Basic Techniques in Computer Graphics

Winter 2017 / 2018

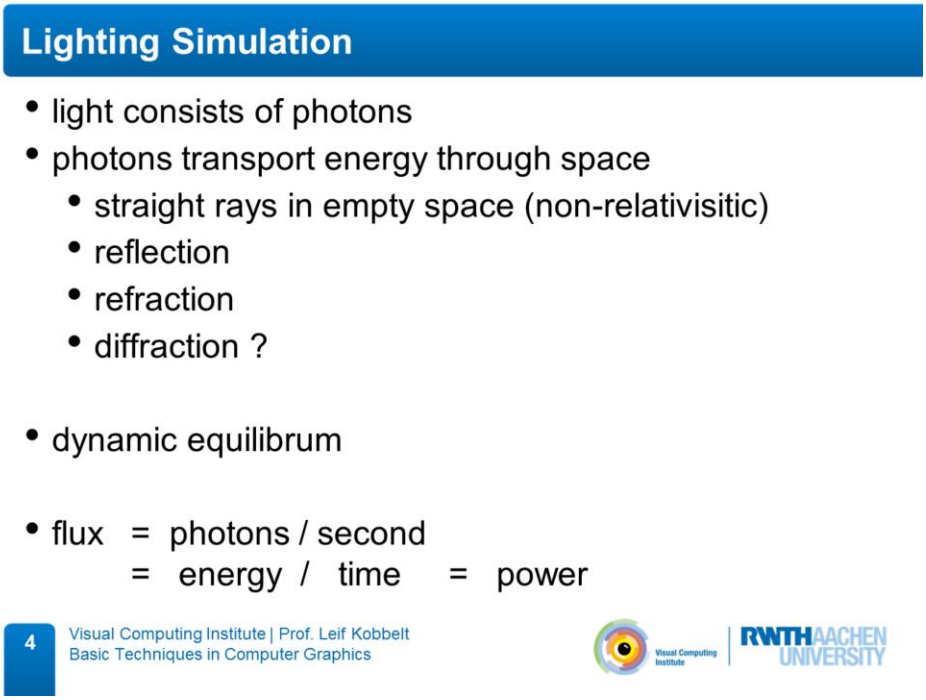
The slide comments are not guaranteed to be complete, they are no alternative to the lectures itself. So go to the lectures and write down your own comments!

Light and Shadow

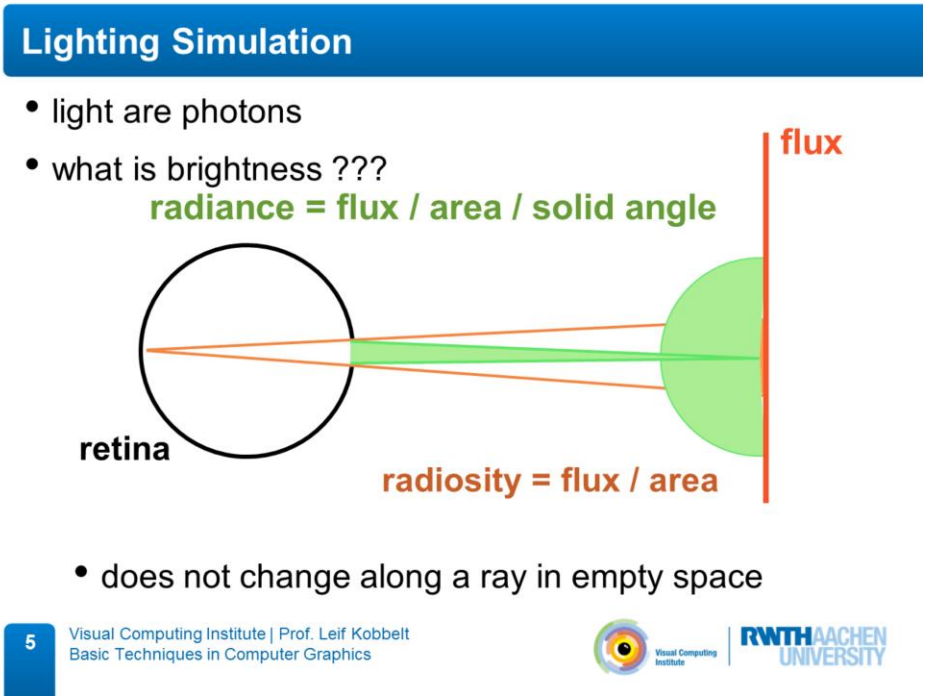
- Lighting is not only about brightness ...
- ...but about light-material interaction!
- Light-material interaction creates colors!

Light and Shadow

- **Lighting Simulation**
- Local Lighting
- Shading
- Shadows

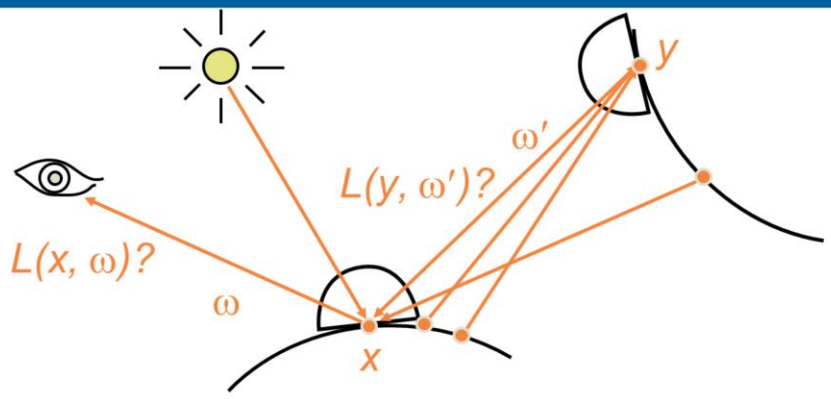


Light behaves like particles (photons), but also like waves. Most often the photon model is most practical for our uses. Assuming that light travels in a non-relativistic way in straight rays in space is also a simplification which is sufficient to model all daily effects and is the model we use here. Reflection and refraction are often seen but diffraction effects are often not important.
The light source(s) add photons into the scene while some get absorbed by the surfaces and the resulting equilibrium is the brightness of the scene. Flux is a measure of brightness as energy per time.



The flux is the total amount of photons emitted by the surface/lightsource per time while the radiosity also takes the area into account.
The radiosity however does not take into account how many of the emitted photons will reach the viewer if they are emitted evenly into the scene. The radiance is the radiosity per solid angle. The unit of the solid angle is srad.

Rendering Equation



$$L(x,\omega) = L_e(x,\omega) + \int_{\omega' \in \Omega_x} f_r(\omega, x, \omega') \cdot (L(\mathrm{ray}(x,\omega'), -\omega') \cos\theta \, d\omega'$$

James T. Kajiya. The rendering equation. SIGGRAPH `86

The rendering equation is in fact a recursive definition. In this form it‘s the equation for one wavelength and changes of the wavelength are ignored here.
L(x,w) is the light that gets emitted/reflected at a position x into the direction w.
Le(x,w) is the emission of that surface (or 0 if it is no light source itself).
The integral is the light that hits that point from all directions (L) weighted with the BRDF, the surface/material property).

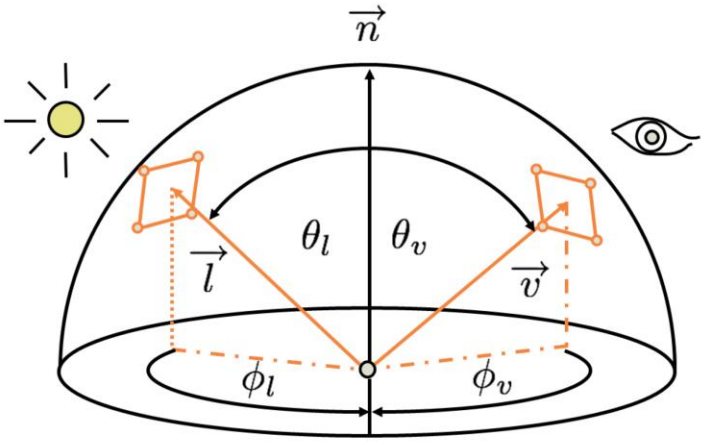
BRDF

$$L(x,\omega) = L_e(x,\omega) + \int_{\omega' \in \Omega_x} f_r(\omega, x, \omega') \, (L(\text{ray}(x,\omega'), -\omega') \cos\theta \, d\omega'$$

- *Bi-directional Reflectance Distribution Function*
- Describes reflections of rays
- Measure real materials (isotrop / anisotrop)

The BRDF tells us how much light gets reflected at a location x in direction w given incoming light from direction w‘.

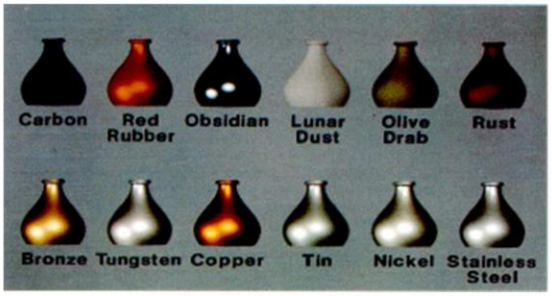
BRDF



For every surface position and every pair of vectors l and v the BRDF tells us how much light gets reflected (for R,G & B). There are even extensions to describe light that enters the surface at one point and exists at a slightly other point (e.g. marble behaves this way). For real materials the BRDF can get measured. Because of the high dimensionality of the BRDF and the complexity of the light transport, both are normally approximated.

BRDF

- Many different models for BRDF
  - Phong
  - Blinn-Phong
  - Cook-Torrance

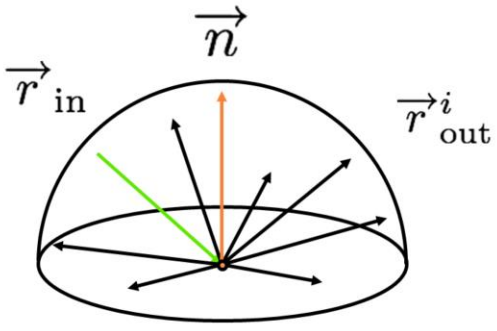


R. Cook and K. Torrance.

A reflectance model for computer graphics. SIGGRAPH `81

Phong, Blinn-Phong and Cook-Torrance are simple functions which approximate BRDFs with few parameters. Most of those approximations decompose the light reflections into three types of reflections to simplify the BRDF: Diffuse, specular and glossy reflections.

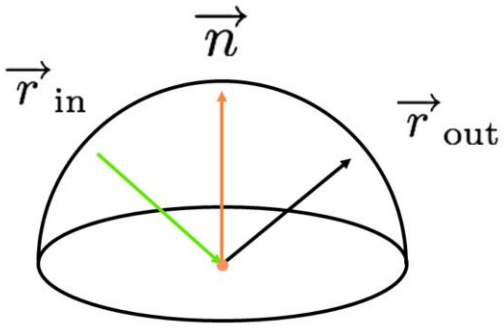
Diffuse Reflection



Diffuse

The diffuse reflection is a uniform distribution of the incoming light in all directions and thus independent of the viewers position. Concrete or paper is mostly diffuse.

