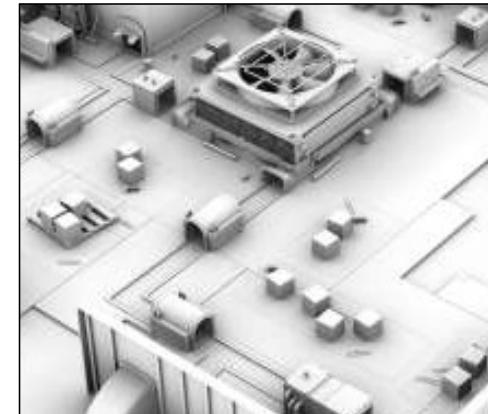
- Launch ray towards the light
  - If ray misses → not in shadow
  - Skip closest hit shader
  - Use miss shader
- Soft shadows
  - Choose random direction from cone
  - Cone width defines penumbra
  - Sample 1 ray per frame
  - Accumulate and filter over time (like TAA)



Umgebungsverdeckung

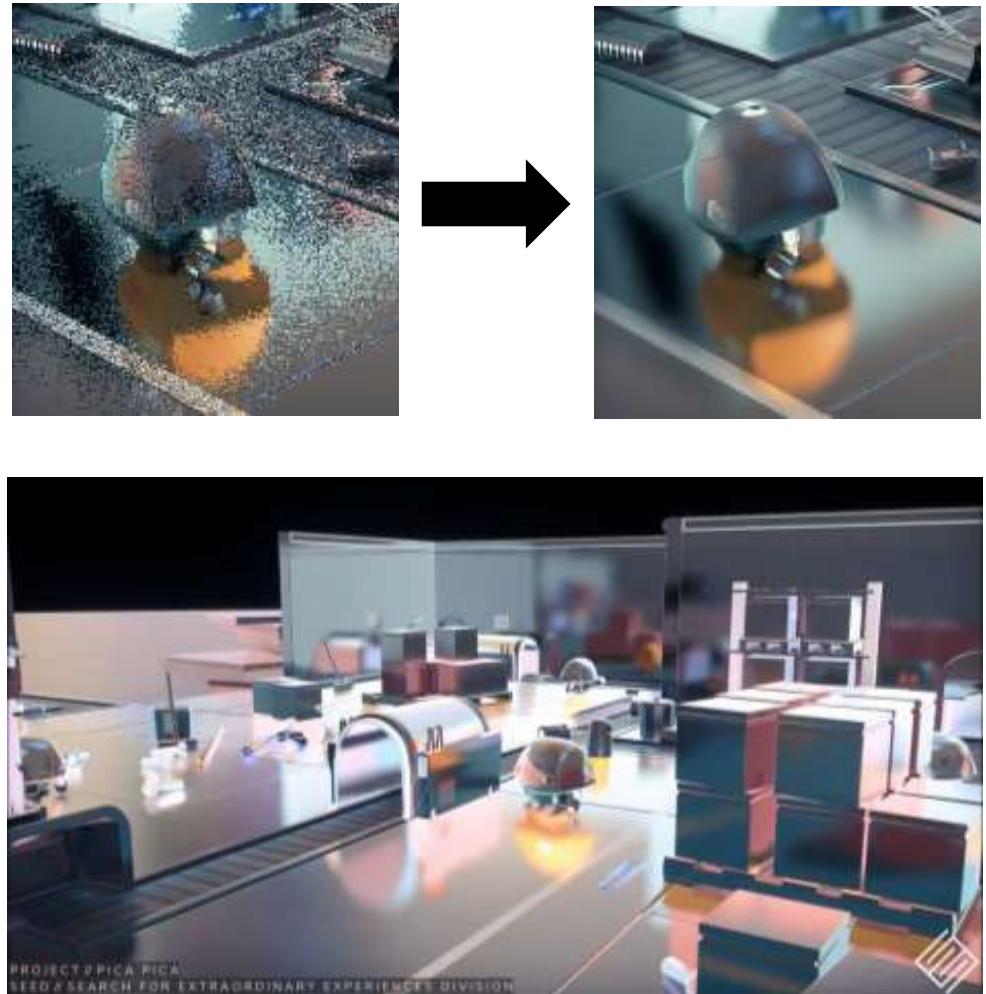
# Raytraced Ambient Occlusion

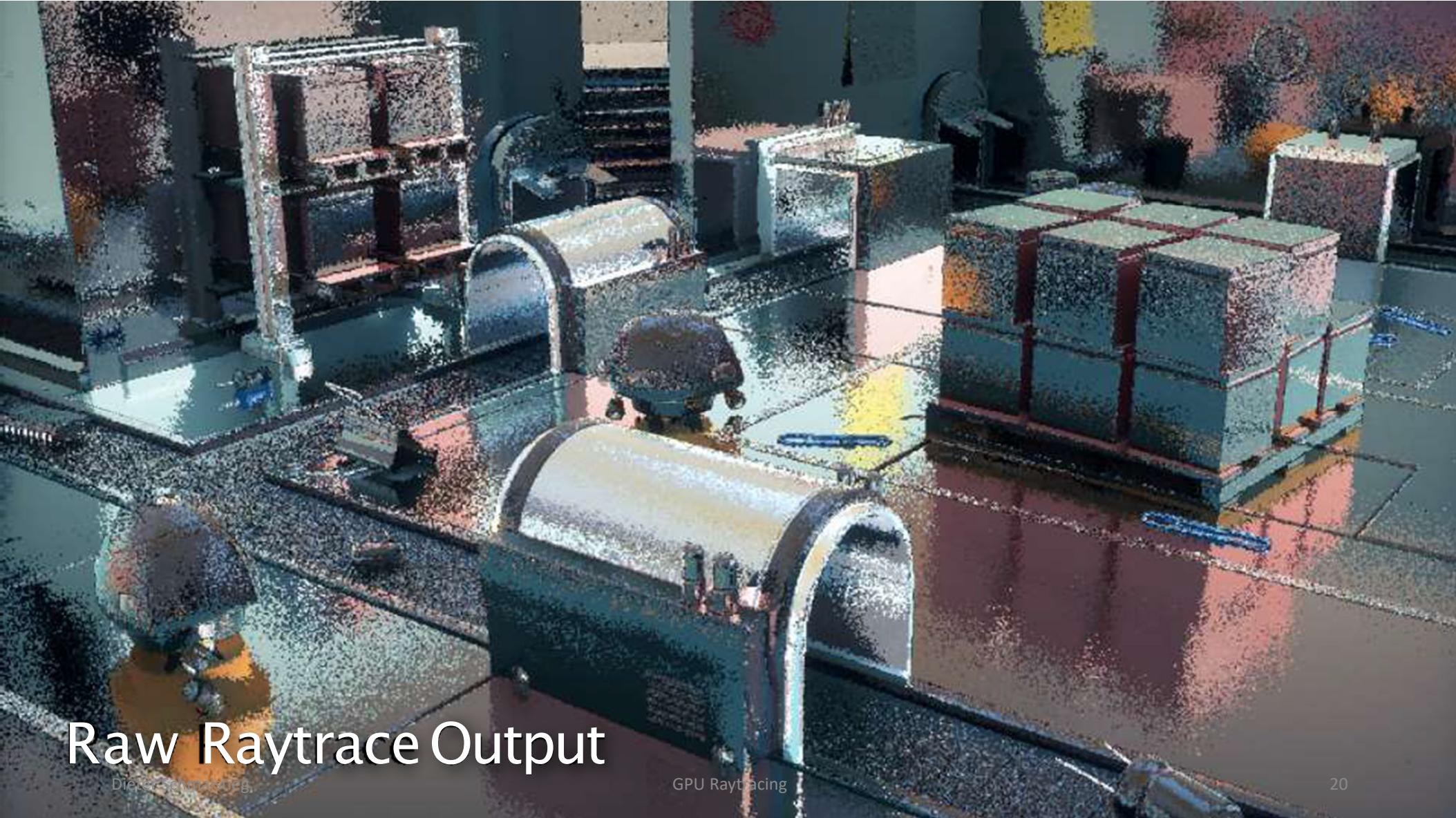
- Randomly chose directions over hemisphere at a surface point
- Like shadows, miss shader tells us if a point is occluded over not
- Compute average



# Raytraced Reflections

- Launch rays from G-Buffer
  - Trace at 50% resolution
  - 25% ray/pixel for reflection
  - 25% ray/pixel for reflected shadow
- Reconstruct at full resolution
- Combine reflection methods
  - Screen-space reflections (SSR)
  - + Raytraced reflections (at SSR gaps)
  - + Environment map (at remaining gaps)
- Both spatial + temporal filtering





# Raw Raytrace Output

Dieser Schauspiel

GPU Raytracing

20



# Spatial Reconstruction

Dieter Schmalstieg

GPU Raytracing

21



# +Temporal Accumulation

Dieter Schmalstieg

GPU Raytracing

22



+Bilateral cleanup

Dieter Schmalstieg



GPU Raytracing

23



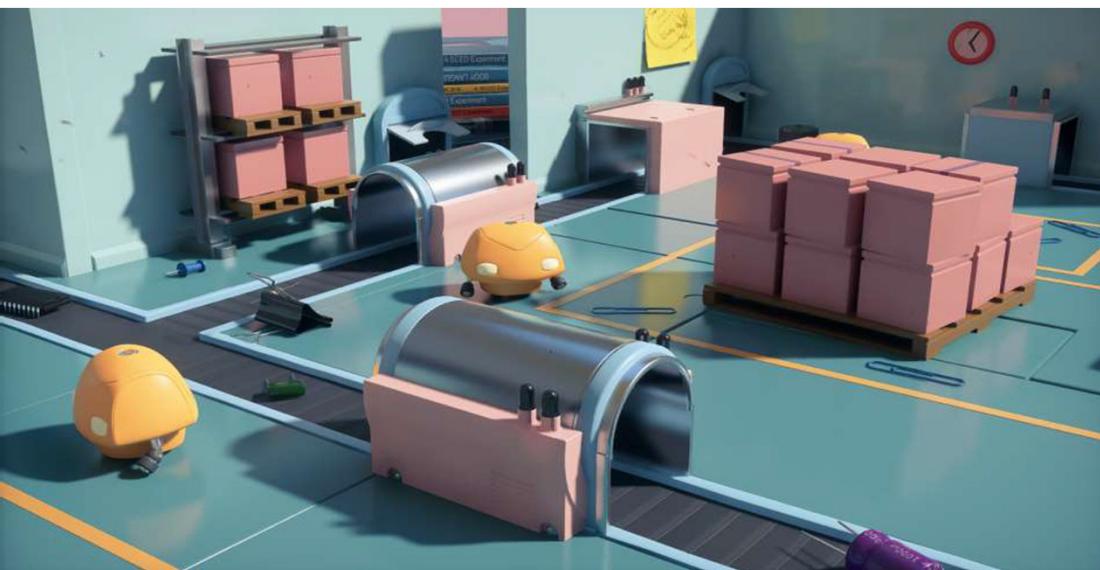
# +Temporal Anti-Aliasing

Dieter Schmalstieg

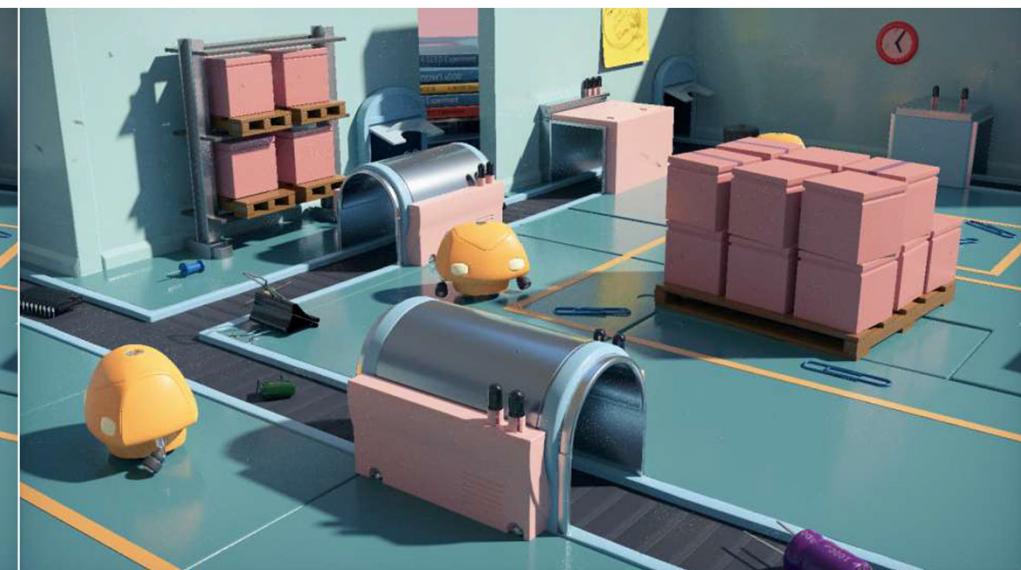
GPU Raytracing

24

# Comparison



Real-time raytracing



Pathtracing, ~15 seconds accumulation

Image courtesy of Epic Games

A wide-angle photograph of a majestic, snow-covered mountain range under a bright, slightly hazy sky. The mountains are rugged with sharp peaks and deep, shadowed valleys. Small, glowing particles, resembling falling snow or distant stars, are scattered across the scene, particularly visible against the light sky.

