Advanced GPU techniques

CGG MFF UK Praha

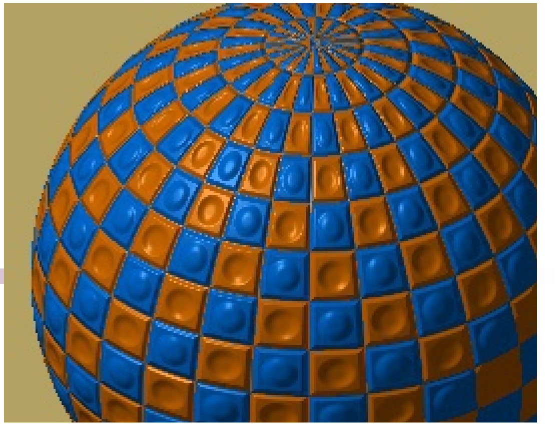
pepca@cgg.mff.cuni.cz

http://cgg.mff.cuni.cz/~pepca/

Content

- advanced lighting
  - environment maps
  - light maps, irradiance maps, refraction, bump-mapping
- multi-pass algorithms
  - buffers (stencil buffer, depth buffer, accumulate buffer)
- shadow casting
  - “shadow buffers“, projected shadows, volume shadows
- CSG rendering
- non-photorealistic techniques
- photorealism: BRDFs, sub-surface scattering, ...

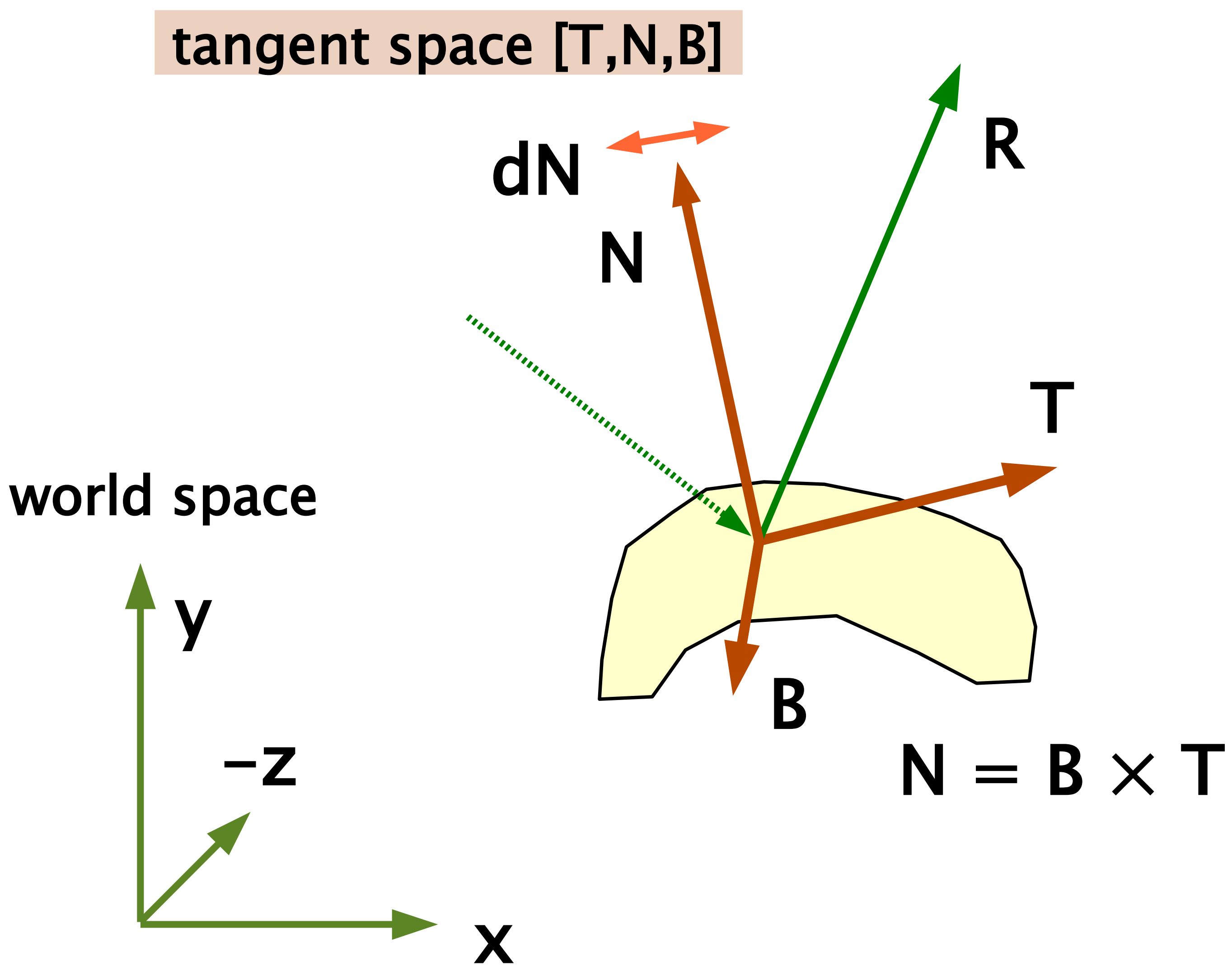
Normal map ("bump–map")



- **"bump-mapping"**
  - modulation of normal vector (originally from 3D model)
  - imitation of **surface imperfections … rough, bumpy** surface
  - data in regular **2D texture** ("$\mathbf{R^2 \rightarrow R^3}$", "[**s, t**] → [$\mathbf{N_x, N_y, N_z}$]") – "**normal map**"

- **"tangent space"**
  - axes: **tangent T**, **normal N**, **binormal B**
  - normal map contains "relative" data (normal vector in tangent space) [$\mathbf{N_t, N_n, N_b}$]
    - ideal normal **[0,1,0]**

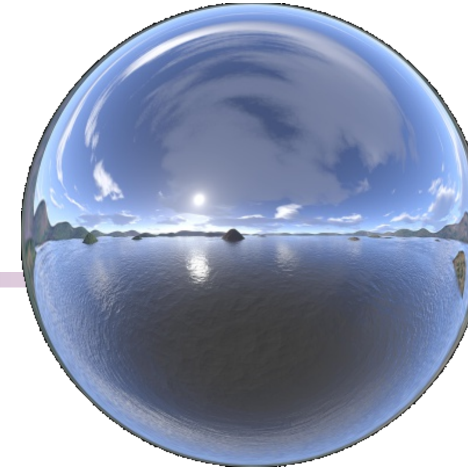
Normal map in tangent space



Tangent space

- if there is no need of world space, we can transform all relevant vectors to the **tangent space** (vertex process.)
  - view vector, light vectors ("half vectors"), ..
  - lighting can be done in tangent space (= **normal map** space)
- or we must stay in world space
  - because of some global techniques ("**environment map**")
  - "**tangent → world**" matrix could be interpolated (orthogonality issues!?)
  - **normal map values** have to be transformed back to the world space

Environment maps

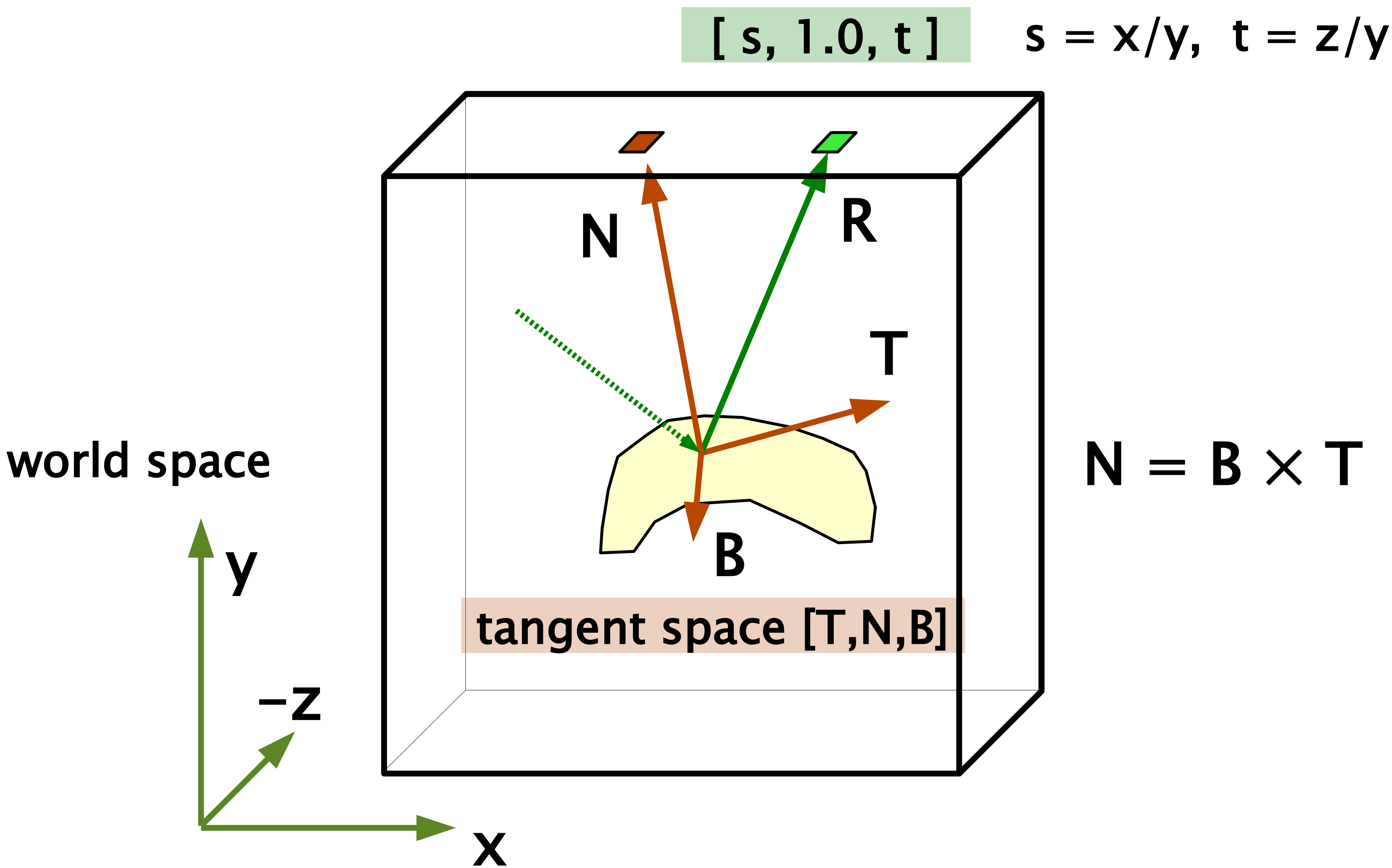


- **concept**
  - HW implements six-part texture – six faces of a cube (“cube-mapping”)
  - addressing by **3D vector** (needs not be normalized)
  - **static** or **dynamic** data (possibility of preprocessing)
- **popular use**
  - perfect <u>mirror reflection</u> (“environment map“)
  - <u>glossy reflection</u>, <u>diffuse component</u> – simulations of real lighting conditions
  - <u>refraction</u> of light
  - combination with “<u>bump-mapping</u>”

“Cube-mapping“

- **other utilization**
  - 3D vector normalization, ...
  - storage of any **computationally demanding** function (up to $\mathbf{R^3} \rightarrow \mathbf{R^3}$) for shaders
- “**environment mapping**”
  - input 3D vector must be in **world coordinate space**
  - transform matrix “**model → world**“ is needed
  - in shader languages there are support functions for **re-flection** and **refraction**

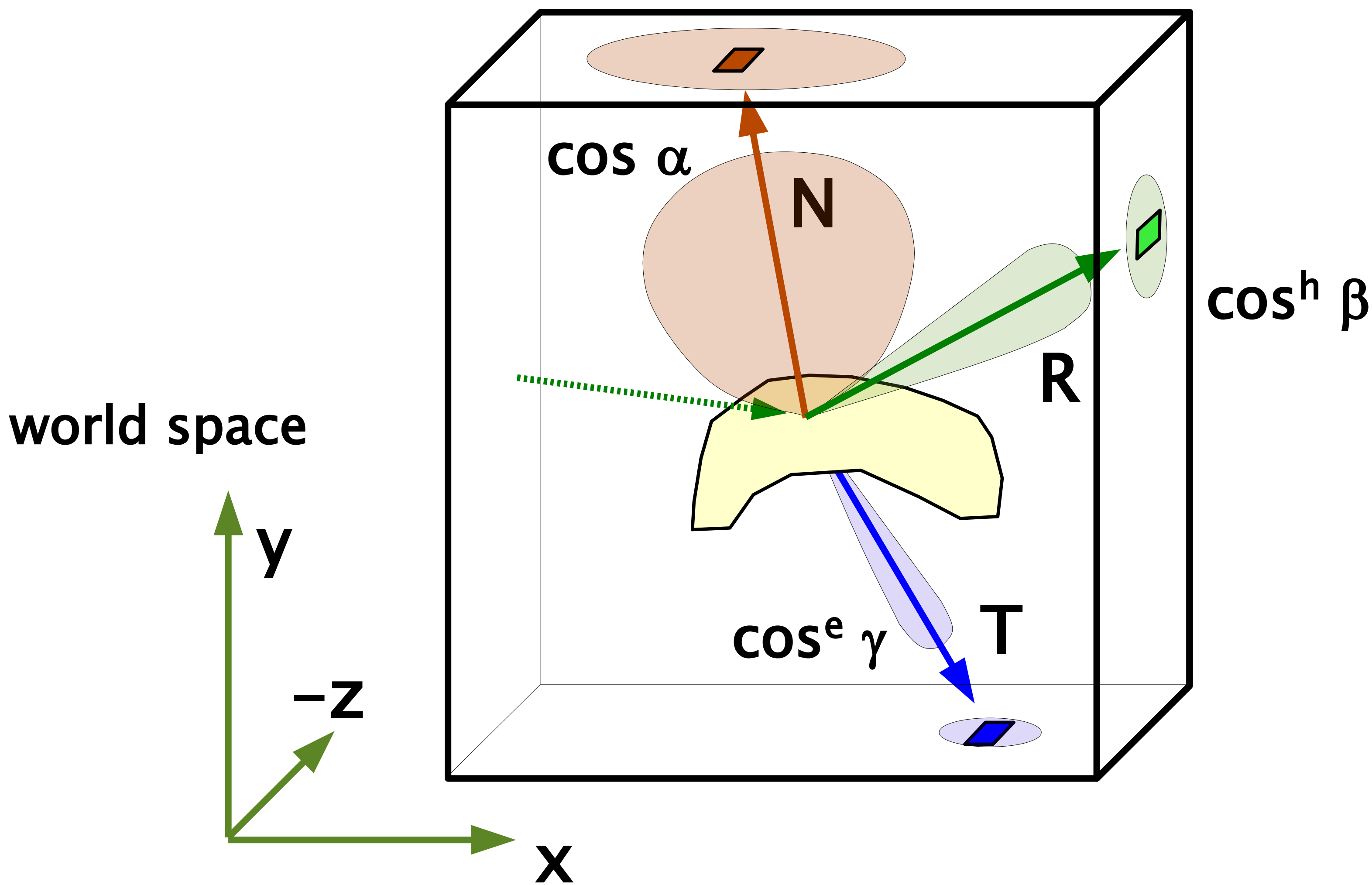
Coordinate spaces



Enhanced lighting

- **diffuse component**
  - cube-map is addressed by the **normal vector N**
  - precomputed **incoming light total** (integral) using the "**cos α**" factor
- **specular component**
  - exact representation of models with qualitative term "**cos β**"
  - cube-map is addressed by the **reflected vector R** ("reflect()" in GLSL)
  - pre-computed **environment blur** using the "**$\cos^h \beta$**" factor

Lighting–related maps



Refraction of light

- **simplified approach**
  - cube-map is addressed by the refraction **vector T**
  - usually the perfect (not blurred) environment image is used
    - we can use **blurred environment “cos^e γ”** as well
- **light dispersion** can be simulated
  - variation of index of refraction for different wavelengths
  - shared environment image

Multi-pass algorithms

- 3D scene (or its part) is **processed on GPU multiple times**
  - different **GPU settings** (buffer setting, depth-test, stencil-test, rendering parameters)
  - different **transformation matrix**, projection
  - **different shaders**
- **data exchange/sharing** between passes
  - **GPU buffers** (frame buffer, depth-buffer, stencil buffer, accumulation buffer, general-purpose buffers)
  - **textures** (shadow map, environment map, ...)