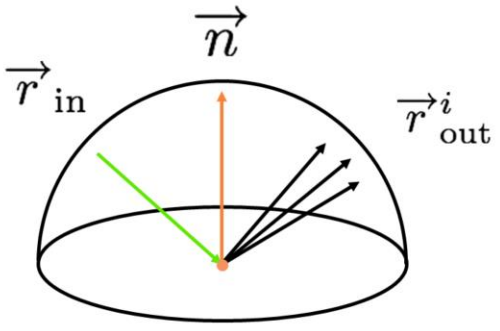
Specular Reflection



Specular

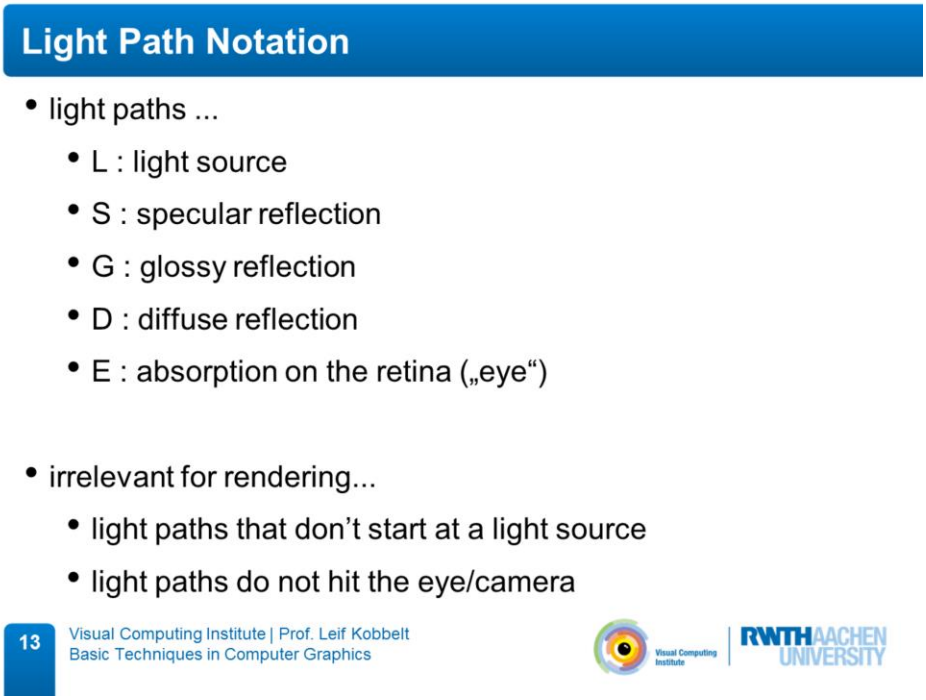
A perfect specular term would behave like a mirror and only reflect in one direction.

Glossy Reflection



Glossy

Glossy objects are no perfect mirrors and reflect in a small outgoing cone. Unpolished metal or plastic has a relatively high glossy term.
In the early days of computer graphics some called the glossy reflection specular reflection as this was close to a specular reflection but real specular reflections were not simulatable with local lighting.



Lets look at all possible light passes that can happen in the scene. In fact only the ones that end up in the eye/camera are of interest to us.
So the most generic lightpass would be:
E (D|G|S)* L
This means: A photon hits the eye E after it got reflected from diffuse (D), glossy (G) or specular (S) surfaces an arbitrary number of times (*) after it left the light source (L)

General Global Illumination

E (D | G | S)* L

The most general form starts at the eye, then has an arbitrary number of diffuse, glossy and specular reflections in an arbitrary order before it reaches the light source.

Ray Tracing

E S* (D | G) L

In ray tracing the light gets traced from the eye into the scene and can get reflected on specular surfaces multiple times but as soon as a diffuse or glossy surface is found, only a direct path to the light gets calculated.

Radiosity

E D* L

The radiosity algorithm is focussing only on diffuse reflections.

Ray Tracing + Radiosity

E S* (D* | G) L

The combination of a radiosity simulation of the scene and ray tracing can simulate more effects.

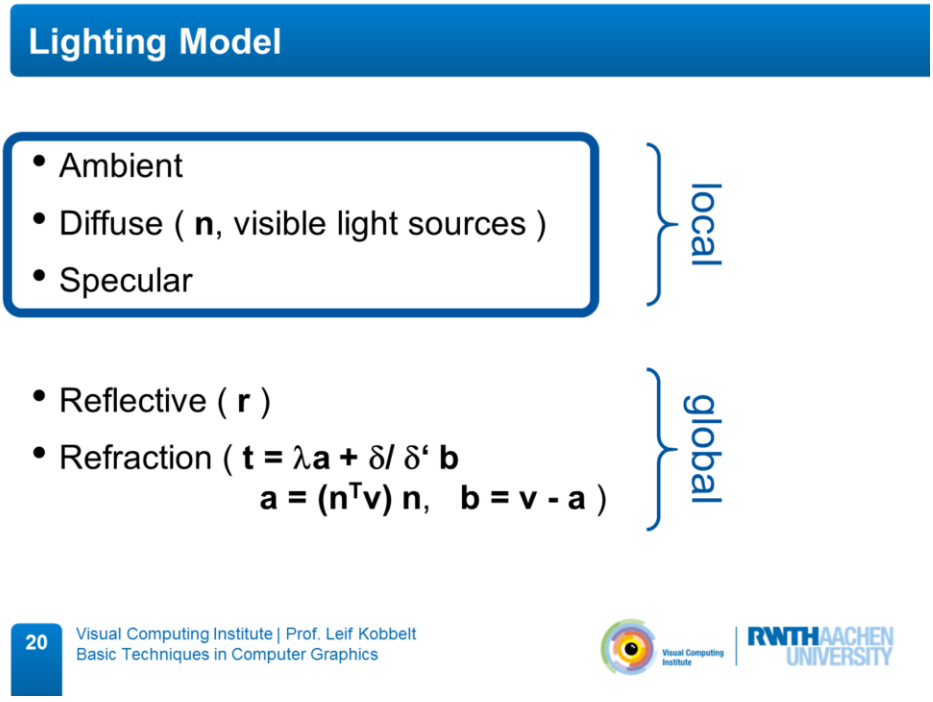
Direct Illumination

E (D | G) L

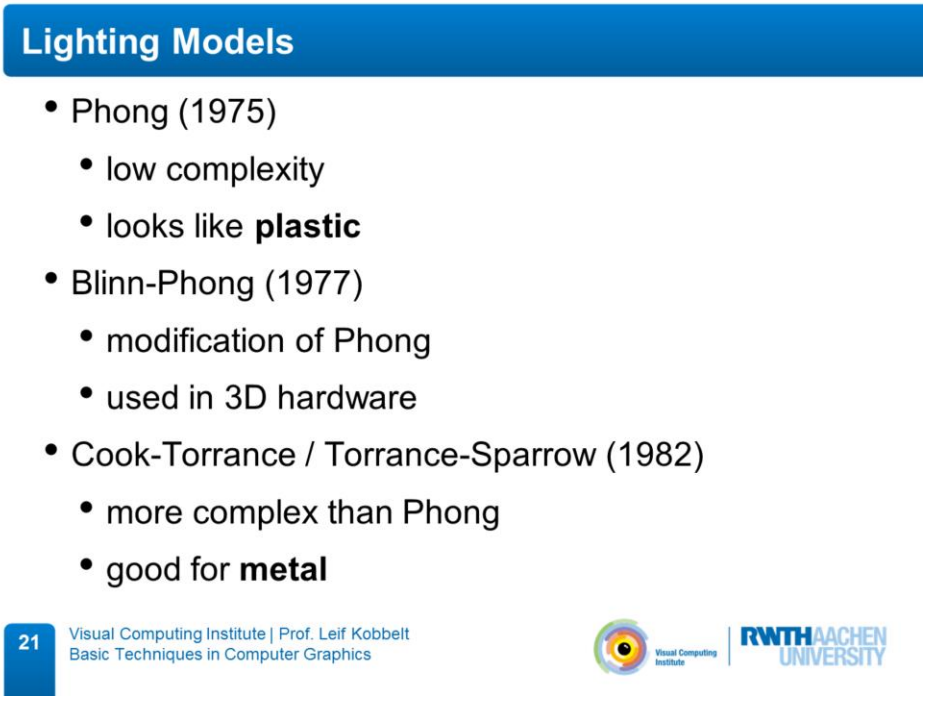
Local lighting models (e.g. Phong) can do only E (D|G) L, after leaving the light source the light gets reflected one time from diffuse or glossy objects - real mirror like reflections are not supported.
While those models are limited to only one surface interactions, this is exactly the limitation we have inside of our rendering pipeline as well: only the currently processed primitive (e.g. triangle) is known to the shaders in which we would implement those lighting calculations.

Light and Shadow

- Lighting Simulation
- **Local Lighting**
- Shading
- Shadows



Local lighting only takes one object into account at a time, global lighting has to know more of the scene.
As many local lighting models originate in the old days of computer graphics, where the glossy term was actually (incorrectly) called the specular term, we will use the same term here.
The ambient term is a simplification of the indirect light in a scene (a global illumination effect), often to just one brightness (and color) value for the whole scene.



Three examples of local lighting models.
Phong is a simple lighting model, that does not use much compute time but often looks like plastic.
Blinn-Phong is a minor variation of Phong and was implemented in hardware for Direct3D and OpenGL GPUs and is still used today (using shaders).
Cook-Torrance is more complex in terms of computation but can simulate surfaces like metal more realistically than Phong/Blinn-Phong.

Lighting Models

- Blinn-Phong: Most used lighting model for rasterization
- All: Simulate only local lighting
  - Can be used for ray-tracing & rasterization
  - Approximates global effects for rasterization
  - Uses normals to compute lighting

Still often used is the Blinn-Phong model. While PC and console games move to more complex models, for example mobile / smartphone games often still use Blinn-Phong.
All of the three mentioned models are only local light simulations and rely on surface normals.

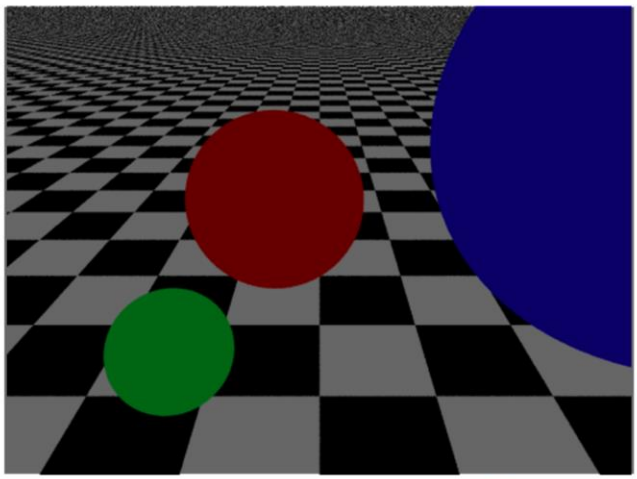
Lighting

- *Ambient* lighting:
  - simulate global effects
- *Diffuse* lighting:
  - dull, matt surfaces
- *Specular* lighting:
  - shiny surfaces, highlights

Ambient lighting can be a simple as one global term but it can also be the result of a offline global illumination algorithm stored e.g. per vertex or in a texture. It will be constant and can’t react to changes in the lighting situation!

