

Shading

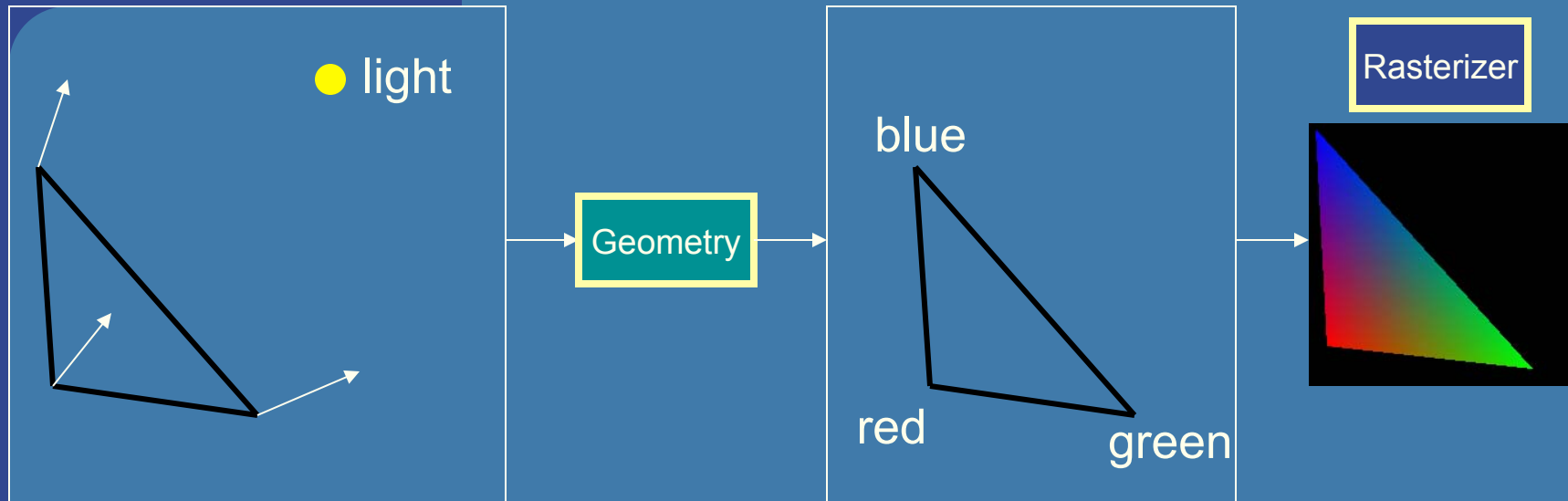
Slides by Ulf Assarsson and Tomas
Akenine-Möller

Department of Computer Engineering
Chalmers University of Technology

Overview of today's lecture

- A simple most basic real-time lighting model
 - Also, OpenGL's old fixed pipeline lighting model
- Fog
- Gamma correction
- Transparency and alpha

Compute lighting at vertices, then interpolate over triangle



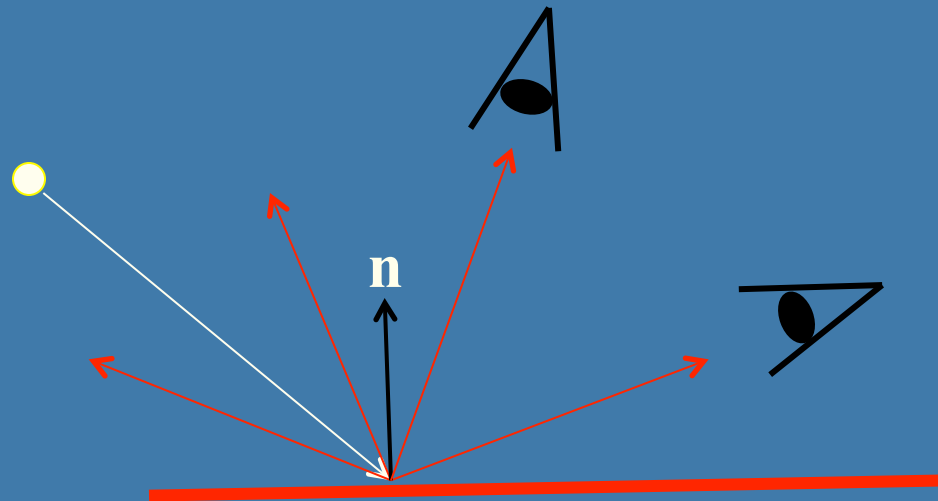
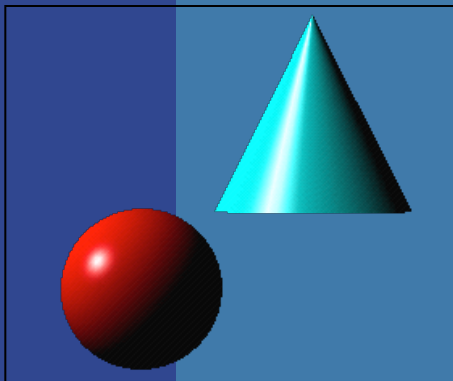
- How compute lighting?
- We could set colors per vertex manually
- For a **little** more realism, compute lighting from
 - Light sources
 - Material properties
 - Geometrical relationships

The ambient/diffuse/specular/emission model

- Also the most basic real-time model:
- Light interacts with material and change colors at bounces:

$$\mathbf{outColor}_{rgb} \sim \mathbf{material}_{rgb} \otimes \mathbf{lightColor}_{rgb}$$

- Diffuse light: the part that spreads in all direction (view independent) due to that the surface is **rough** on microscopic level

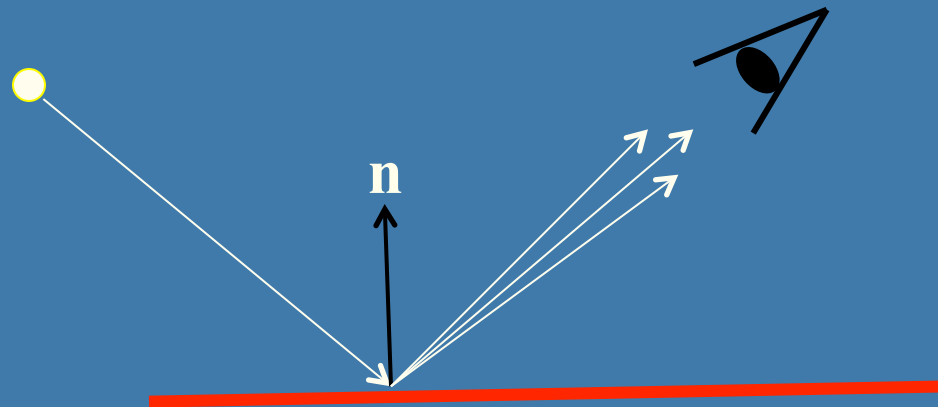
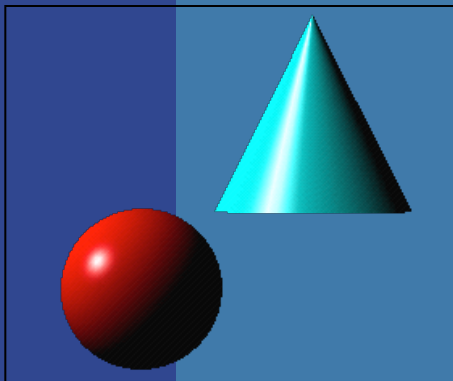


The ambient/diffuse/specular/emission model

- Also the most basic real-time model:
- Light interacts with material and change colors at bounces:

$$\mathbf{outColor}_{rgb} \sim \mathbf{material}_{rgb} \otimes \mathbf{lightColor}_{rgb}$$

- Diffuse light: the part that spreads in all direction (view independent)
- Specular light: the part that spreads mostly in the reflection direction (often same color as light source)

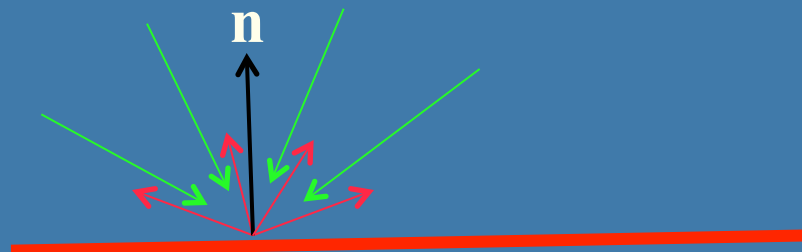
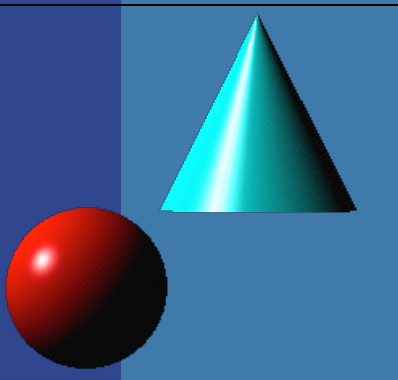