Dieter Schmalstieg

Levels of Detail

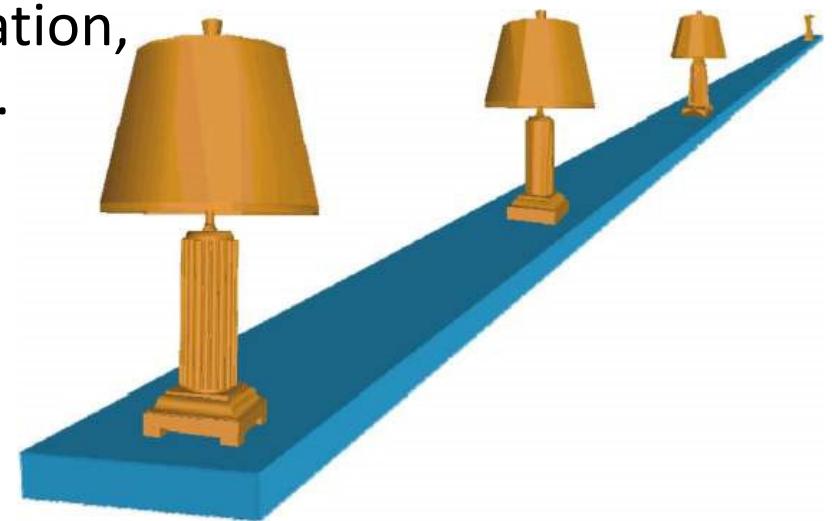
Based on material from Michael Wimmer and Markus Grabner

# Case Study: Unreal Engine 5

- Siggraph Tutorial <https://www.youtube.com/watch?v=TMorJX3Nj6U>
- Reveal Trailer [https://www.youtube.com/watch?v=qC5KtatMcUw&ab\\_channel=UnrealEngine](https://www.youtube.com/watch?v=qC5KtatMcUw&ab_channel=UnrealEngine)
- Leading game engine made by Epic
  - Open source :-)
  - High code complexity :-)
- Supports for huge models → our topics today
  - Level of detail
  - Visibility culling
  - Virtual textures and geometry

# LOD - Basic Idea

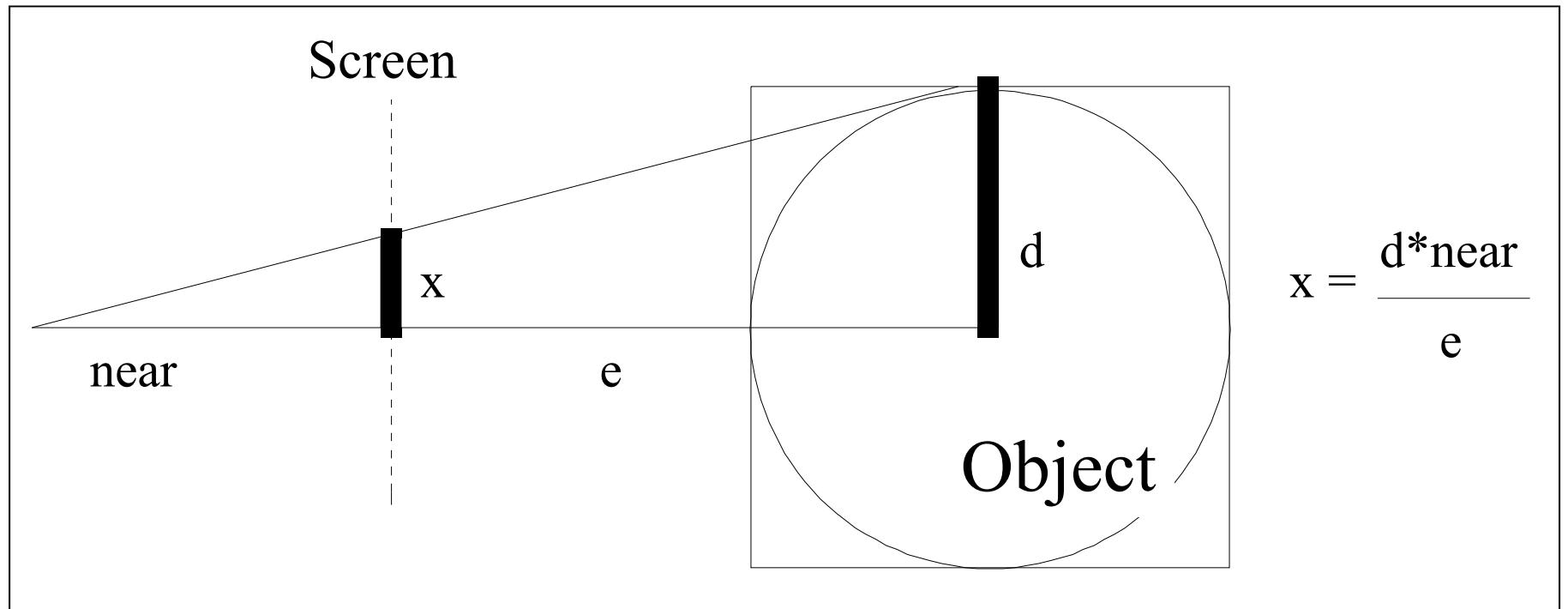
- Problem: even after visibility, model may contain too many polygons
- Idea: Simplify the amount of detail used to render small or distant objects
- Known as levels of detail (LOD)  
A.k.a. multiresolution modeling,  
polygonal/geometric simplification,  
mesh reduction/decimation, ...



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

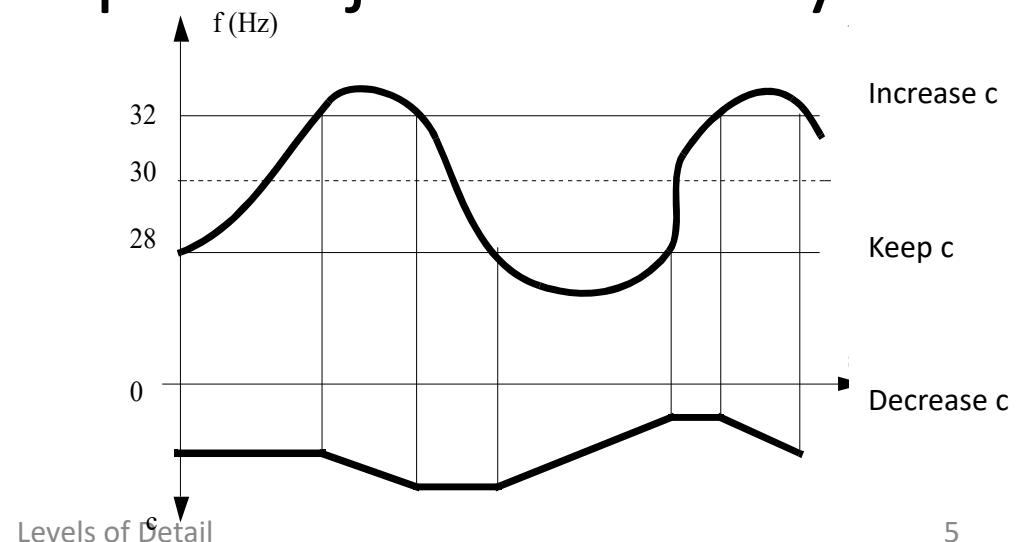
# Static LOD Selection

- 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]

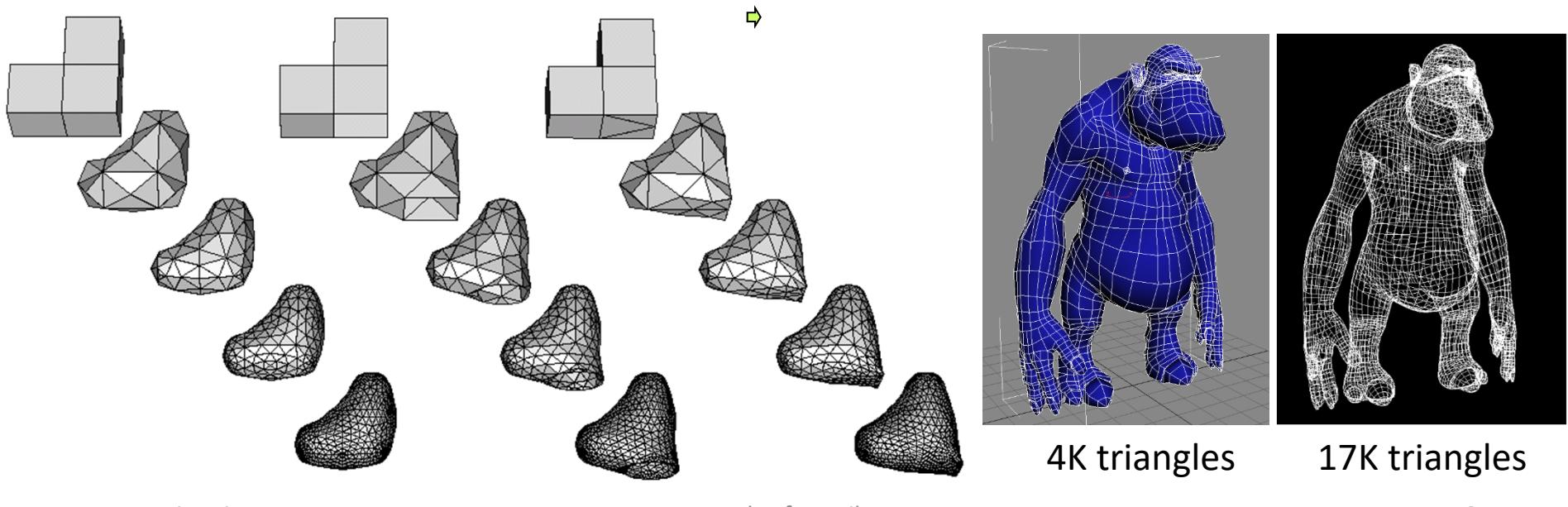
- COST = time for drawing object with a given LOD
- Goal: best possible image quality
- BENEFIT = contribution of object to image quality
  - Most important: screen-size of object
- Optimization (*Rucksack*) problem
  - Sum(BENEFITS) → max, BUT
  - Sum(COSTS) ≤ FRAMETIME

# 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](https://www.youtube.com/watch?v=I20Zyr4U_Xk)
  - Interpolate triangle shapes from 1<sup>st</sup> to 2<sup>nd</sup> LOD
  - + Best quality
  - Requires geometric correspondence between LODs

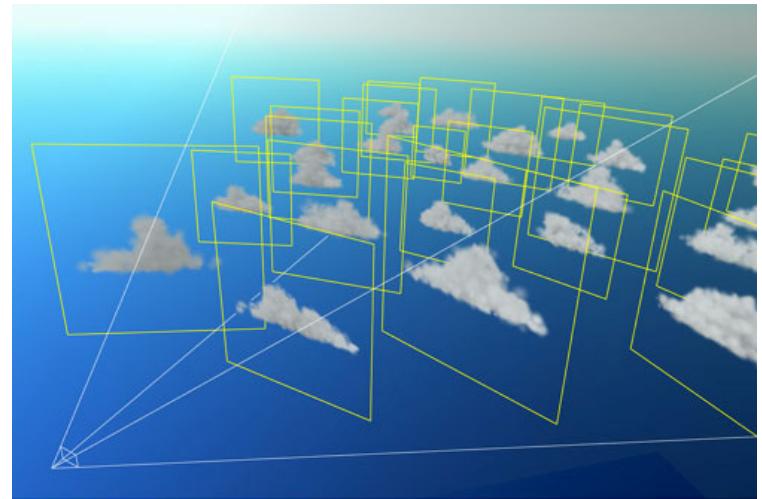
# LOD by Subdivision Surfaces

- Curved surface defined by repeated subdivision steps on a polygonal model
- Subdivision rules create new vertices, edges, faces based on neighboring features
- Compute in geometry shader



# LOD by Shading and Rendering

- Shading and illumination
  - From simple local to complex global illumination
- Geometry vs. images
  - Textures, impostors