Lighting

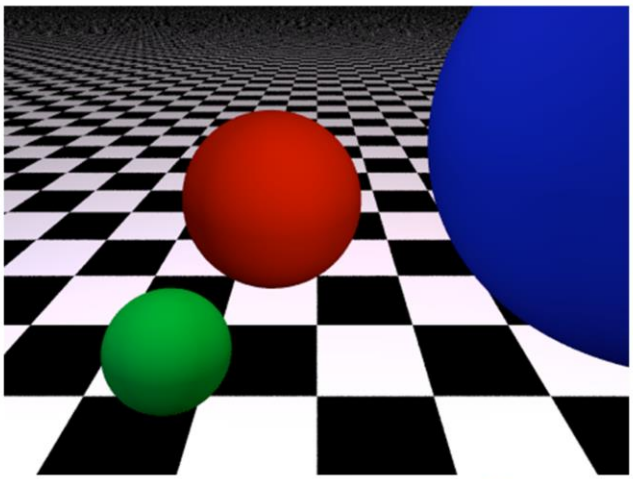
- Ambient



The ambient light is uniform and independent of the geometry or view position.
Note: The image is brighter than it should be as the ambient term is relatively low in this scene, however, with a realistic brightness, the image would be too dark for the projector.

Lighting

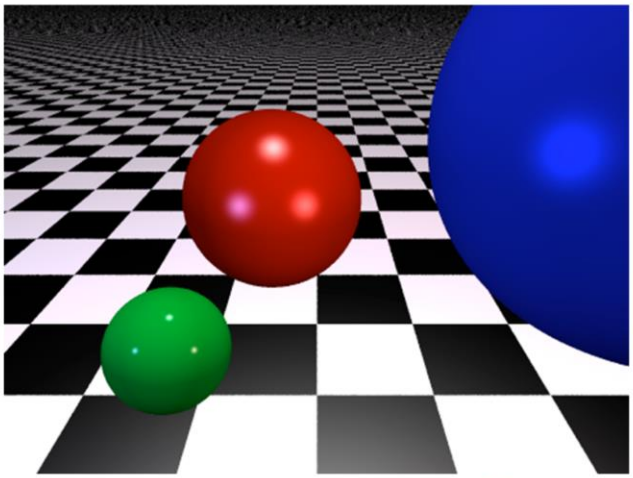
- Ambient + diffuse



Adding diffuse lighting will not only give the objects a 3D impression as it takes the normals and the light position into account, it will also increase the overall brightness.

Lighting

- Ambient + diffuse + specular



The specular term will add glossy highlights to the objects.

Lighting Models

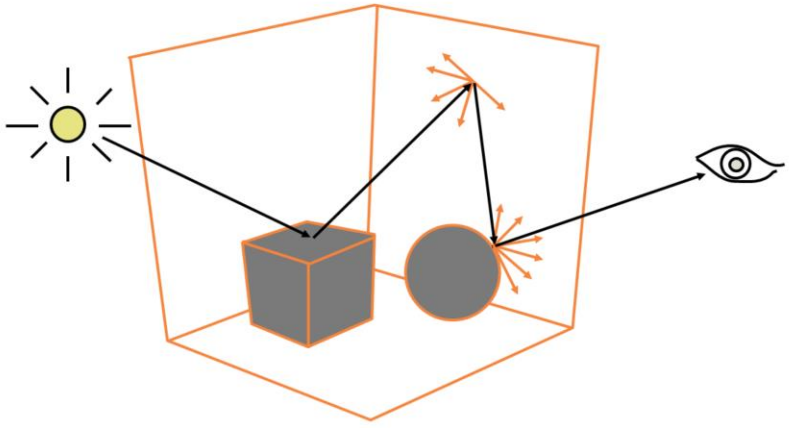
$C(p,n,v,l) = C_a + C_d(p,n,l) + C_{sp}(p,n,v,l)$

- C_a … constant environment term
- C_d … depends on light source *l* and surface normal *n*
- C_{sp} … depends on *l*, *n* and viewing direction *v*

(Phong/Blinn-Phong/Cook-Torrance differ here!)

p = position of the point
n = normal of that point
v = position of the viewer
l = position of the light source
Lighting is often implemented to be performed in „eye-space“, in the local coordinate system of the viewer. In such cases v is (0,0,0)!

Ambient Term



Ambient Term

- Approximate global light exchange
- Independent from actual light sources
- C_A : global ambient intensity

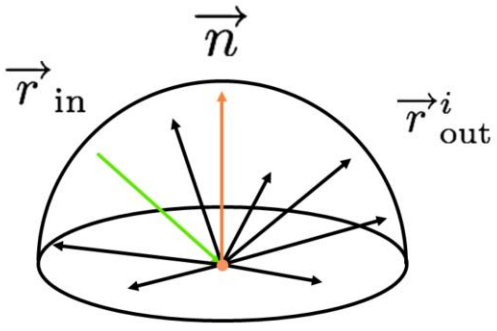
$$C_a = C_A \, \alpha_a$$

C_* : RGB color (3D vector)

α_* : material property (3x3 diagonal matrix)

In case the material properties are only a diagonal matrix, a per-component multiplication of a 3D material and the light ‚color vector' is also possible.

Diffuse Term

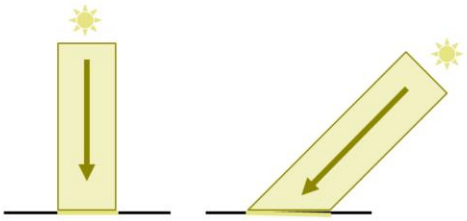


Diffuse

The diffuse term depends on the incoming light direction but as it models a uniform scattering, there is no outgoing direction.

Diffuse Term

- Uniform reflection
  (independent from viewer position)
- Brightness ~ received energy
  (Lambertian reflection)

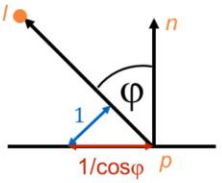


constant intensity
area ~ 1/cosφ
brightness ~ cosφ

One way to model the diffuse reflection is the Lambertian reflection law. It assumes an ideal scattering in all directions (maximum roughness) and in that case the brightness of the surface only depends on the energy density of the incoming light. Phi is the angle between the normal at point p and the negative incoming light direction. E = L/A => cos phi.

Diffuse Term

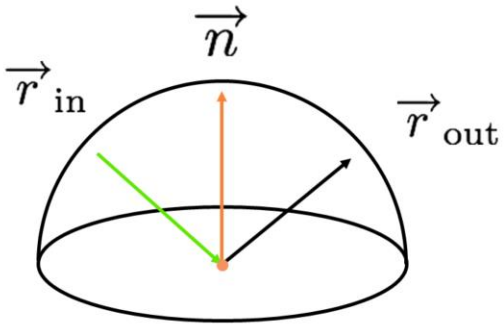
- Uniform reflection
  (independent from viewer position)
- Brightness ~ received/incoming energy
  (**Lambertian** reflection)



$$C_d(p,n,l) = C_l \cdot \alpha_d \cdot cos\varphi$$
$$= C_l \cdot \alpha_d \cdot \frac{n^T(l-p)}{\|l-p\|}$$

The diffuse term can get negative, so a max(Cd, 0) has to be performed if it should get implemented.

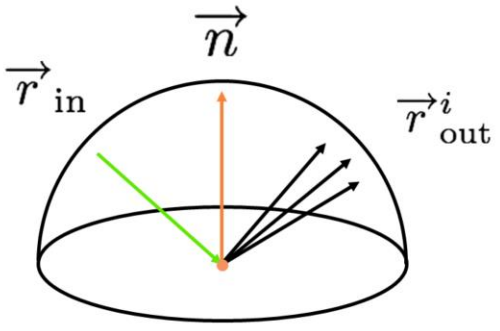
Specular Reflection



Specular

The specular term in Phong does not model this specular reflection...

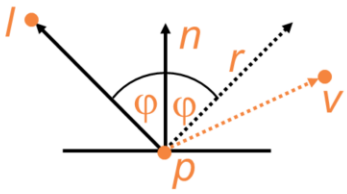
Glossy Reflection



Glossy

...but a glossy reflection.

Specular Term: Phong



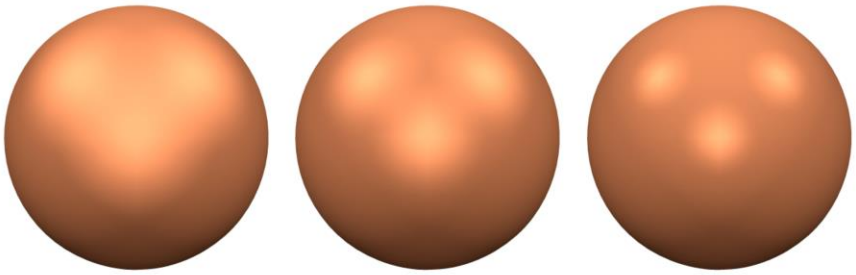
$$r = 2nn^T(l - p) - (l - p)$$

$$C_{\text{sp}}(p, n, v, l) = C_l \alpha_{\text{sp}} \cdot \left(\frac{r^t(v - p)}{\|r\| \|v - p\|} \right)^s$$

The cosine term of the specular term is now the cosine between the reflected ray and the viewing direction. If the viewer is looking directly into the reflected vector, the specular term is 1. The specular exponent s can be used to control the size of the specular highlight.
The specular term can get negative as well, so a max(specular, 0) has to be performed if it should get implemented.

Specular Term

Different values for the specular exponent s:



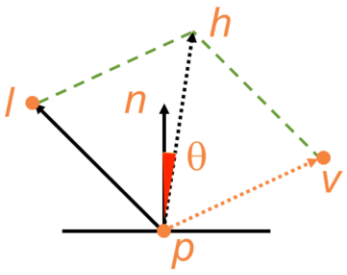
An example of how the exponent s controls the shape of the highlights. From left to right the exponent gets larger.

Specular Term: Blinn-Phong

- also called Phong-Blinn

$$\omega_l := \frac{l-p}{\|l-p\|} \quad \omega_v := \frac{v-p}{\|l-p\|}$$

$$h = \frac{\omega_l+\omega_l}{\|\omega_l+\omega_l\|}$$

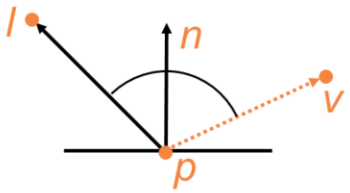


$$C_{\mathrm{sp}}(p,n,v,l) = C_l \alpha_{\mathrm{sp}} \cdot \left(n^T h\right)^s$$

Using this specular term is a bit less computationally expensive compared to Phong. The highlights look similar but not identical and normally the specular exponent has to get adjusted when switching from Phong to Blinn-Phong.

Specular Term: Cook-Torrance

- Reflection (Cook-Torrance)



$$C_{\mathrm{sp}}(p,n,v,l) = C_l \cdot \alpha_{\mathrm{sp}} \cdot \left(\frac{F}{\pi} \frac{DG}{(n \cdot (l-p))(n \cdot (v-p))} \right)$$

F = Fresnel term
D = Facet slope distribution
G = Geometric attenuation term

Ambient and diffuse terms are the same as in Phong, just the specular term gets exchanged.