The ambient/diffuse/specular/emission model

- Also the most basic real-time model:
- Light interacts with material and change colors at bounces:

$$\mathbf{outColor}_{rgb} \sim \mathbf{material}_{rgb} \otimes \mathbf{lightColor}_{rgb}$$

- Diffuse light: the part that spreads in all direction (view independent)
- Specular light: the part that spreads mostly in the reflection direction (often same color as light source)
- Ambient light: incoming background light from all directions and spreads in all directions (view-independent and light-position independent color)

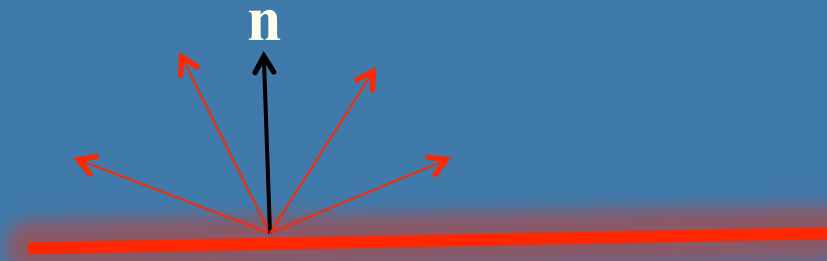
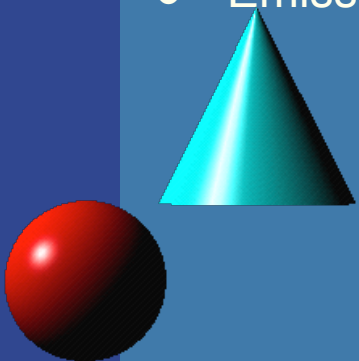


The ambient/diffuse/specular/emission model

- Also the most basic real-time model:
- Light interacts with material and change colors at bounces:

$$\mathbf{outColor}_{rgb} \sim \mathbf{material}_{rgb} \otimes \mathbf{lightColor}_{rgb}$$

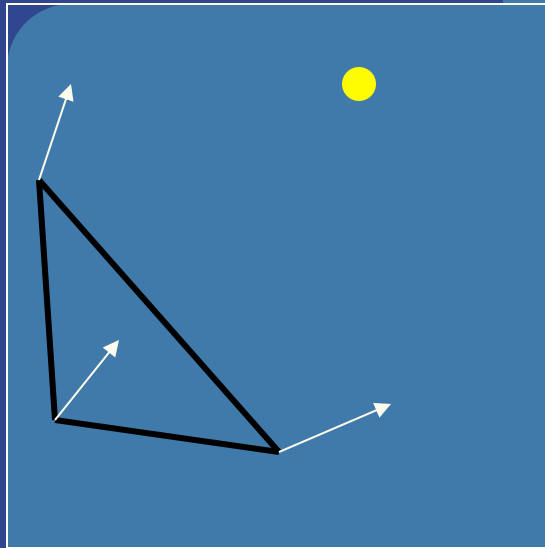
- Diffuse light: the part that spreads in all direction (view independent)
- Specular light: the part that spreads mostly in the reflection direction (often same color as light source)
- Ambient light: incoming background light from all directions and spreads in all directions (view-independent and light-position independent color)
- Emission: self-glowing surface



A basic lighting model

Light:

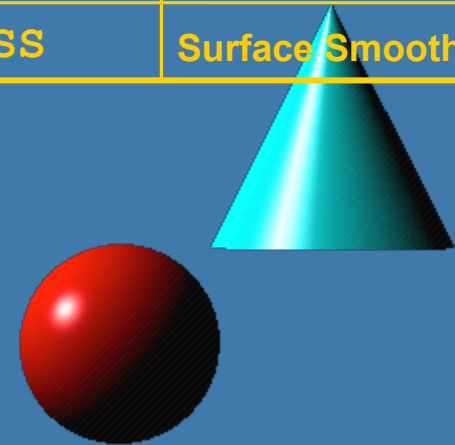
- Ambient (r,g,b,a)
- Diffuse (r,g,b,a)
- Specular (r,g,b,a)



Material:

- Ambient (r,g,b,a)
- Diffuse (r,g,b,a)
- Specular (r,g,b,a)
- Emission (r,g,b,a) = "självlysande färg"

DIFFUSE	Base color
SPECULAR	Highlight Color
AMBIENT	Low-light Color
EMISSION	Glow Color
SHININESS	Surface Smoothness



Ambient component: \mathbf{i}_{amb}

- Ad-hoc – tries to account for light coming from other surfaces
- Just add a constant color:

$$\mathbf{i}_{amb} = \mathbf{m}_{amb} \otimes \mathbf{s}_{amb}$$

i.e., $(i_r, i_g, i_b, i_a) = (m_r, m_g, m_b, m_a) (1_r, 1_g, 1_b, 1_a)$



Old: `glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient)`

Diffuse component : i_{diff}



- $\mathbf{i} = \mathbf{i}_{amb} + \mathbf{i}_{diff} + \mathbf{i}_{spec}$

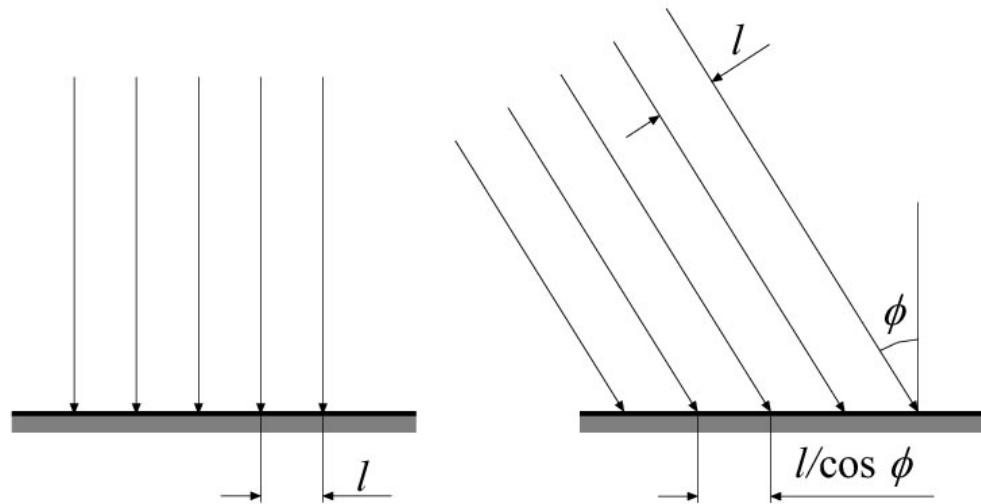
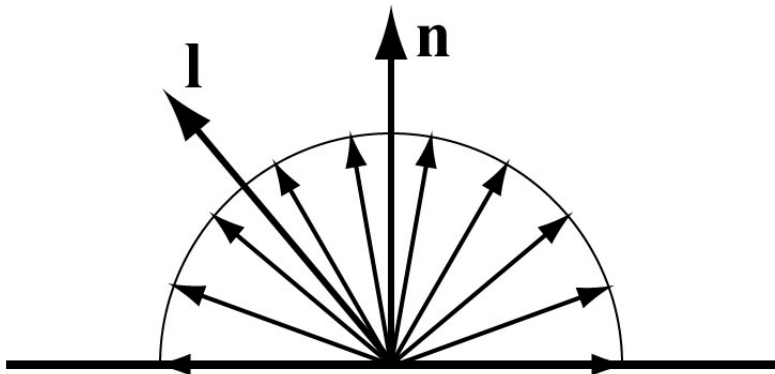
- Diffuse is Lambert's law: $i_{diff} = \mathbf{n} \cdot \mathbf{l} = \cos \phi$

- Photons are scattered equally in all directions

$$\mathbf{i}_{diff} = (\mathbf{n} \cdot \mathbf{l}) \mathbf{m}_{diff} \otimes \mathbf{s}_{diff}$$

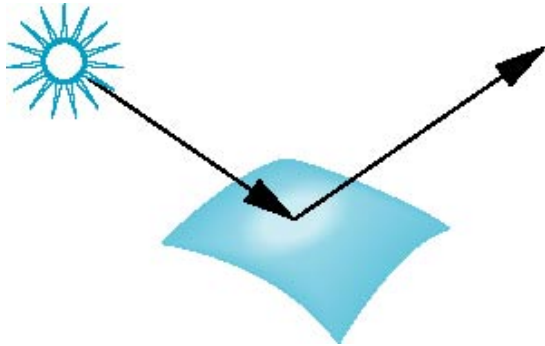
\mathbf{n} and \mathbf{l} are assumed being unit vectors