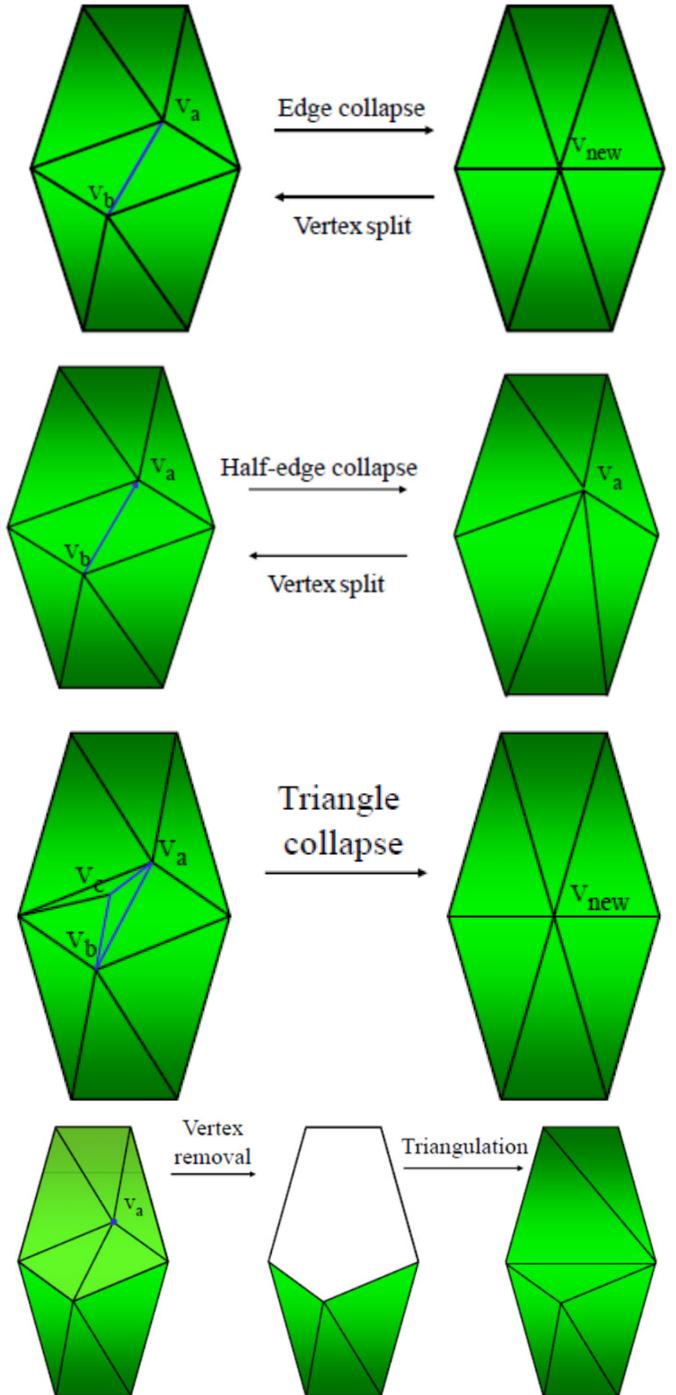


# LOD by Geometric Simplification

- Iteratively reduce number of primitives
  - Vertices, edges, triangles
- Topology simplification
  - Reducing number of holes, tunnels, cavities
- Does not change rasterization
  - Fragment shader load remains roughly identical

# Local Simplification

- Edge collapse
- Vertex-pair collapse
- Triangle collapse
- Cell collapse
- Vertex removal



# Lazy Greedy Local Simplification

- Fewer cost evaluations

compute costs for each possible operation

insert them into queue

set „dirty“ flags to false

while the queue is not empty

    extract head of queue (i.e., element with smallest error)

    if head is dirty

        re-compute cost

        set „dirty“ flag to false

        re-insert into queue

    else

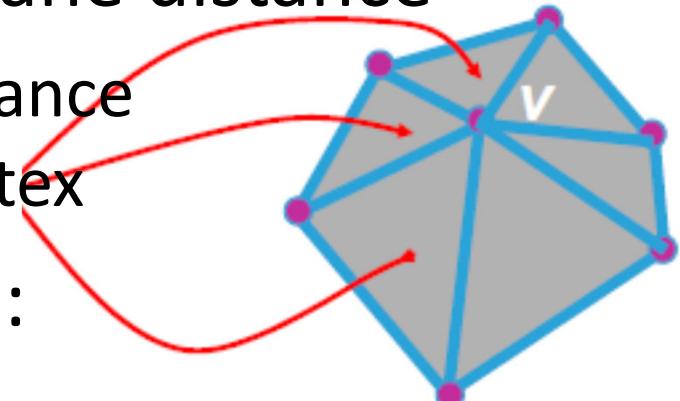
        perform operation

        for each neighbor

            set „dirty“ flag to true

# Quadric Error Metric

- Error measure = vertex-to-plane distance
    - Minimize sum of squared distance to all planes attached at a vertex
  - Plane equation for each face:
- $$p: Ax + By + Cz + D = 0$$
- Distance to vertex  $v$  for a single plane:
- $$\Delta v = p^T \bullet v = [A \ B \ C \ D] \bullet [x \ y \ z \ 1]^T$$



Quadric Error Metrics (QEM) ist ein Verfahren zur Optimierung der Polygonnetzreduktion in der Computergrafik. Es nutzt eine Metrik, die die Abweichung zwischen einem ursprünglichen 3D-Modell und einer reduzierten Version misst, um die Polygone, die am wenigsten von der Reduktion betroffen sind, beizubehalten und diejenigen zu entfernen, die die größte Abweichung aufweisen. QEM betrachtet jeden Eckpunkt des Polygonnetzes als eine Quadrik (eine Art von dreidimensionalem Parabel) und misst die Abweichung, die durch die Verringerung der Anzahl der Eckpunkte entsteht. Auf diese Weise kann QEM die Qualität des reduzierten Modells maximieren, indem es sicherstellt, dass die wichtigsten geometrischen Eigenschaften erhalten bleiben.

# Using the Quadric Error Metric

$$\begin{aligned}
 \Delta(v) &= \sum_{p \in planes(v)} (p^T v)^2 \\
 &= \sum_{p \in planes(v)} (v^T p)(p^T v) \\
 &= \sum_{p \in planes(v)} v^T (pp^T) v \\
 &= v^T \left( \sum_{p \in planes(v)} pp^T \right) v \\
 \Delta(v) &= v^T Q v
 \end{aligned}$$

- $pp^T$  = plane equation squared:

$$pp^T = \begin{bmatrix} A^2 & AB & AC & AD \\ AB & B^2 & BC & BD \\ AC & BC & C^2 & CD \\ AD & BD & CD & D^2 \end{bmatrix}$$

- $\sum pp^T = Q$  is also a matrix
- Sandwich product  $\Delta v$  computes distance metric  $v \rightarrow$  all planes
- For edge collapse of vertices  $v_1 + v_2$ , just add  $Q_1 + Q_2$



**Visibility**

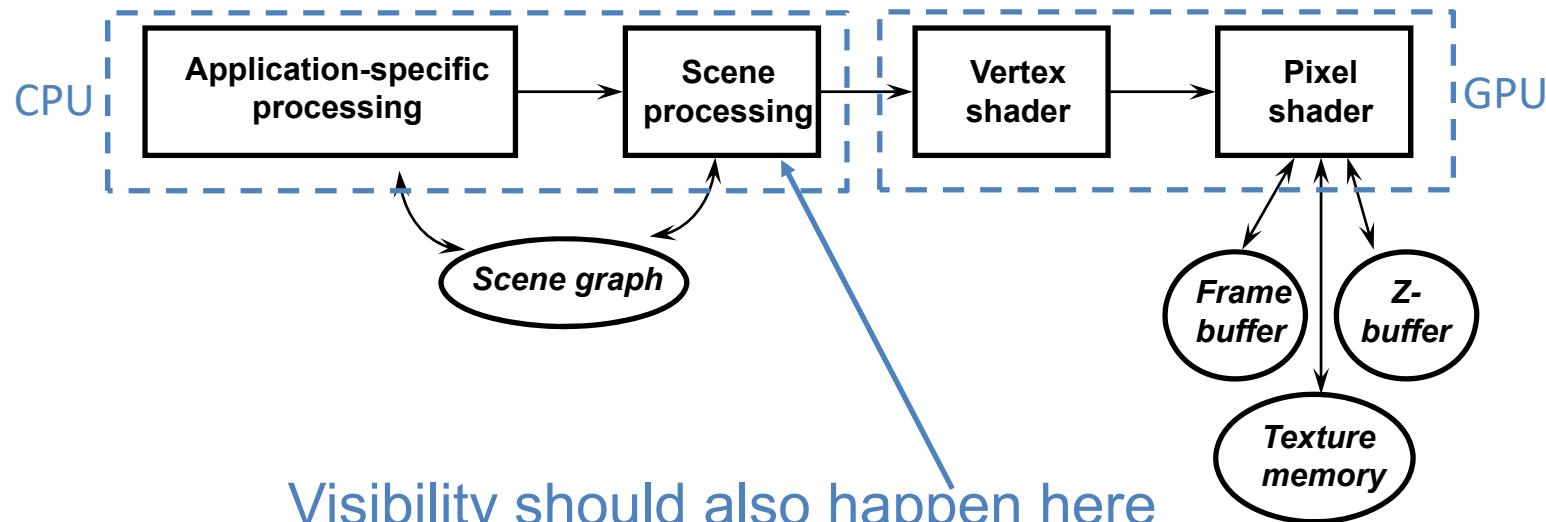
# Recap: Depth Buffering

- Hardware to determine per-pixel visibility
  - 2D buffer for storing z-values per pixel
  - Pixel shader result only stored if z-value test passed
  - Allows drawing of unsorted geometry
- Sorting still greatly improves performance
  - *Early-z testing*: run depth test *before* pixel shader
  - Drawing objects close to camera early increases chances that a later pixel shader is not called

# Why is the Z-Buffer Not Enough?

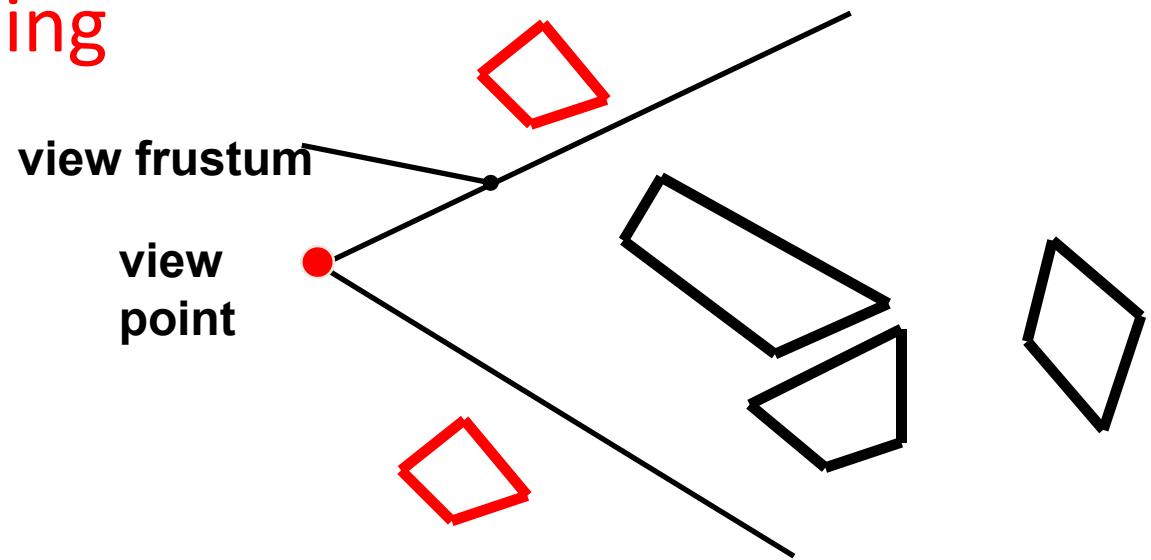
(Even with depth sorting:) Z-buffer...

- ...Does not eliminate depth-complexity (overdraw)
- ...Does not eliminate vertex processing of occluded polygons

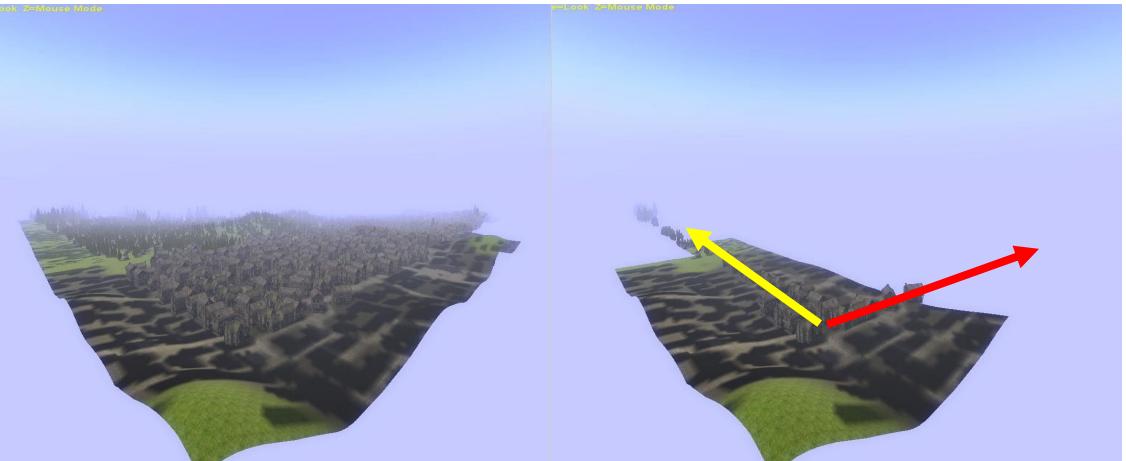


# View Frustum Culling

- View-frustum culling
- Occlusion culling
- Backface culling



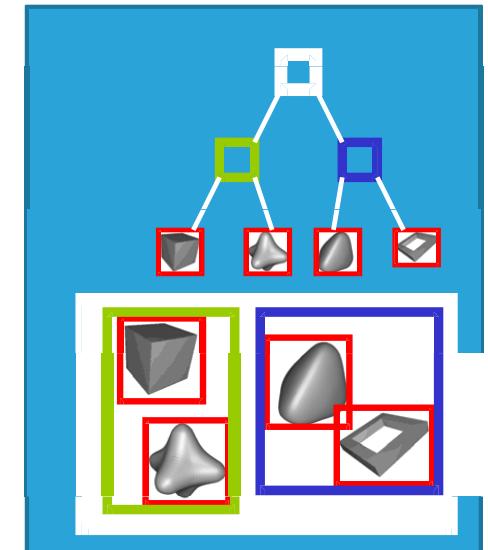
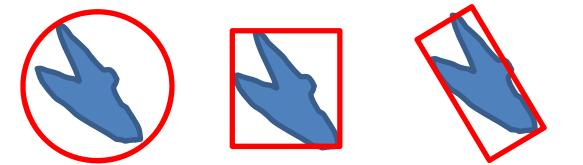
Dieter Schmalstieg



