

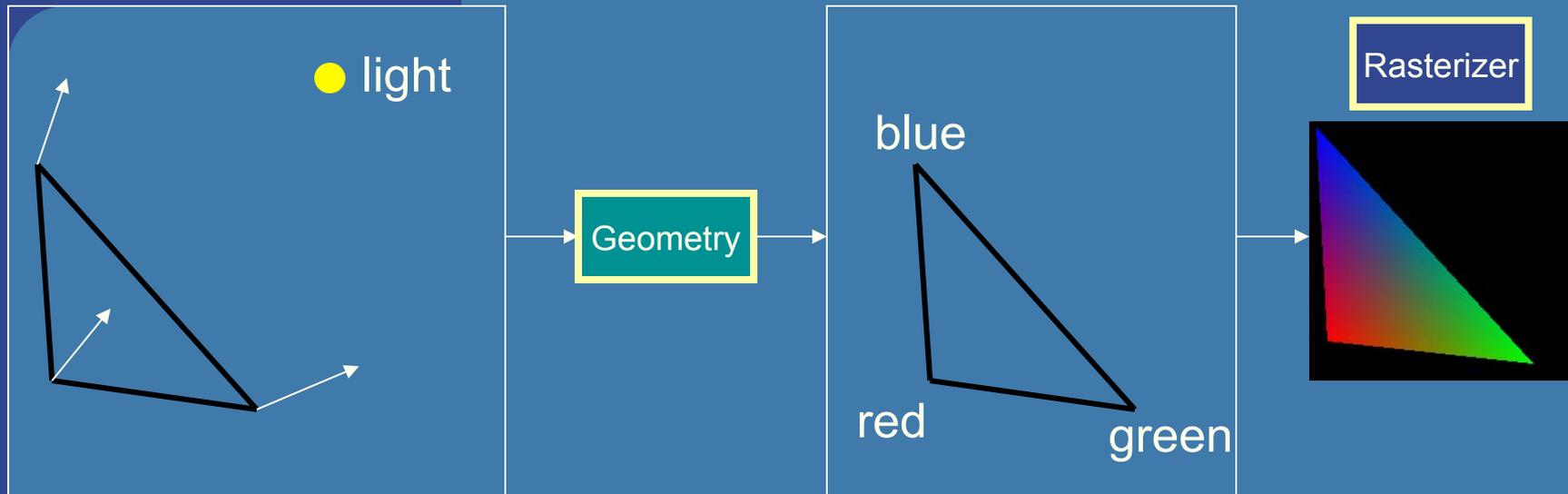
Shading

Most slides by Tomas Akenine-Möller
Department of Computer Engineering
Chalmers University of Technology

Overview of today's lecture

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

Compute lighting at vertices, then interpolate over triangle

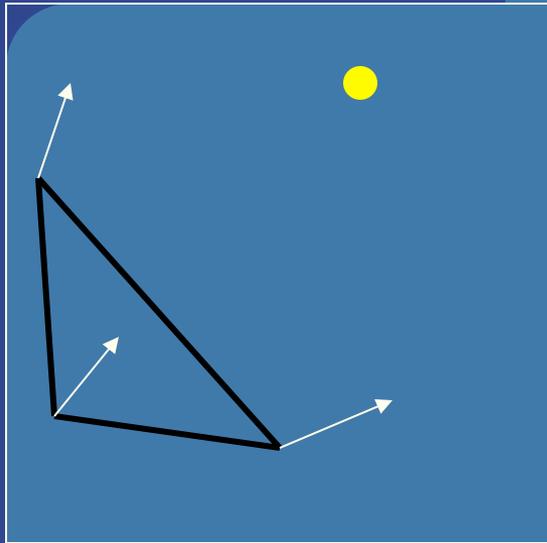


- How compute lighting?
- We could set colors per vertex manually
- For a **little** more realism, compute lighting from
 - Light sources
 - Material properties
 - Geometrical relationships

A basic lighting model

Light:

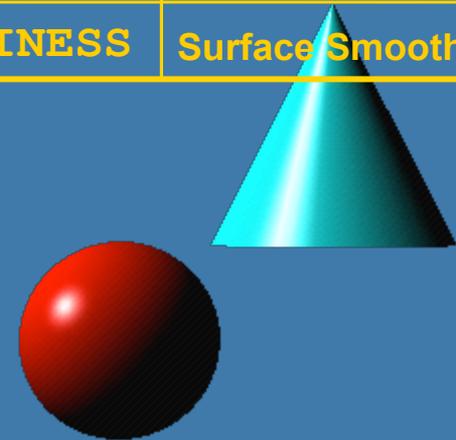
- Ambient (r,g,b,a)
- Diffuse (r,g,b,a)
- Specular (r,g,b,a)



Material:

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

GL_DIFFUSE	Base color
GL_SPECULAR	Highlight Color
GL_AMBIENT	Low-light Color
GL_EMISSION	Glow Color
GL_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}