○ light source

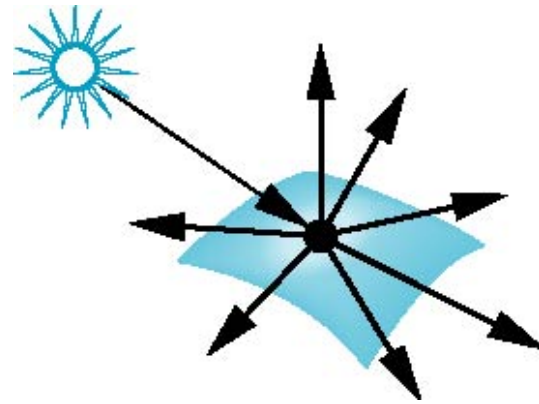


Lambertian Surfaces

- Perfectly diffuse reflector
- Light scattered equally in all directions



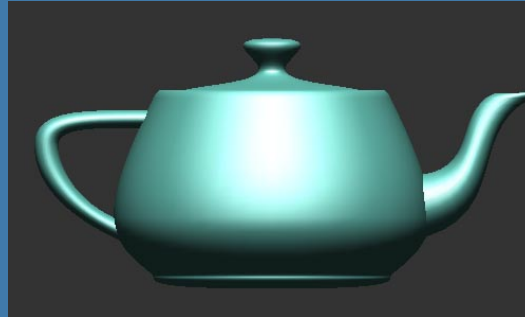
**Highly reflective
surface (specular)**



**Fully diffuse surface
(Lambertian)**

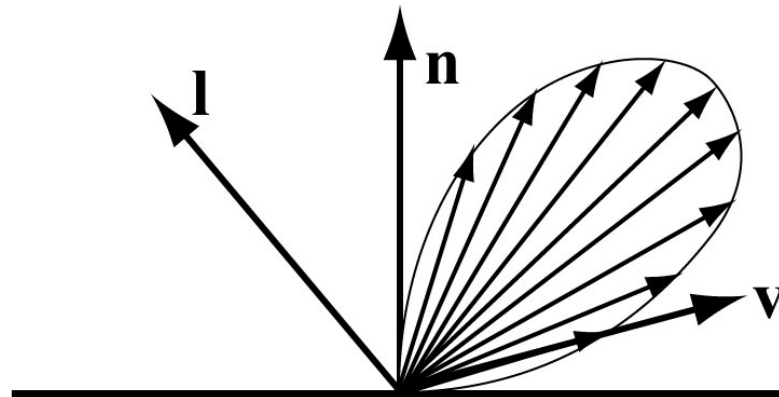
Lighting

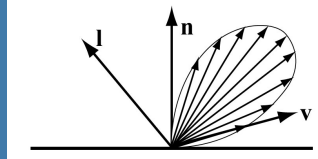
Specular component : i_{spec}



- Diffuse is dull (left)
- Specular: simulates a highlight

○ light source



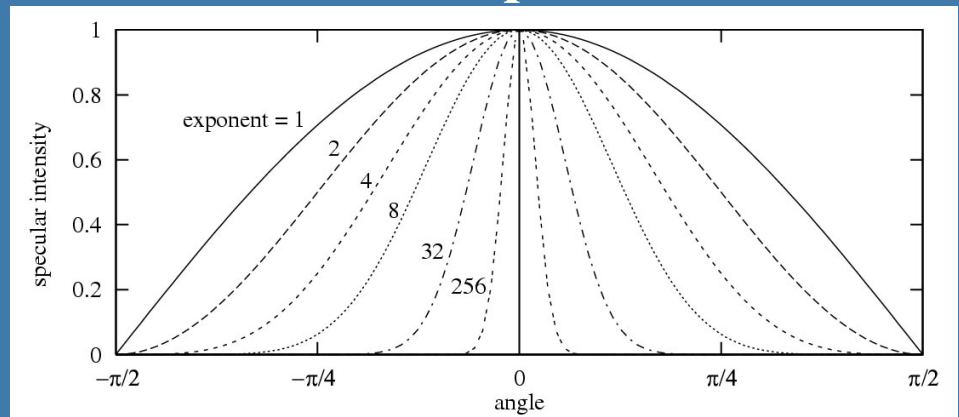
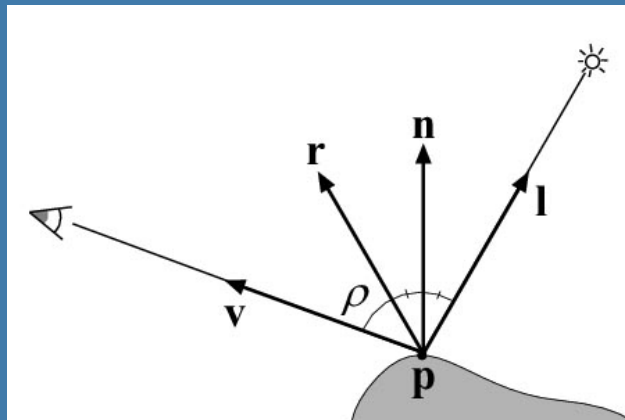
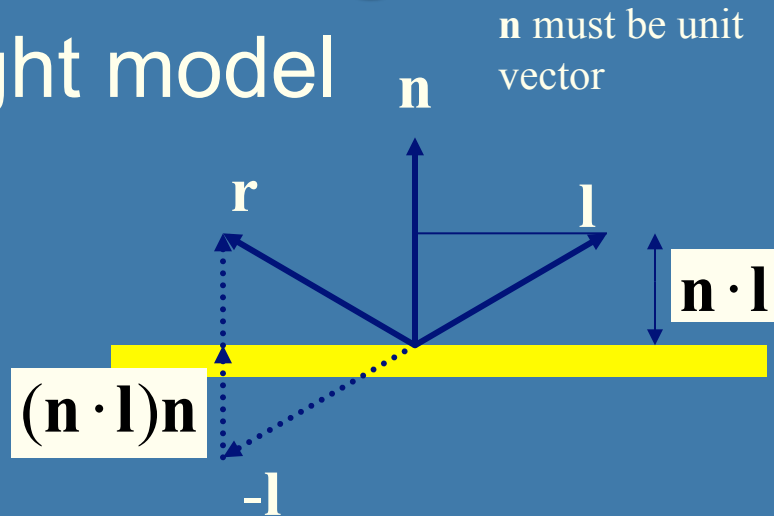


Specular component: Phong

- Phong specular highlight model
- Reflect l around n :

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

$$i_{spec} = (\mathbf{r} \cdot \mathbf{v})^{m_{shi}} = (\cos \rho)^{m_{shi}}$$



$$\mathbf{i}_{spec} = ((\mathbf{n} \cdot \mathbf{l}) < 0) ? 0 : \max(0, (\mathbf{r} \cdot \mathbf{v}))^{m_{shi}} \mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

- Next: Blinns highlight formula: $(\mathbf{n} \cdot \mathbf{h})^m$



Halfway Vector

Blinn proposed replacing $\mathbf{v} \cdot \mathbf{r}$ by $\mathbf{n} \cdot \mathbf{h}$ where

$$\mathbf{h} = (\mathbf{l} + \mathbf{v}) / |\mathbf{l} + \mathbf{v}|$$

$(\mathbf{l} + \mathbf{v})/2$ is halfway between \mathbf{l} and \mathbf{v}

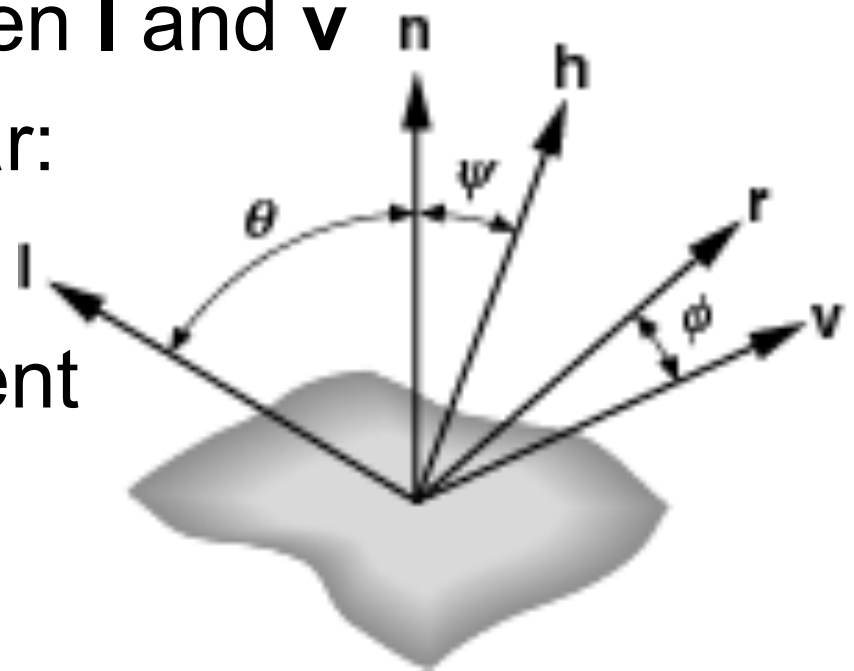
If \mathbf{n} , \mathbf{l} , and \mathbf{v} are coplanar:

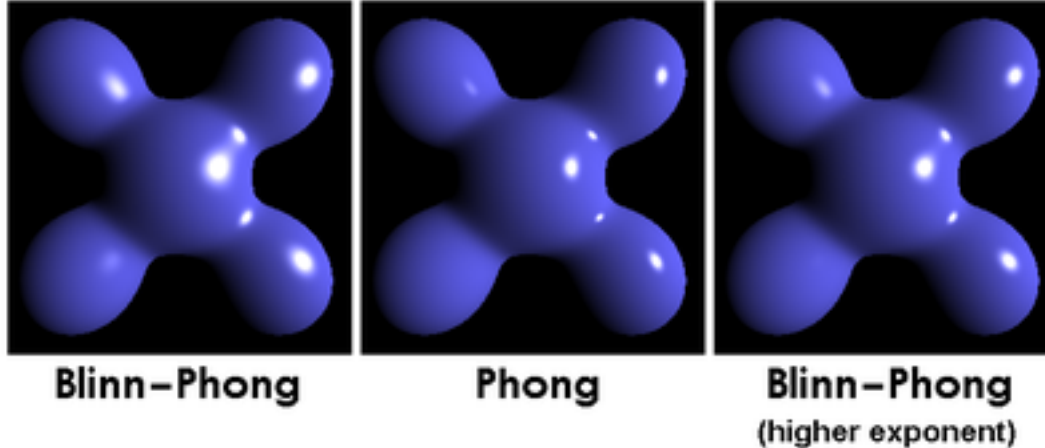
$$\psi = \phi/2$$

Must then adjust exponent

so that $(\mathbf{n} \cdot \mathbf{h})^{e'} \approx (\mathbf{r} \cdot \mathbf{v})^e$

$$(e' \approx 4e)$$





Efficiency

The Blinn rendering model is less efficient than pure Phong shading in most cases, since it contains a square root calculation. While the original Phong model only needs a simple vector reflection, this modified form takes more into consideration. However, as many CPUs and GPUs contain single and double precision square root functions (as standard features) and other instructions that can be used to speed up rendering -- the time penalty for this kind of shader will not be noticed in most implementations.

However, Blinn-Phong will be faster in the case where the viewer and light are treated to be at infinity. This is the case for directional lights. In this case, the half-angle vector is independent of position and surface curvature. It can be computed once for each light and then used for the entire frame, or indeed while light and viewpoint remain in the same relative position. The same is not true with Phong's original reflected light vector which depends on the surface curvature and must be recalculated for each pixel of the image (or for each vertex of the model in the case of vertex lighting).

In most cases where lights are not treated to be at infinity, for instance when using point lights, the original Phong model will be faster.

Lighting

$$i = i_{amb} + i_{diff} + i_{spec}$$



+



+



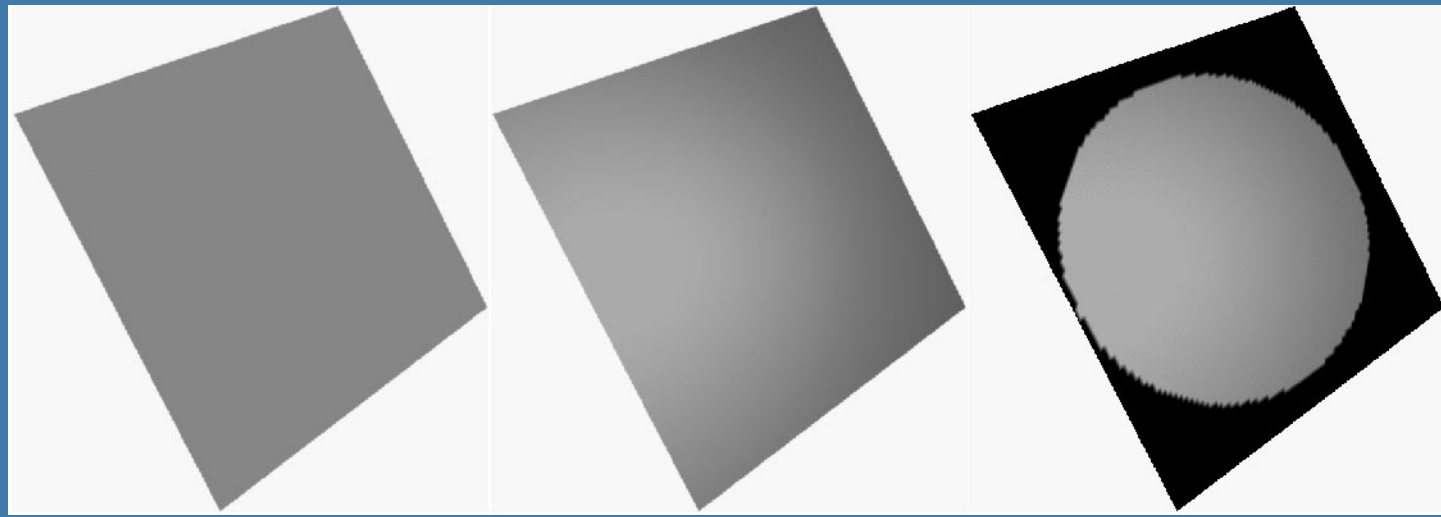
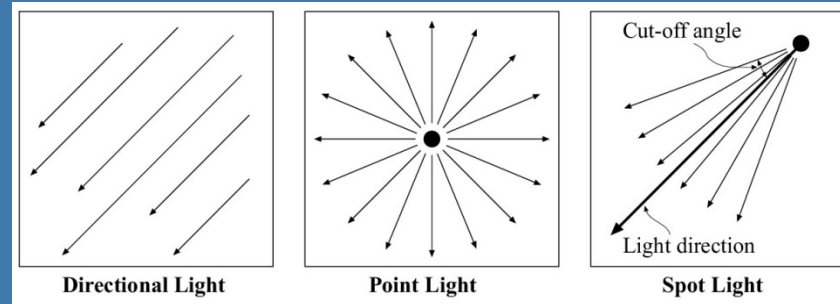
=



- This is just a hack!
- Has little to do with how reality works!

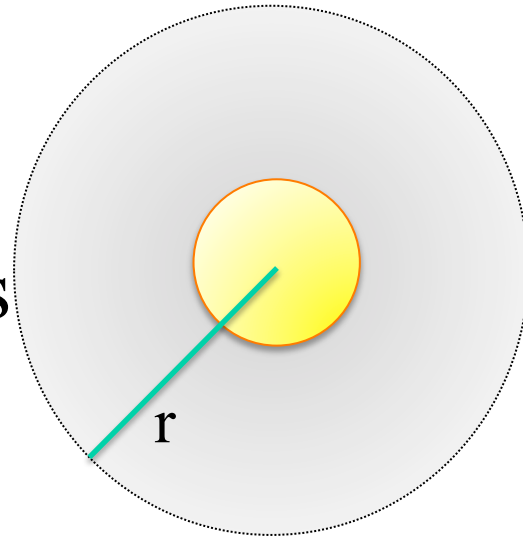
Additions to the lighting equation

- Depends on distance: $1/(a+bt+ct^2)$
- Can have more lights: just sum their respective contributions
- Different light types:



Clarifications

- Energy is emitted at equal proportions in all directions from a spherical radiator. Due to energy conservation, the intensity is proportional to the spherical area at distance r from the light center.
- $A = 4\pi r^2$
- Thus, the intensity scales $\sim 1/r^2$



OpenGL Example – (fixed pipeline)