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



# Frustum Culling with BVH

Function  $Cull(node)$

{

  if not ( $intersect(node, frustum) = \text{EMPTY}$ )

    if  $node = \text{LEAF}$  then  $draw(node)$

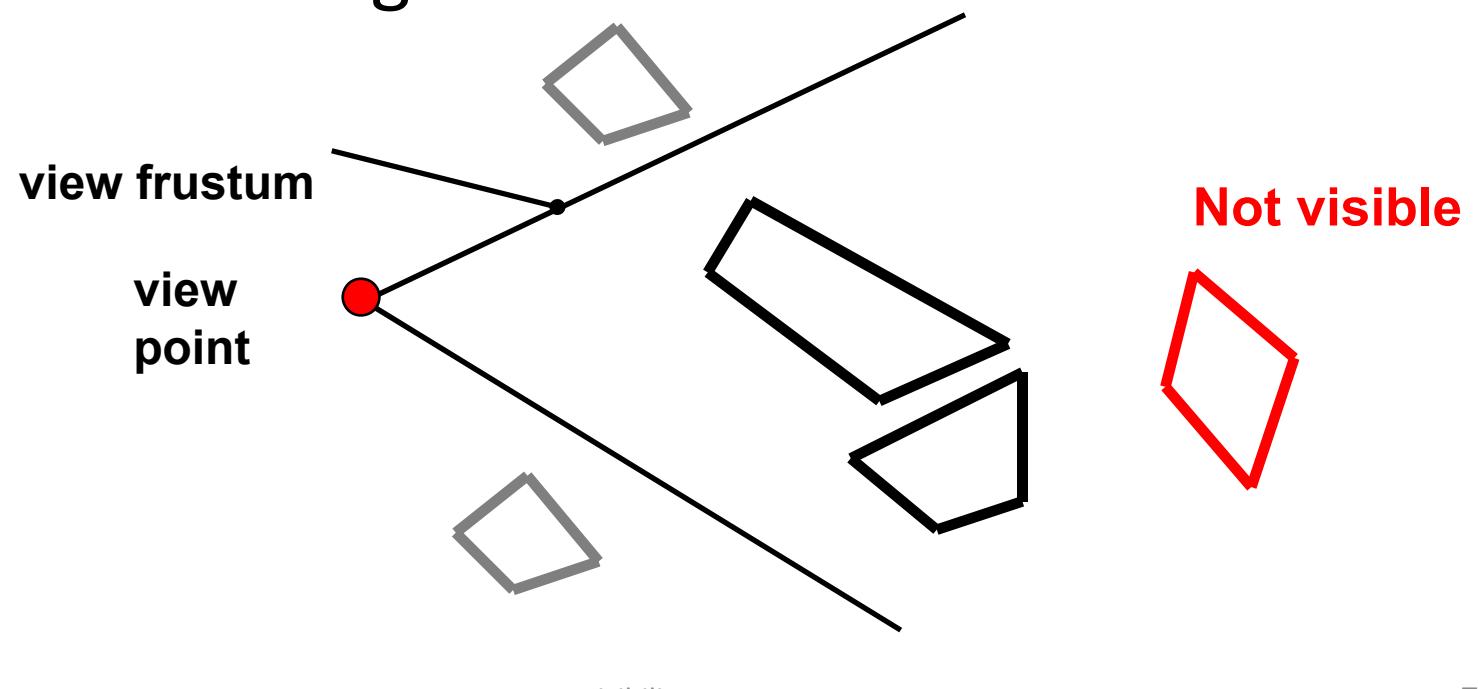
    else for all children  $C$  of  $node$