Accumulation buffer, environment

- **accumulation buffer** usage:
  - anti-aliasing
  - motion blur simulation
  - depth of field simulation
  - iterated scene pass with slightly different rendering settings – **transform (projection) matrix** mostly
- dynamic computation of **environment image**:
  - we want an animation to be reflected on other objects
  - **cube-map**: we need to do scene rendering 6-times
    - reduction based on animation specifics

Shadow casting

- several approaches
  - **sharp shadows** (one pass)
  - **soft shadows** (more “passes”, accumulation of results)

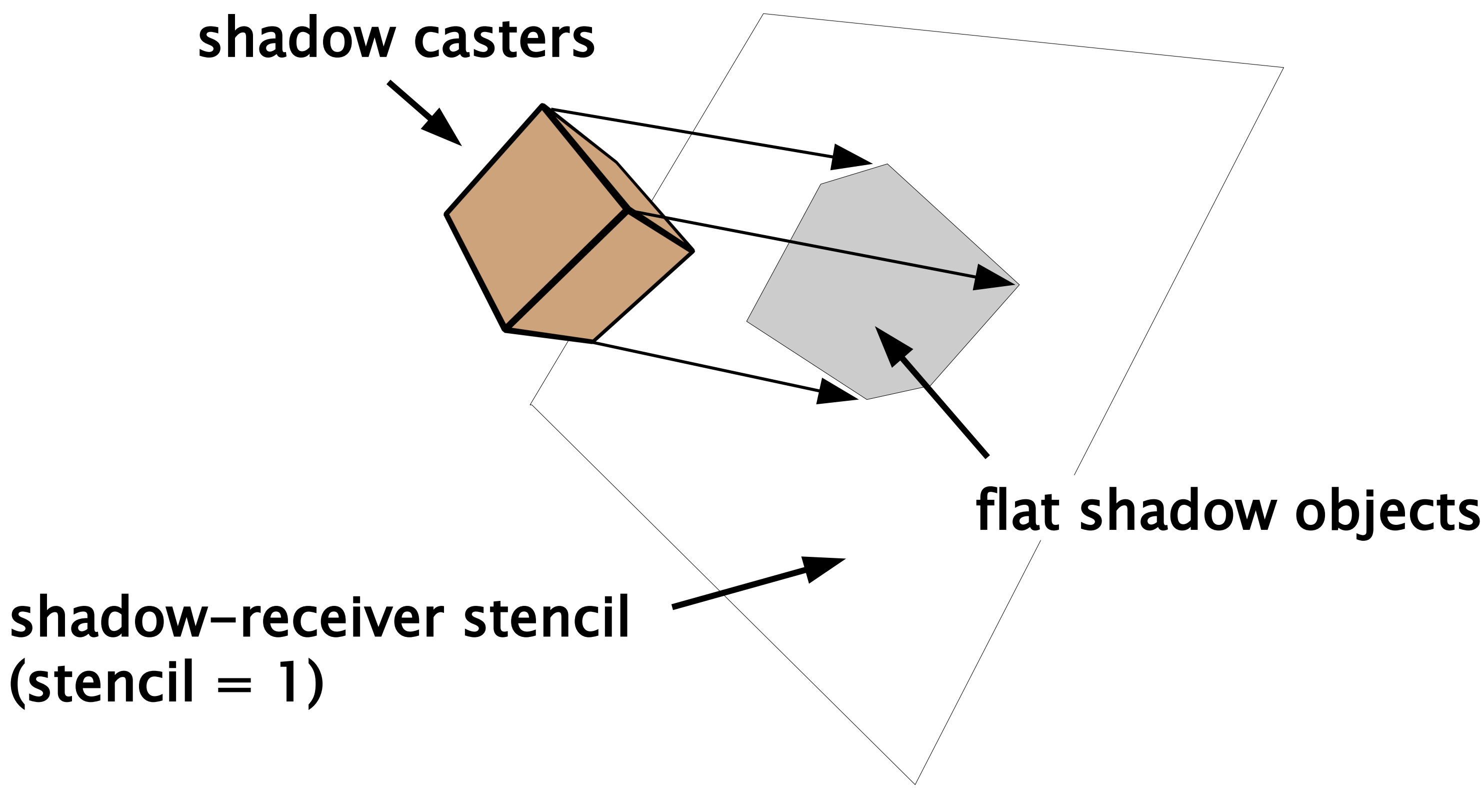
- single **shadow-receiving plane**
  - simple approach, not generally usable
- **shadow mapping**
  - shadow “depth-buffer”, supported in HW
- **shadow volumes**
  - precise but very computationally intensive

Shadow-receiving plane

- sharp shadows … point light source
- use of **stencil buffer** and **multiple scene passes**
  - stencil prevents shadow duplication
- simple algorithm
  - single **shadow-receiving plane**
  - shadow could be **opaque** (destroying the original surface color) or **transparent** (only reducing the amount of light)

Shadow casting to single plane

- **projection matrix** (math sense of the word) from 3D world into shadow-receiving plane



Shadow casting to a plane

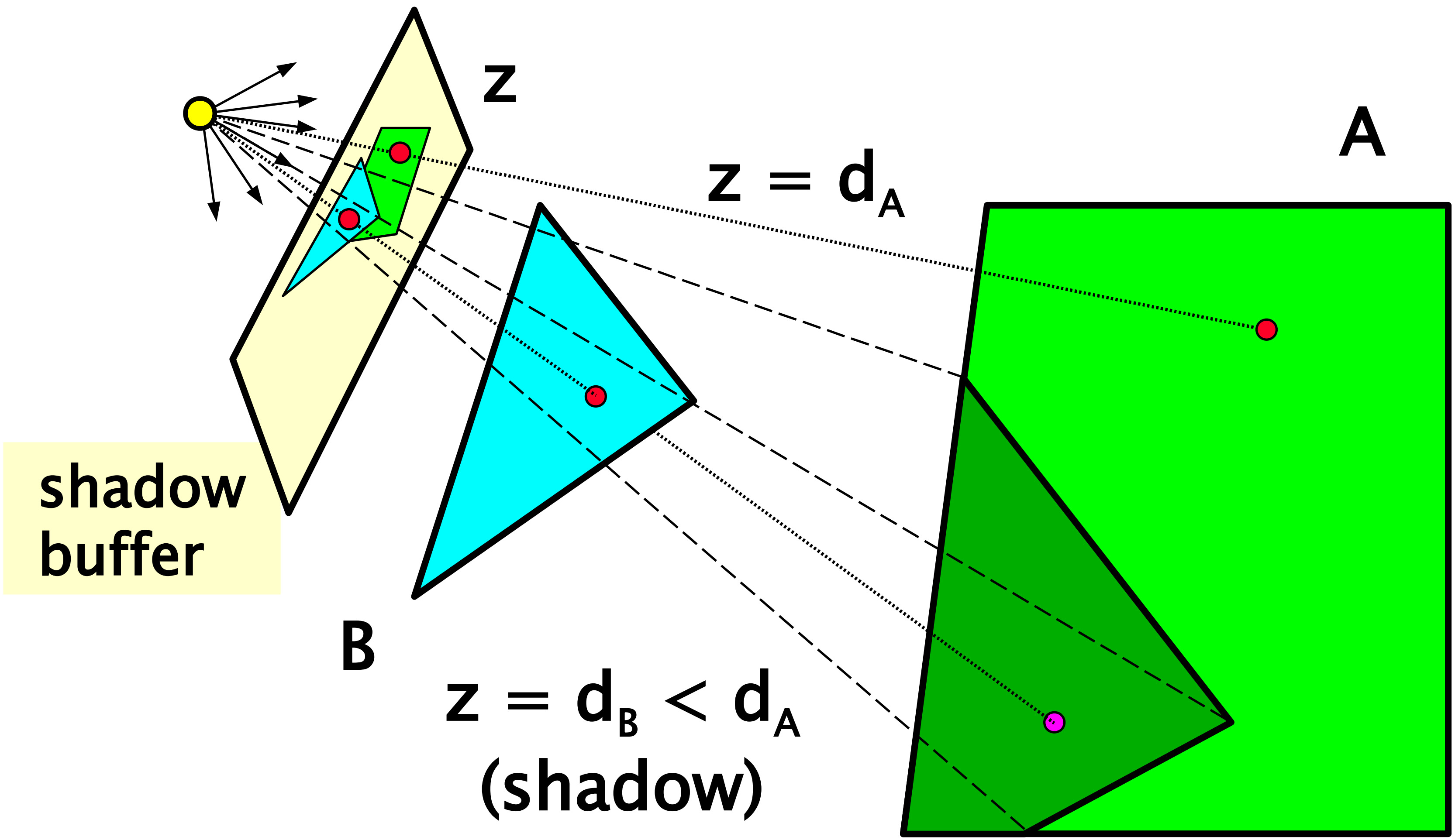
- procedure
  1. the whole scene is rendered using **ordinary projection**
     - shadow-receiver **sets stencil to 1**
     - all the other objects **zero this bit**
  2. all potential **shadow-casters** are rendered to the shadow-receiving plane
     - depth-test is **off**
     - **special transformation matrix**
     - shadows are drawn only to the **(stencil==1)** pixels
     - if **semi-transparent shadows** are required, the first write into the frame-buffer should also **zero the stencil**

Shadow mapping

1. scene is rendered from the **light-source viewpoint**
   - no need to modify frame buffer, only **depth-buffer** has to be updated
2. depth-buffer is moved into a texture ("**shadow map**")
   - regular projection according to the camera
   - use of **projective texture coordinates**
   - **GPU can test** actual distance of a fragment from the light source (in the world space) against the pre-computed val-ue stored in the **shadow-map texture**:

```
float4 shadow = tex2Dproj( shadowMap, texCoordProj );
```

Shadow mapping



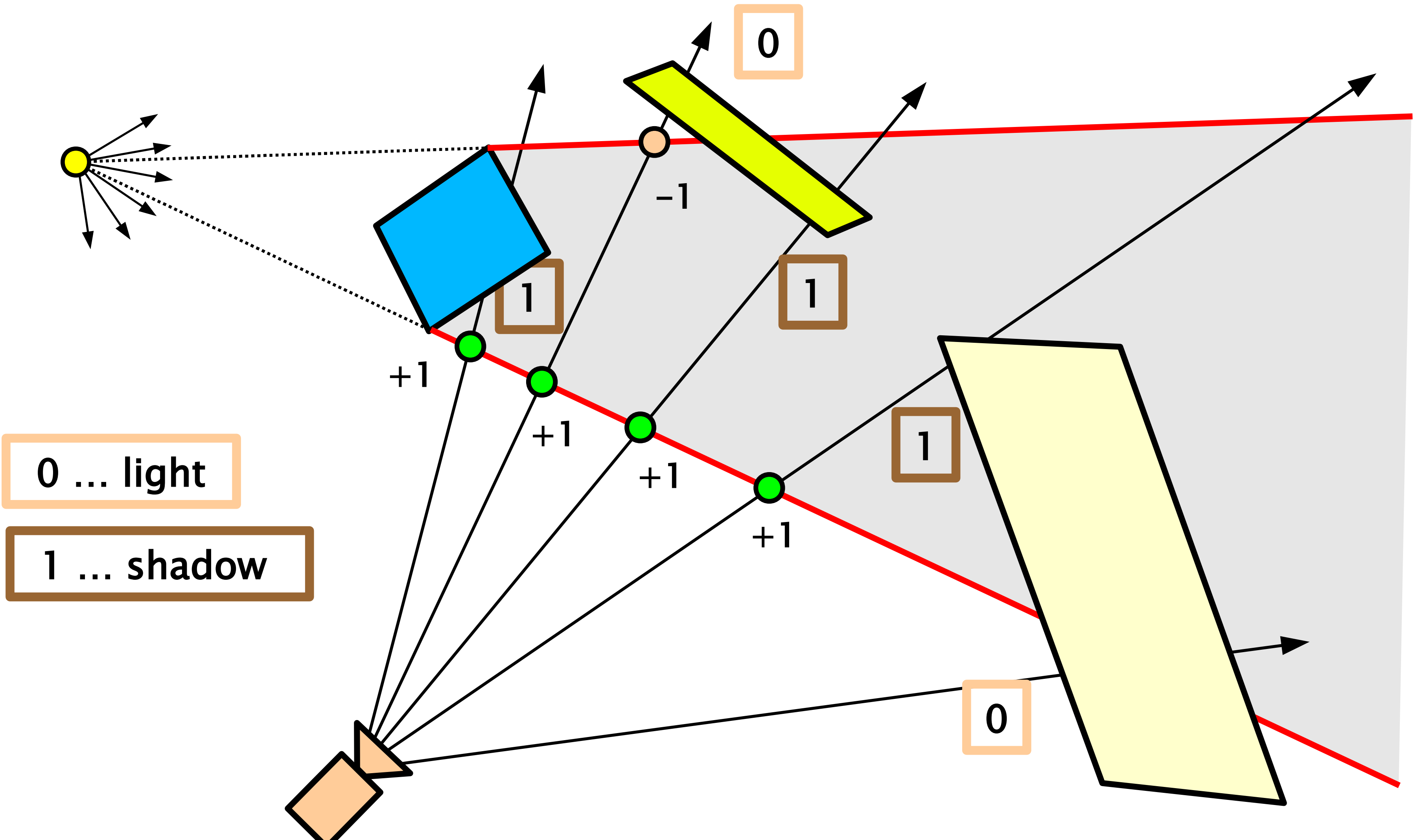
Shadow volumes

- every shadow-caster casts an infinite **"shadow volume"** (shadow solid)
- **lateral faces** of a shadow solid are considered, but invisible, virtual quadrilaterals
  - virtual ray from the camera is tested against these faces
  - GPU can rasterize the virtual faces and "draw" them into the stencil buffer (no need to change frame buffer)
- at the end stencil buffer values define **shadows** in the scene
  - this has to be done separately for **each point light source**

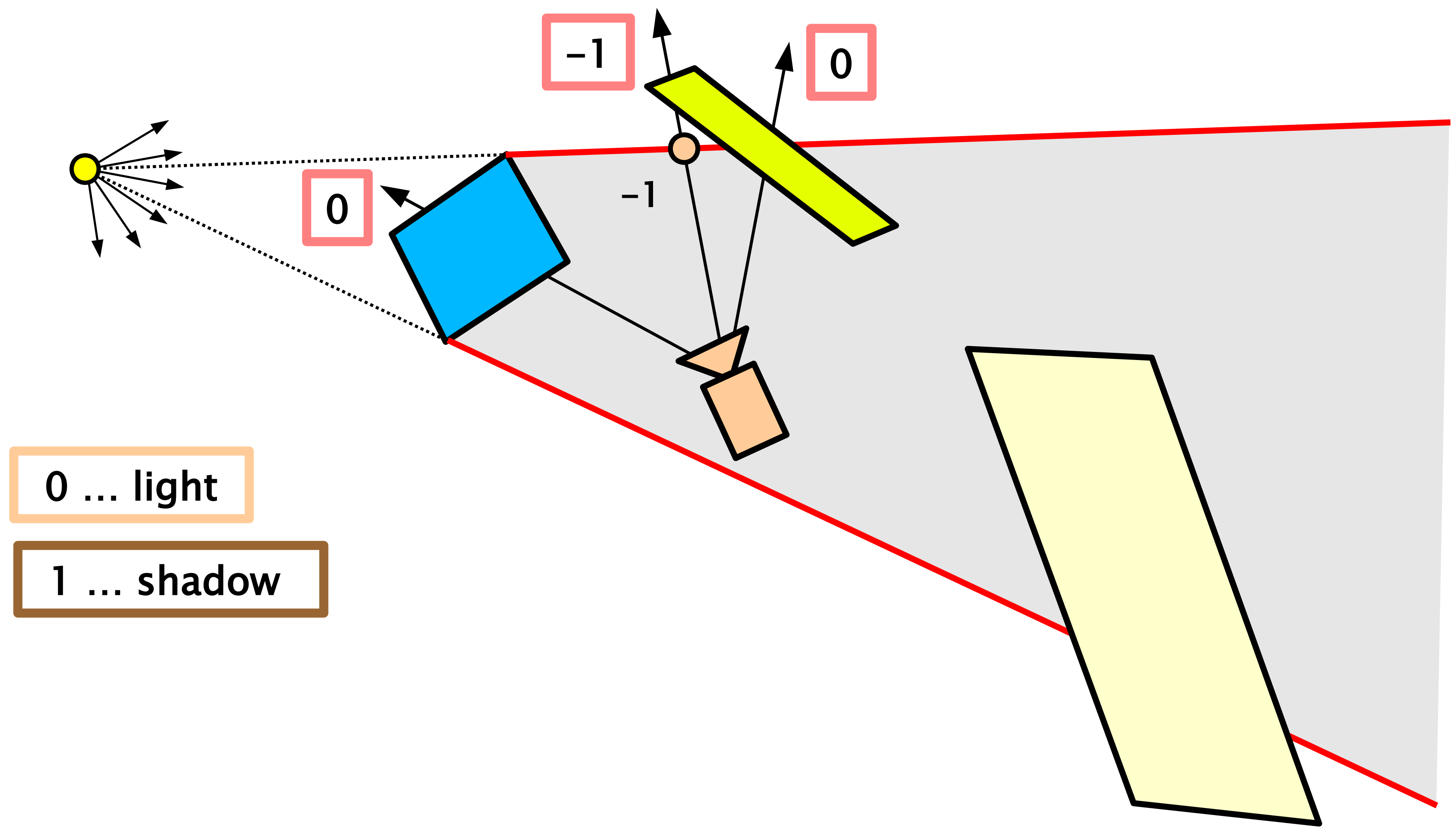
Shadow volumes I

- common **first phase**: rendering of the **real scene**
  - writing to the depth-buffer, lighting: “ambient”
- shadow face is either **front-facing** or **back-facing**
  - shadow volumes **do not modify depth-buffer** (but are tested against it)
- **second phase**: only lateral shadow volume faces are rasterized:
  - **front-facing visible** face **increments** the stencil
  - **back-facing visible** face **decrements** the stencil
- **third phase**: stencil==0 means “light”
  - contribution of the light source is added