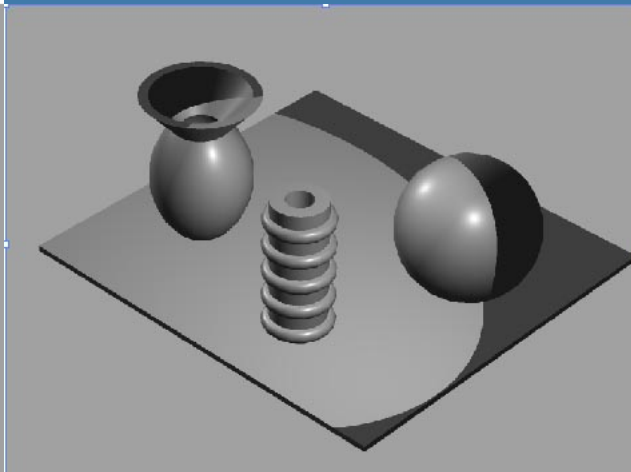
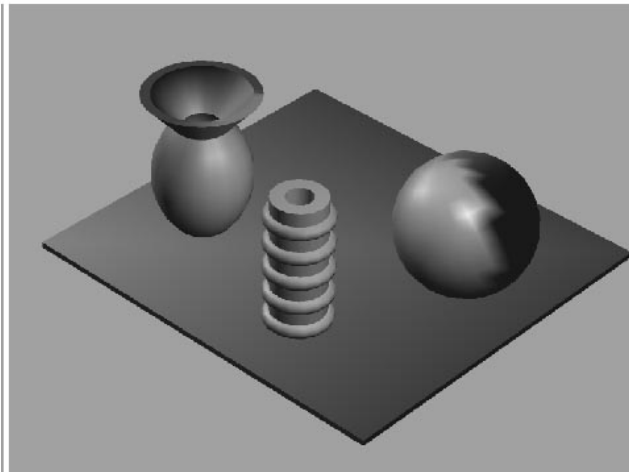
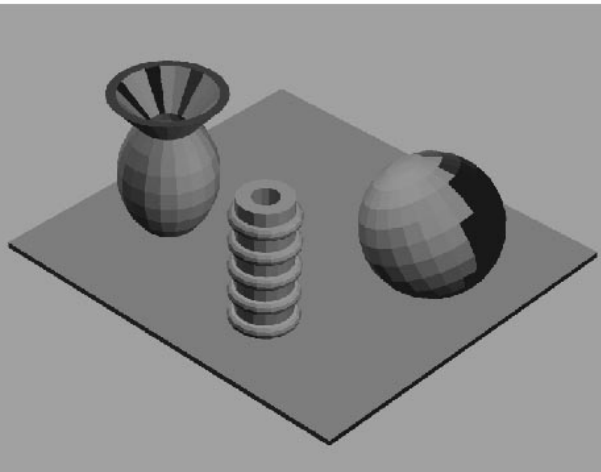
Sets constant (no) attenuation:

```
glLight(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 1.0);  
glLight(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.0);  
glLight(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.0);
```

Attenuation often looks unnatural in OpenGL, due to only 8-bit colors (too small intensity range)

Shading

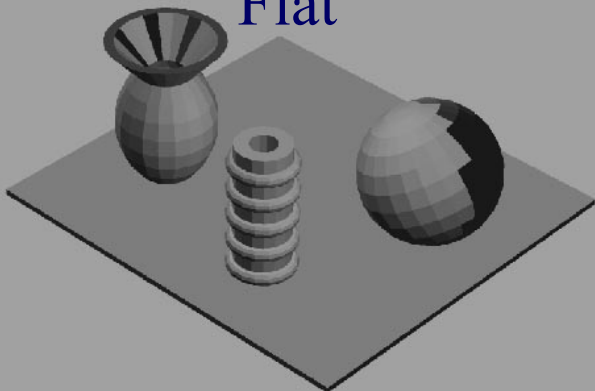
- Shading: do lighting (at e.g. vertices) and determine pixel's colors from these
- Three common types of shading:
 - Flat, Gouraud, and Phong



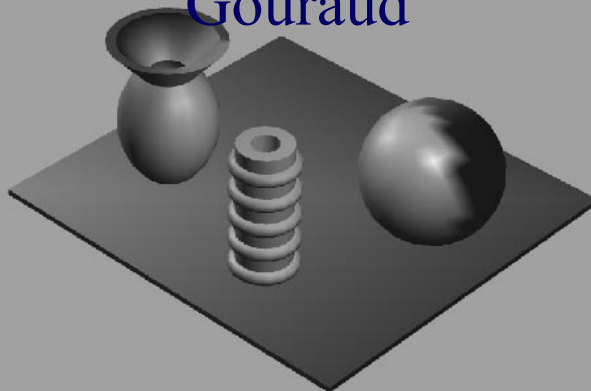
Shading

- Three common types of shading:
 - Flat, Gouraud, and Phong
- In standard Gouraud shading the lighting is computed per triangle vertex and for each pixel, the color is interpolated from the colors at the vertices.
- In Phong Shading the lighting is not computed per vertex. Instead the normal is interpolated per pixel from the normals defined at the vertices and full lighting is computed per pixel using this normal. This is of course more expensive but looks better.

Flat



Gouraud



Phong



```
// Vertex Shader
#version 130
```

Gouraud Shading Code

```
in vec3 vertex;
in vec3 normal;
uniform vec4 mtrlAmb, mtrlDiffuse, mtrlSpec, mtrlEmission;
uniform vec4 lightAmb, lightDiffuse, lightSpec, lightEmission;
uniform float shininess;
uniform mat4 modelViewProjectionMatrix, normalMatrix, modelViewMatrix;
uniform vec4 lightPos; // in view space
out vec3 outColor;

void main()
{
    gl_Position = modelViewProjectionMatrix*vec4(vertex,1);
    // ambient
    outColor = lightAmb * mtrlAmb;

    // diffuse
    vertex = vec3(modelViewMatrix * vec4(vertex,1));
    normal = normalize(normalMatrix * normal);
    vec3 lightDirection = normalize(lightPos - vertex.xyz);
    float intensity=max(0, dot(normal, lightDirection))
    outColor += lightDiffuse*mtrlDiffuse*intensity;

    // specular
    vec3 viewVec = -vertex.xyz; // because we are in view space
    vec3 reflVec = -lightDirection + normal*(2*dot(normal*lightDirection))
    intensity=pow(max(0,(dot(reflVec, viewVec)), shininess));
    outColor += lightSpec * mtrlSpec * max(0,intensity);
}
```

For one light source

```
// Fragment Shader:
#version 130
in vec3 outColor;
out vec4 fragColor;

void main()
{
    fragColor = vec4(outColor,1);
}
```

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

$$\mathbf{i}_{spec} = ((\mathbf{n} \cdot \mathbf{l}) < 0) ? 0 : \max(0, (\mathbf{r} \cdot \mathbf{v}))^{m_{shi}} \mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