$Cull(C)$

}

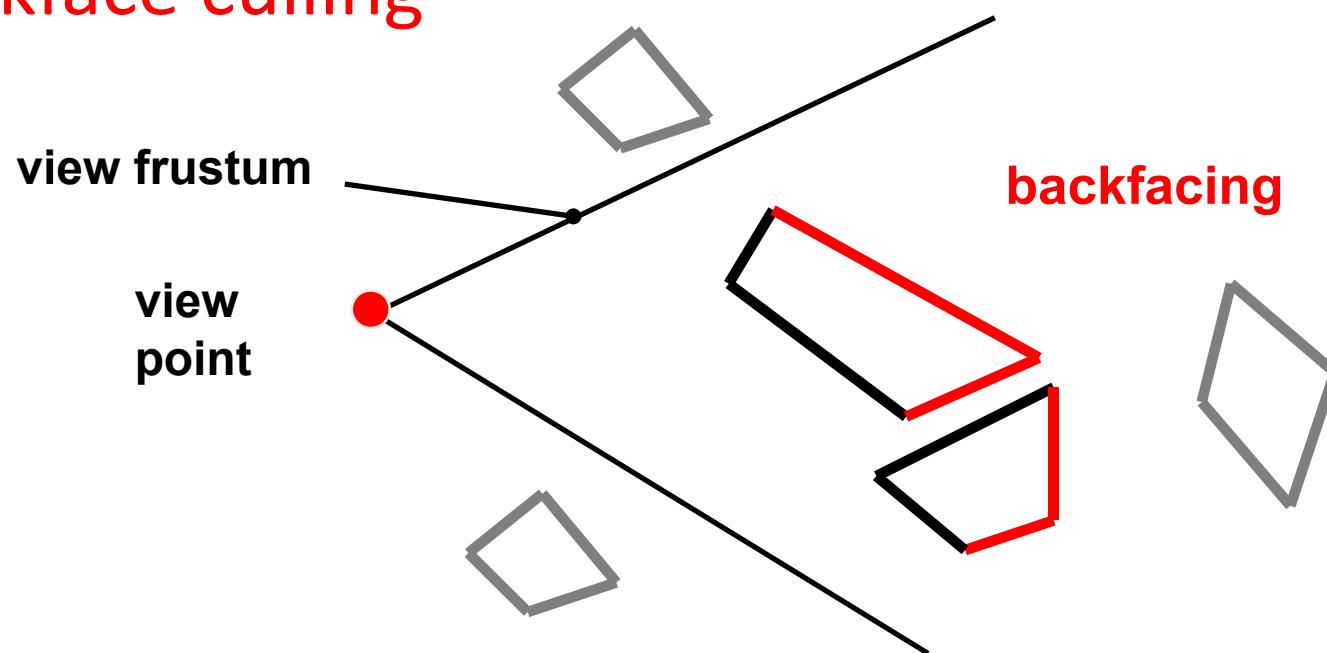
# Occlusion Culling

- View-frustum culling
- Occlusion culling
- Backface culling



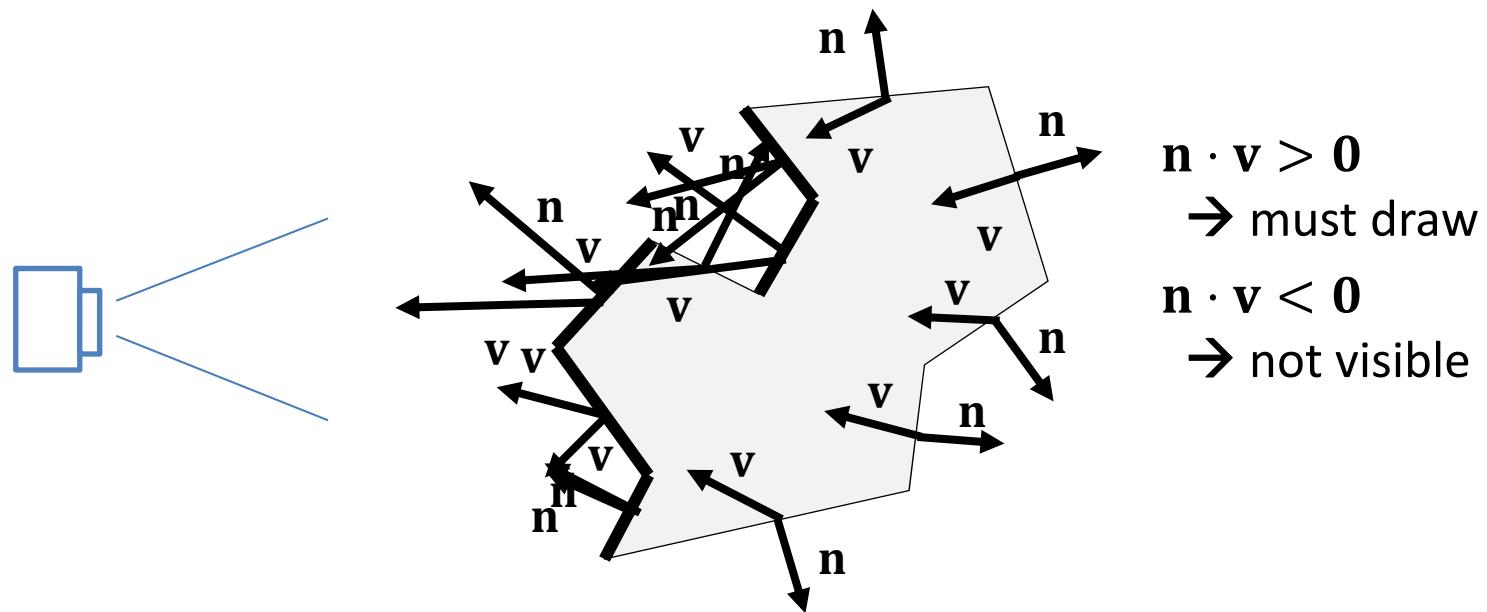
# Backface Culling

- View-frustum culling
- Occlusion culling
- Backface culling



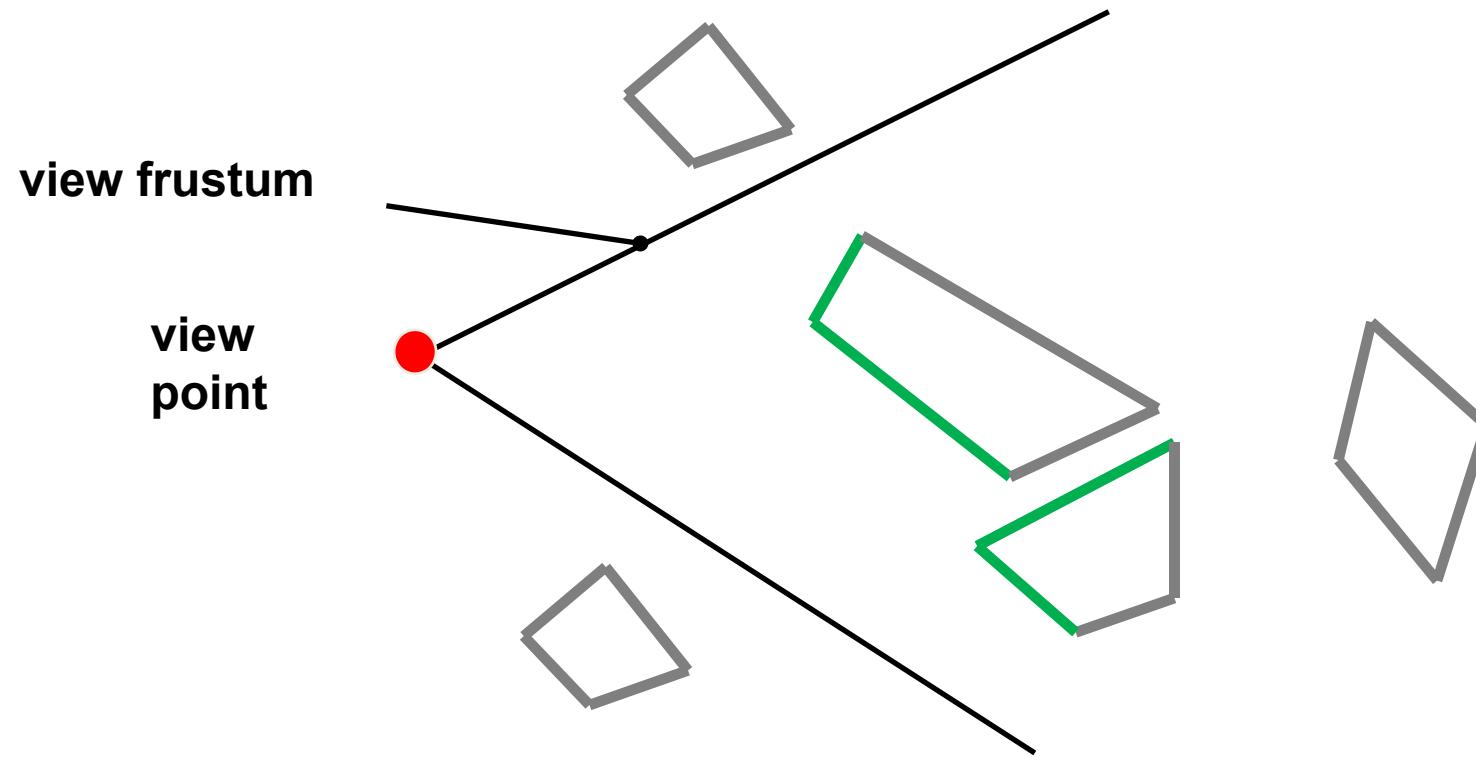
# Backface Culling

- 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  $\mathbf{v}$



# Visibility Culling Result

Result of frustum + occlusion + backface culling



# Exactly Visible Set

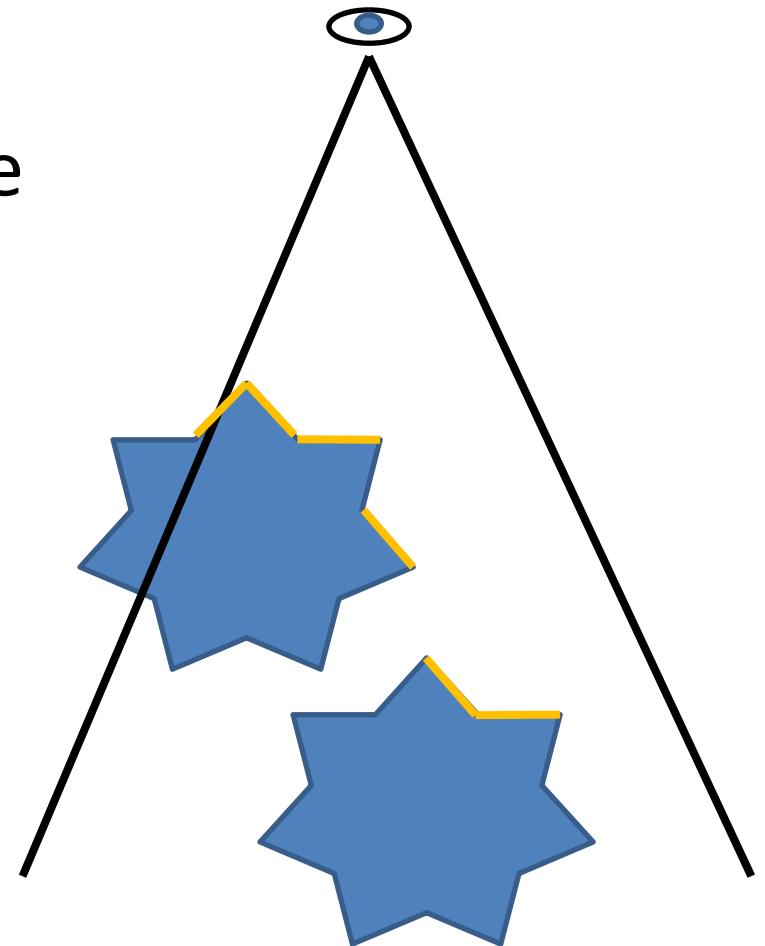
- All primitives that are visible
- No more, no less



(a)



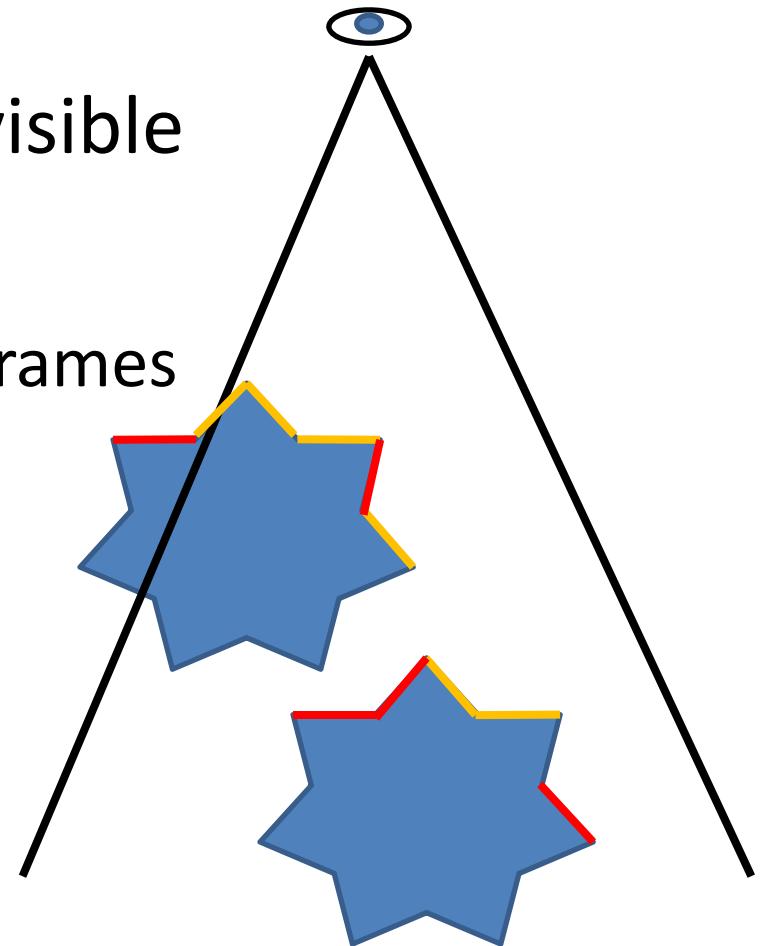
(b)



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

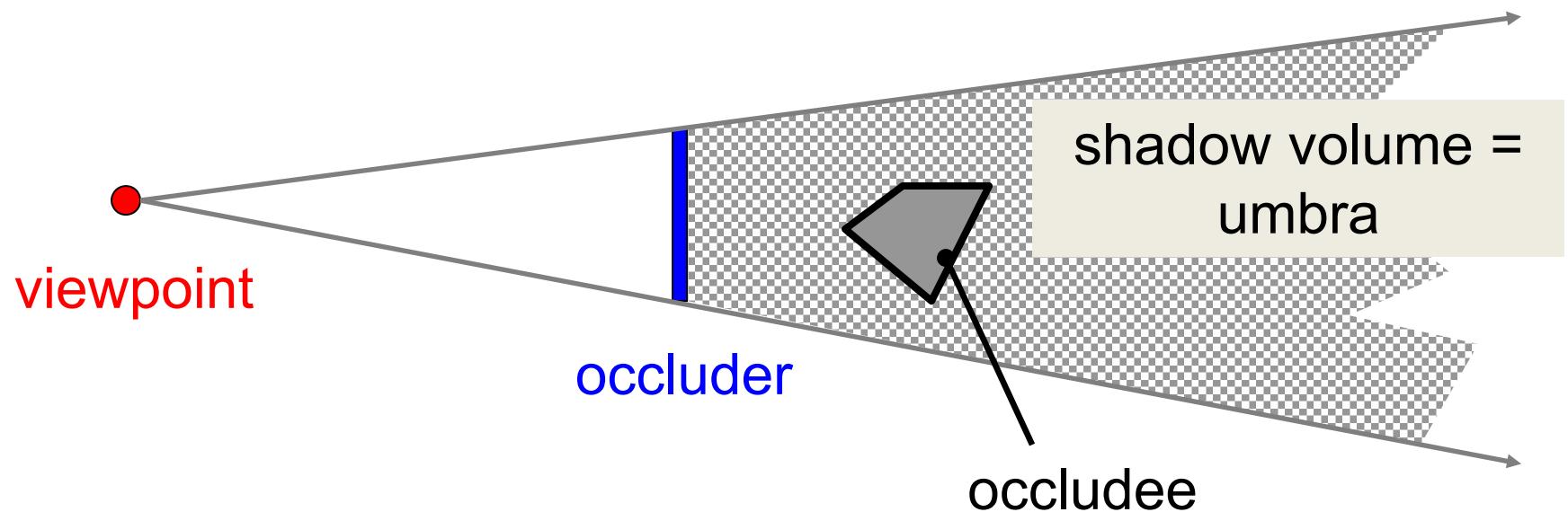
- The exact visible set (EVS) is unknown:  $O(N^9)\dots$
- Potentially visible set (PVS) =  
set of objects that *could* be visible
- PVS can be
  - Aggressive,  $PVS \subseteq EVS$
  - Conservative,  $PVS \supseteq EVS$  (preferred)
  - Approximate,  $PVS \sim EVS$
- PVS can be precomputed
  - But we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

# Naïve Occlusion Culling

- Select some promising occluders
  - Large objects
  - Close to camera → large in screen space
- Test all other objects against occluders
  - Remove the objects that are occluded
  - Maybe speed up by testing bbox instead of object
- Why does it not work well for real-life scenes?