Specular Term: Cook-Torrance

$$C_{sp}(p,n,v,l) = C_l \cdot \alpha_{sp} \cdot \left(\frac{F}{\pi} \frac{DG}{(n \cdot (l-p))(n \cdot (v-p))} \right)$$

$$F = F_\lambda + (1 - F_\lambda)(1 - cos\theta)^5$$ Schlick's approximation

$$D = \frac{1}{m^2 cos^4\theta} e^{-(\frac{tan\theta}{m})^2}$$

Cook-Torrance: D = Beckmann Distribution

(Torrance-Sparrow: D = Gauss Distribution)

m = surface roughness

θ = angle between n and h

$$G = min\left\{1, \frac{2(n \cdot h)(n \cdot (v-p))}{(v-p) \cdot h}, \frac{2(n \cdot h)(n \cdot (l-p))}{(v-p) \cdot h}\right\}$$

See Cook, Torrance, "A Reflectance Model for Computer Graphics", 1982 for details

D is the facet slope distribution on the surface, it's modeled with the so called Beckmann distribution, but other distributions can be used as well.

Attenuation

- Point light source
- Energy decreases quadratically w.r.t. distance || p - l ||
- Attenuation factor

$$\mathrm{att}(p,l) = \frac{1}{\mathrm{att}_{\mathrm{lin}}\|p-l\| + \mathrm{att}_{\mathrm{quad}}\|p-l\|^2}$$

A quadratic attenuation is more realistic but because local lighting ignores a realistic calculation of the ambient factor, a linear attenuation can lead to more pleasing results.

Spotlights

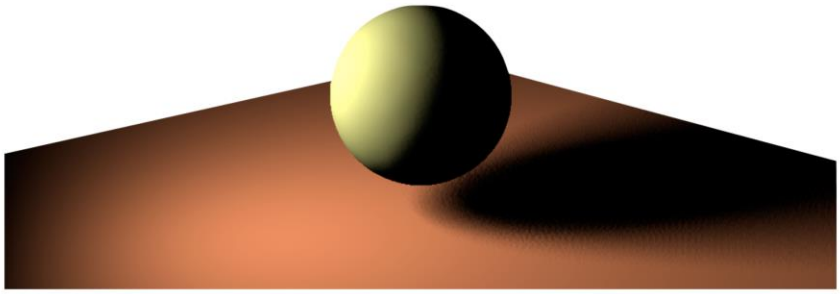
- Point light: isotropic
- Spotlight: anisotropic (direction d_l)

- Spotlight factor:

$$\text{spot}(p, l) = \left(\frac{d_l^T (p - l)}{\|p - l\|} \right)^f$$

A more focussed light source.

Preview: Area Light Sources



In the real world most light sources are area lights and do not originate from one point in space. This leads to soft shadows depending on the size of the light source and the distance to the objects. They can not be simulated in such a simple way as the point light or spotlight.

Multiple Light Sources

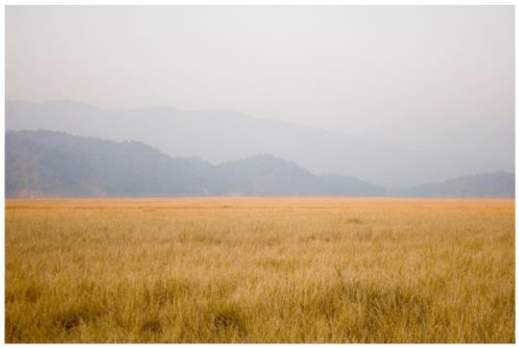
- Sum up contributions

$$C_{\text{orig}} = C_a \cdot \alpha_a + \sum_l \text{spot}(p,l) \cdot \text{att}(p,l) \cdot [C_d(p,n,l) + C_{\text{sp}}(p,n,v,l)]$$

When a scene has multiple light sources, the light contributions will get summed up. The ambient term however is still one scene constant.

Depth Cueing

- Damping of color / brightness by atmospheric effects
- Fade into grey-blue



$$C_{\text{final}} = b \cdot C_{\text{orig}} + (1 - b) \cdot C_{\text{DC}}, \quad b \in [B_{\text{min}}, B_{\text{max}}]$$

(Image courtesy of johnharveyphoto.com)

There are still effects missing, for example the damping of light based on the air the photons have to travel through. Depending on the distance to the objects (the amount of air in the way) a damping factor can get applied.

Lighting: Summary

$$C_{\text{final}} = b \cdot C_{\text{orig}} + (1 - b) \cdot C_{\text{DC}}$$

$$C_{\text{orig}} = C_a \alpha_a + \sum_l \text{spot}(p, l)\text{att}(p, l)[C_d(p, n, l) + C_{\text{sp}}(p, n, v, l)]$$

$$C_d(p, n, l) = C_l \cdot \alpha_d \cdot \frac{n^T(l - p)}{\|l - p\|}$$

$$C_{\text{sp}}(p, n, v, l) = C_l \cdot \alpha_{\text{sp}} \cdot \left(\frac{r^t(v - p)}{\|r\|\|v - p\|} \right)^s$$

The Phong lighting model in total including the color damping.
l = light source position
v = view center = eye/camera coordinates, often (0,0,0)T if lighting is done in camera/eye coordinate system
p = point that should be lit, if lighting is done in eye-space, p = Mmv * p‘ where Mmv is the modelView matrix and p‘ the original point
n = normal at p, if p was transformed, so must be n, but not with the same matrix, but a transposed inverse variant (called normal matrix): (MmvT)^-1

Lighting: Summary

$$C_{\text{final}} = b \cdot C_{\text{orig}} + (1 - b) \cdot C_{\text{DC}}$$

$$C_{\text{orig}} = C_a \alpha_a + \sum_l \text{spot}(p, l)\text{att}(p, l)[C_d(p, n, l) + C_{\text{sp}}(p, n, v, l)]$$

$$C_d(p, n, l) = C_l \cdot \alpha_d \cdot \frac{n^T(l - p)}{\|l - p\|}$$

scene parameter

$$C_{\text{sp}}(p, n, v, l) = C_l \cdot \alpha_{\text{sp}} \cdot \left(\frac{r^t(v - p)}{\|r\|\|v - p\|} \right)^s$$

The Phong lighting model in total including the color damping.
l = light source position
v = view center = eye/camera coordinates, often (0,0,0)T if lighting is done in camera/eye coordinate system
p = point that should be lit, if lighting is done in eye-space, p = Mmv * p‘ where Mmv is the modelView matrix and p‘ the original point
n = normal at p, if p was transformed, so must be n, but not with the same matrix, but a transposed inverse variant (called normal matrix): (MmvT)^-1

Lighting: Summary

$$C_{\text{final}} = b \cdot C_{\text{orig}} + (1 - b) \cdot C_{\text{DC}}$$

$$C_{\text{orig}} = C_a \alpha_a + \sum_l \text{spot}(p,l)\text{att}(p,l)[C_d(p,n,l) + C_{\text{sp}}(p,n,v,l)]$$

$$C_d(p,n,l) = C_l \cdot \alpha_d \cdot \frac{n^T(l-p)}{\|l-p\|}$$

$$C_{\text{sp}}(p,n,v,l) = C_l \cdot \alpha_{\text{sp}} \cdot \left(\frac{r^t(v-p)}{\|r\|\|v-p\|} \right)^s$$

scene parameter

material parameter

The Phong lighting model in total including the color damping.
l = light source position
v = view center = eye/camera coordinates, often (0,0,0)T if lighting is done in camera/eye coordinate system
p = point that should be lit, if lighting is done in eye-space, p = Mmv * p‘ where Mmv is the modelView matrix and p‘ the original point
n = normal at p, if p was transformed, so must be n, but not with the same matrix, but a transposed inverse variant (called normal matrix): (MmvT)^-1

Transformation of Normals

- Given: plane $n_x x + n_y y + n_z z + d = 0$
- Before *affine* transformation:
  $[n_x, n_y, n_z, d] \cdot [x, y, z, 1]^T = 0$
- After *affine* transformation:
  $[n'_x, n'_y, n'_z, d'] \cdot \mathbf{M} \cdot [x, y, z, 1]^T = 0$
  $\Rightarrow [n'_x, n'_y, n'_z, d']^T = (M^T)^{-1} \cdot [n_x, n_y, n_z, d]^T$
- Attn: *planar projections are singular!*
  ($\Rightarrow$ transform normals in object space *before* perspective projection)

The normal at a point P (P = (x,y,z,1)) defines a tangent plane. Before the transformation the plane equation is (nx, ny, nz, d)*(x,y,z,1)T = 0. The same is true after the transformation of course: (nx', ny', nz', d')*(x',y',z',1)T = 0. The vector P' is M*P, n' is unknown.
Solving for n' we find that n' = MT-1 * n. So the matrix to transform the normals with is the inverse of the transpose of the transformation matrix M of the points.
This matrix is called the "normal matrix" and should get calculated one per object instead of calculating it per vertex (the inverse can be slow inside of a shader).

Light and Shadow

- Lighting Simulation
- Local Lighting
- **Shading**
- Shadows

Where do normals come from?

- Defined by the artist
- Estimate **vertex normal vector**
  - average face normals
  - weighted by area:

    $\sum (pi - p) \times (pi_{+1} - p) = \sum p_i \times pi_{+1}$
  - weighted by inverse area
  - weighted by opening angle
  - ...

We can provide an individual normal per surface point (via a texture, defined by an artist), but we can also provide a normal per vertex computed from the mesh itself. For this, there are multiple ways to calculate the normal.
Normals defined by an artist can also mean, that he provides a higher resolution (more surface details) mesh to define the normals of the lower detailed one from.

Shading

- Possible: Lighting at vertices
- But: Need color for all fragments of a primitive
- How to get colors inside of primitive?

  → **Shading**

- Different strategies for color computation / interpolation

The interpolation of attributes of the vertices over the triangle is called shading.

Shading

- **Flat Shading**
  - Per face lighting
  - Piecewise constant color
  - Mach banding effect
- Gouraud Shading
- Phong Shading

Shading is the way the final color gets defined on all (visible) surface points.
Flat shading is a way where all polygons get shaded in the same color, this reduces computation time because the lighting has only be evaluated once per surface.
Even today the graphics hardware has the option to use the data of one vertex to be used for all fragments of the triangle to get a flat shaded result.

Mach Banding

The natural contrast enhancement of the human visual system leads to visible artifacts, which is a good argument against flat shading.
Depending on your screen/printer/beamer contrast you might see the mach banding effect only on one of the images.

Shading

- Flat Shading
- **Gouraud Shading**
  - Per vertex lighting
  - Linear interpolation of colors (object space, screen space)
  - Lose small highlights
- Phong Shading

In Gouraud shading the lighting gets calculated per vertex (in the vertex shader) and the result gets interpolated over the triangle. All attributes from the vertices which will get passed to the fragment shader (the varyings) are normally interpolated in the same way as gouraud shading works.

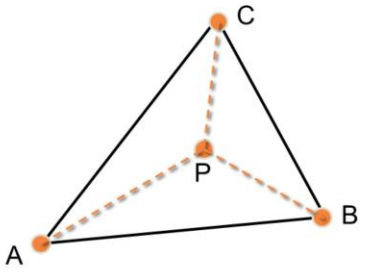
Shading

- Flat Shading
- Gouraud Shading
- **Phong Shading**
  - Linear interpolation of normals (object space)
  - Per pixel lighting

In phong shading (don‘t mix it up with the phong lighting model) the lighting gets calculated per fragment (in the fragment shader), thus, the parameters might get provided per fragment as well (via textures) or per vertex (as vertex attributes) and the parameters get interpolated.