Phong Shading Code For one light source

```
// Vertex Shader
#version 130

in vec3 vertex;
in vec3 normal;
uniform vec3 mtrlAmb;
uniform vec3 lightAmb;
uniform vec4 lightPos;
uniform mat4 modelViewProjectionMatrix;
uniform mat4 normalMatrix;
uniform mat4 modelViewMatrix;
out vec3 outColor;
out vec3 N;
out vec3 viewVec;
out vec3 lightDirection;

void main()
{
    gl_Position = modelViewProjectionMatrix *
        vec4(vertex,1);

    // ambient
    outColor = lightAmb * mtrlAmb;

    N= normalize(normalMatrix * normal);
    lightDirection = normalize(lightPos - vertex.xyz);
    viewVec=-vec3(modelViewMatrix*vec4(vertex,1));
}
```

```
// Fragment Shader:
#version 130

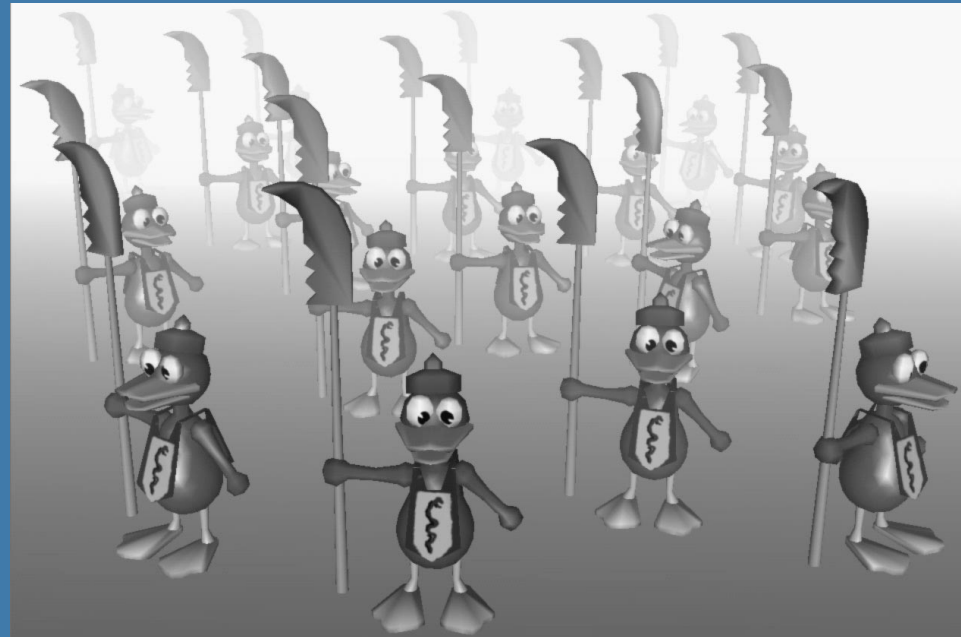
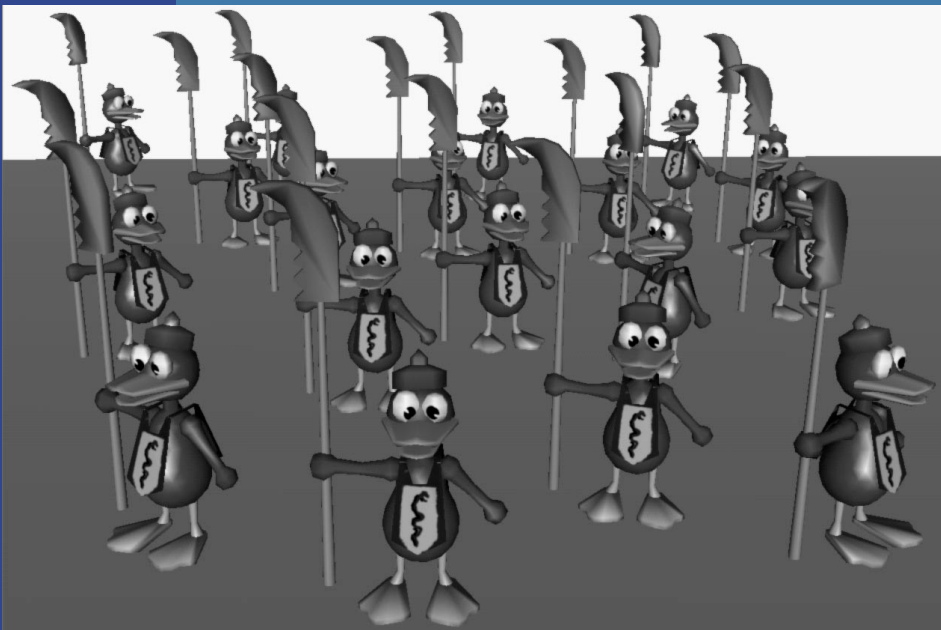
in vec3 outColor, lightDirection, N, pos;
in vec3 viewVec;
uniform vec3 mtrlDiffuse, mtrlSpec, mtrlEmission;
uniform vec3 lightDiffuse, lightSpec, lightEmission;
uniform float shininess;
out vec4 fragColor;

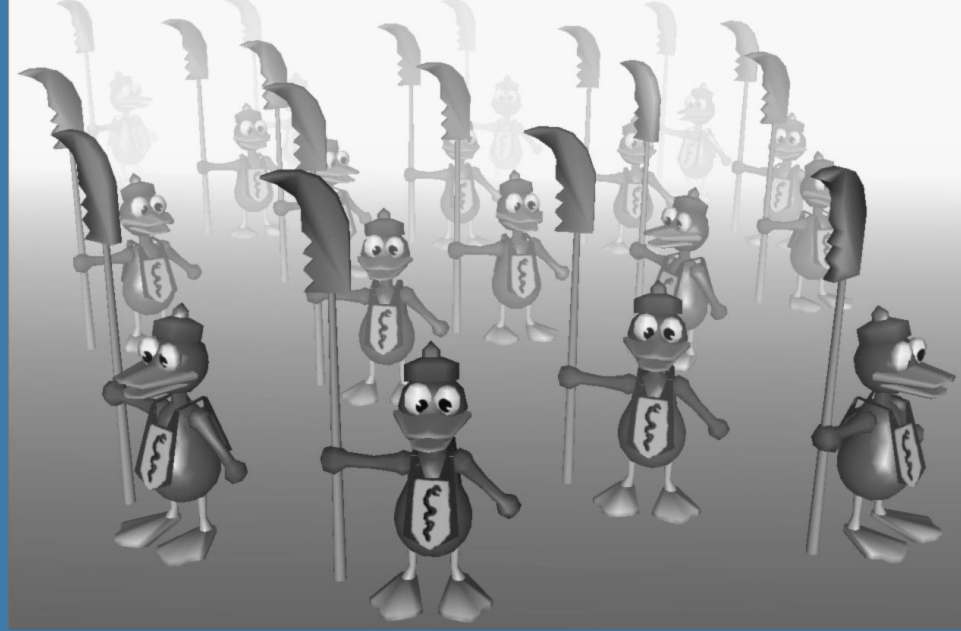
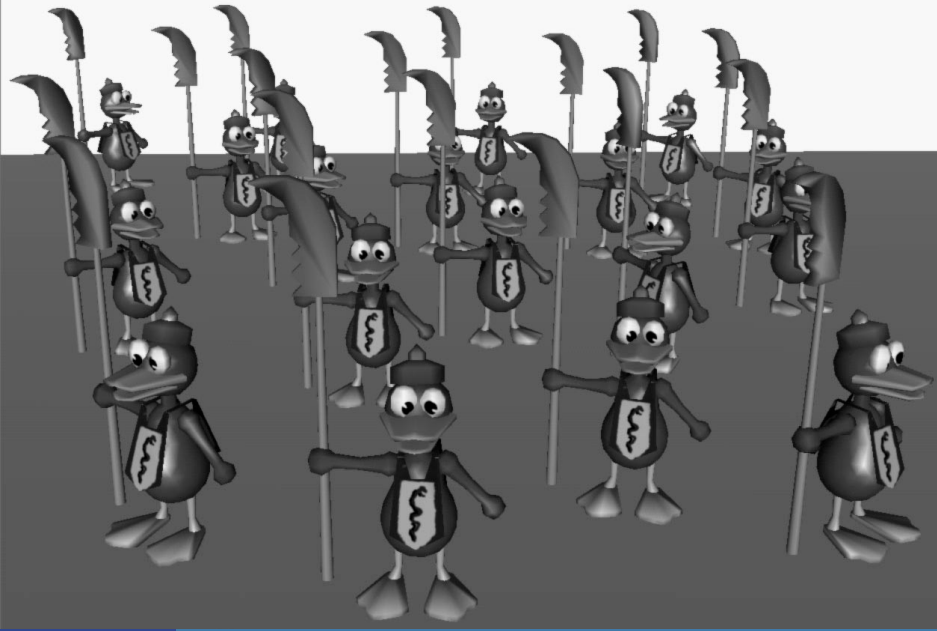
void main()
{
    N = normalize(N); // renormalize due to the interpolation
    // diffuse
    float intensity=max(0, dot(N, lightDirection))
    outColor += lightDiffuse*mtrlDiffuse*intensity;

    // specular
    vec3 reflVec = -lightDirection + N*(2*dot(N*lightDirection));
    intensity=pow(max(0,(dot(reflVec, viewVec)), shininess));
    outColor += lightSpec * mtrlSpec * max(0,intensity);
    fragColor = vec4(outColor,1);
}
```

Fog

- Simple atmospheric effect
 - A little better realism
 - Help in determining distances





- Color of fog: \mathbf{c}_f color of surface: \mathbf{c}_s

$$\mathbf{c}_p = f\mathbf{c}_s + (1 - f)\mathbf{c}_f \quad f \in [0,1]$$

- How to compute f ?
- 3 ways: linear, exponential, exponential-squared
- Linear:

$$f = \frac{z_{end} - z_p}{z_{end} - z_{start}}$$

Fog

The equation for GL_EXP fog is

$$f = e^{-(\text{density} \cdot x)}$$

Rationale:

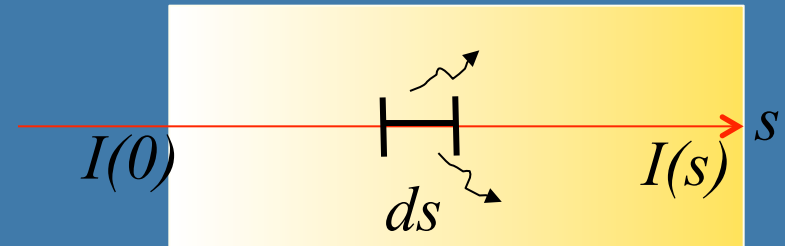
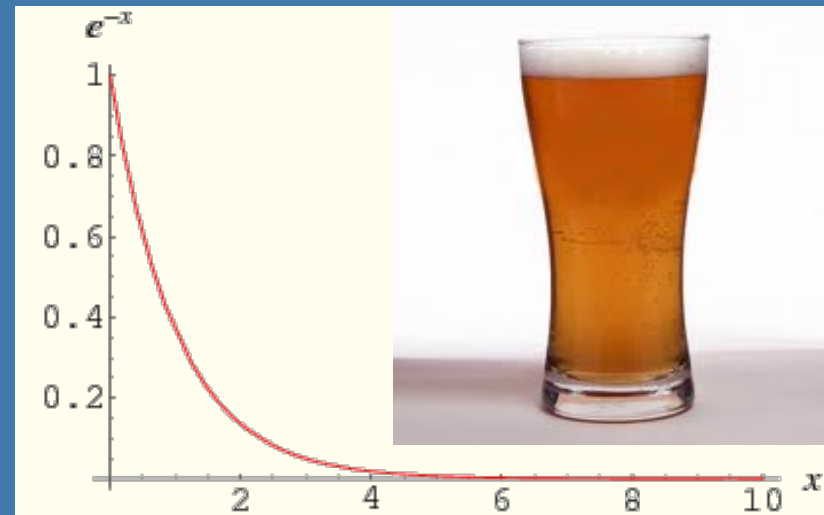
Constant intensity decrease at greater distance due to out scattering and absorption.