Shadow volumes I



Shadow volumes I – failure



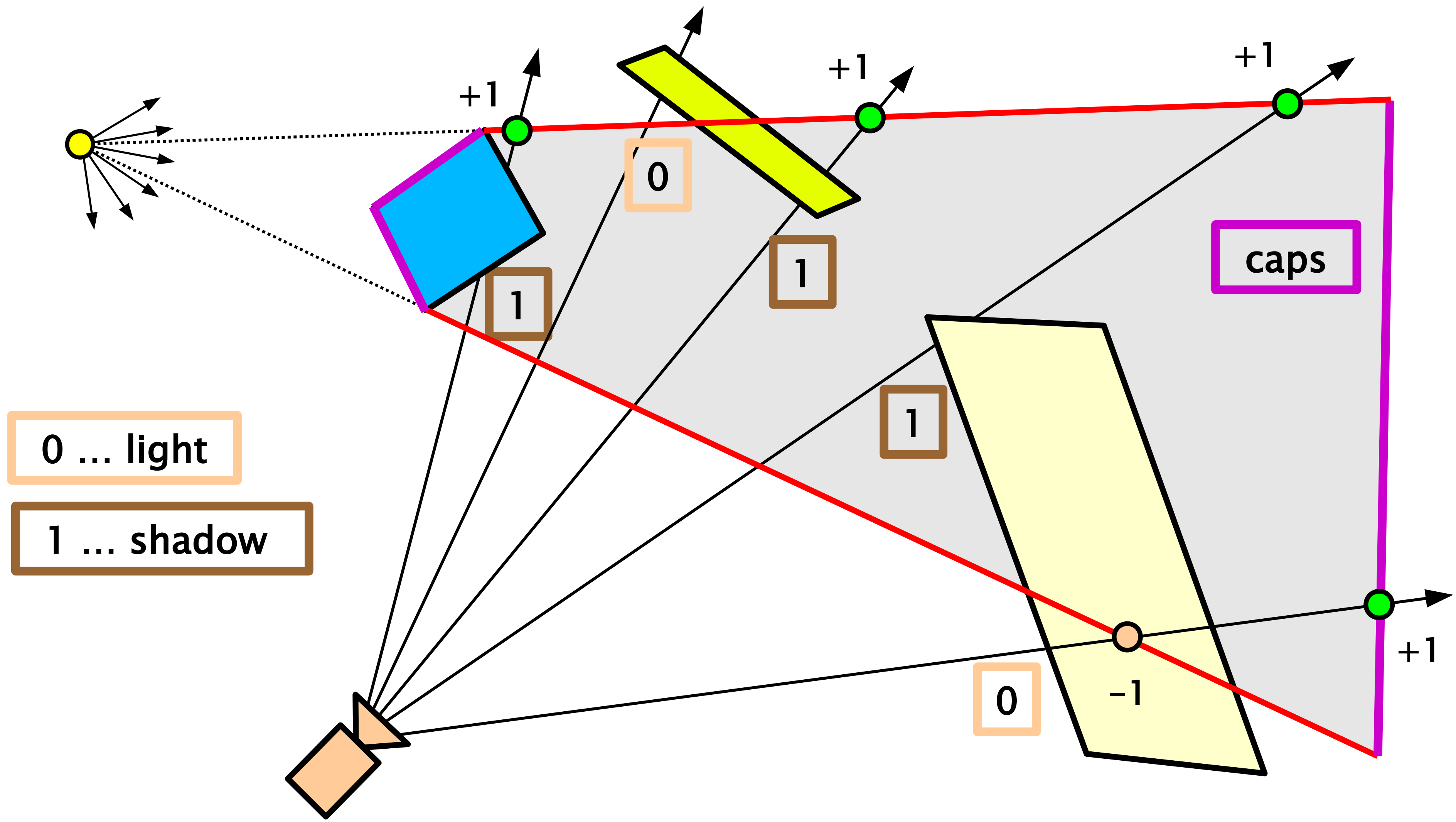
Shadow volumes II

- **camera** can be placed **anywhere**
  - shadow solid is perfectly sealed using “**caps**”: one is formed by an <u>illuminated part</u> of an object, the second one lies in <u>infinity</u>

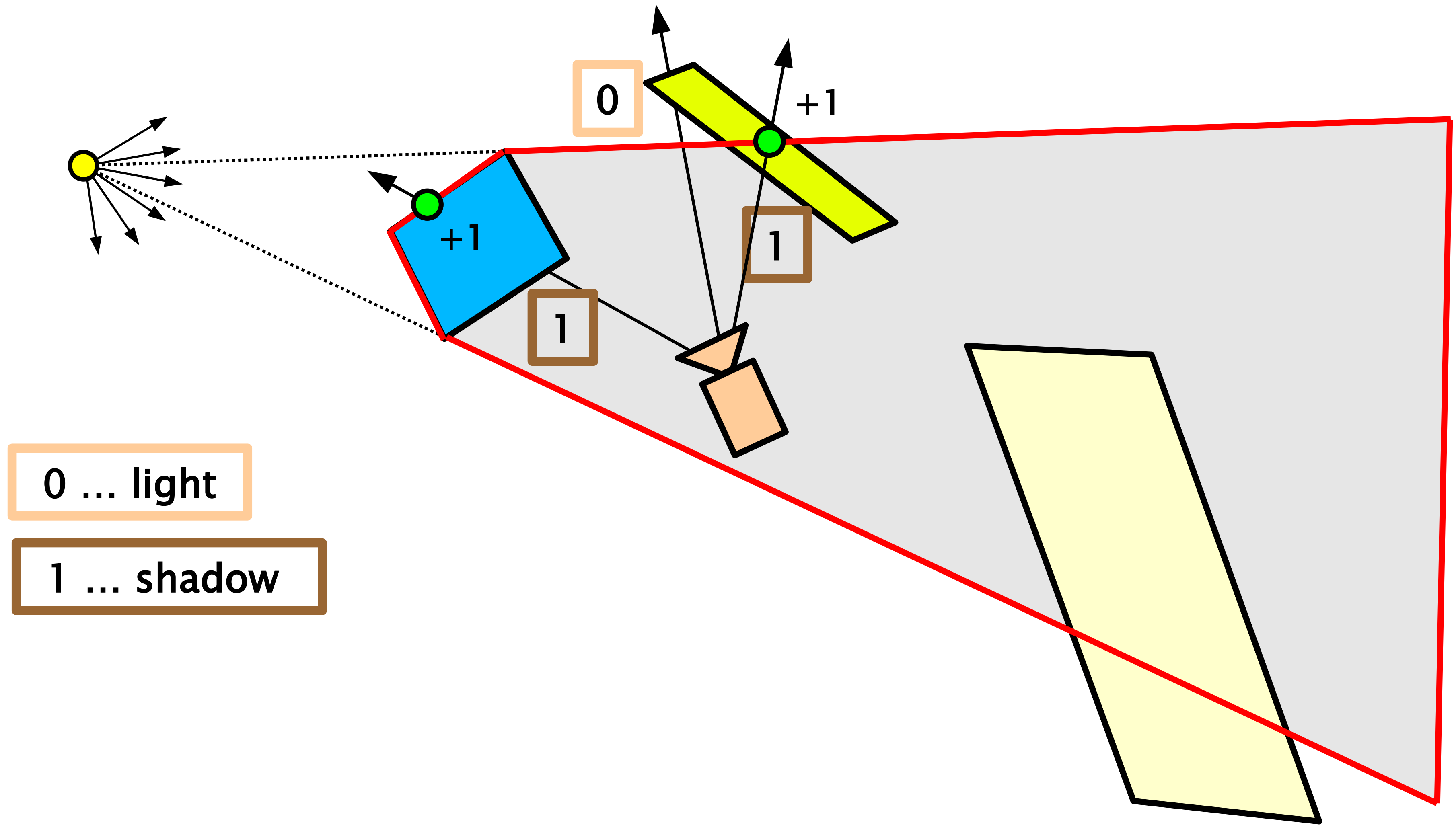
- **second phase**: lateral shadow faces and both “caps“
  - **front-facing <u>invisible</u>** face **<u>decrements</u>** the stencil
  - **back-facing <u>invisible</u>** face **<u>increments</u>** the stencil

- **third phase**: stencil==0 means “light”
  - contribution of the light source is added

Shadow volumes II



Shadow volumes II – correct



Vertices in infinity

- lateral faces in the back “cap” need to have vertices **in infinity**
  - more distant than any other objects in a scene
- vertex projection [**x**, **y**, **z**, **1**] to infinity: [**x**, **y**, **z**, **0**]
- **projection matrix** for value “**far = ∞**“

$$A = \frac{2n}{r-l} \qquad B = \frac{r+l}{r-l} \qquad C = \frac{2n}{t-b} \qquad D = \frac{t+b}{t-b}$$

$$M(n,\infty,r,l,t,b) = \begin{bmatrix} A & 0 & 0 & 0 \\ 0 & C & 0 & 0 \\ -B & -D & 1 & 1 \\ 0 & 0 & -2n & 0 \end{bmatrix}$$

Projection to infinity

- projection of an **intrinsic point** (including homogeneous division):

$$[\, x,\ y,\ z,\ 1 \,] \cdot M \ = \left[\ \frac{x}{z}A - B,\ \frac{y}{z}C - D,\ 1 - \frac{2n}{z}\ \right]$$

- projection of an **extrinsic point**:

$$[\, x,\ y,\ z,\ 0 \,] \cdot M \ = \left[\ \frac{x}{z}A - B,\ \frac{y}{z}C - D,\ 1\ \right]$$

Front face / back face

- from the **point of view of camera**
  - **GPU can filter** (“**face cull**“) according to vertex order in NDS:

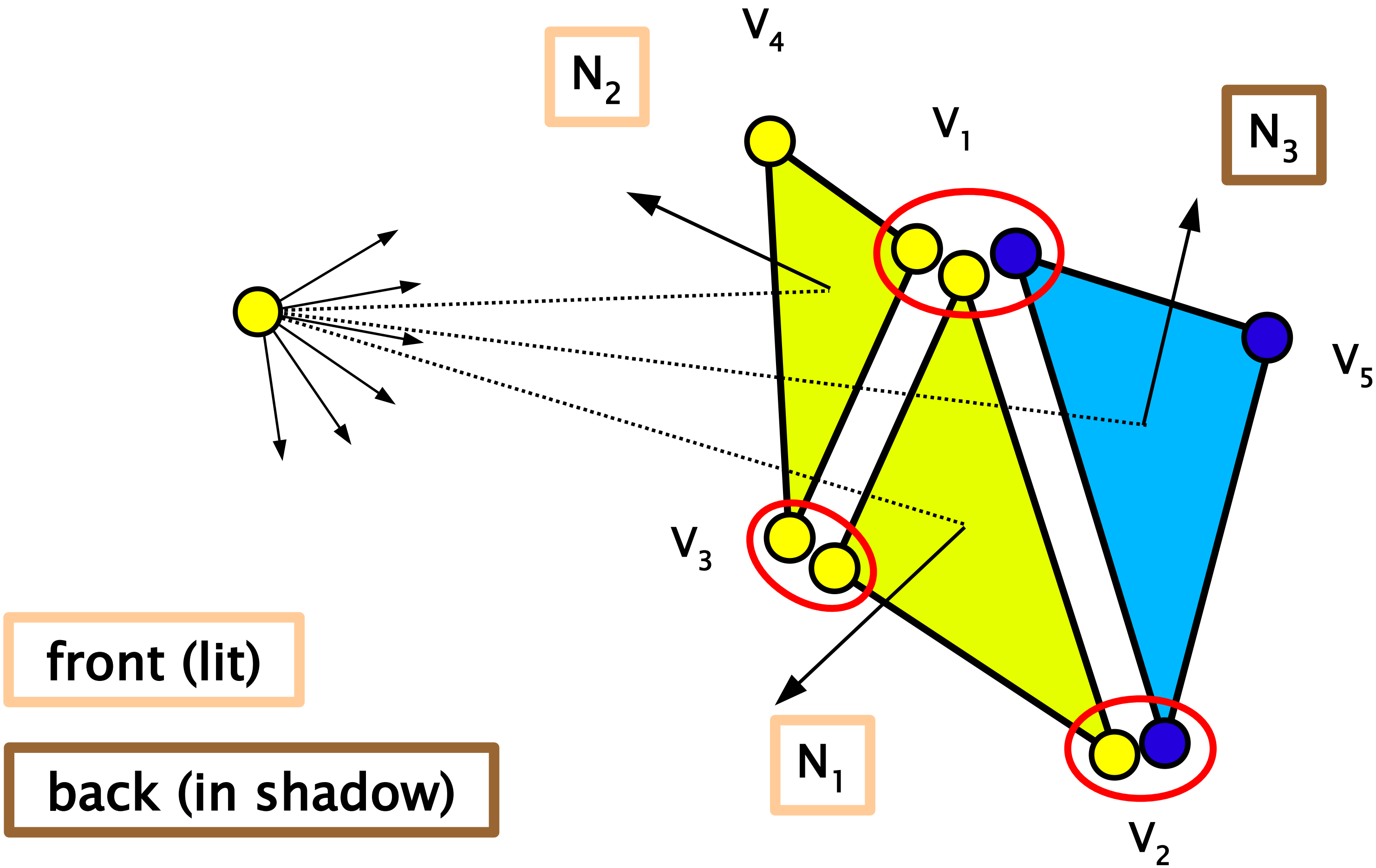
```
glEnable( GL_CULL_FACE );
glFrontFace( GL_CCW );
glCullFace( GL_BACK );  // draw front faces only
…
```

- from the **point of view of light source**
  - computed on **CPU** (normal vectors)
  - **programmable GPU** can help (vertex processing)
    - elimination of incorrect primitives (degeneration, clipping)
    - **drawback**: all potential primitives have to be sent to GPU

Face elimination techniques

- there is no good way of **canceling a primitive** in old-fashioned OpenGL
  - geometry shader can do the job (clip-distance, cull-distance /OpenGL ≥4.5/)
- every primitive would be supplied with information useful for the elimination
  - normal vector of the face
- example of **primitive elimination**
  - all coordinates can be set to **[2, 0, 0, 1]** (outside of the NDS frustum)
  - vertex sharing among primitives must not be used !

Face elimination example



Shadow volume surface

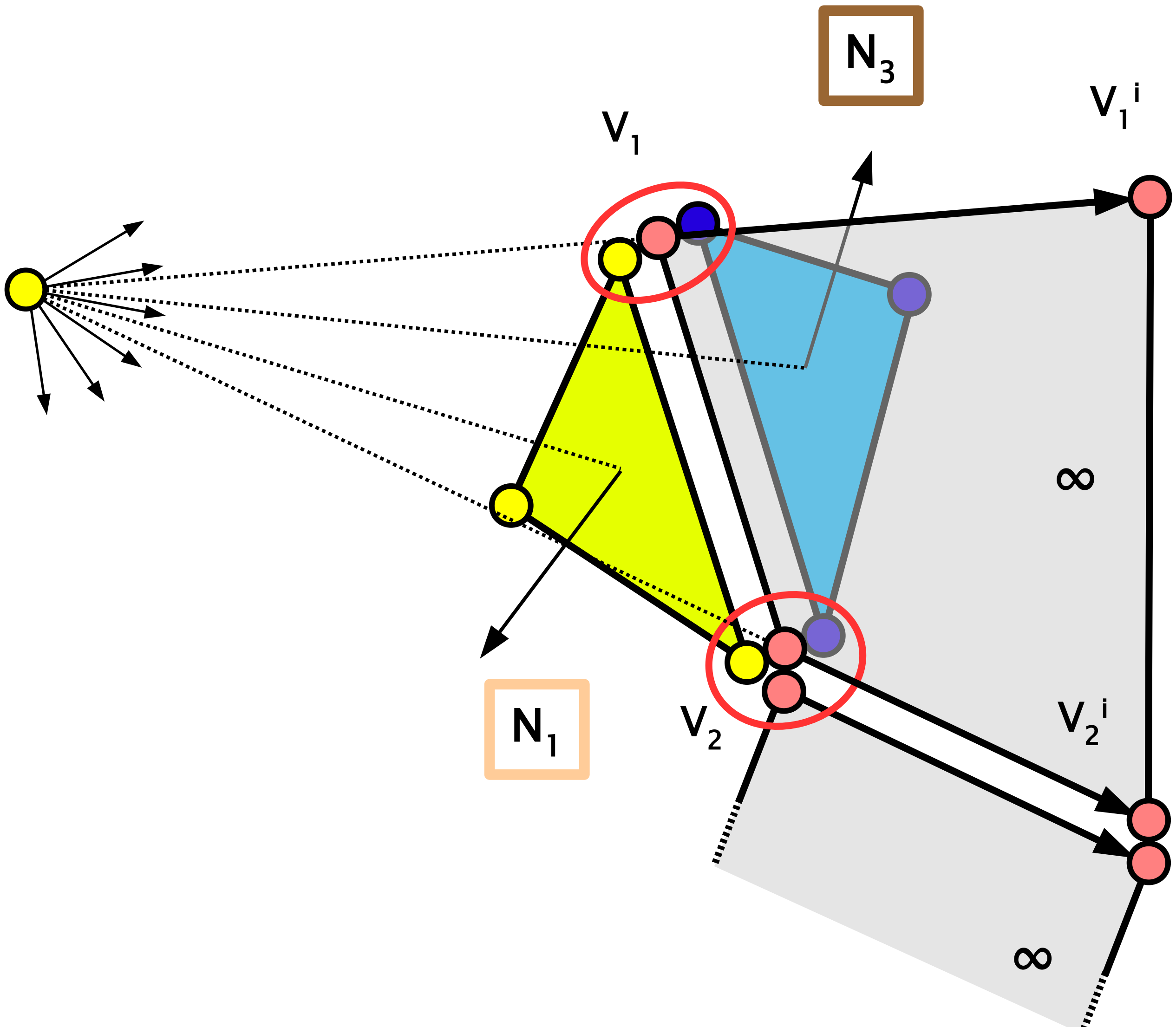
- **infinite quadrangles** projected from contour edges of shadow solid
  - "contours" according to light source (front / back faces)