Bilinear Interpolation

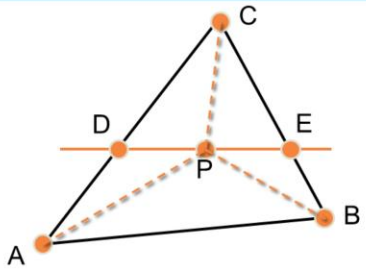
- Barycentric coordinates:
  - $P = \alpha A + \beta B + \gamma C$ with $\alpha + \beta + \gamma = 1$
  - Unique
  - Ratio of areas
  - $AP = \alpha A_A + \beta A_B + \gamma A_C$



The barycentric coordinates are used for the interpolation of data over a triangle.

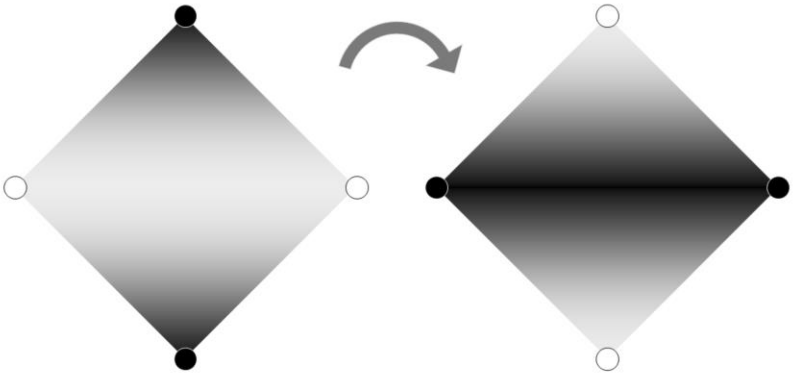
Bilinear Interpolation

- scanline oriented
  - along edges:
    - $D = (1-\lambda_0)\, A + \lambda_0\, C$
    - $E = (1-\lambda_1)\, B + \lambda_1\, C$
  - along scanline:
    - $P = (1-\lambda_2)\, D + \lambda_2\, E$
      $= (1-\lambda_2)(1-\lambda_0)\, A + \lambda_2(1-\lambda_1)\, B + [\, (1-\lambda_2)\,\lambda_0 + \lambda_2\,\lambda_1\,]\, C$
      $= \alpha A + \beta B + \gamma C$
  - depends on orientation of triangle?
    - Triangles : NO (uniqueness of the barycentric coordinates)
    - General polygons : YES



The bilinear interpolation can get decomposed into linear operations when it gets evaluated per scan line.

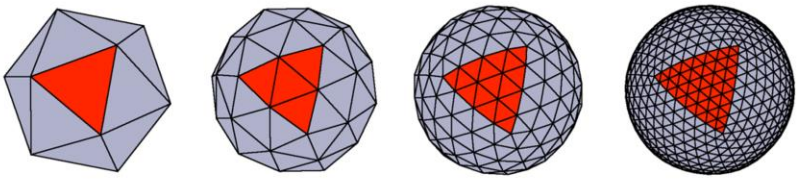
Bilinear Interpolation



Given a polygon with different colors at the corners, it‘s not well defined how the interpolation of the colors (or any parameters) should get performed. This is however well defined for triangles.

Rendering Polygonal Meshes

- Assign material properties to vertices
- Shade facets by interpolating
  - colors
  - normal vectors



Light and Shadow

- Lighting Simulation
- Local Lighting
- Shading
- **Shadows**

Shadow Computation

- Shadows are a result of multiple objects: occluder and occludee
- Global **volumetric** property of the scene
- Special case of visibility computation (light visible from a scene point)

Shadows are very important for the realistic impression of the scene. To calculate a shadow we have to test whether there is any occluder between the fragment being shaded and the light source. This requires some kind of knowledge of the whole scene.

Shadow Computation

- Lighting computation

$$C(p,n,v) = C_a + \sum_{\text{lights}l} S(p,j) \cdot C_l(p,n,v,l)$$

$$S(p,l) = \begin{cases} 0, & \text{light } l \text{ is blocked at point } p \\ 1, & \text{point } p \text{ is visible from light } l \end{cases}$$

Adding a shadowing test into our lighting calculation. In case of point lights the shadow term is just a binary value. Area light sources would have a value between 0 and 1 but those are more complex and wont be covered here.

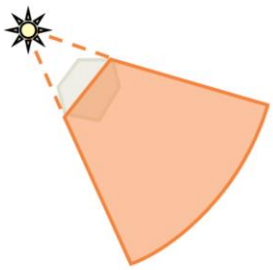
Shadow Computation: Rasterization

- Different Shadow Algorithms for rasterization:
  - **Shadow Volumes**
    - Shadow computation in object space
    - **Continuous** shadows
  - **Shadow Maps**
    - Shadow computation in image
    - **Discrete** shadows
    - **Perspective Shadow Maps**

For rasterization, shadows have to be created with some additional tricks.
Two techniques are shadow volumes and shadow maps. There exist many variants of shadow maps, for example perspective shadow maps.

Shadow Volumes

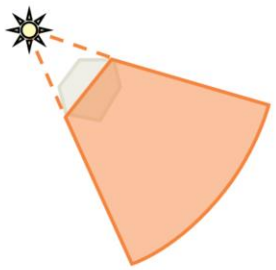
- Polygonal representation of regions in shadow
  - Silhouette cone (3D representation)
  - Each silhouette edge spans an infinite face
  - Pre-computed for static scenes
  - O(#light sources * #occluder)



The first step in the shadow volumes algorithm is to determine the volume of a shadow by calculating the silhouette of an object (as seen from the light source) and extending this into the scene in the „view“ direction of the light source.

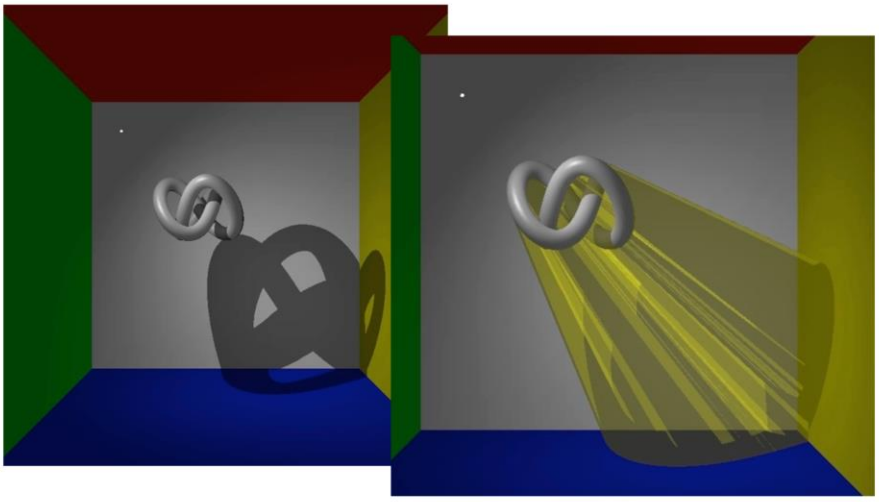
Shadow Volumes

- Shadow volume generation
  - Detect and extrude silhouette edges
  - Test normals of incident triangles
  - Requires closed two-manifold mesh



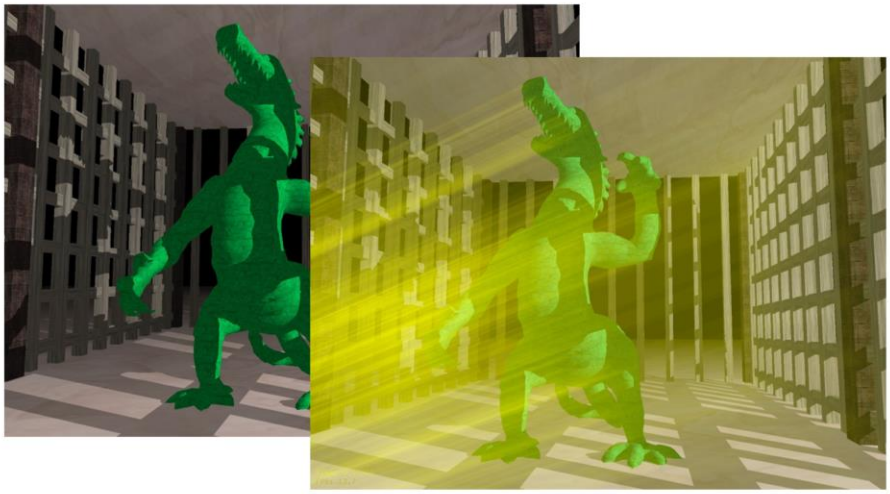
A silhouette is always guaranteed to be a closed loop (assuming a closed surface).

Shadow Volumes



In fact there can be multiple intersecting volumes from one light source. A point that lights at least in one shadow volumes should not be lit by the corresponding light source. There can also be multiple volumes from multiple light sources (which would create half shadows as well).
(left: the final rendering, right: the volumes visualized)

Shadow Volumes



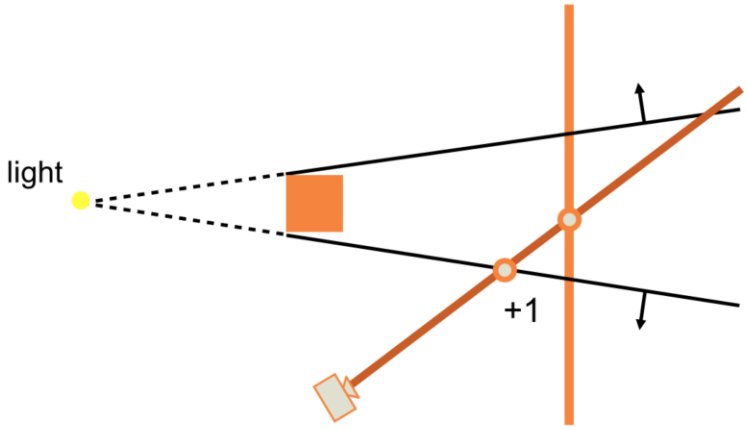
Another example with visualized shadow volumes. The visible overdraw can be used to determine whether a fragment lays within a shadow.