$$dI = -C I ds$$

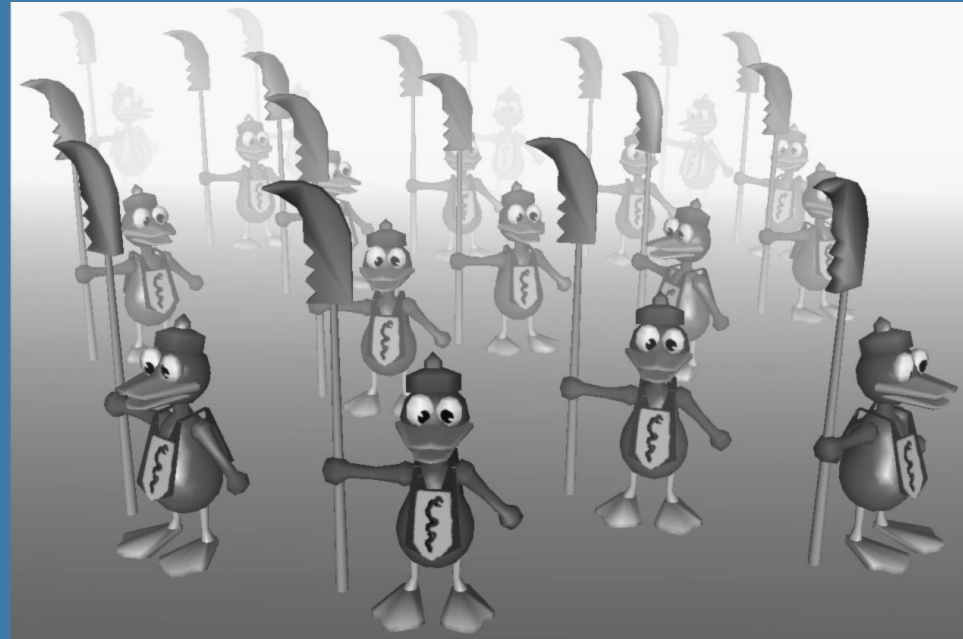
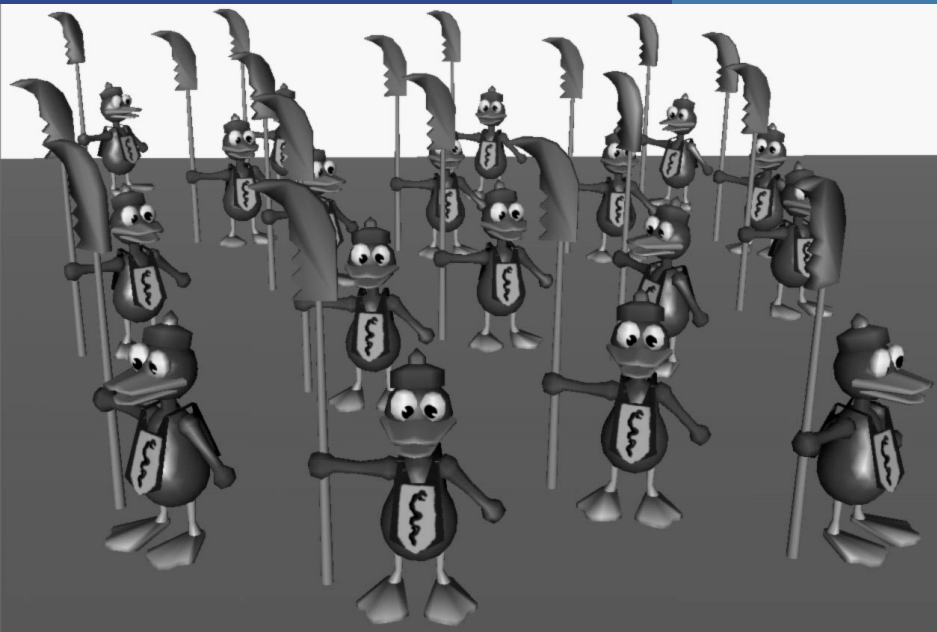
$$I(s) = I(0) e^{-\text{density} \cdot s}$$

The equation for GL_EXP2 fog is

$$f = e^{-(\text{density} \cdot c)^2}$$



Fog example

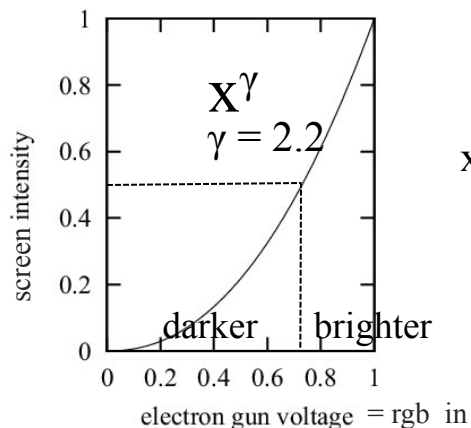
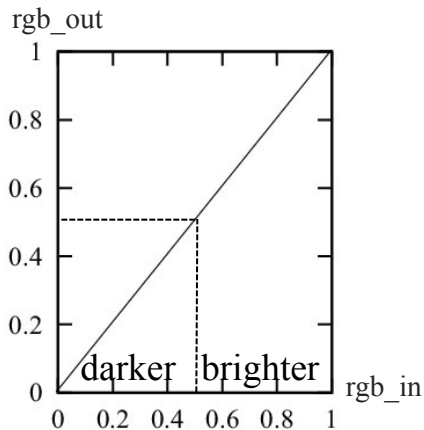


- Often just a matter of
 - Choosing fog color
 - Choosing fog model
 - Old OpenGL – just turn it on. New OpenGL – program it yourself in the fragment shader

Fog in up-direction

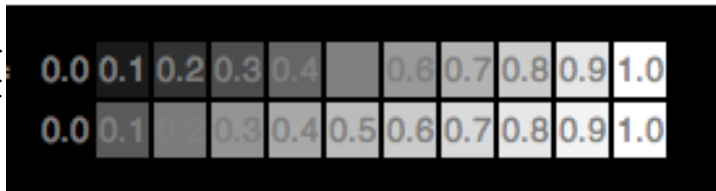


Gamma correction



better distribution for humans.
Our eyes have non-linear sensitivity and monitors have limited brightness

x^γ : perceived lin. intensity:
linear intensity:

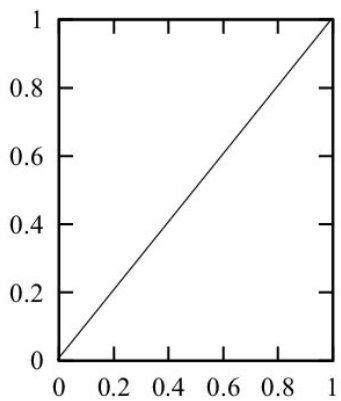


Intensities: x^γ vs linear

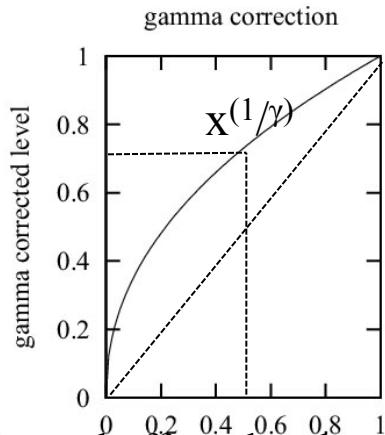
Lighting computes rgb color intensities in linear space from $[0,1]$

However, CRT-monitor output is non-linear. Has more precision for darker regions. Very Good! But we need to adapt the input to utilize this. Else, our images will be too dark.

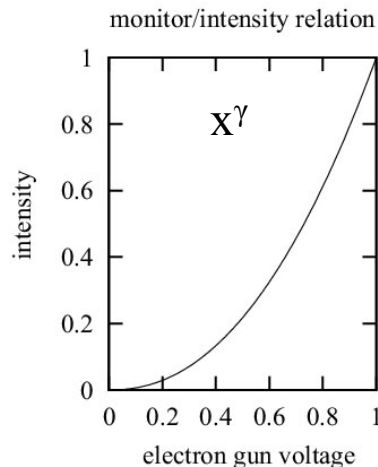
So, store color intensities with more precision for darker colors: convert to $x^{(1/\gamma)}$ before storing in 8 bits in the frame buffer. Conversion to $x^{(1/\gamma)}$ is called gamma correction.