- if the edge $[\,\mathbf{x_1}, \mathbf{y_1}, \mathbf{z_1}, 1\,] - [\,\mathbf{x_2}, \mathbf{y_2}, \mathbf{z_2}, 1\,]$ is on the **contour**, **infinite quad** will be generated:

$$[\,x_1, y_1, z_1, \mathbf{1}\,],\ [\,x_2, y_2, z_2, \mathbf{1}\,],\ [\,x_2, y_2, z_2, \mathbf{0}\,],\ [\,x_1, y_1, z_1, \mathbf{0}\,]$$

- **contour edge** decision on the GPU
  - every "edge" has additional **4 vertices** (waste..)
  - normal vectors of <u>two incident faces</u> must be present

Shadow volume surface example

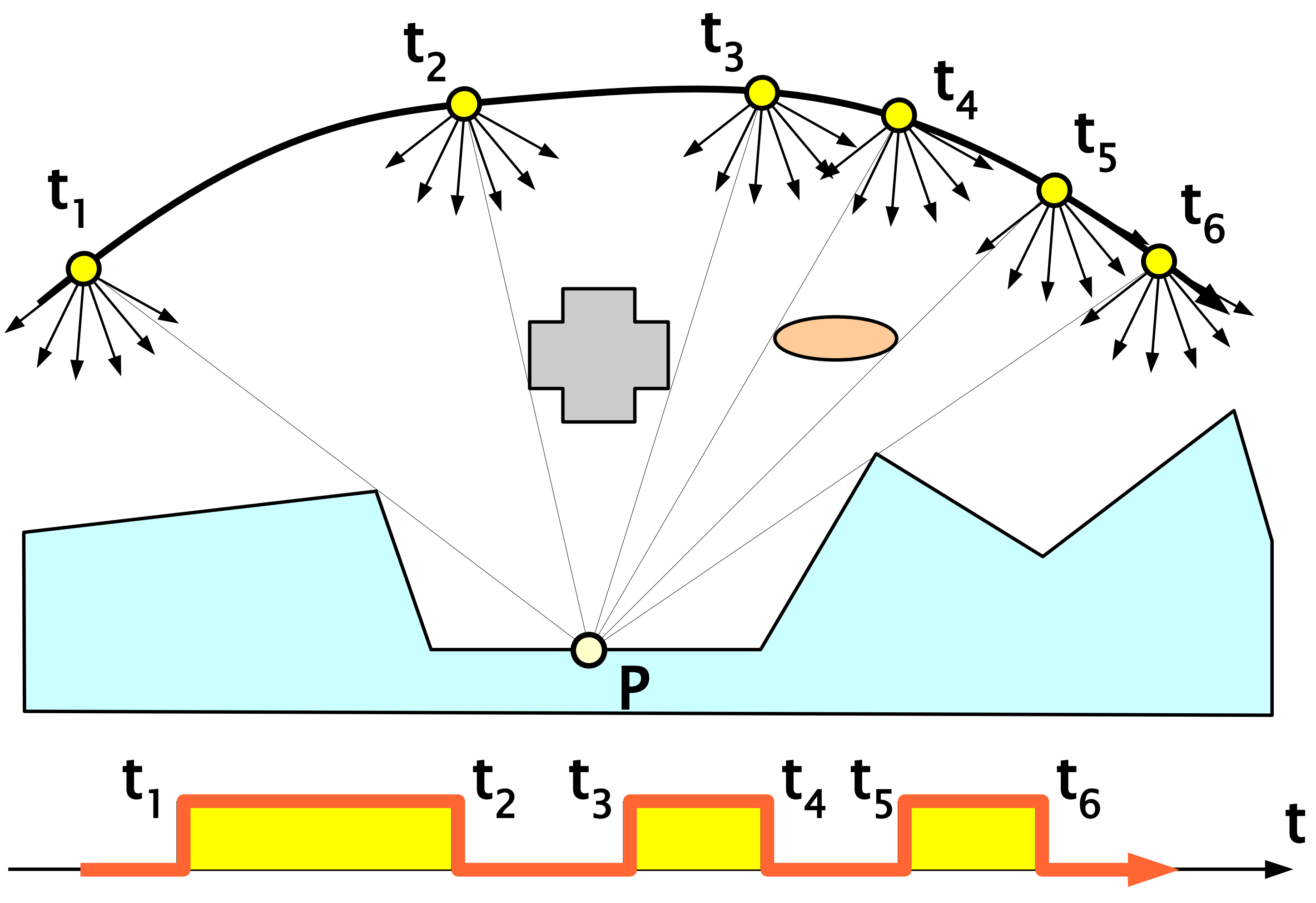


Soft shadows (occlusion interval maps)

- **special** method for **static scene** and light source moving along **static curve**
  - e.g.: static exterier scene and the Sun
- precomputed **occlusion intervals** for every surface point in the scene!
  - **indicator function** for the light (dependent on time)
  - time consuming (stochastic Ray-tracing – 256 rays/px)
  - **result map** stored in special texture (beginnings and ends of time intervals)
  - **soft shadows** are interpolated in real-time on the GPU

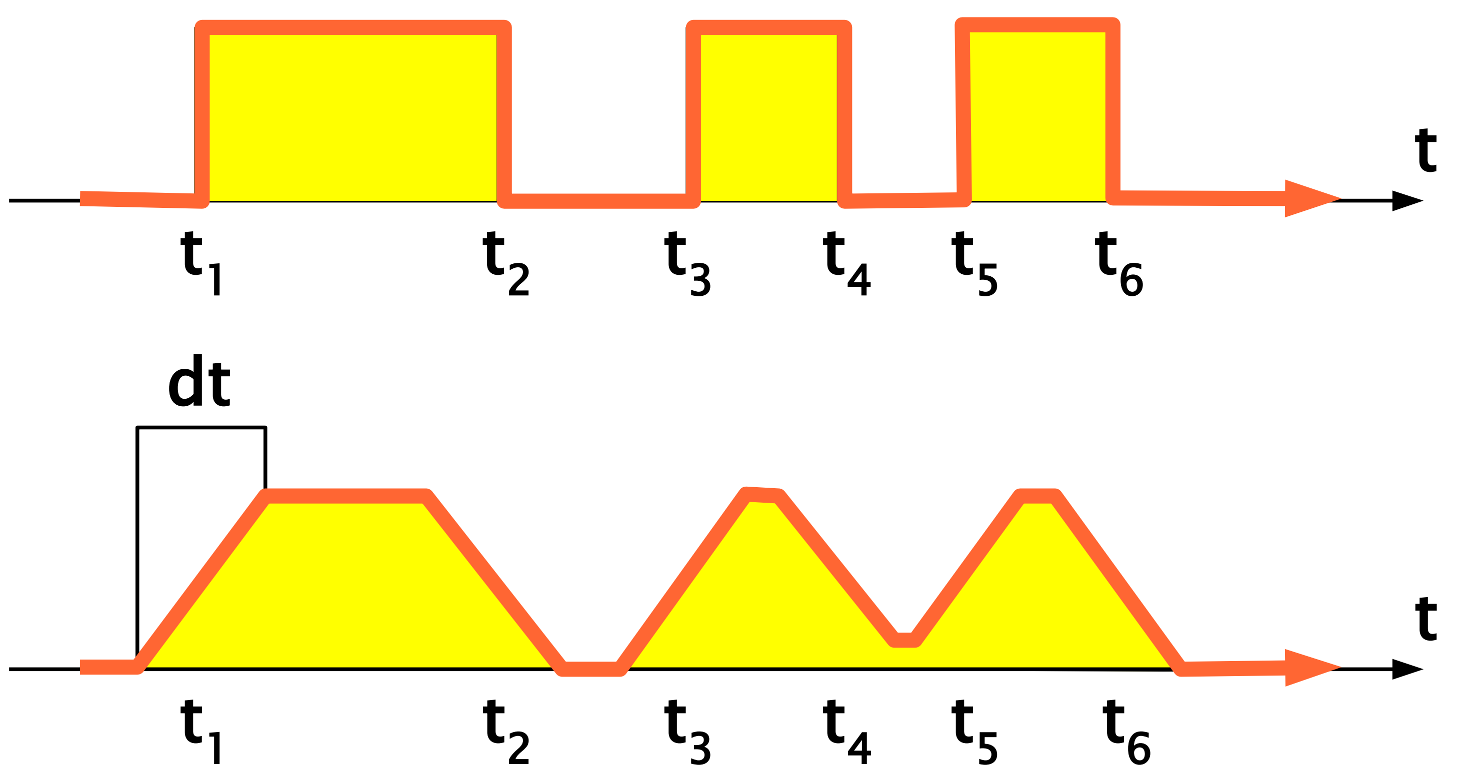
Occlusion intervals

- light source is moving along a **static path**:



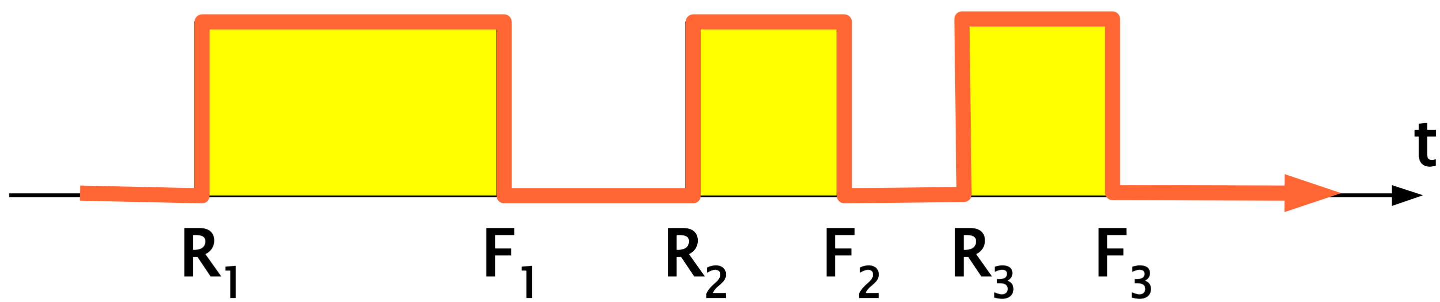
Blurring the occlusion map

- to obtain soft shadows, **original occlusion map** should be blurred (fragment shader on the GPU):



Efficient interpolation

- **beginnings** (“$\mathbf{R_i}$“ - rise) and **ends** (“$\mathbf{F_i}$“ - fall) of the intervals are stored separately in 2 textures:



$$V_{lin}(t) = \int_0^1 V_{point}(u)\, W_{dt}(t-u)\; du$$

$$V_{lin}(t) = \sum_{i=1}^{n} \frac{1}{dt} \cdot max\left(0,\;\; min\left(t+\frac{1}{2}dt\,,\, F_i\right) - max\left(t-\frac{1}{2}dt\,,\, R_i\right) \right)$$

Fragment shader for interpolation

- R_i and F_i passed in **two textures** (up to 4 intervals)
- “**t-dt/2**“, “**t+dt/2**“ and “**1/dt**“ are **uniforms**

```
half softShadow ( sampler2D riseTex,
                  sampler2D fallTex,
                  float2 texCoord,
                  uniform half intStart,    // t-dt/2
                  uniform half intEnd,      // t+dt/2
                  uniform half intInvWidth ) // 1/dt
{
  half4 rise = h4tex2D( riseTex, texCoord );
  half4 fall = h4tex2D( fallTex, texCoord );
  half4 minT = min( fall, intEnd );
  half4 maxT = max( rise, intStart );
  return dot( intInvWidth, saturate( minT - maxT ) );
}
```

Results

CSG rendering on the GPU

- elementary solids converted to **polyhedra**
- **set operations** evaluated on the GPU:
  - **union** is trivial (default depth-buffer based rendering)
  - **intersection** and **subtraction**: use of stencil buffer, considering <u>front</u> vs. <u>back</u> faces
- 1989: **Goldfeather** et al.
  - **normalization** of a CSG tree – decomposition to union of "**products**" (intersections and differences)
  - implementation uses **several depth-buffers** and a **stencil buffer** (needs to copy depth-buffers)

Sequential convex subtraction

- 2000: **Stewart** et al. – Sequenced Convex Subtraction ("**SCS**")
  - does not need depth-buffer copying, complex depth-tests
  - all elementary solids have to be **convex**
  - **O(n)** – intersection of **n** solids
  - **O(n²)** – difference of **n** solids (**O(kn)** limited occlusion)

- algorithm phases
  1. **preprocessing** (CSG normalization, sorting of subtraction sequences /front-to-back/)
  2. **depth-buffer processing** (for every product + merge)
  3. final rendering to **frame-buffer**

Intersection of n solids

- init: `depth = near; stencil = 0;`
- passing through **front faces** of individual solids

```
if ( front > depth ) depth = front;
```

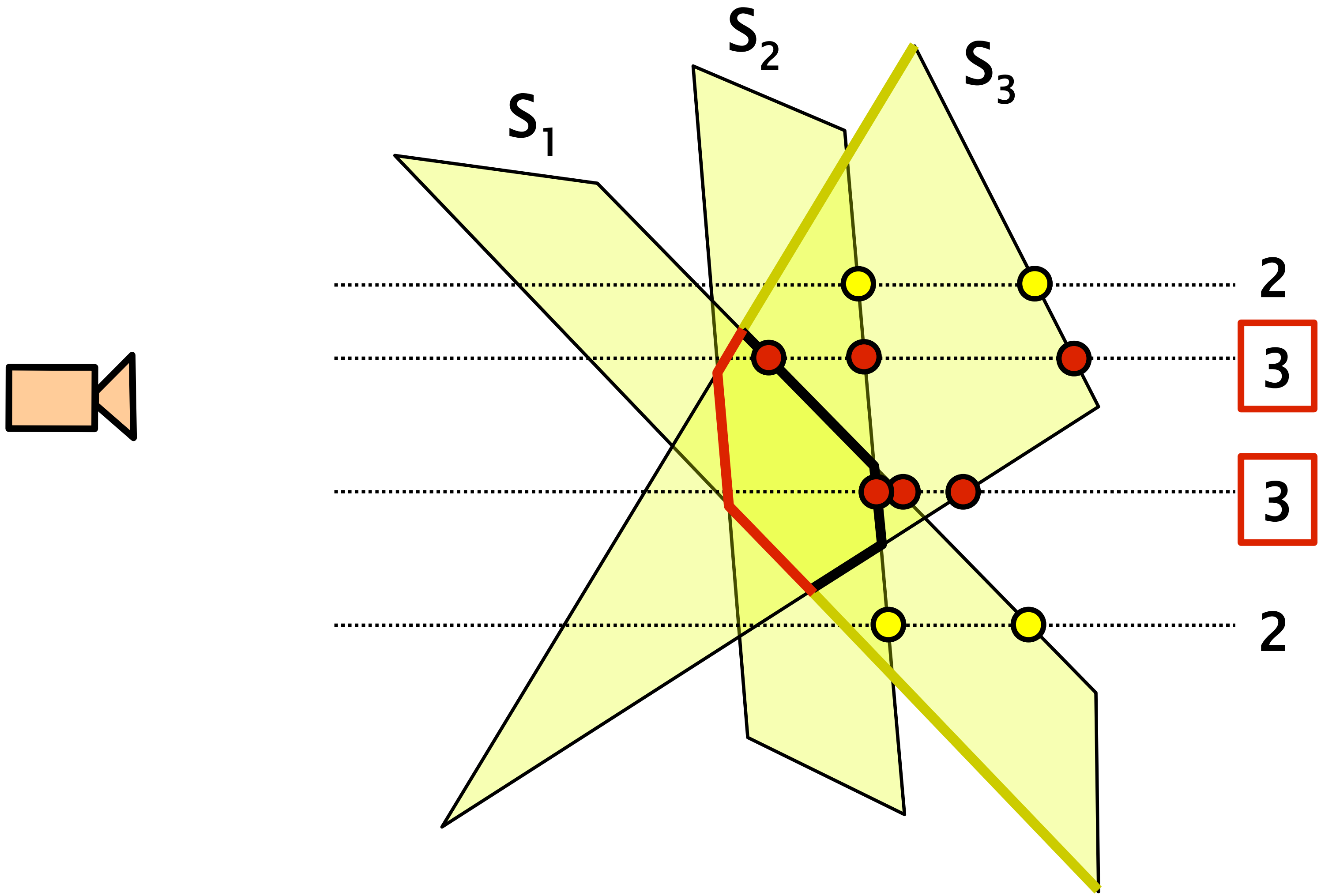
- passing through **back faces** (occlusions)

```
if ( back > depth ) stencil++;
```