Shadow Volumes

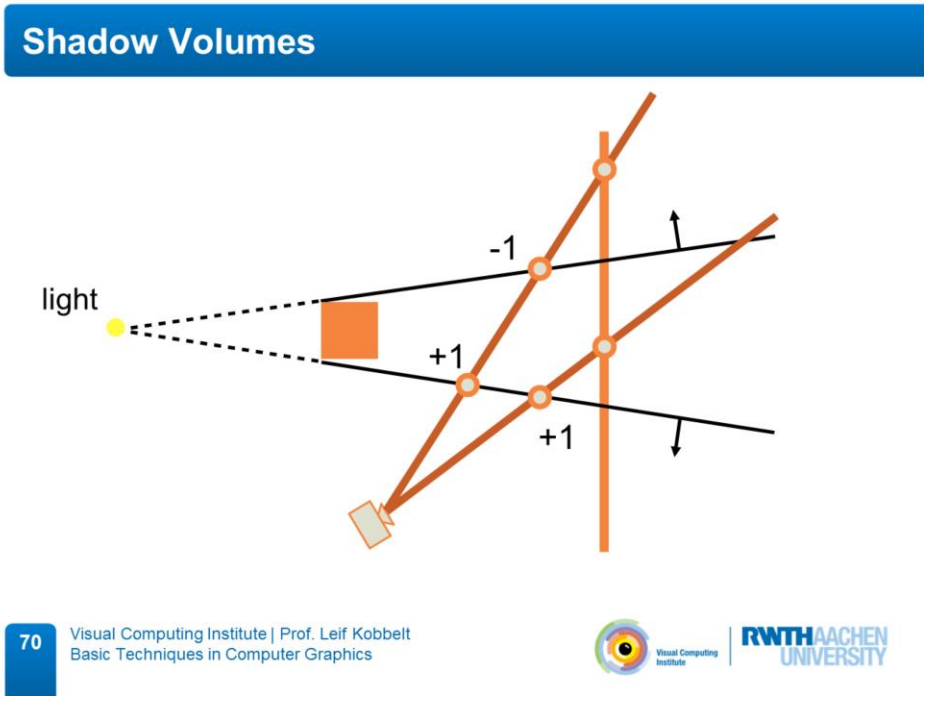
- Ray Casting
  - Count intersections with shadow volume
  - Intersect more front-facing than back-facing shadow volume polygons → in shadow
  - Special case: viewer in shadow

How is the shadow computed from the volume? If a ray gets shot per pixel into the scene the intersections with the (not rendered) shadow volume can get counted to determine whether the object that got rendered at that point is in shadow or not.
If the viewer is inside of a shadow volume, the ray cast would as well start inside of the shadow volume. In that case the algorithm has to get adjusted for this.

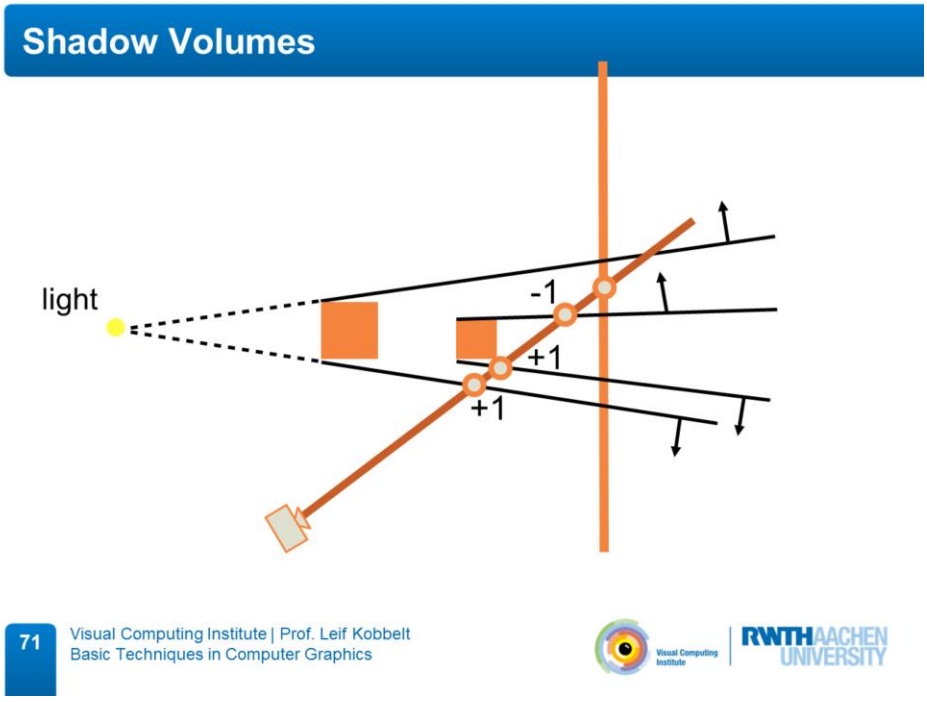
Shadow Volumes



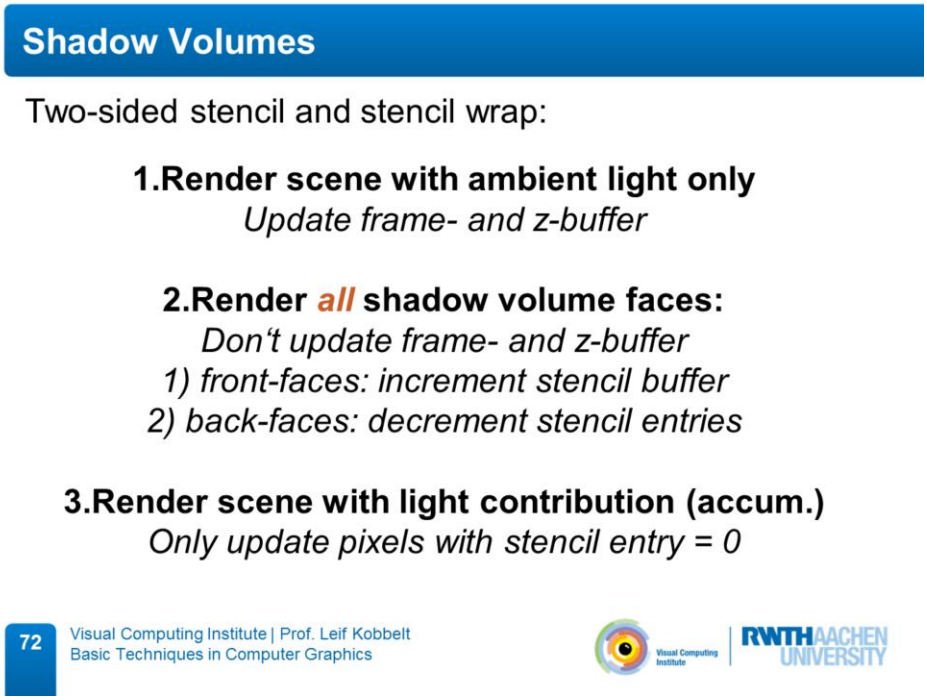
The ray from the camera intersects one front facing shadow volume polygon before it hits the visible surface, so we count +1. This means this point is in shadow.



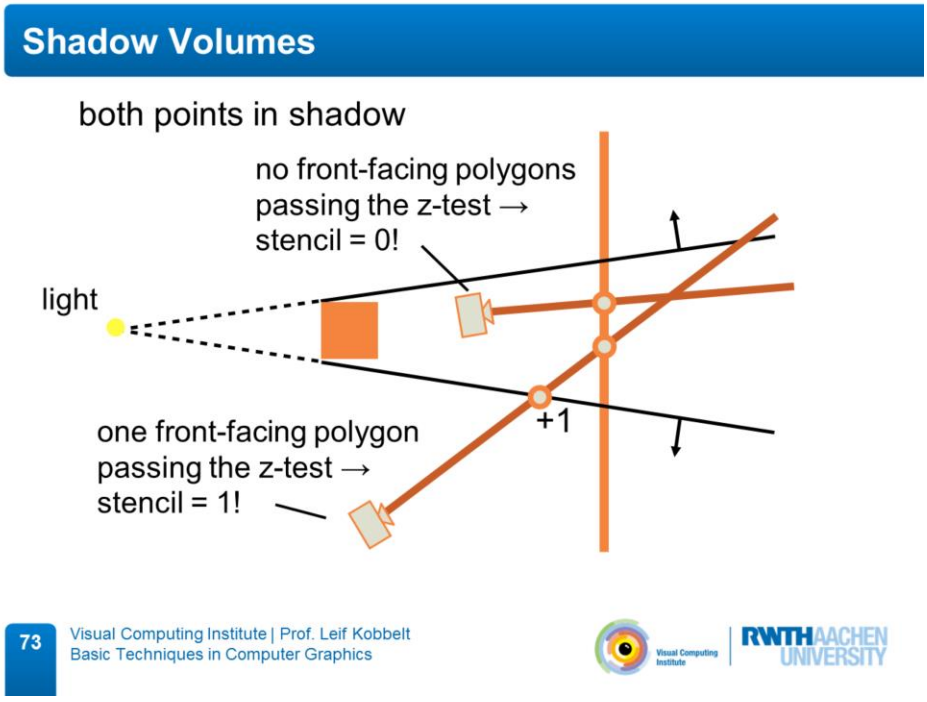
A second ray intersects a front facing shadow volume polygon (+1) and one back facing one (-1) so the result is 0, which means the point is lit.



A more complex example with two objects in shadow: The ray intersects two front facing shadow volumes and one back facing one so the value is at the intersection with the geometry still >1 so the object is in shadow.



The ambient light is independent of light sources and thus independent of shadows, so we can render it beforehand and also fill the depthbuffer.
The stencil buffer is an extra buffer in OpenGL in the same resolution of the final image. This buffer can be used to „filter out“ fragments that would be drawn at a point on the screen where the stencil buffer has a specific value. So this buffer is really well suited for this task.
Front-facing and back-facing triangles (relative to the viewer) can get easily determined, even by the hardware itself. So OpenGL can be forced just to render front or back-facing triangles.
Note: This algorithm has to be executed once per light source.



Shadow volumes have a problem if the camera is also in the volume, because the number of intersections with the volumes are wrong (so far we assumed the camera to be outside of the volumes!).

Shadow Volumes

Reverse algorithm (z-fail)

1.Render scene with ambient light only

Update frame- and z-buffer

2.Render *all* shadow volume faces:

Don‘t update frame- and z-buffer

***Reverse depth test**, update stencil on z-fail*

1) back-faces: increment stencil entries

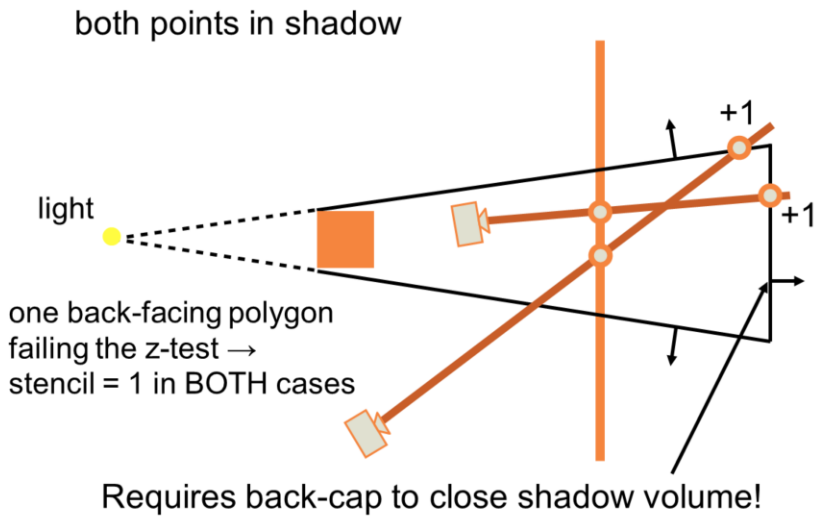
2) front-faces: decrement stencil buffer

3.Render scene with light contribution (accum.)

Only update pixels with stencil entry = 0

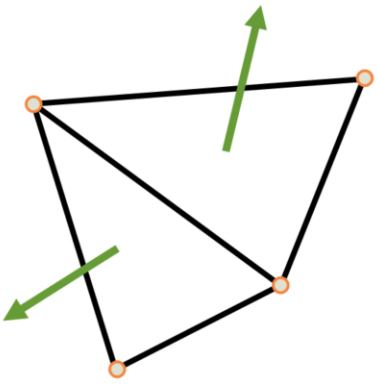
Idea: count the intersections of the rays with the shadow volumes behind the visible geometry, not the ones in-front of it! Don’t update the stencil buffer if the depth-test passes, but if it fails. Note that the shadow volumes need to be closed at the end as in practice the volume faces will not go to infinity but just far enough to be outside of the scene.