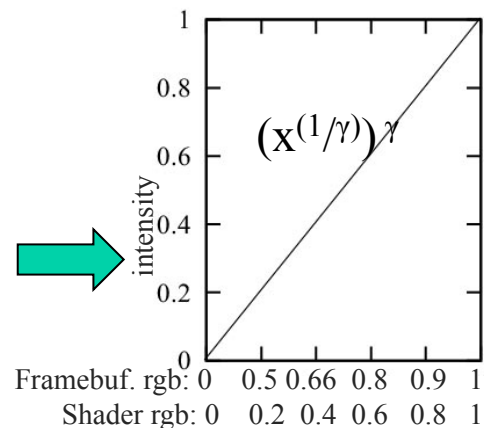
$x^{(1/\gamma)}$



Frame buffer rgb colors.
“Dark pixels are made brighter”

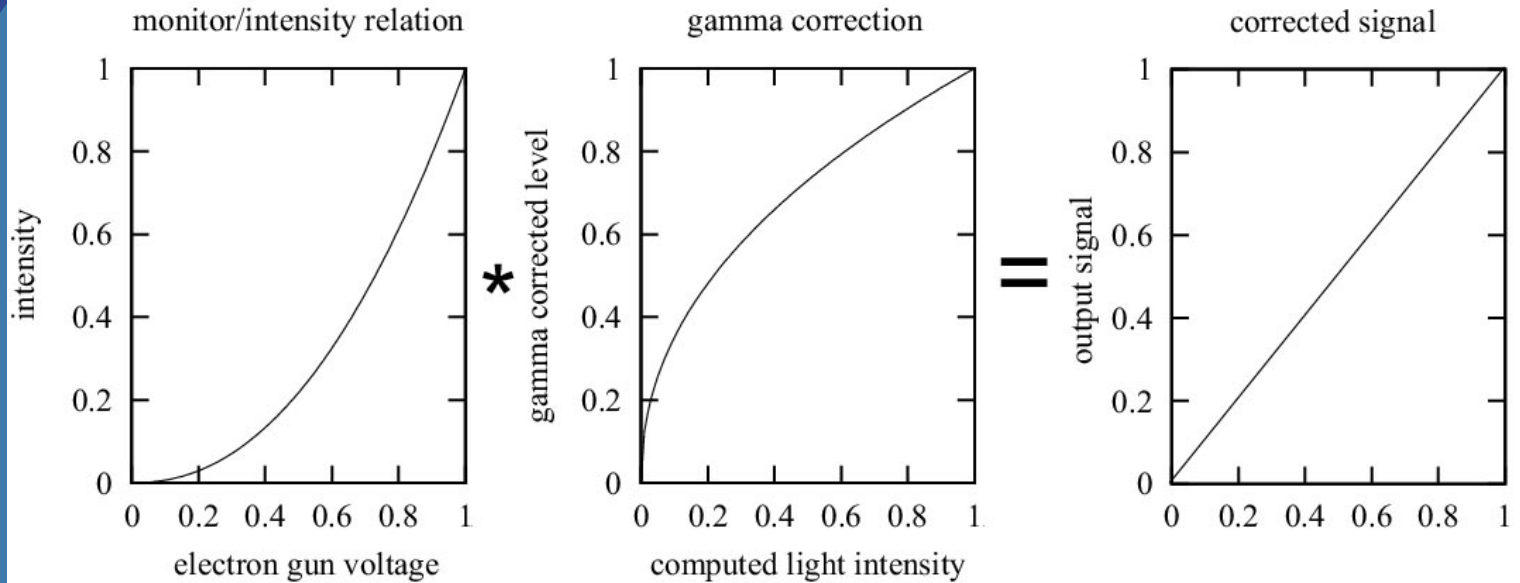


Displayed by CRT



Linear output again, but redistributed precision.

Gamma correction



- If input to gun is 0.5, then you don't get 0.5 as output in intensity
- Instead, gamma correct that signal: gives linear relationship

Gamma correction

$$I = a(V + \varepsilon)^\gamma$$

- I =intensity on screen
- V =input voltage (electron gun)
- $a, \varepsilon, \text{ and } \gamma$ are constants for each system
- Common gamma values: 2.2-2.6
- Assuming $\varepsilon=0$, gamma correction is:

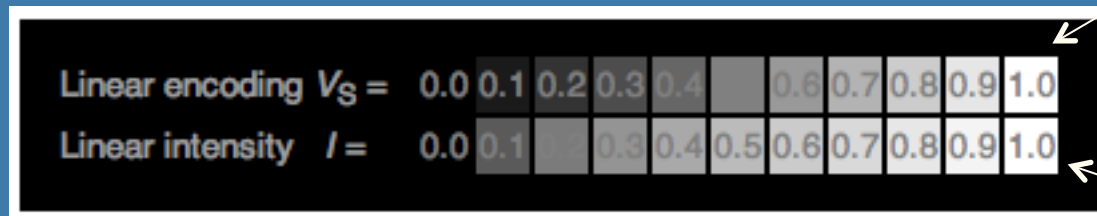
$$C = C_i^{(1/\gamma)}$$

Why is it important to care about gamma correction?

- Portability across platforms
- Image quality
 - Texturing
 - Anti-aliasing
- One solution is to put gamma correction in hardware...
- sRGB assumes $\gamma=2.2$
- Can use `EXT_framebuffer_sRGB` to render with gamma correction directly to frame buffer

Gamma correction today

- Standard is 2.2
- Happens to give more efficient color space when compressing intensity from 32-bit floats to 8-bits. Thus, still motivated.



better distribution for humans

Gamma of 2.2

Truth

On most displays (those with gamma of about 2.2), one can observe that the linear intensity output (bottom) has a large jump in perceived brightness between the intensity values 0.0 and 0.1, while the steps at the higher end of the scale are hardly perceptible. A linear input that has a nonlinearly-increasing intensity (upper), will show much more even steps in perceived brightness.

Transparency and alpha

- Transparency
 - Very simple in real-time contexts
- The tool: alpha blending (mix two colors)
- Alpha (α) is another component in the frame buffer, or material for a triangle
 - Represents the opacity
 - 1.0 is totally opaque
 - 0.0 is totally transparent
- The over operator:

$$\mathbf{c}_o = \alpha \mathbf{c}_s + (1 - \alpha) \mathbf{c}_d$$

Rendered object

$$\mathbf{c}_o = \alpha \mathbf{c}_s + (1 - \alpha) \mathbf{c}_d$$

Transparency

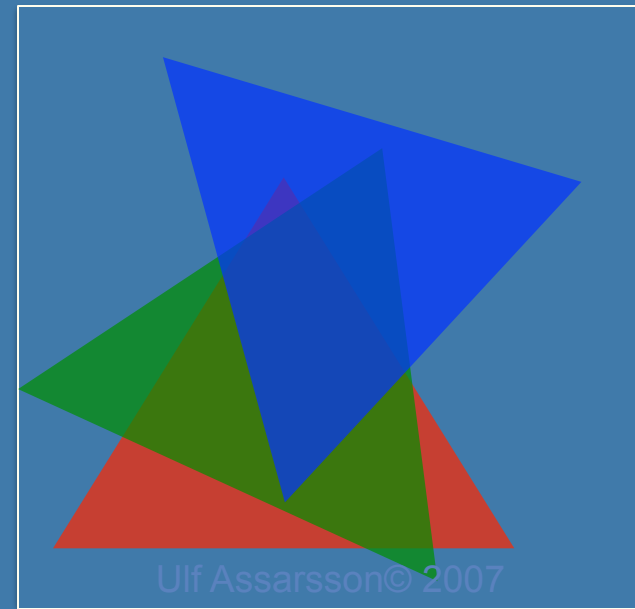
Rendered object

Background

- Need to sort the transparent objects
 - Render back to front (blending is order dep.)
 - See next slide...
- Lots of different other blending modes
- Can store $RGB\alpha$ in textures as well



So the texels with $\alpha=0.0$
do not not hide the
objects behind



Transparency

- Need to sort the transparent objects
 - **First, render all non-transparent triangles as usual.**
 - **Then, sort all transparent triangles and render back-to-front with blending enabled. (and using standard depth test)**
 - **The reason is to avoid problems with the depth test and because the blending operation (i.e., over operator) is order dependent.**

$$\mathbf{c}_o = \alpha \mathbf{c}_s + (1 - \alpha) \mathbf{c}_d$$

Blending

- Used for
 - Transparency
 - `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`
 - Effects (shadows, reflections)
 - (Complex materials)
 - Quake3 used up to 10 rendering passes, blending together contributions such as:
 - Diffuse lighting (for hard shadows)
 - Bump maps
 - Base texture
 - Specular and emissive lighting
 - Volumetric/atmospheric effects
 - Enable with `glEnable(GL_BLEND)`