- removing pixels with **occlusion number < n**

```
if ( stencil < n )
{
  stencil = 0;
  depth = far;
}
```

Intersection – example



Subtracting sequences

- determining correct **subtracting sequence**
  - **front-to-end** subtraction
  - e.g. **X – A – B** is replaced by universal **X – A – B – A**
  - **A,B,A** is a correct universal **subtracting sequence**
  - see “sequences containing all occlusion permutations“
- subtracting **from the front**: passing all subtr. solids

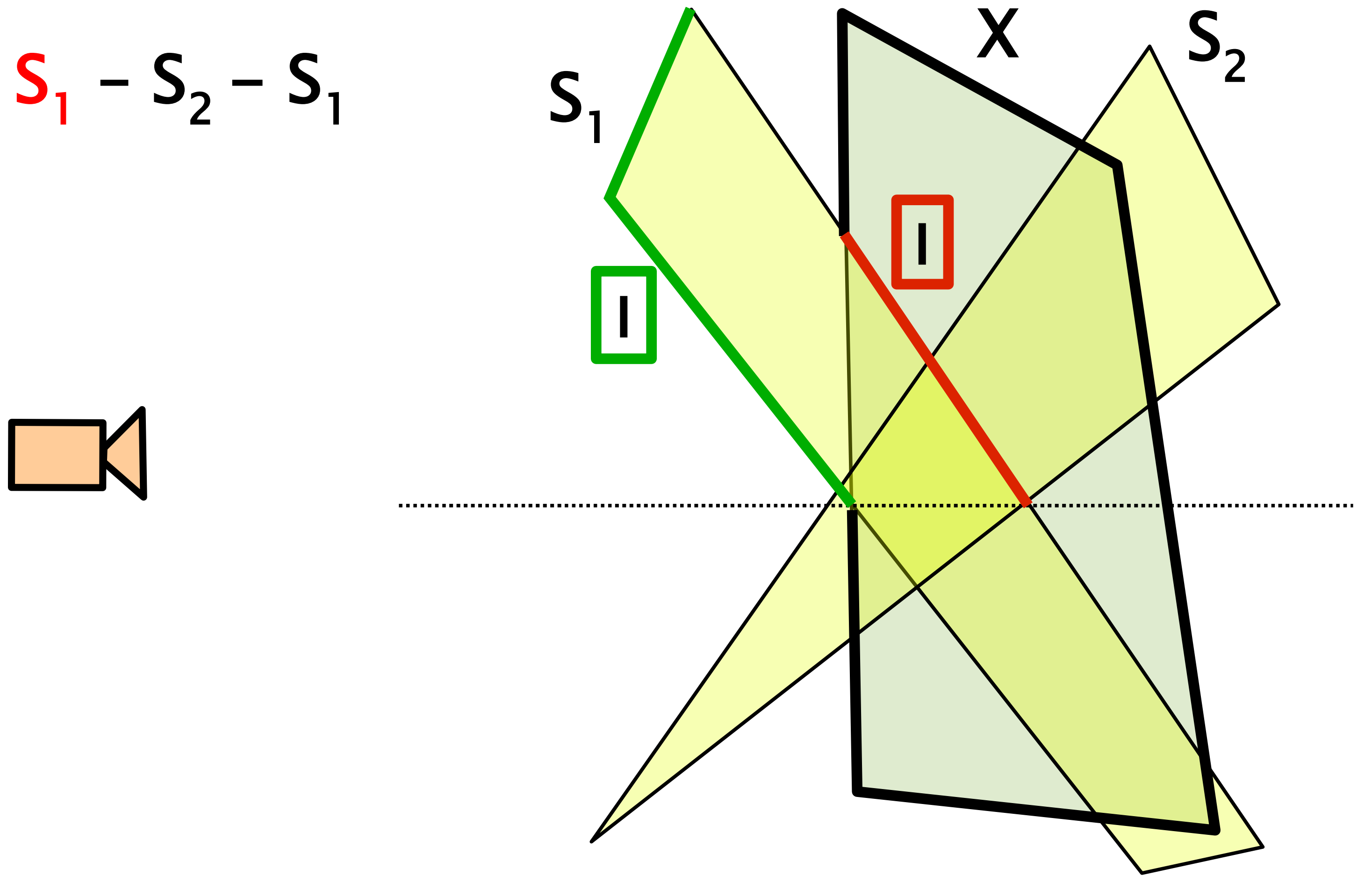
```
if ( front < depth ) stencil = 1;
else                 stencil = 0;
```

- every **back-face** is processed immediately as well

```
if ( back > depth && stencil == 1 )
  depth = back;
```

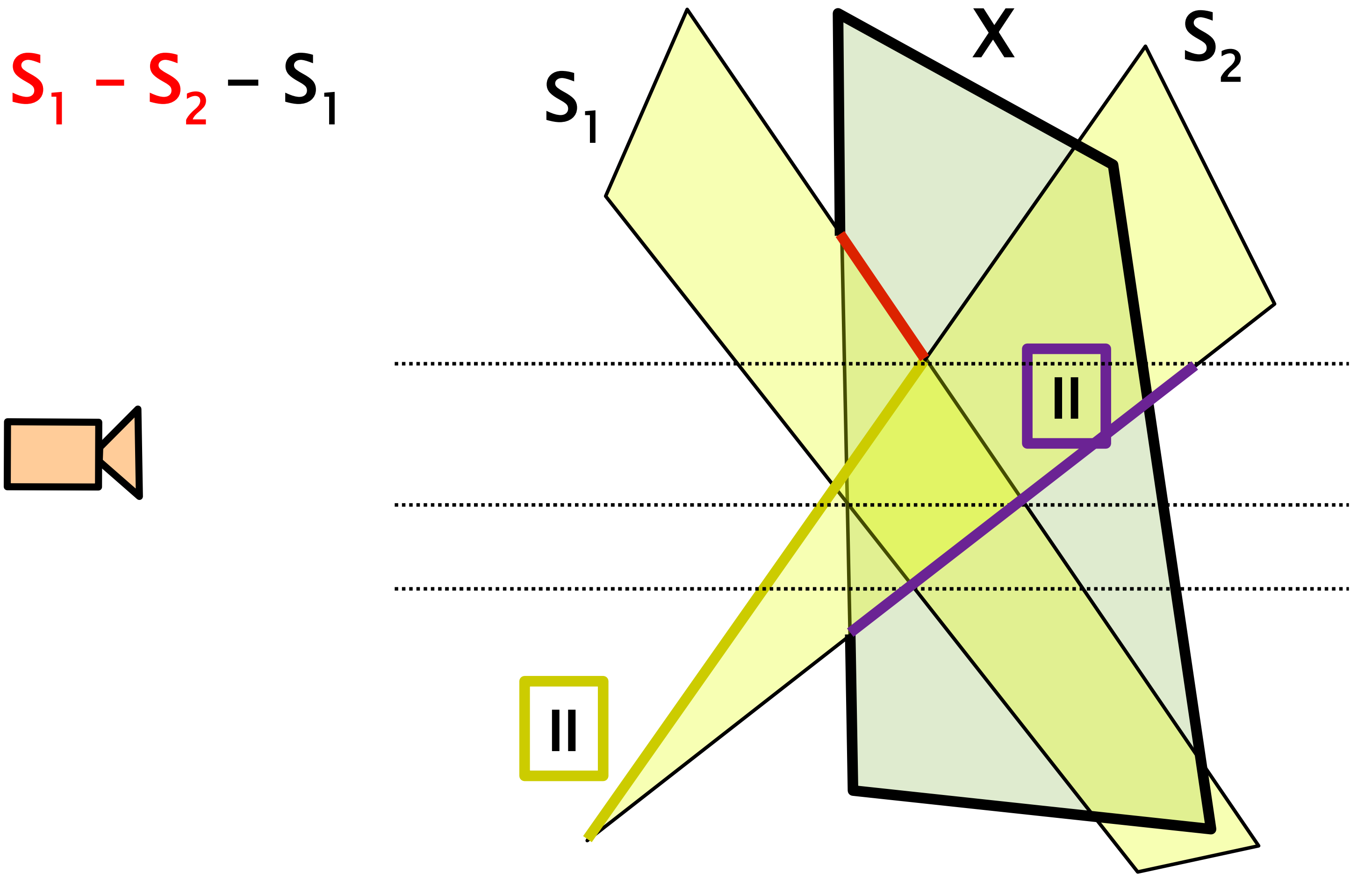
Subtraction – step I

$X - S_1 - S_2 - S_1$



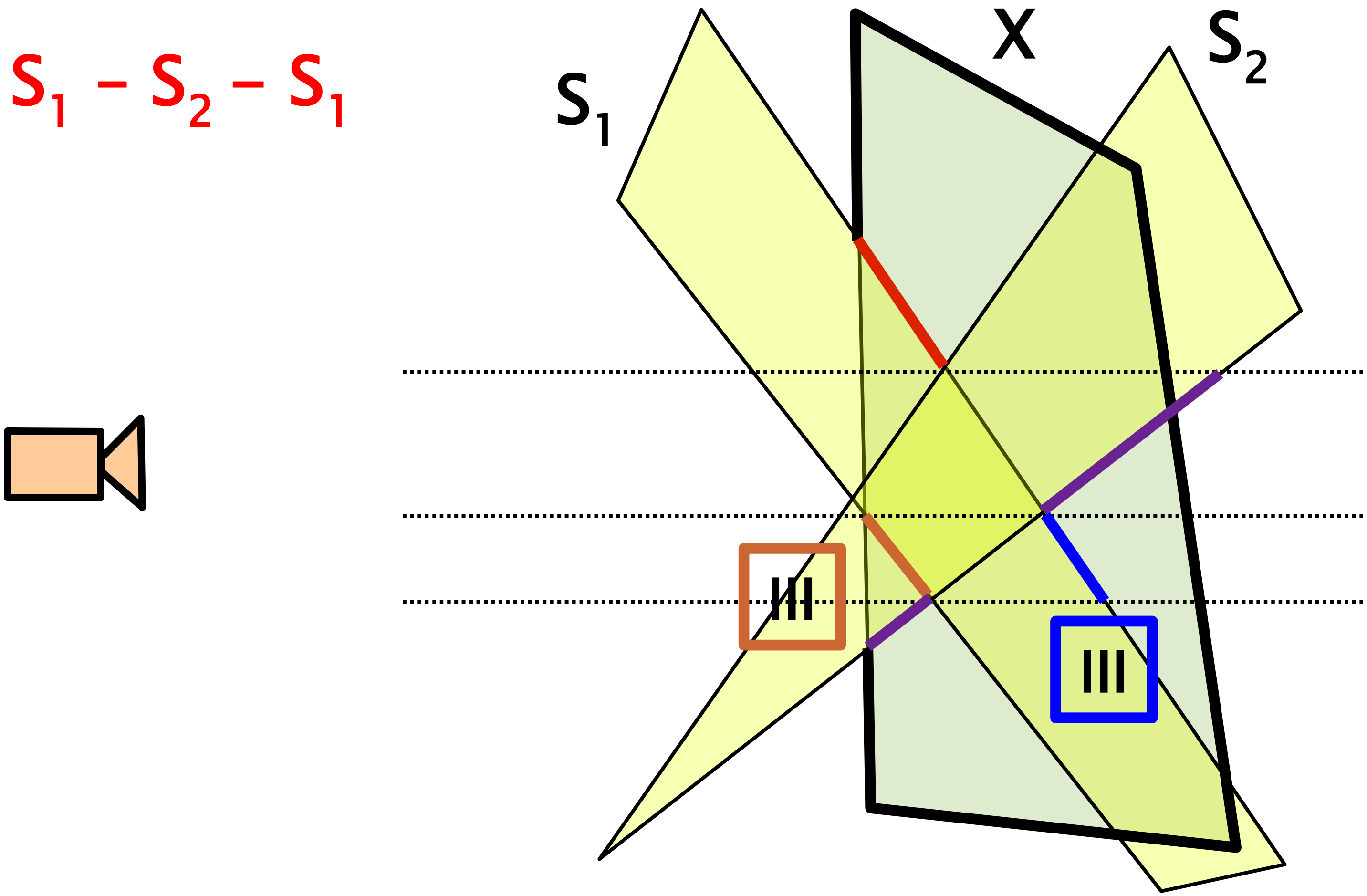
Subtraction – step II

$X - S_1 - S_2 - S_1$



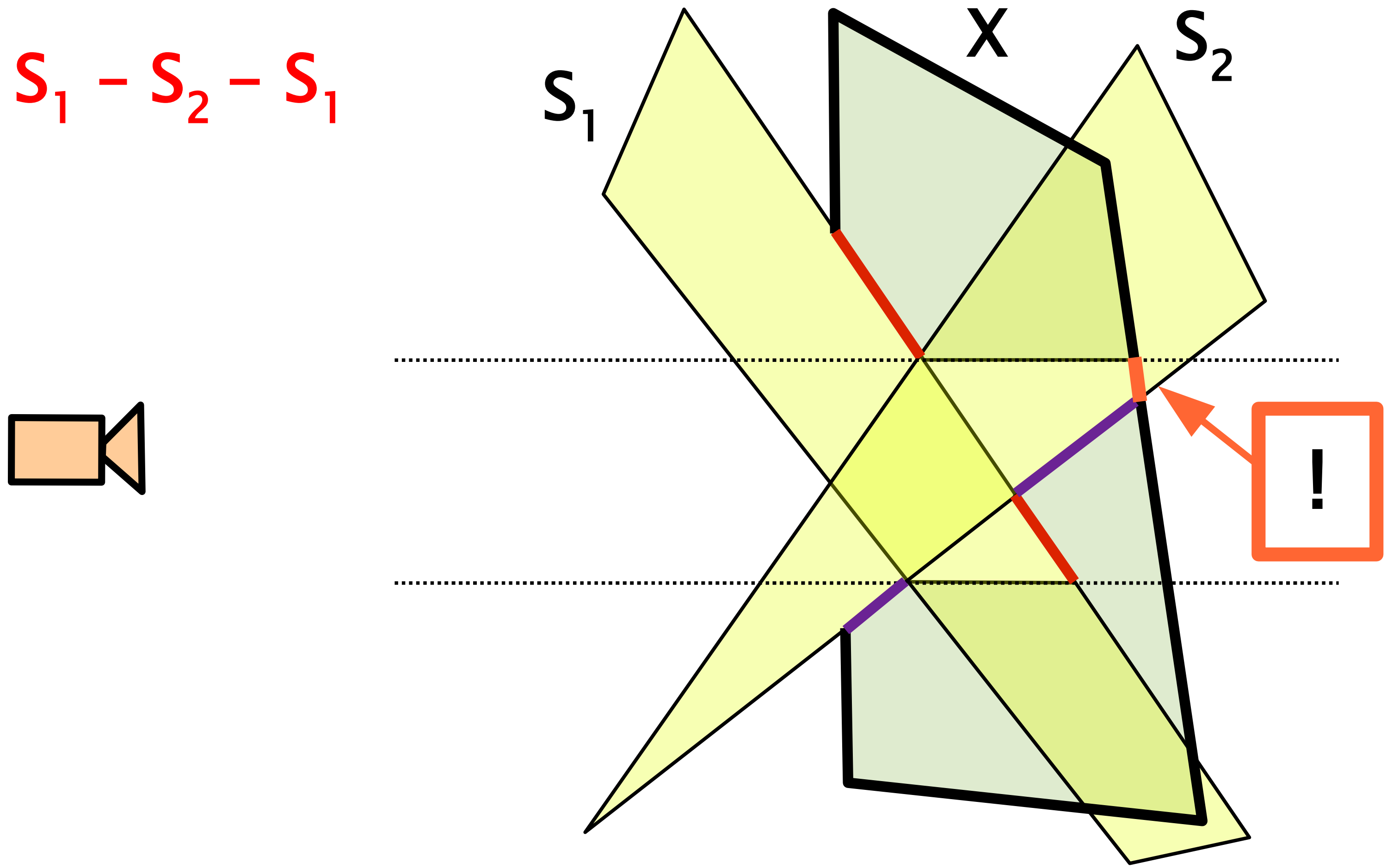
Subtraction – step III

$X - S_1 - S_2 - S_1$



Subtraction – result

$X - S_1 - S_2 - S_1$



Completely subtracted parts

- removing parts of **common intersection**, which were eliminated completely:
- init: `stencil = 0;`
- passing through all **intersection solids** (back faces only – looking for empty results)

```
if ( back < depth ) stencil = 1;
```

- elimination of **completely subtracted parts**

```
if ( stencil == 1 ) depth = far;
stencil = 0;
```