Shadow Volumes



An additional face to close the volume is needed here.

GPU Silhouette

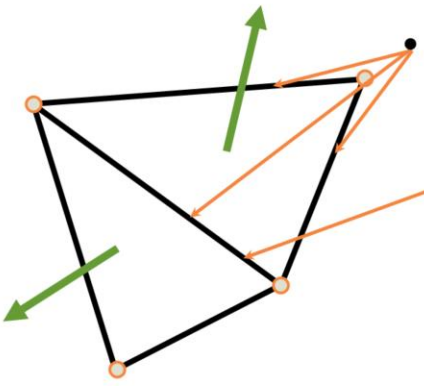


point light

- Can be done in a Geometry Shader:
  - Input: Triangle with adjacent triangles
  - Output: Same Triangle
  - + 0/2/4 or 6 silhouette triangles

A geometry shader can also get information about its adjacent triangles (but can‘t modify them).

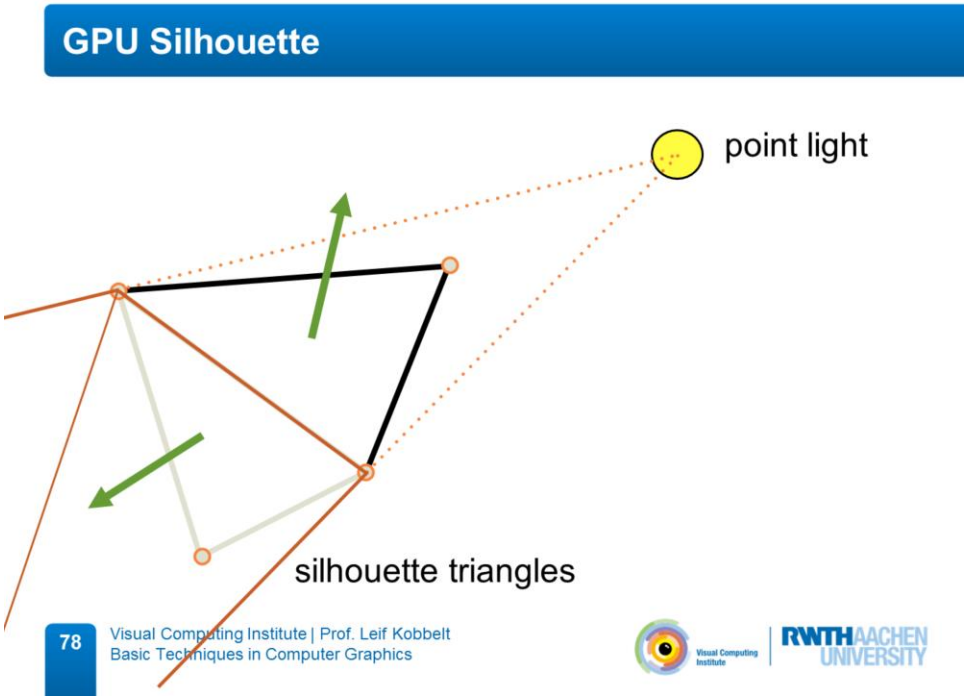
GPU Silhouette



point light

- for each edge:
  - test for silhouette edge (based on normals)
  - if true: create 2 silhouette triangles

For each edge based on the normals of the triangles (provided or given as additional precomputed attributes) a silhouette can get detected.



For each edge a quad (two triangles) will get created from that edge to „infinity“ as an extension of the vector light position vertex. In fact the quads have to be of finite size so a length that will be large enough to extend the shadow volume out of the scene is sufficient.
If a bottom cap of the volume should get created, more output triangles have to be created.
A precomputed shadow volume is created in the same way.

Shadow Maps

1. Render scene seen from light source
2. Store z-buffer (holds distance to light)
   - ➔ *discrete representation of the shadow volume*
   - ➔ „Shadow Map“
3. Render scene from eye point
   - Question: Light point / pixel?
   - Re-project in onto shadow map
   - Distance point-to-light > depth stored in map
     → *in shadow*

The alternative are Shadow Maps that render the scene from the point of the light source and use that depth values to test whether a fragment is lit or not.

Shadow Maps

OpenGL Rendering Algorithm

1.Render scene seen from light source
store depth buffer in shadow map

2.Render scene using shadow map texture
project each fragment onto shadow map
depth comparison as special texture operation

As pass 1 will only have to generate depth values, no (complex) fragment shader is needed. The framebuffer to render into doesn't even need a color buffer - just a depth buffer. The modelview matrix for pass 1 will be based on the light position, direction and opening angle (if it's a spot light).
The second pass will get the depth buffer as a texture as well as the modelview matrix of the light source which was used in pass 1 to project all fragments into the shadow map. Comparing the depth values will define the shadow term of the light source.
For each light one shadow map is needed.

Shadow Maps

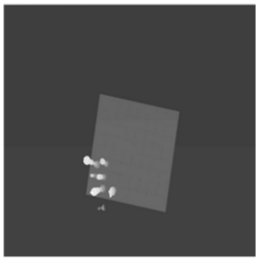
The resolution of the Z-Buffer of the first pass (== resolution of the shadow map) limits the resolution of the shadows and lead to jaggies. For hard shadows, the shadow map algorithm is inferior to shadow volumes.

Perspective Shadow Maps

Scene as seen from the light source

(discarded) colored view

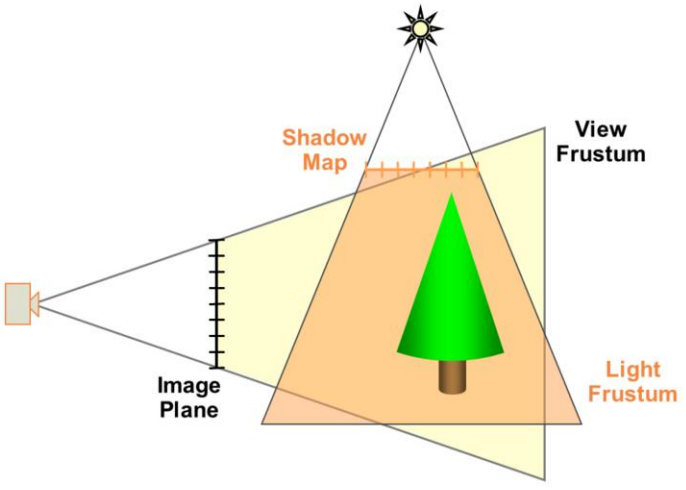
depth map -> shadow map



Final rendering

A scene with one light source and a close-up rendering of a piece of the checker board. In this case only a small part of the shadow map is used to determine the shadows in the scene.

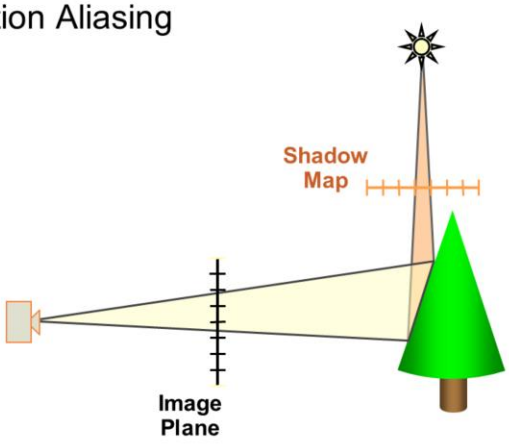
Perspective Shadow Maps



The image plane as well as the shadow map have a finite resolution.

Perspective Shadow Maps

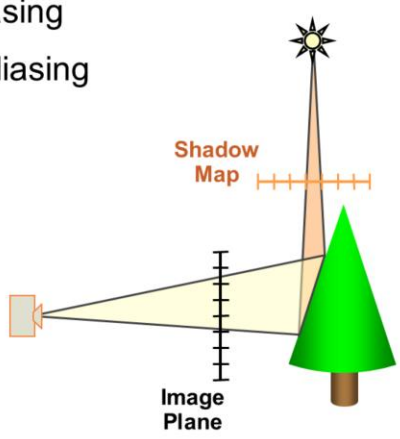
- Projection Aliasing



One Shadow map pixel gets mapped onto a set of screen pixels. This gets worse the nearer the camera gets.

Perspective Shadow Maps

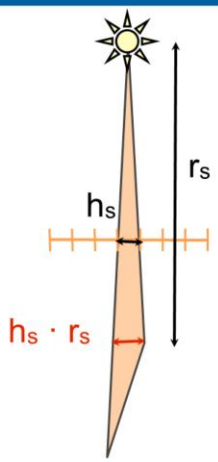
- Projection Aliasing
- Perspective Aliasing



The same situation but with a closer camera shows, that the area the one shadow map pixel gets projected onto gets larger.

Perspective Shadow Maps

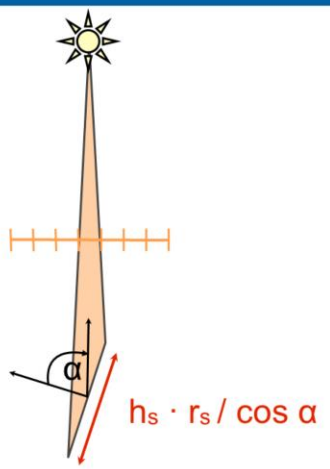
h_s : Pixel width



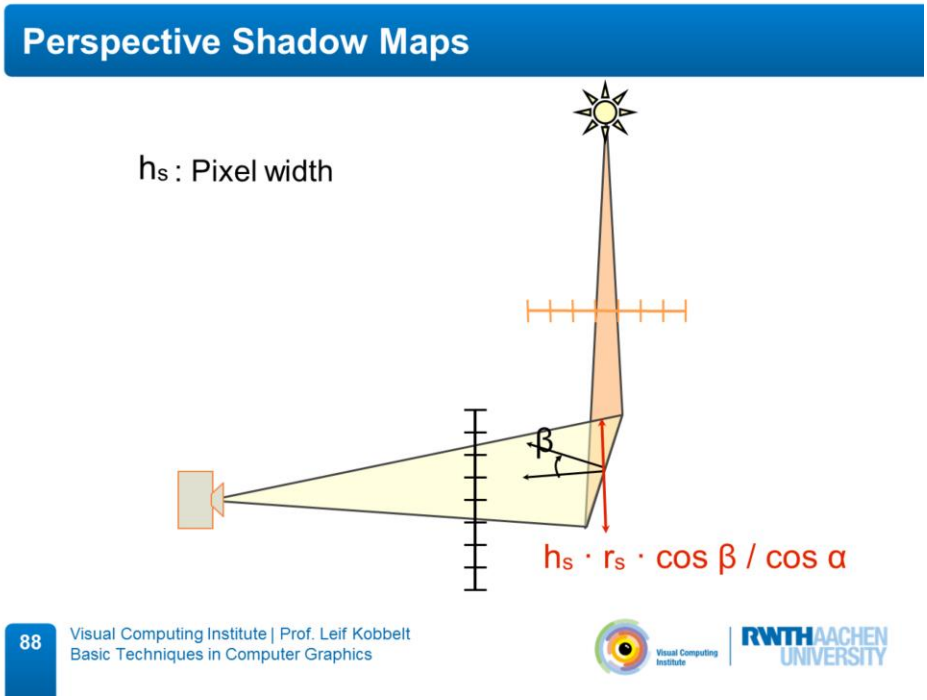
We can calculate how much a shadow map pixel gets scaled into the scene. Assuming that the distance of the light to the shadow map plane is 1 the size hs*rs is a result of the intercept theorem (german: Strahlensatz).