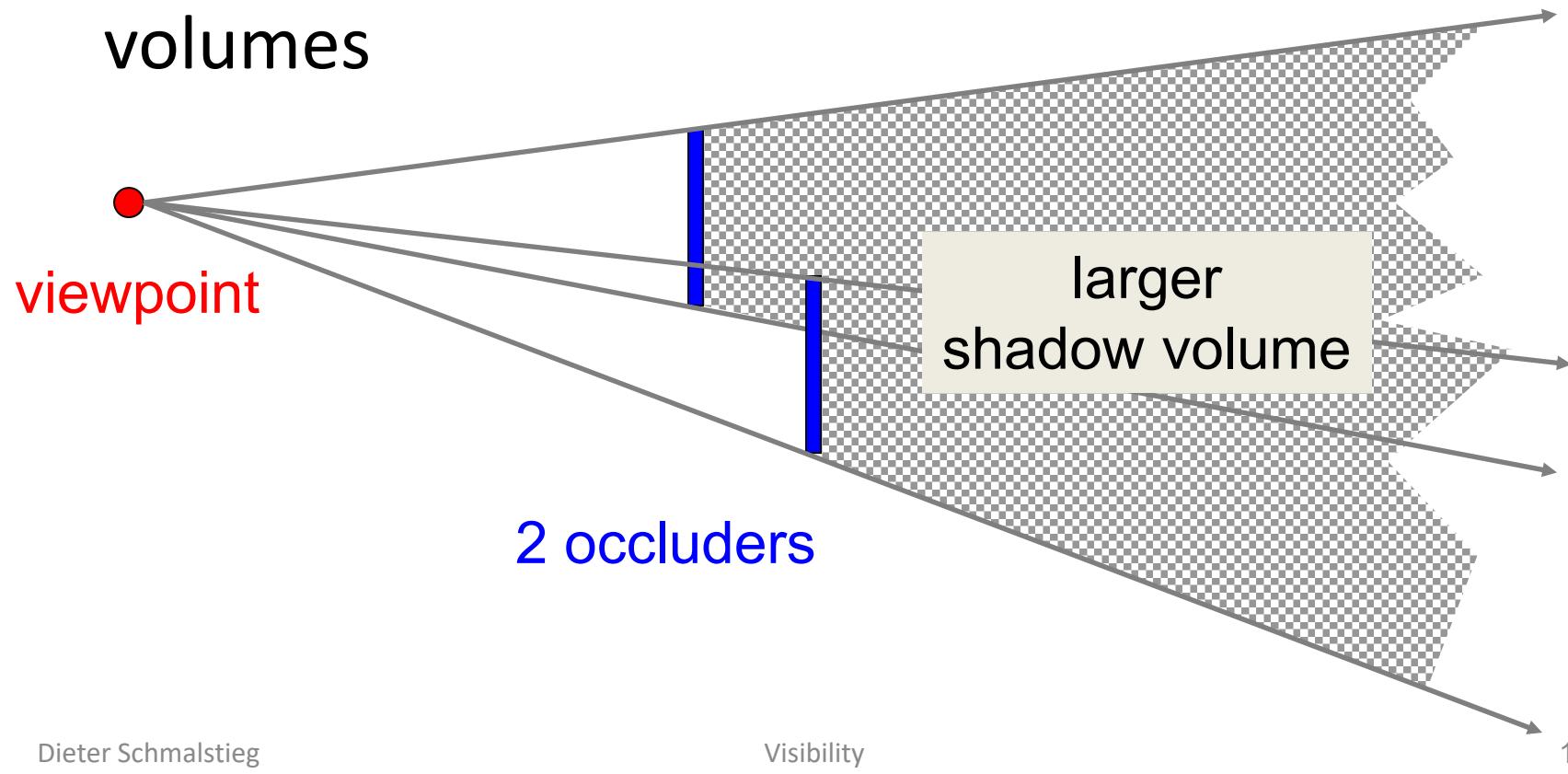
# Visibility from a Point

- Terms: occluder, occludee, shadow volume, umbra



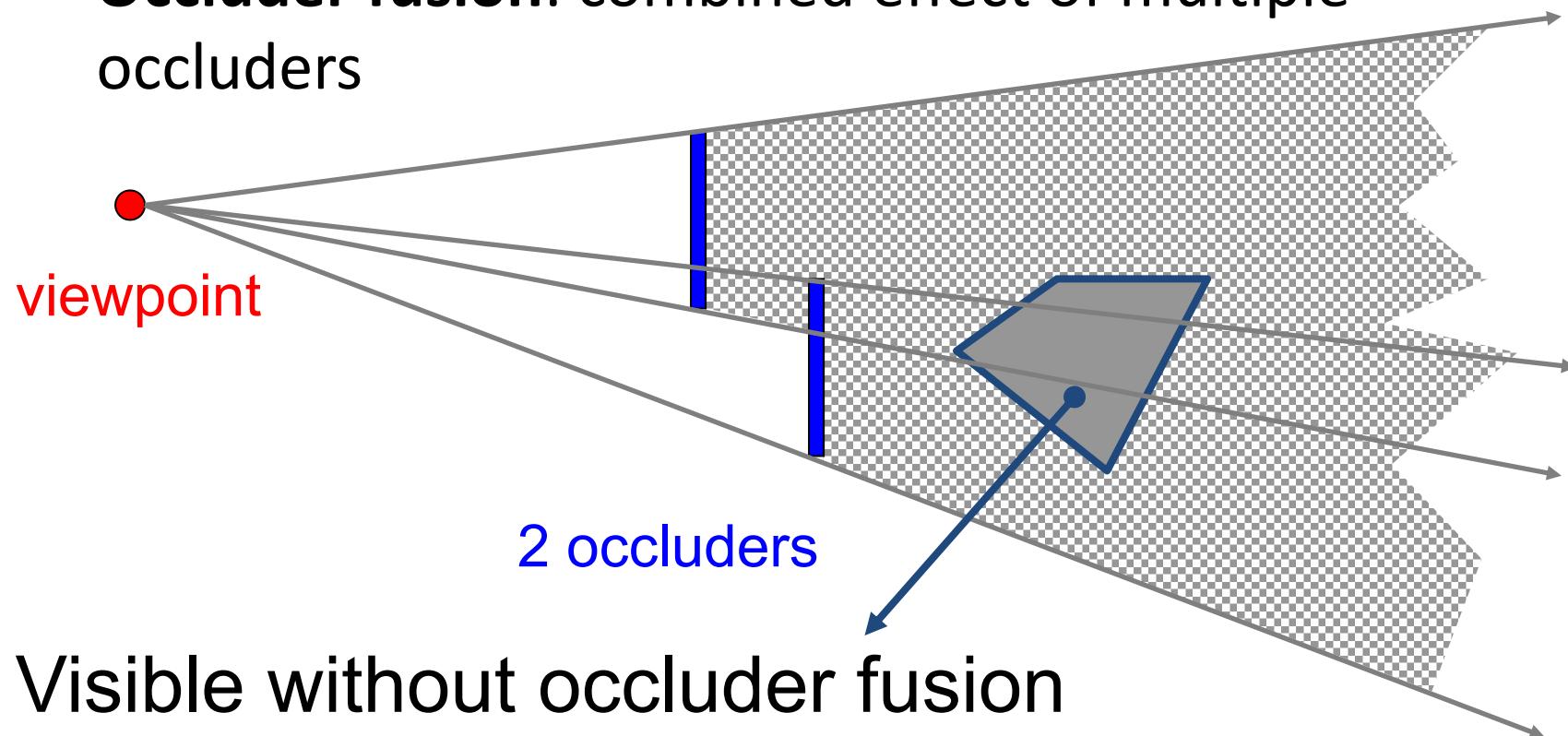
# Visibility from a Point

- Complete shadow volume for occluders  $\text{occ}_1, \dots, \text{occ}_n = \mathbf{union}$  of all individual shadow volumes



# Occluder Fusion

- **Occluder fusion:** combined effect of multiple occluders



# Occlusion Culling in Practice

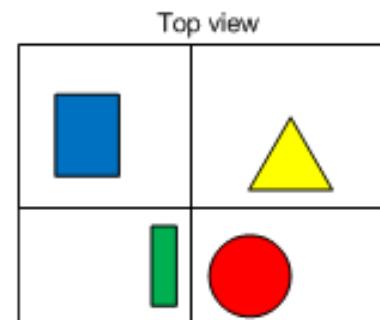
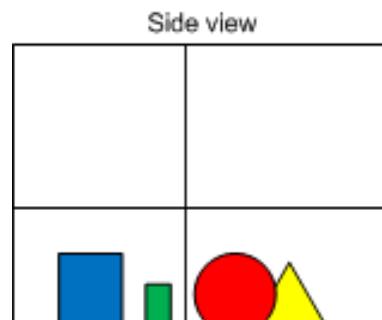
- Use two spatial data structures
  1. **Scene data structure (SDS)**
    - Stores the objects in the scene
  2. **Shadow volume data structure (SVDS)**
    - Generated by occluders (selected from scene), or
    - Generated by virtual occluders (synthesized)
- Cull SDS using SVDS

# Simple Algorithm for Point Visibility

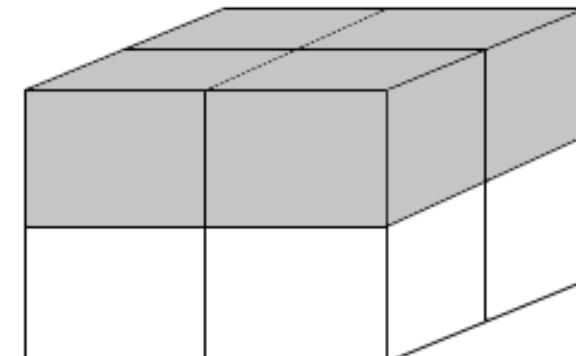
- Shadow volume data structure ( $SVDS$ ) = empty
- For each occluder  $occ_i$ 
  - Calculate shadow volume  $SV_i$
  - Add  $SV_i$  to  $SVDS$
- For every object  $o_j$ 
  - Test  $o_j$  against the  $SVDS$
  - Cull  $o_j$  if occluded

# Spatial Data Structures

- For quickly finding/culling large portions of scene with a single test
- All kinds of (hierarchical) data structures used
- E.g., bounding boxes, bounding volume hierarchy, grids, quadtree, octree, k-d tree, BSP tree



Simple 3D grid

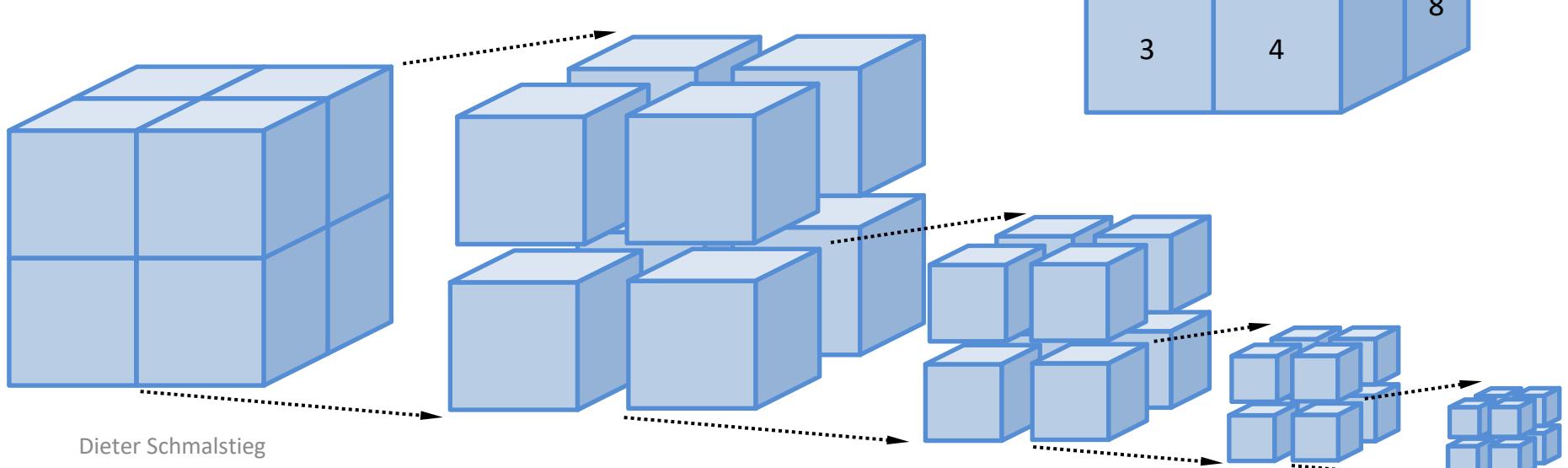
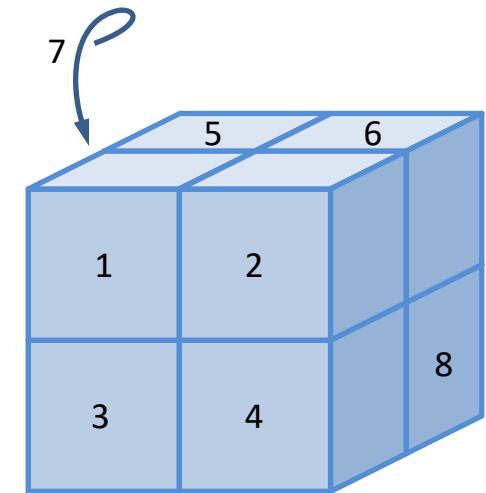


Visibility

# Example: Octree

- 3D equivalent of quadtree
- Hierarchical subdivision of a cube into 8 octants

Region in 3D space



# Simple Algorithm, Revisited

- Shadow volume data structure ( $SVDS$ ) = empty
- For each occluder  $occ_i$ 
  - Calculate shadow volume  $SV_i$
  - Add  $SV_i$  to  $SVDS$
- For every object  $o_j$ 
  - Test  $o_j$  against the  $SVDS$
  - Cull  $o_j$  if occluded

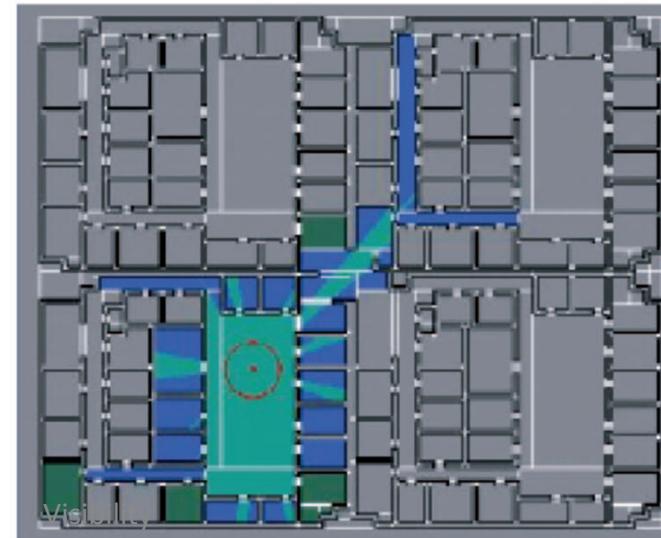
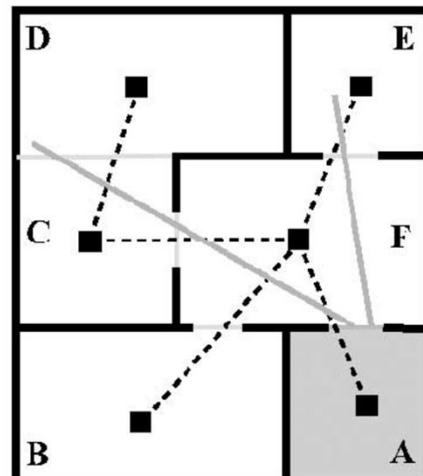
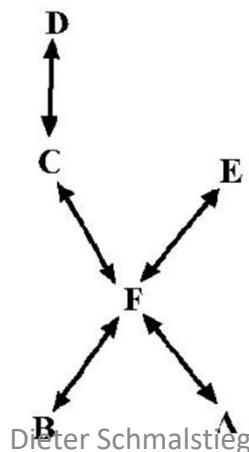
What shall we use for  $SVDS$ ?