Merging products & final rendering

- **product result** = its ”**depth buffer**“
- merging results of one product (i.e. **union** operation)

  `if ( depth < depth`$_{total}$` ) depth`$_{total}$` = depth;`

- **final rendering**
  - different logic for intersections and subtractions
- **intersected solid** (for every pixel):

  `if ( front == depth`$_{total}$` ) draw( front );`

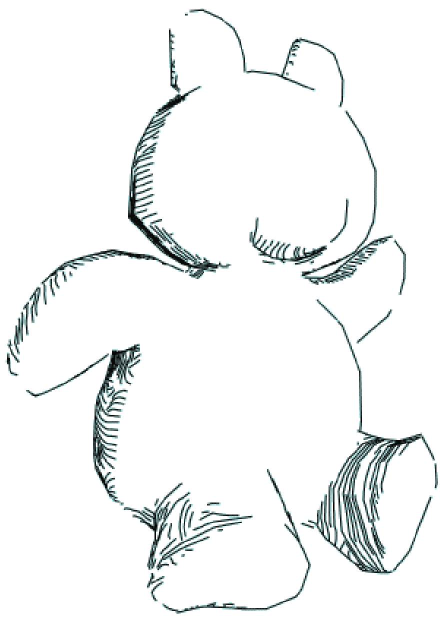
- **subtracted solid** (for every pixel):

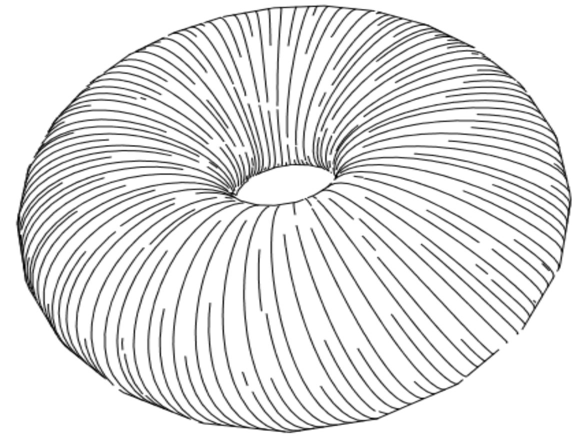
  `if ( back  == depth`$_{total}$` ) draw( back );`

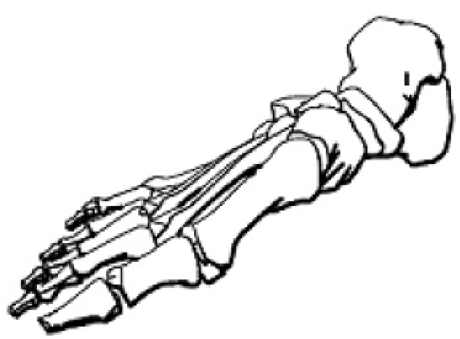
Non-photorealistic rendering (NPR)

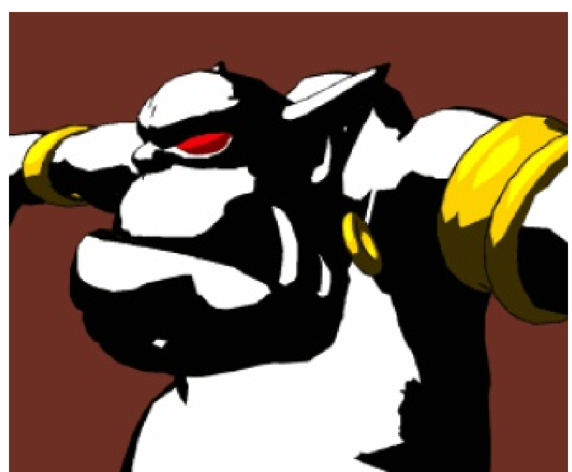
- **goal:** results similar to human 2D graphics
  - contour emphasis
  - **pen-and-ink drawing** simulation (hatching)
  - imitation of **painting techniques** (oil, watercolor)
  - "**cartoon-style**" shading
- **approaches** (techniques)
  - special **textures** (coarse shading tones, ..)
  - **procedural textures** (fragment shader)
  - **post-processing** (for specific painting techniques)
  - \+ combinations

NPR examples









Contours, silhouettes

- very important for **human vision** system
  - borderline between front-facing and back-facing parts
  - often connected to a **hatching system** (emphasizing curvature, slope of the surface or just for shading)
  - purely **geometric information** (for polyhedra)
- **contouring methods**
  - edges between **front-faces** and **back-faces**
  - discontinuities of the **depth-buffer** (post-processing)
  - **discontinuities** (edges) in other output data (see deferred shading, multiple output targets, ..)

Simple contouring method

- **no need** for **explicit definition of contours**
  - solids have to be regular (closed)
  - two phases

1. **front-facing** faces only
   - no special rendering style
   - using "depth-buffer"
   - see "`glEnable( GL_CULL_FACE )`", "`glCullFace()`"
2. **edges** of **back-facing** faces only
   - more **thick line** ("`glLineWidth()`") – contour lines will stick out

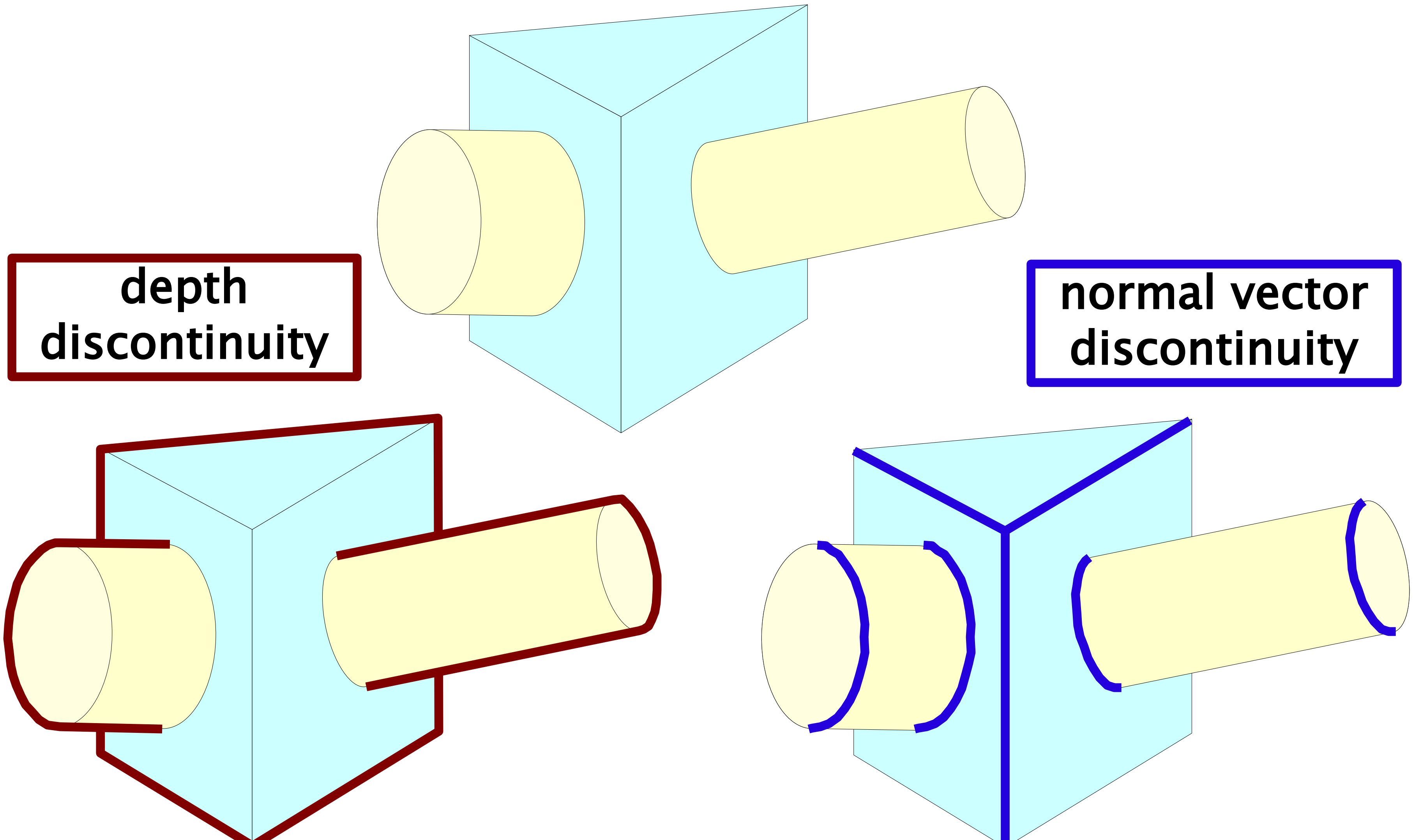
Image processing contours

- **post-processing** of regularly rendered 3D scene
  - source: **depth buffer**, **normal map**, combinations, ..
- restricted **Sobel filter** works well (2 directions only):

$$S_h = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \qquad S_v = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

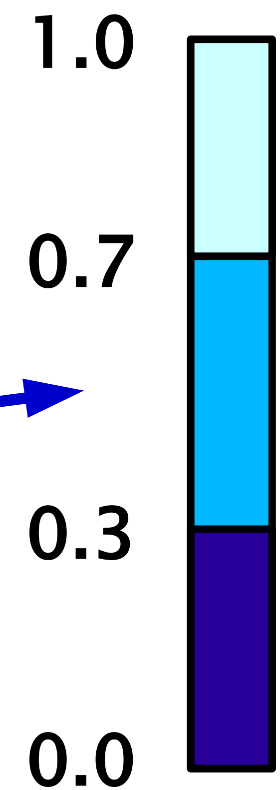
$$Edge \approx \sqrt{H^2 + V^2}$$

Examples – depth and normals



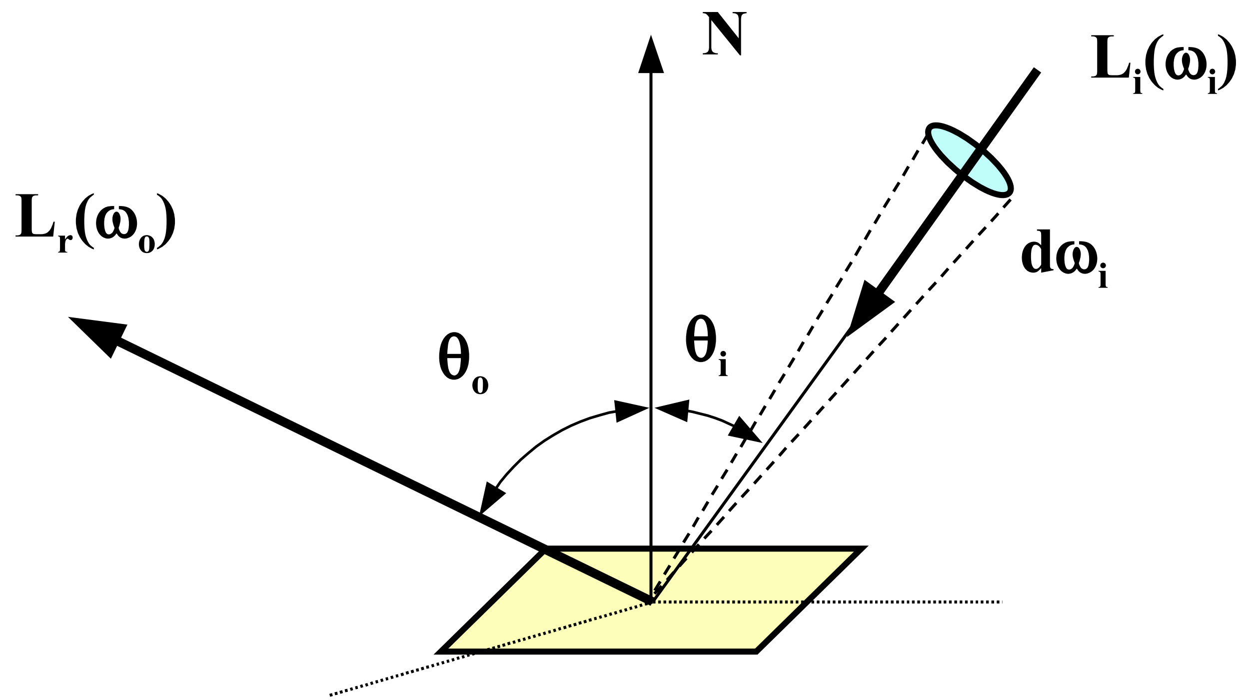
Cartoon–style

- **light model** similar to “Blinn-Phong“
  - diffuse term “**cos α**“
  - optional specular term “**cos^{h} β**“
- **diffuse term** indexes simple “**ramp texture**“
  - only small number of color tones
  - no texture filtering for sharp outlines!
- optional **specular term** with priority
  - thresholding for white-color highlight



BRDF (local reflectance)

(“Bidirectional Reflectance Distribution Function“)



$$f_r(\omega_i, \omega_o) = \frac{\partial L_r(\omega_o)}{\partial E(\omega_i)} = \frac{\partial L_r(\omega_o)}{L_i(\omega_i) \cos\theta_i \, \partial \omega_i}$$

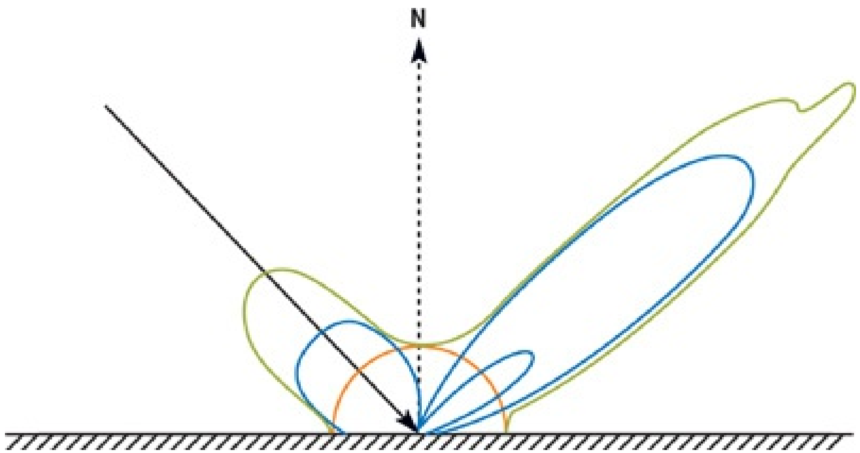
Example of more complex BRDF

- 1977: **Lafortune** introduces efficient reflectance function representation using "**lobes**"
  - based on term similar to "$\mathbf{cos^n \beta}$"
  - a "**lobe**" is represented by a function "$\mathbf{s(\omega_i,\omega_o)}$"
  - lobe direction can be derived from incoming and reflected vector, **"C" vector** is used for the definition
  - **tangent coordinate space** [**t**,**n**,**b**] is used
  - exponent "**n**" defines lobe width

$$f(\omega_i \rightarrow \omega_o) = \rho_d + \sum_j \rho_{s,j} \cdot s_j(\omega_i, \omega_o)$$

$$s(\omega_i, \omega_o) = \left(C_t \omega_{i,t} \omega_{o,t} + C_n \omega_{i,n} \omega_{o,n} + C_b \omega_{i,b} \omega_{o,b} \right)^n$$

Lafortune model – example



© 2004, David McAllister, NVIDIA

Lobe orientations

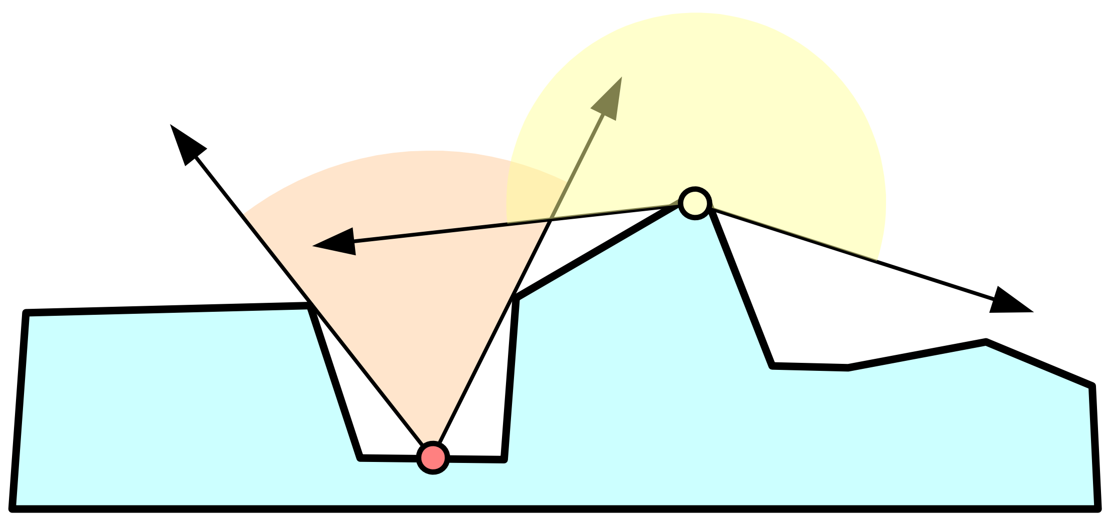
- $C_t = C_b = -1,\ C_n = 1$
  - usual **Phong lobe** (rotated by 180° around normal)
- $C_t = C_b$
  - **isotropic BRDF** (surface orientation does not matter)
- $|C_n| < |C_t|$
  - **non-mirror** specular **maximum** (closer to tangent)
- $C_t > 0, C_b > 0$
  - **back-reflection** (see Oren-Nayar model)
- $\mathrm{sign}(C_t) \neq \mathrm{sign}(C_b)$
  - **anisotropic reflection** (brush strokes, rifts, grinding)