Perspective Shadow Maps

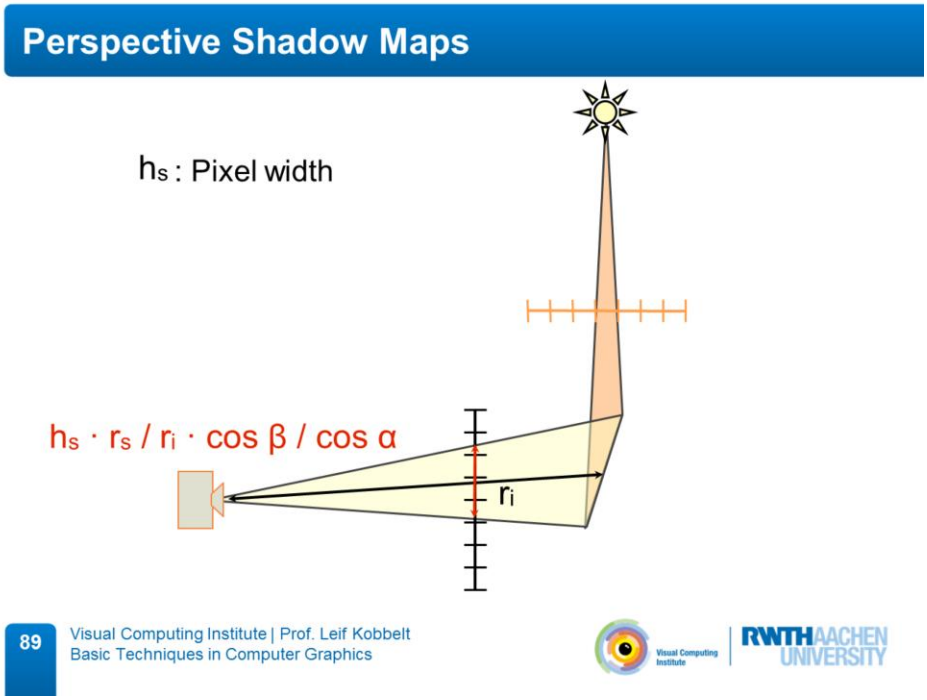
h_s : Pixel width



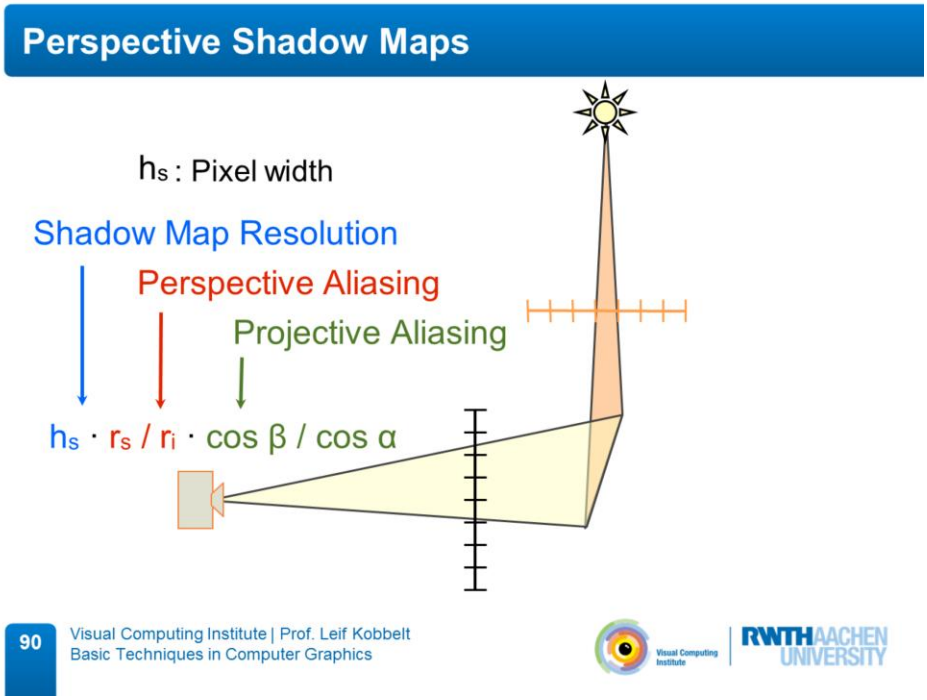
1 / cos(alpha) is the stretching of the pixel based on the angle of the surface normal a shadow map pixel is projected onto and the vector of the light direction.



Projecting this into the camera we have another angle (surface normal to viewing direction)...



... and a second resolution (viewport of the rendering).



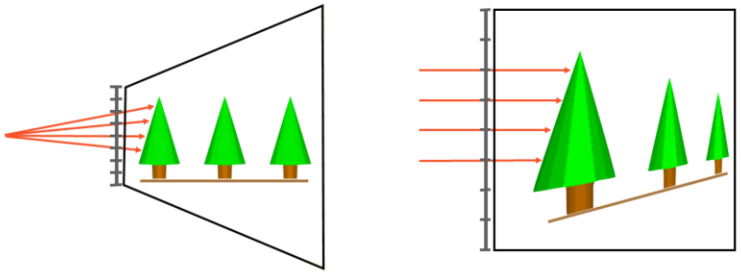
An increased shadow map resolution would reduce hs and thus reduce the aliasing but texture memory is a finite resource.
One can not do anything about cos(beta)/cos(alpha) since it depends on the geometry of the scene.
So the only thing we can do something about is r_s/r_i, the perspective aliasing.

Perspective Shadow Maps

- Avoid perspective aliasing
- Acquire Shadow Map in post-perspective space
  - Normalized Device Coordinates
  - Parallel viewing rays
- Optimal configurations
  - Directional light parallel to image plane
  - Point light in the camera plane

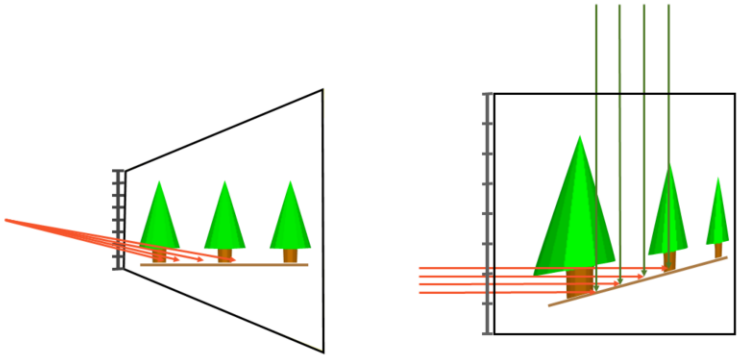
Idea: compute the shadow map in NDC.

Perspective Shadow Maps



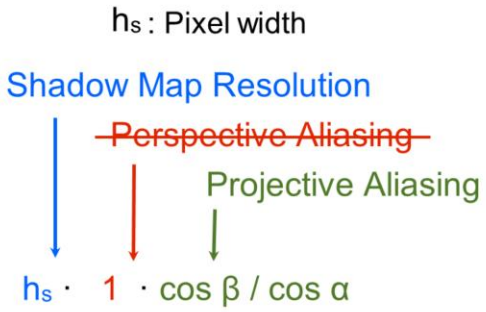
left: euclidian space
right: NDC

Perspective Shadow Maps

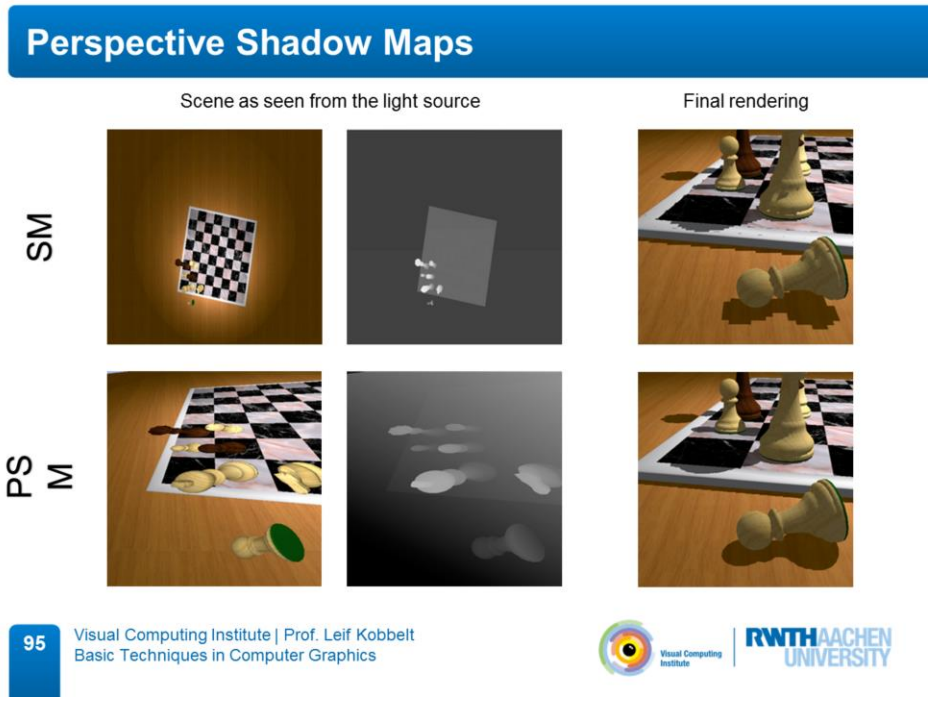


right: uniform sampling in all distances. This means that shadow map pixels that are closer to the camera are not larger anymore than those further away.

Perspective Shadow Maps



We cannot entirely avoid aliasing artifacts. But with PSM, the amount of aliasing is reduced to h_s * cos(beta)/cos(alpha).
The remaining aliasing is due to projective aliasing and the shadow map resolution which we cannot do anything about.
If the light position = eye position and viewing dir = light dir, then the cosine term is 1, further reducing the remaining aliasing error. If additionally the shadow map's resolution is the same as the framebuffer's resolution, there won't be any aliasing artifacts.



Top row: naive shadow maps.
Down row: perspective shadow maps.
Downside of the PSM is that the shadow map has to be recomputed whenever the camera moves. But for both shadow map algorithms the map has to be recalculated whenever something in the scene moves anyway.

Shadow Maps

+ General method for computing shadows
all objects that can be rasterized

- Does not work for omni-directional lights
requires six-sided shadow cube maps

- Resource consumption
high-resolution, high-precision depth textures

- Discrete sampling, shadow map resolution
alias artifacts, epsilon thresholds

Shadow Volumes

+ Exact shadows, minimal aliasing
in contrast to shadow maps

+ Works for omni-directional point lights
in contrast to shadow maps

- Needs closed manifold mesh
shadow maps can handle more general geometry

Special Texture Maps

- 1D / 3D Textures
- *Reflection mapping*
  - View environment from within the object
  - Use image as textures (e.g. cube)
- *Environment mapping*
  - Precompute reflected environment
  - Use reflected vector to access texture

Light and Shadow

- Lighting Simulation
- Local Lighting
- Shading
- Shadows