# 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*

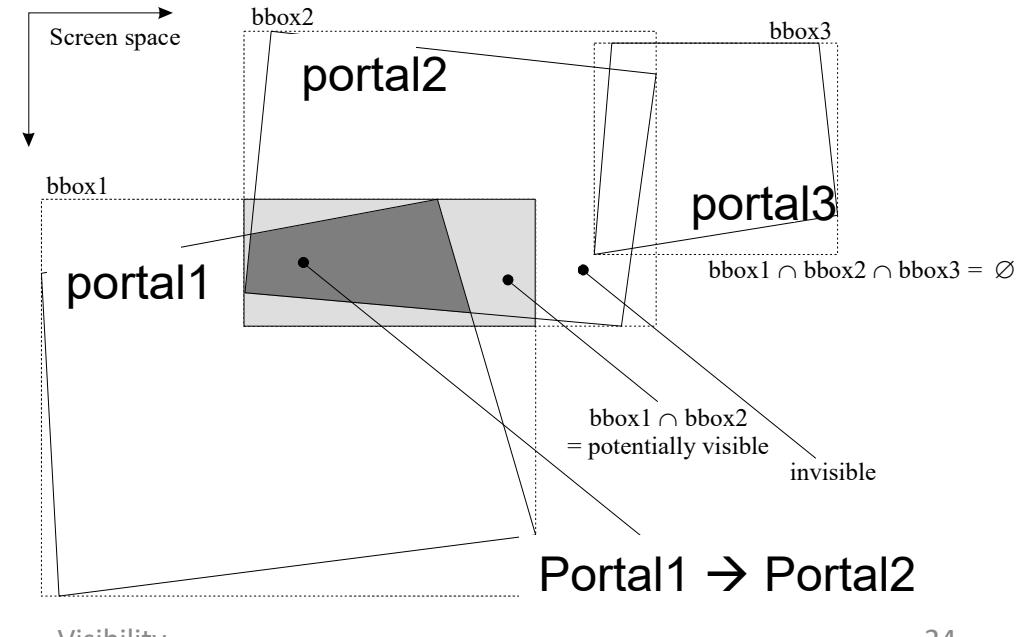


Adjacency graph



# 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



# Regular Depth Buffer

- *Depth Prepass* in deferred rendering
  - Pass 1: Rasterize the geometry,  
only write depth + object id to G-buffer
  - Pass 2: Read + collect all visible ids from G-buffer
- Can be expensive
  - For large scenes with many primitives
  - For high framebuffer resolutions

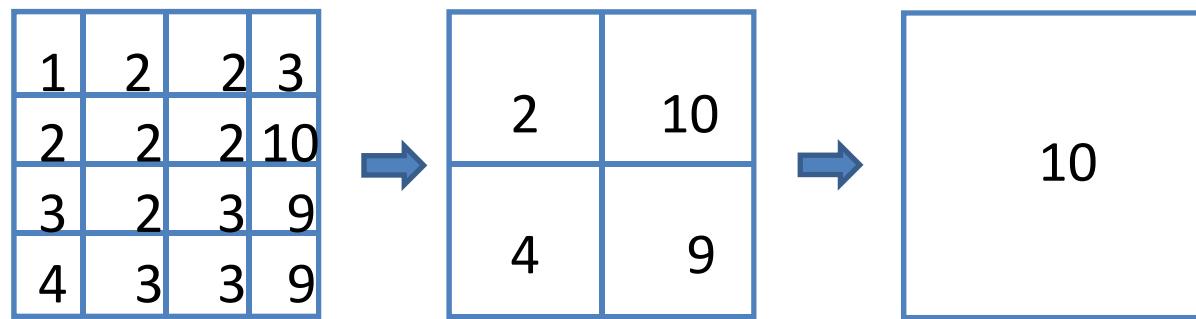
# Virtual Occluders

- Expensive to use objects with many primitives as occluders
- Cannot use *bounding volumes* of objects, since this is not a conservative test
- But we can use *bounded volumes* completely contained inside objects as *virtual occluders*
- Virtual occluders can be simple boxes

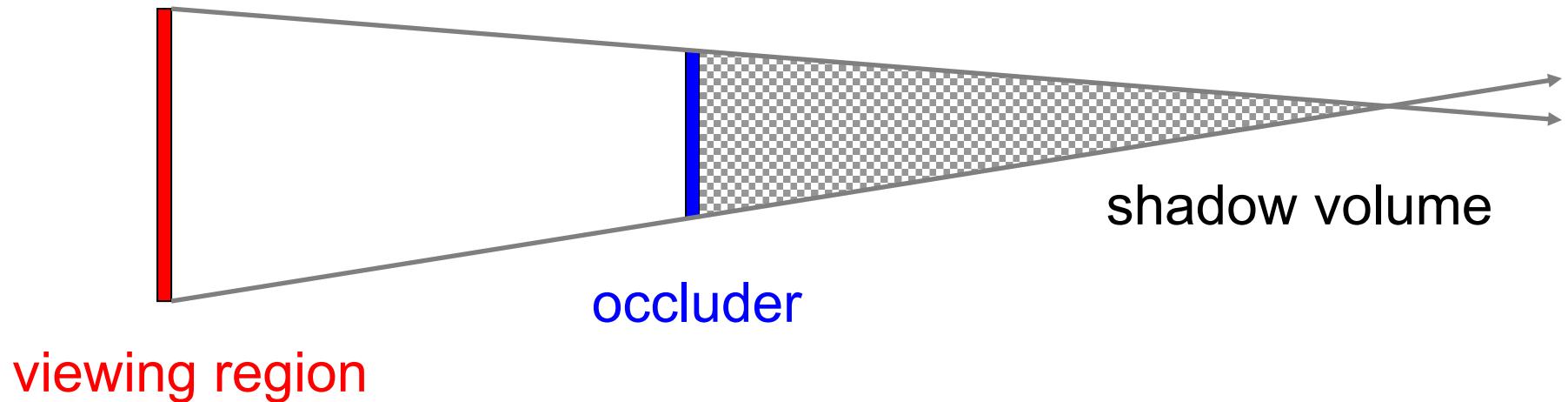


# Hierarchical Depth Buffer

- Regular depth buffer can be expensive for high resolution
- Replace depth buffer with a depth *pyramid*
  - Bottom of the pyramid: full-resolution depth buffer
  - Higher levels: smaller resolution depth buffers, where a pixel represents max. z-value of group of pixels on lower level
- Hierarchically rasterize polygon, starting from highest level
  - If polygon is further than recorded pixel, early exit
  - If polygon is closer, hierarchically test lower levels
  - If bottom of pyramid is reached and polygon still closer, propagate the value up the pyramid



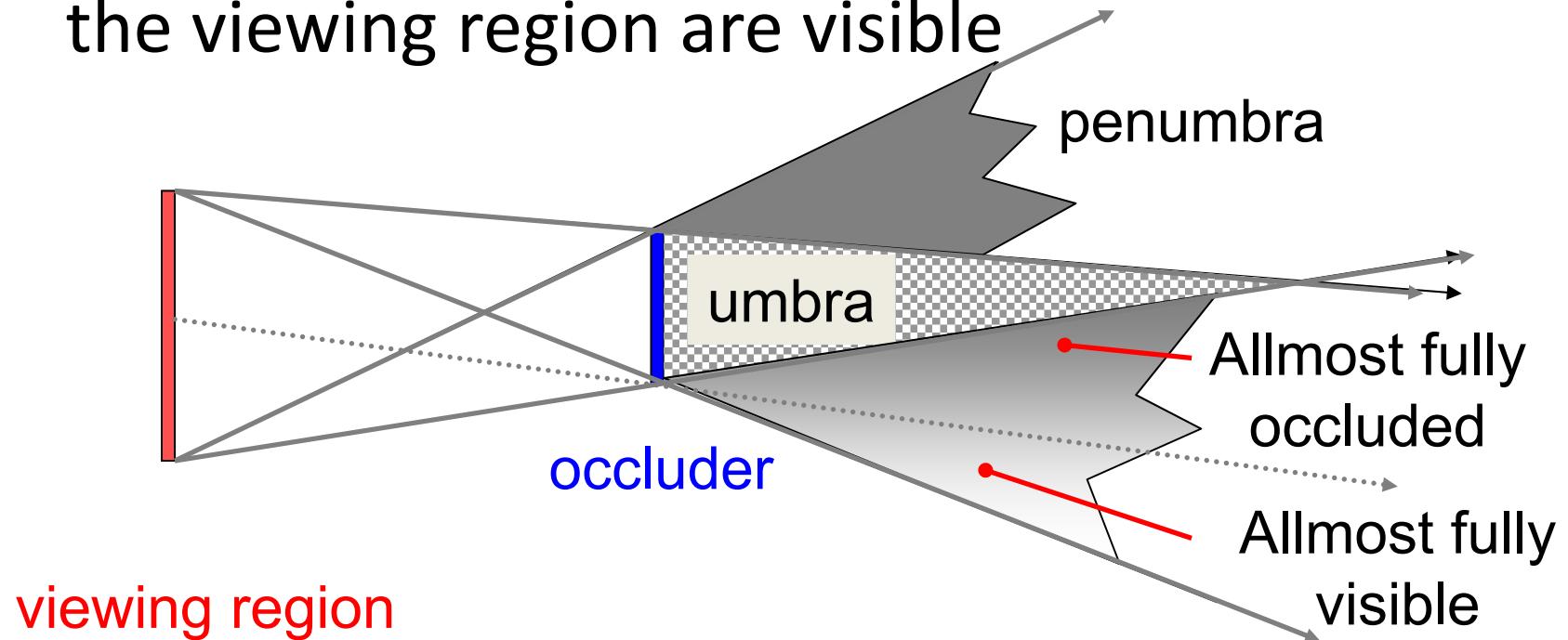
# Visibility from a Region (Example in 2D)



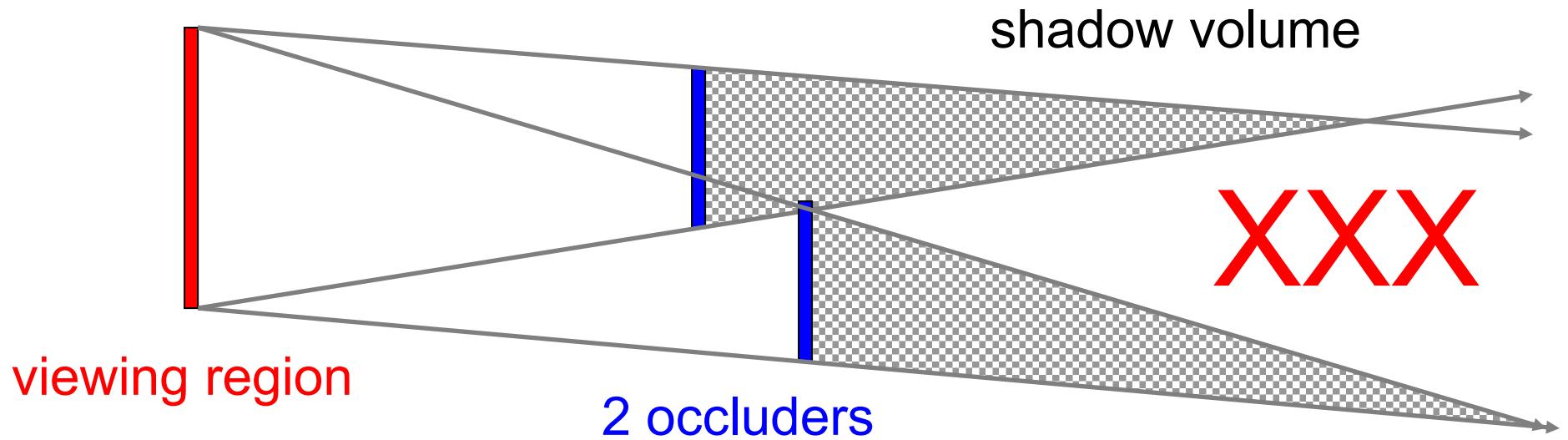
- If we want to precompute visibility, we must discretize viewpoints into view regions
- Only works for static scenes

# Umbra and Penumbra

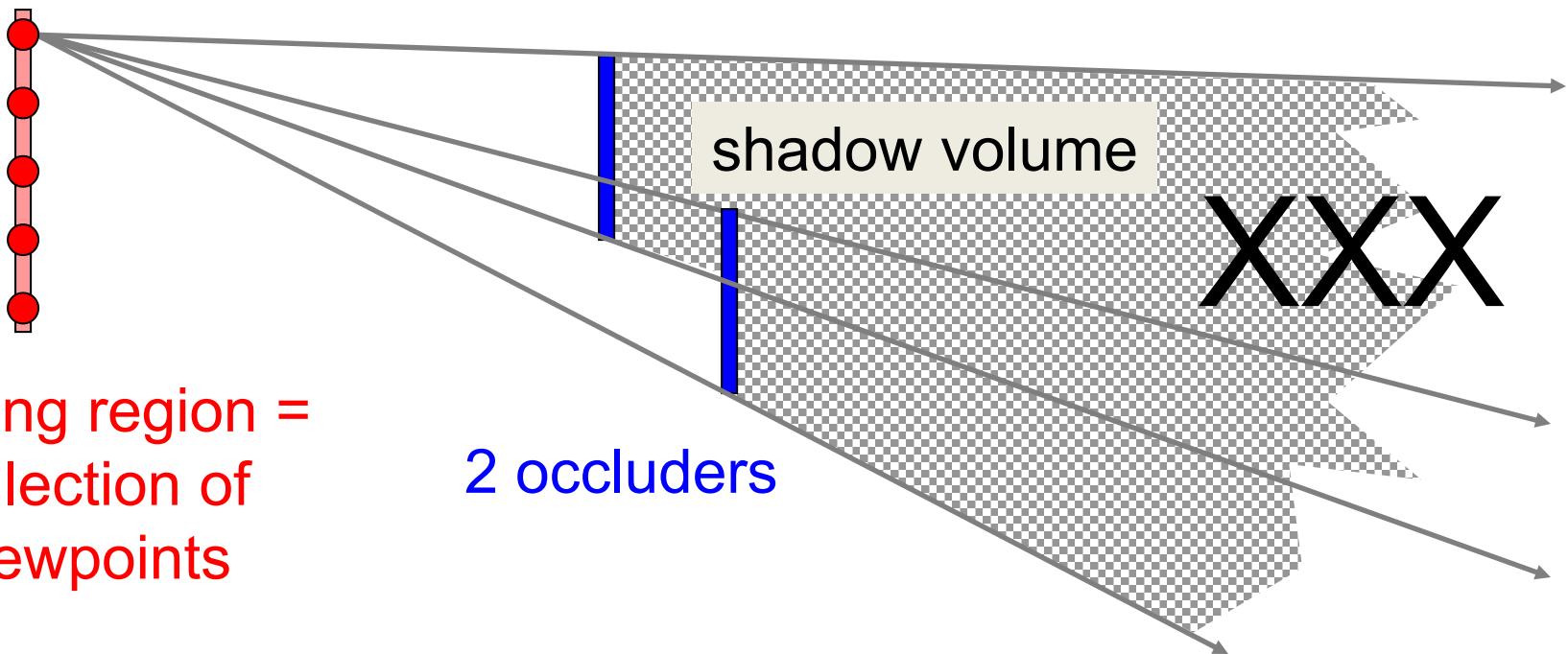
- **Umbra** = full shadow, penumbra = half-shadow
- **Umbra** is a simple in/out classification
- **Penumbra** additionally encodes which parts of the viewing region are visible