Implementation

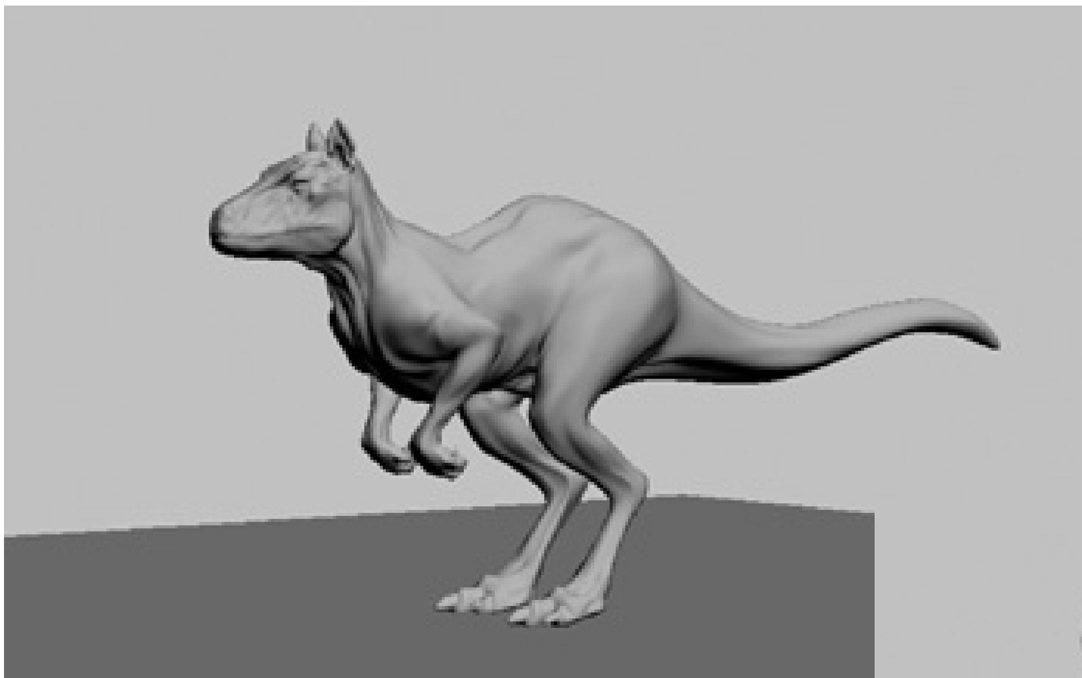
- **reflection factors** (~"albedo") $\boldsymbol{\rho}$ are [R,G,B] triples
  - stored separately for each term (one texture per term)
- **four lobe parameters** [$\mathbf{C_t}$, $\mathbf{C_n}$, $\mathbf{C_b}$, $\mathbf{n}$] in one texture
  - **one to three lobes** sufficient for realistic BRDF
- **environment map**
  - environment image can be **blurred** in pre-processing phase, using **exponent** $\mathbf{n} = \mathbf{0}, \mathbf{1}, \mathbf{4}, \mathbf{16}, \mathbf{64}$ and $\mathbf{256}$
  - in different MIP-map levels or in a 3D texture

Ambient occlusion

- constant “**ambient term**“ is not good enough
  - does not consider occlusion (even self-occlusion)
  - “ridges” are equally lighted as “valleys”
- pre-computed **average (potential) contribution** of surround light to the surface point

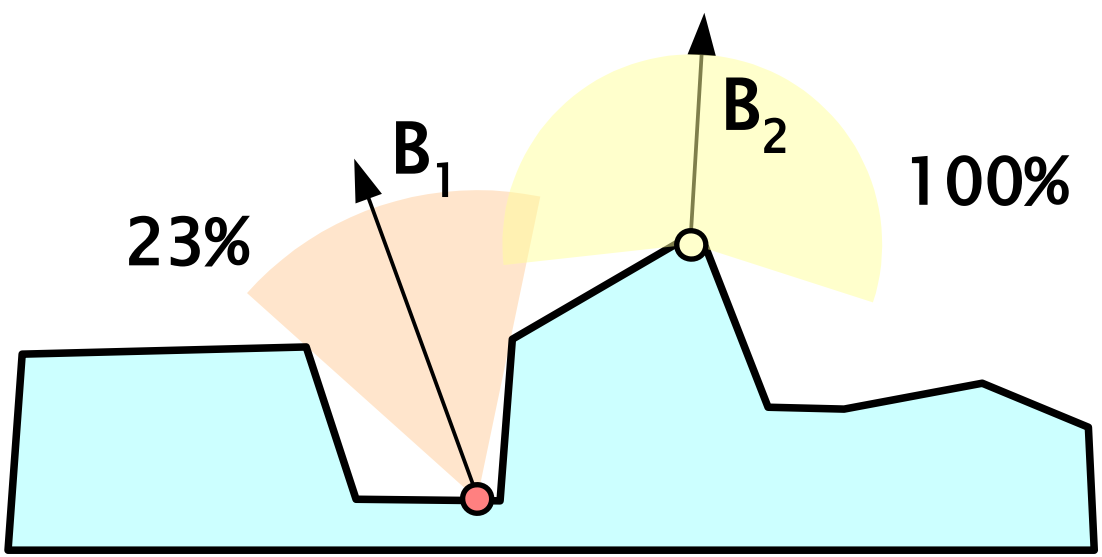


Ambient occlusion example



Pre-processing: accessibility

- for **every surface point** compute:
  - **percentage** of unoccluded rays from an environment (self-occlusion elimination) - "accessibility coefficient"
  - **dominant light direction** ("best lit from") – "**B**"
  - **technique**: Ray-tracing or special GPU computation



Accessibility map utilization

- **accessibility coefficient**
  - multiplication factor for ambient light approximation (instead of the "$\mathbf{k_A}$" constant)

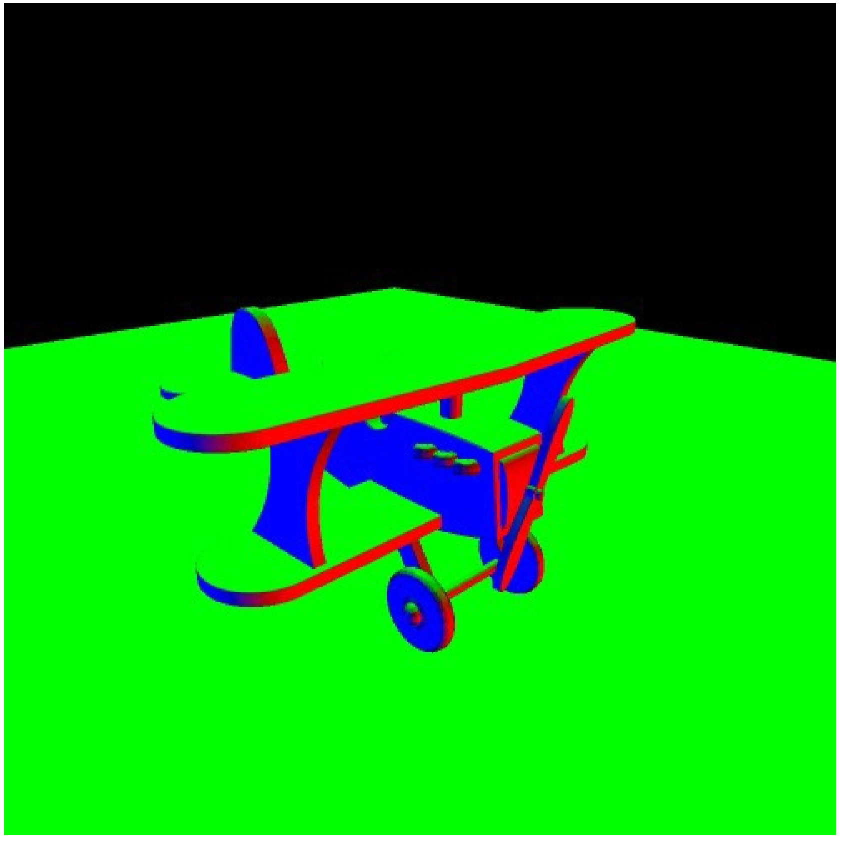
- **dominant vector "B"**
  - addressing for the "environment light map"
    - map should be blurred in advance ("**cos α**")
  - texture data are multiplied by the **accessibility coefficient** as well

Accessibility example I

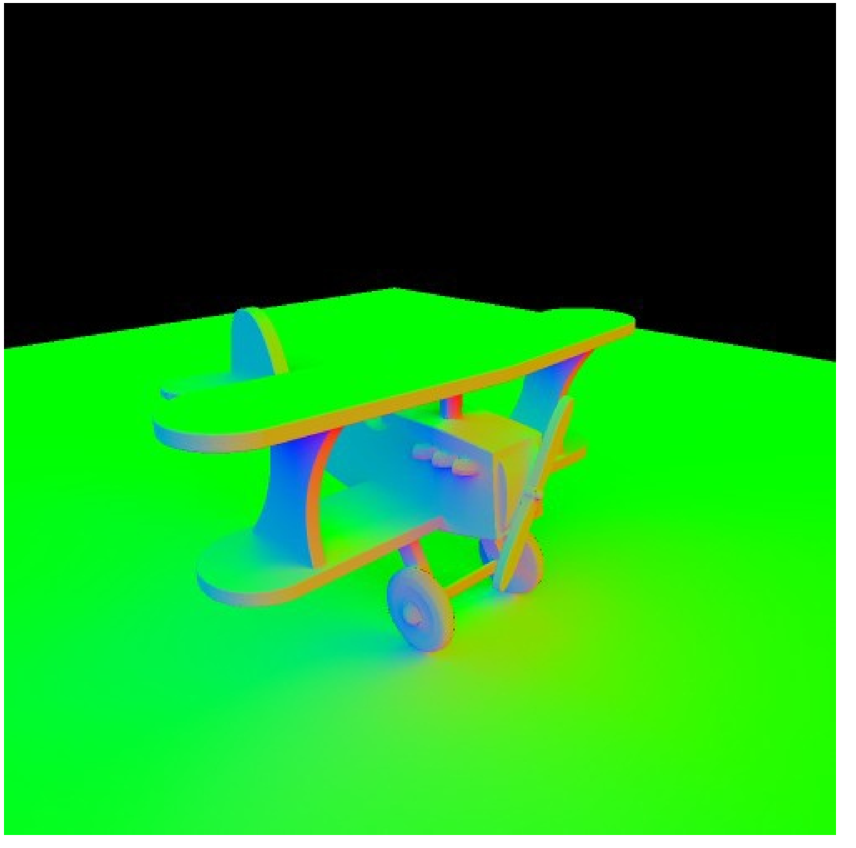
Phong shading

Accessibility coefficient

Accessibility example II (normals)



Model normals



Average unoccluded ray B

Accessibility example III (environment)

Phong shading

Environment lighting

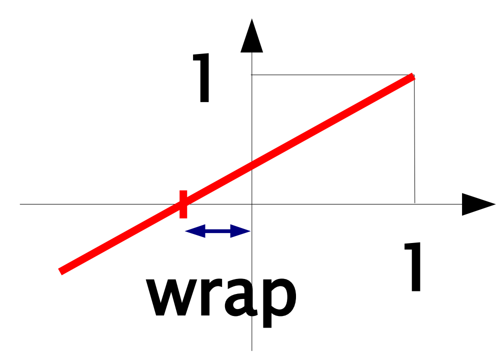
Subsurface scattering

- very important for “**photo-realism**“
  - human skin (“Shrek 2“, “Finding Nemo“)
  - other transcluent materials (wax, milk, marble, amber,..)
  - **precise implementation** is very expensive (see “Participating media” term in photorealistic graphics)
- **simplified approaches** in real-time graphics
  - “**wrap lighting**“ – lighting extends “around the corner“
  - absorption simulation using “**depth map**”
  - absorption computed in **tangent space**

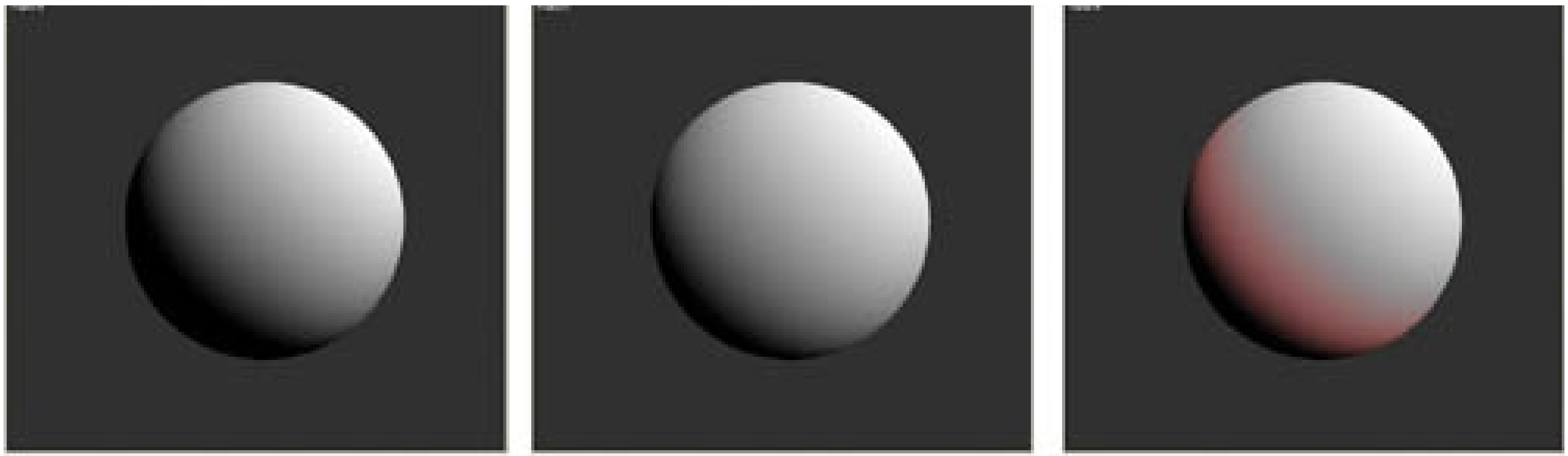
Wrap lighting

- naïve method
  - ignores shape and thickness of the object
  - does not try to compute light diffusion at all
- modifies the **diffuse term "cos α"** – extends its influence to **adjacent not illuminated** parts of the surface (behind the "terminator")
  - simple linear transform of the dot product $\mathbf{L} \cdot \mathbf{N}$
  - **tint of the transition** can be added (reddish for skin)

$$Diff = max\left(0, \frac{\cos\alpha + wrap}{1 + wrap}\right)$$



Wrap lighting example



Regular shading Light wrap Tinted wrap

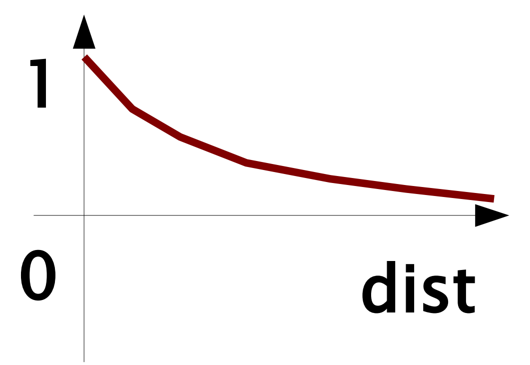
© 2004 Simon Green, NVIDIA

Absorption using depth-maps

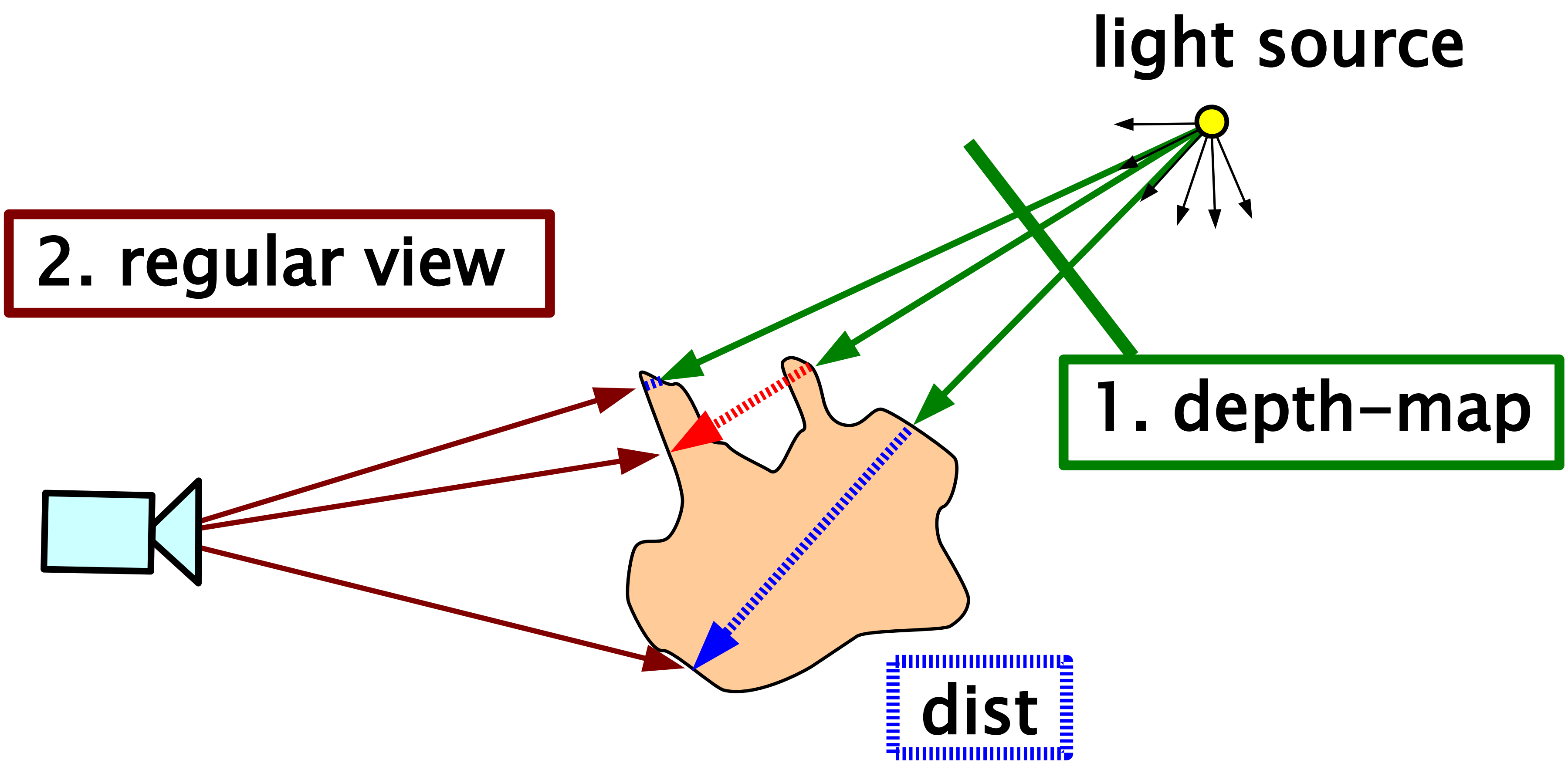
- HW-implemented “**depth-map**”
  1. “depth-map” from the point of view of **light source**
  2. solid **thickness** is known in render-time (fragment-shader)

- thickness is used for **attenuation approximation**
  - simple **exponential dependency** (can be cached in 1D texture)

$$Scatter = C_{light} \cdot e^{-\sigma \cdot dist}$$



Depth-map attenuation



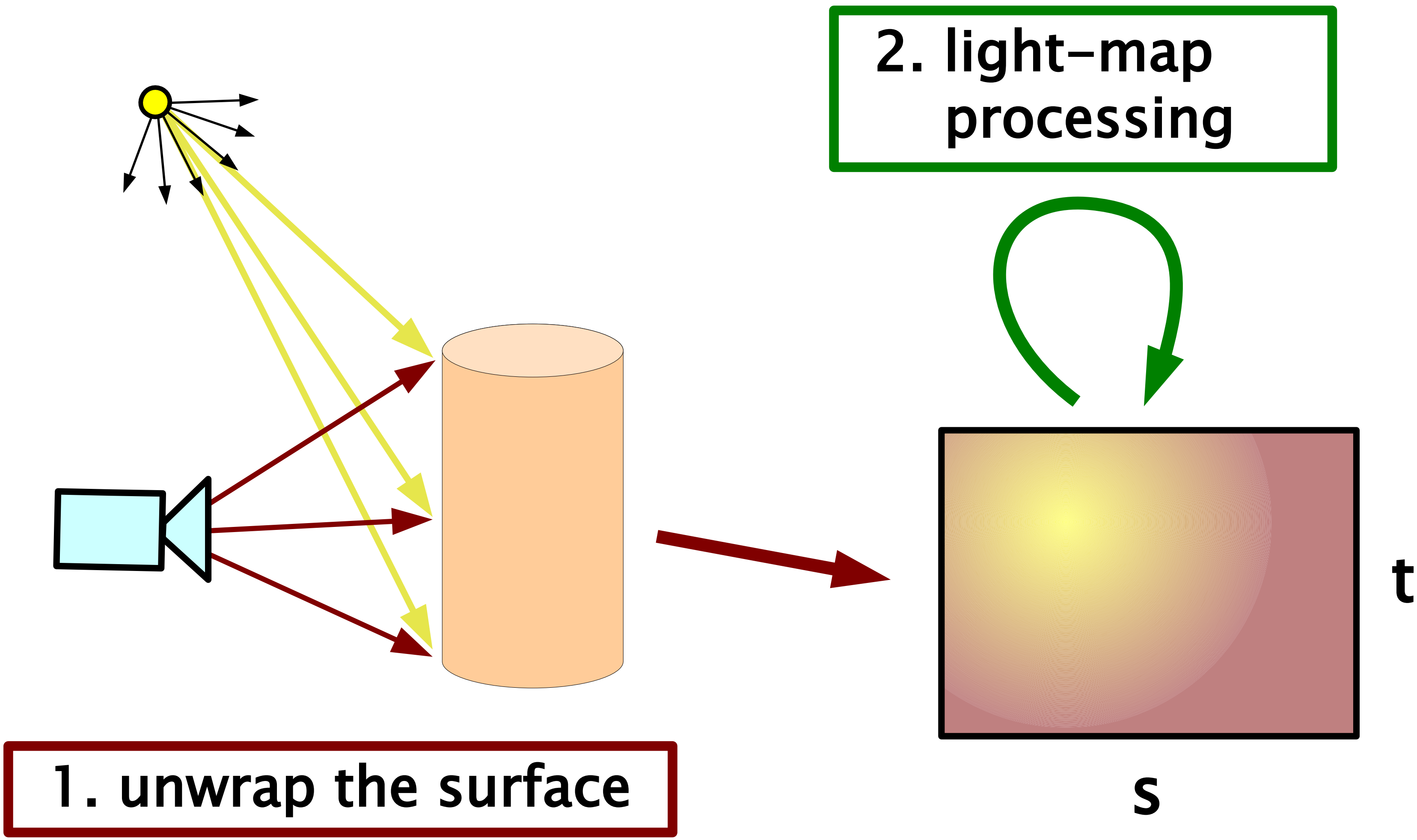
Depth-map example

© 2004 Simon Green, NVIDIA

Texture-space approaches

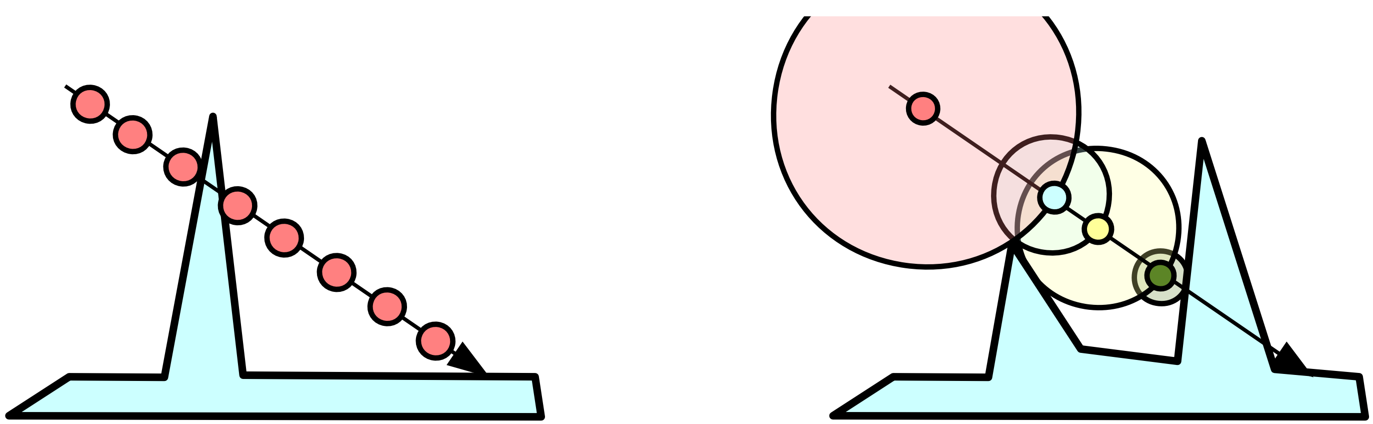
- **1st pass:** primary lighting, results are written to a texture [**s**, **t**] (see GPU technique "render targets")
  - vertex shader must provide texture coordinates and transform them into the NDS = **[-1, 1]²**
  - **good quality parametrization** of the surface !
  - fragment shader needs regular 3D coordinates as well
- **subsequent passes:** light-map processing (digital image filtering), computing capabilities ?
- **the last pass:** regular rendering
  - **light map** is used as a texture

Light map



Displacement mapping

- concept by **Ken Perlin** (1989, "**hypertexture**", rendered using new "**ray marching**" method)
  - surface point position is modulated by a "**displacement function**"
  - actual modification of point position (vs. "bump map")
- fragment position is computed by "**sphere tracing**" (Hart 1996) – originally for implicit surfaces (point-surface distance)



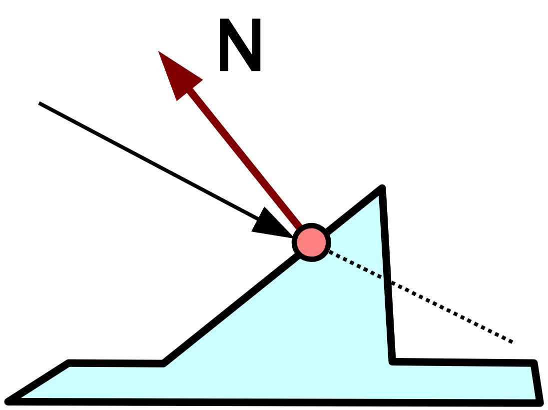
Distance volume implementation

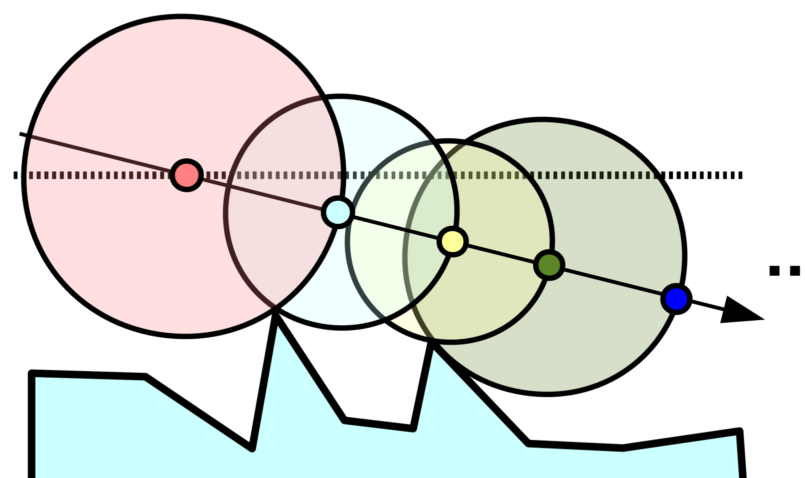
- ray casting in **texture coordinates**
  - **3D tangent space** with unit = 1 texel
  - **init**: direction vector computation ("**dir**")
- "**distance map**": each point receives distance to the closest real-surface point ("$\mathbf{R}^3 \rightarrow \mathbf{R}$")
  - pre-computation (Danielsson 1980 – **O(n)** time)

```
float3 dir = normalize( in.tanEyeVec );
float3 texCoord = in.texCoord;
for ( int i = 0; i < NUM_ITERATIONS; i++ )
{
  float dist = f1tex3D( distanceTex, texCoord );
  texCoord += dist * dir;
}
```

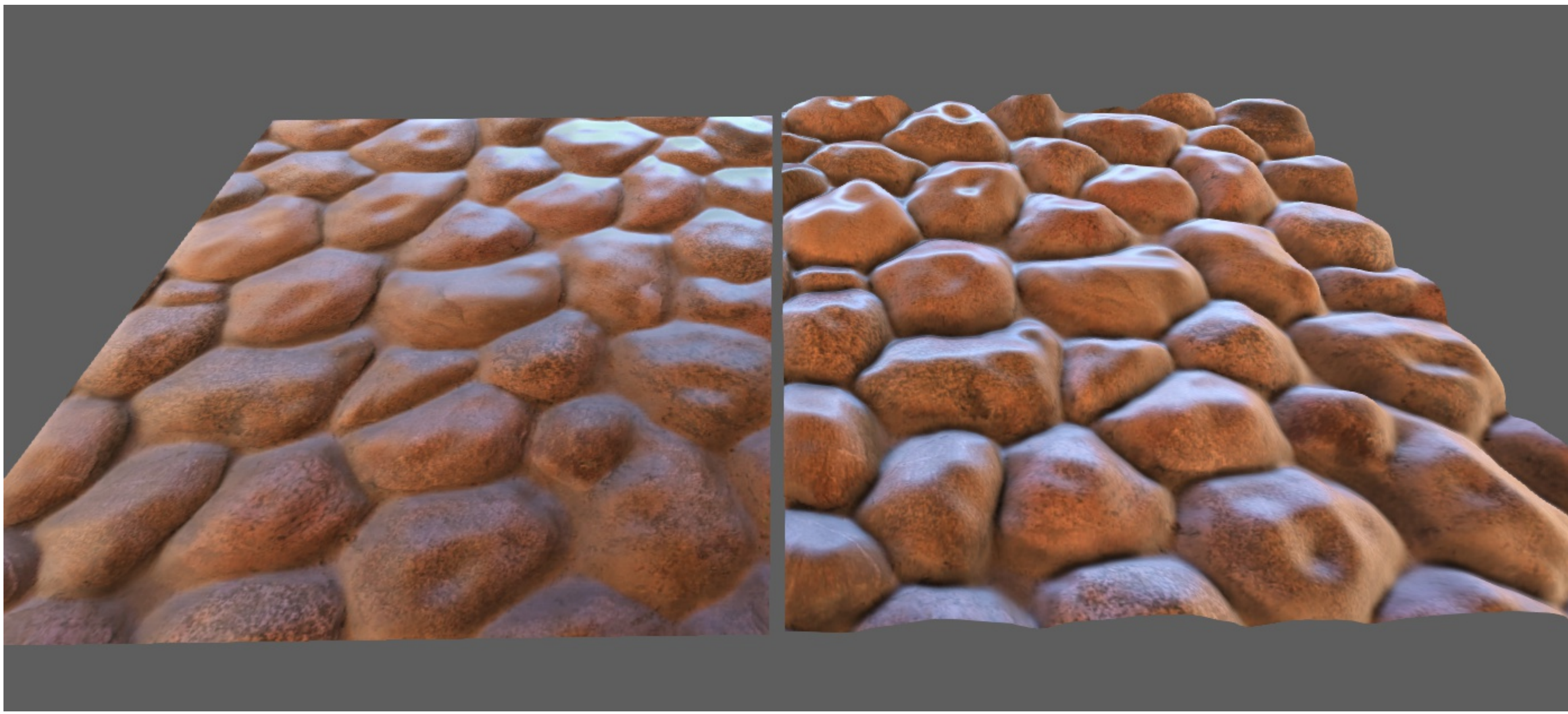
Finishing

- after **N iteration steps** we have the result:
  - the ray **hitted the surface** (normal vector, lighting, ..)
  - the ray **missed the surface** (completely transparent fragment is returned)
    - virtual "GPU geometry" should be bigger than simulated shape





Parallax mapping/occlusion example



© ΩMEGA, YouTube (from Skyrim)

Sources I

- Tomas Akenine-Möller, Eric Haines: ***Real-time rendering, 2nd edition***, A K Peters, 2002, ISBN: 1568811829
- Randima Fernando, Mark J. Kilgard: ***The Cg Tutorial***, Addison-Wesley, 2003, ISBN: 0321194969
- OpenGL ARB: ***OpenGL Programming Guide, 4th edition***, Addison-Wesley, 2004, ISBN: 0321173481
- ed. Randima Fernando: ***GPU Gems***, Addison-Wesley, 2004, ISBN: 0321228324

Sources II

- William Donelly: ***Generation Soft Shadows Using Occlusion Interval Maps***, GPU Gems, Ch 13
- Eric Lafortune et al.: ***Non-Linear Approximation of Reflectance Functions***, SIGGRAPH 1997
- David McAllister: ***Spatial BRDFs***, GPU Gems, Ch 18
- Matt Pharr, Simon Green: ***Ambient Occlusion***, GPU Gems, Ch 17
- Simon Green: ***Real-Time Approximations to Subsurface Scattering***, GPU Gems, Ch 16