# Visibility from a Region



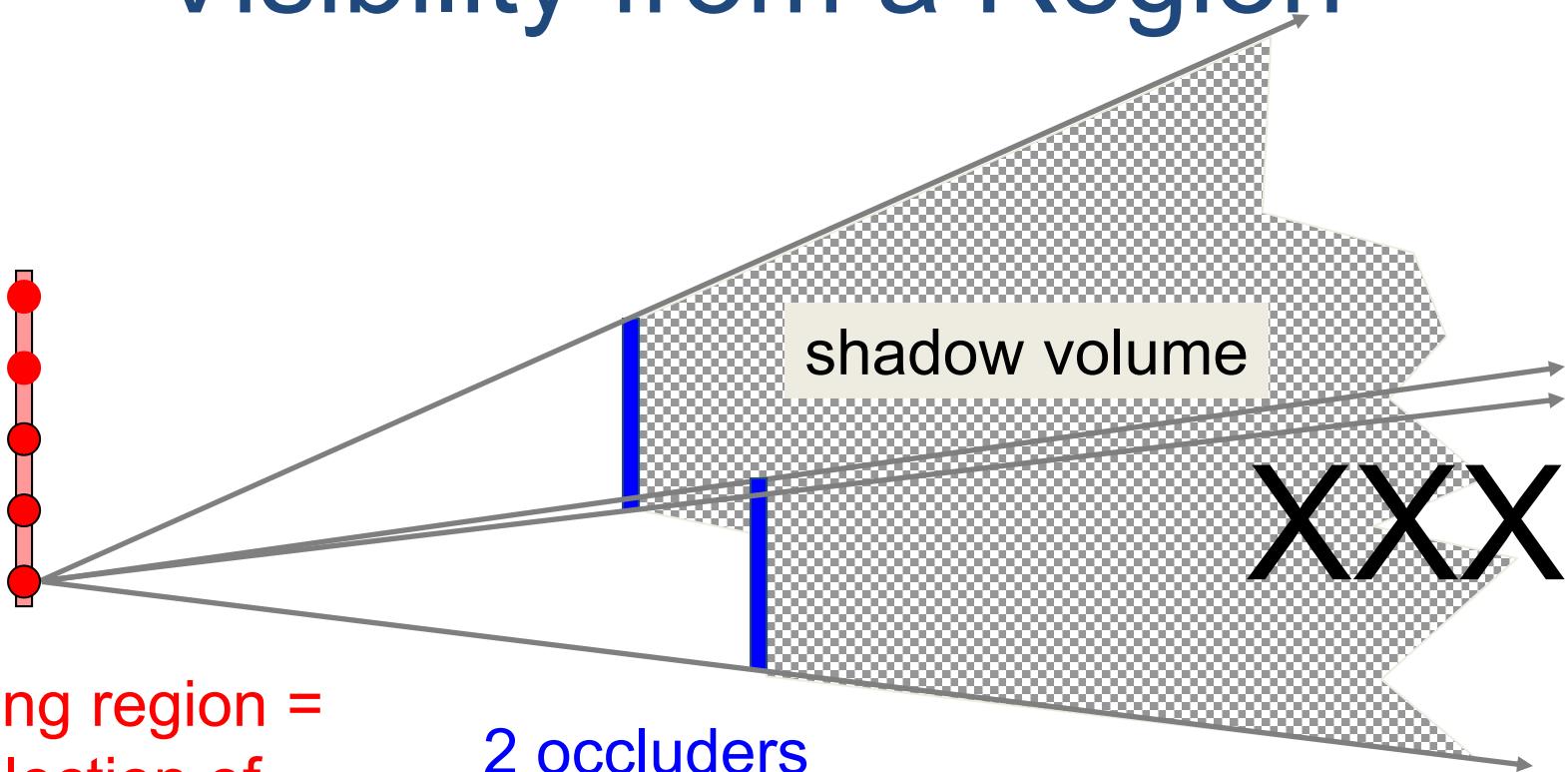
# Visibility from a Region



viewing region =  
collection of  
viewpoints

2 occluders

# Visibility from a Region

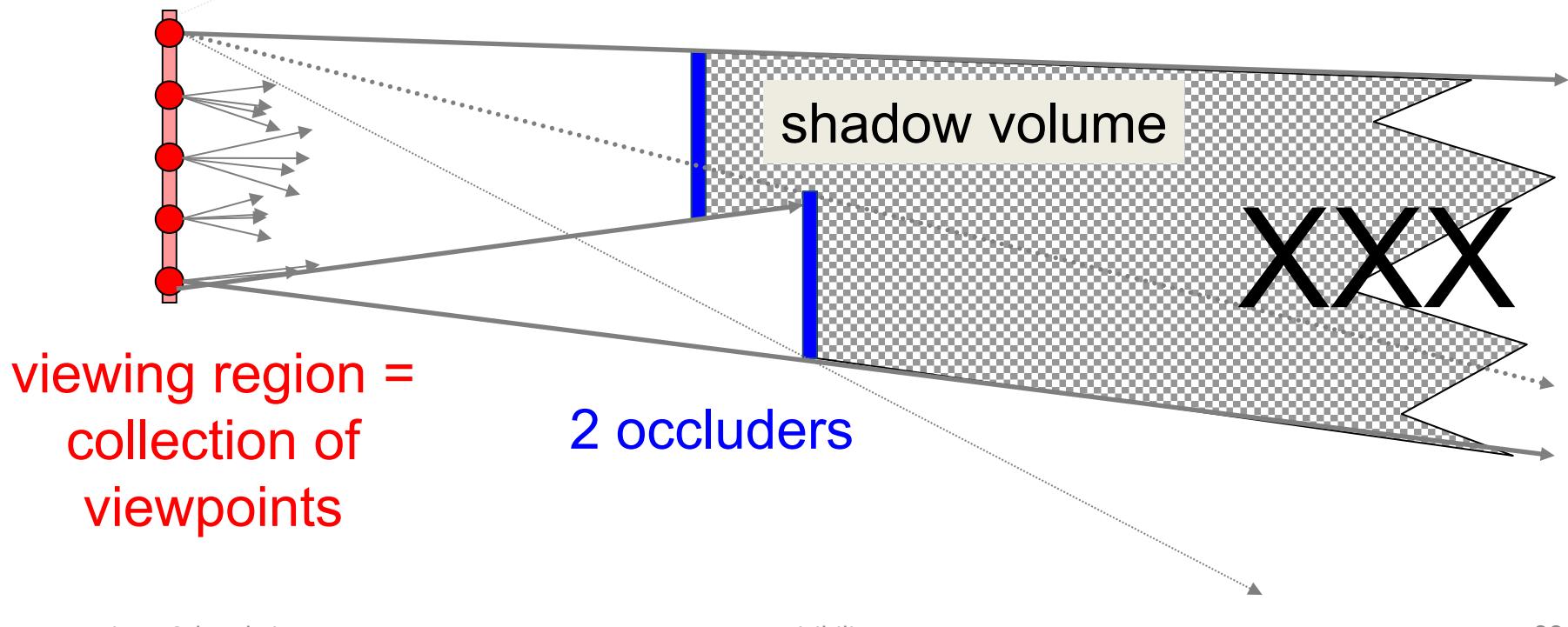


viewing region =  
collection of  
viewpoints

2 occluders

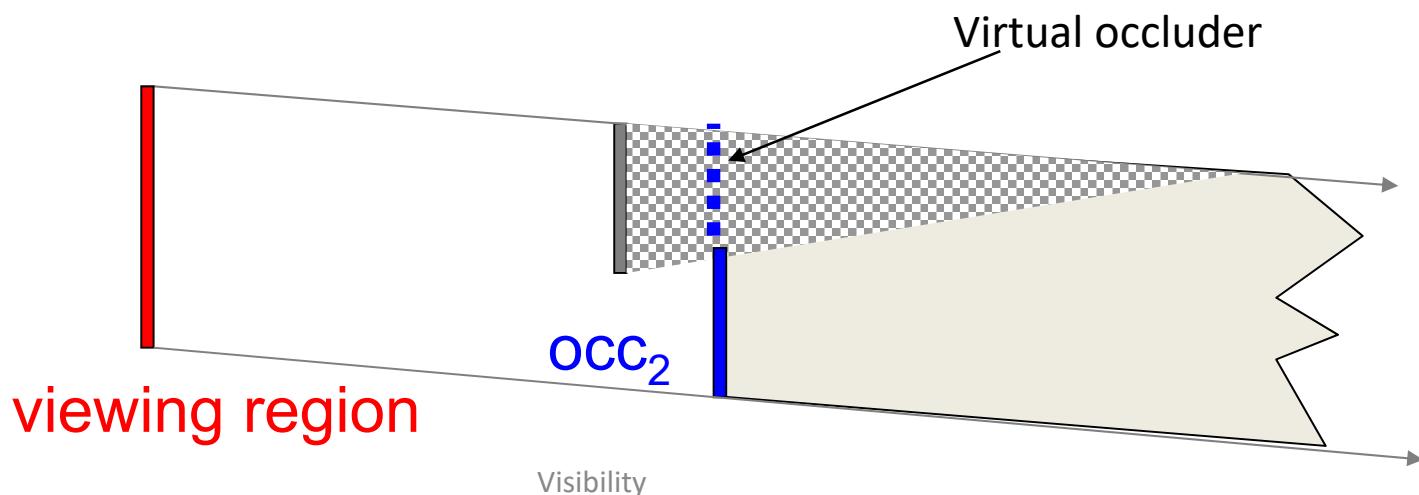
# Visibility from a Region

- Area XXX is always occluded → complete shadow volume is **more** than the union of individual shadow volumes



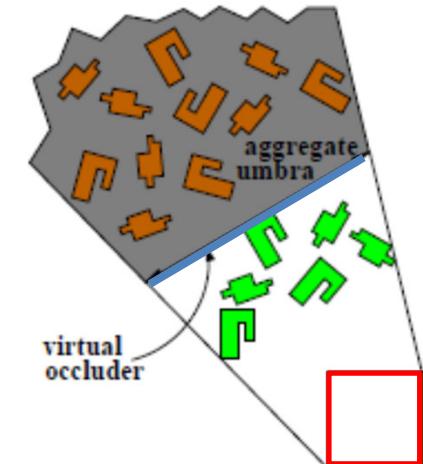
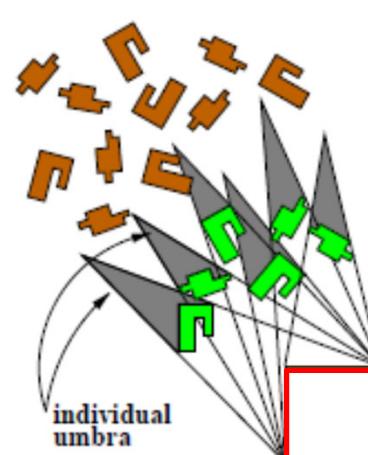
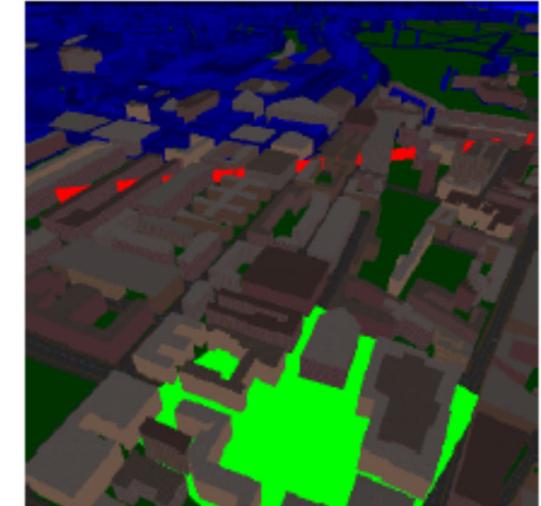
# Virtual Occluders for Regions

- Shadow volume data structure = empty
- For each occluder  $\text{occ}_i$  in front-to back order
  - Expand occluder inside existing shadow volume as far as possible
  - Calculate shadow volume  $\text{SV}_i$
  - Add  $\text{SV}_i$  to shadow volume data structure
- Test the scene against the SVDS



# Virtual Occluders from Fusion

- Occluder fusion via intermediate data structure
- Search inside umbra for adjunct occluders and add it to umbra
- Virtual occluder is guaranteed to be occluded from cell
- Virtual occluder grows large



V. Koltun, Y. Chrysanthou, and D. Cohen-Or, "Virtual Occluders: An Efficient Intermediate PVS Representation," Rendering Techniques 2000: Proc. 11th Eurographics Workshop Rendering, pp. 59-70, June 2000.

Dieter Schmalstieg

# Questions?