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

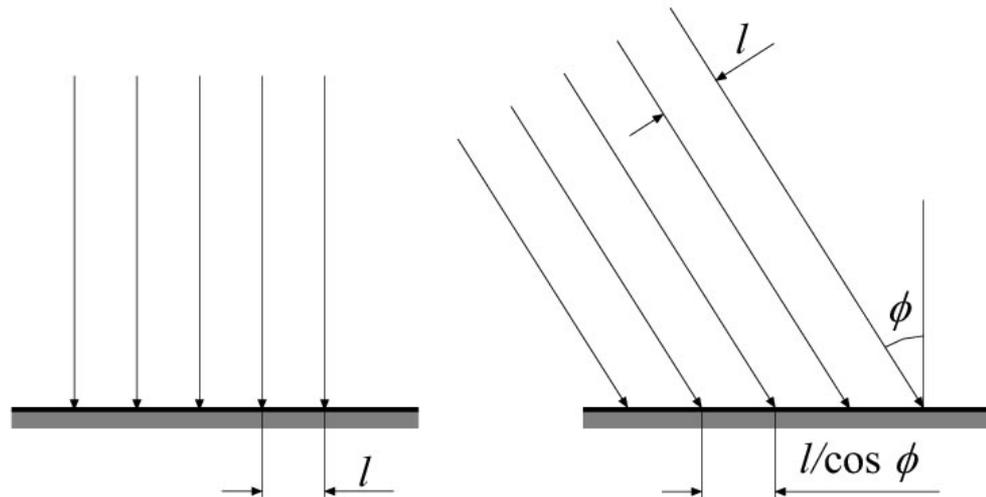
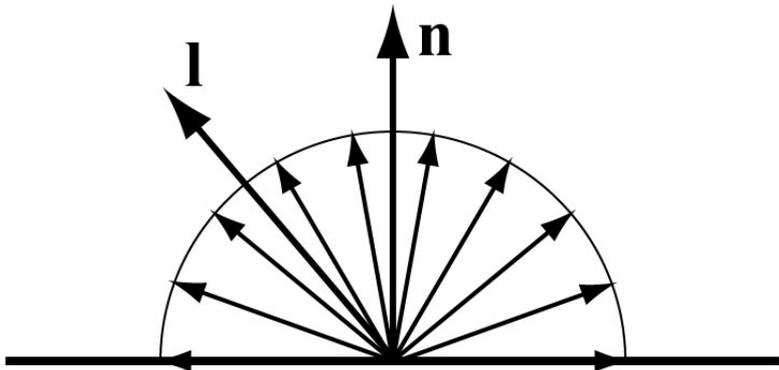
- Diffuse is Lambert's law: $i_{diff} = \mathbf{n} \cdot \mathbf{l} = \cos \phi$

- Photons are scattered equally in all directions

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

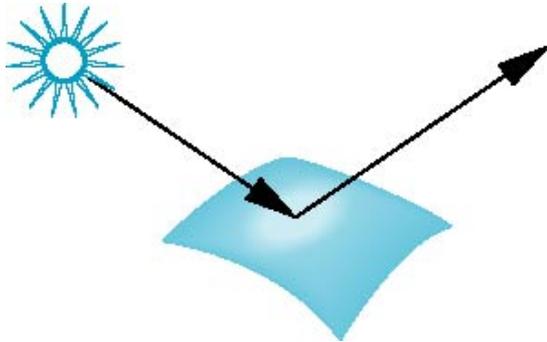
\mathbf{n} and \mathbf{l} are assumed being unit vectors

○ light source

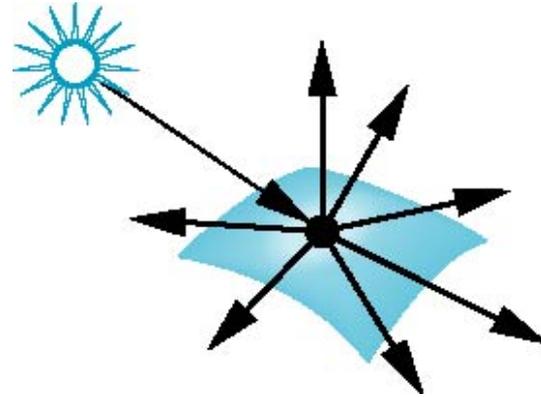


Lambertian Surfaces

- Perfectly diffuse reflector
- Light scattered equally in all directions



**Highly reflective
surface (specular)**



**Fully diffuse surface
(Lambertian)**

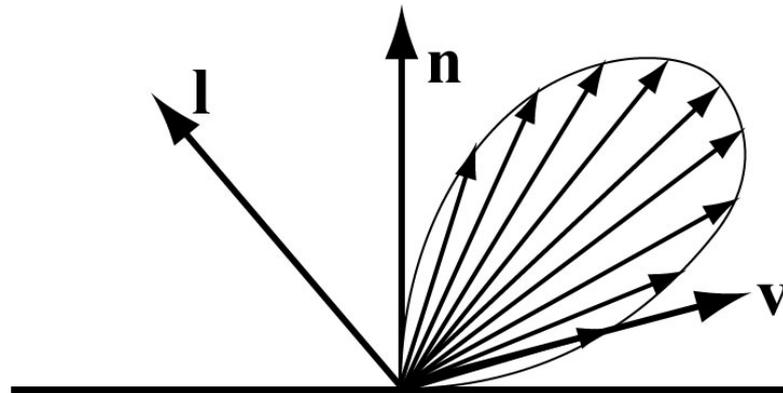
Lighting

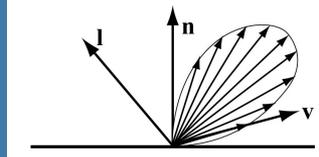
Specular component : i_{spec}



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

○ light source



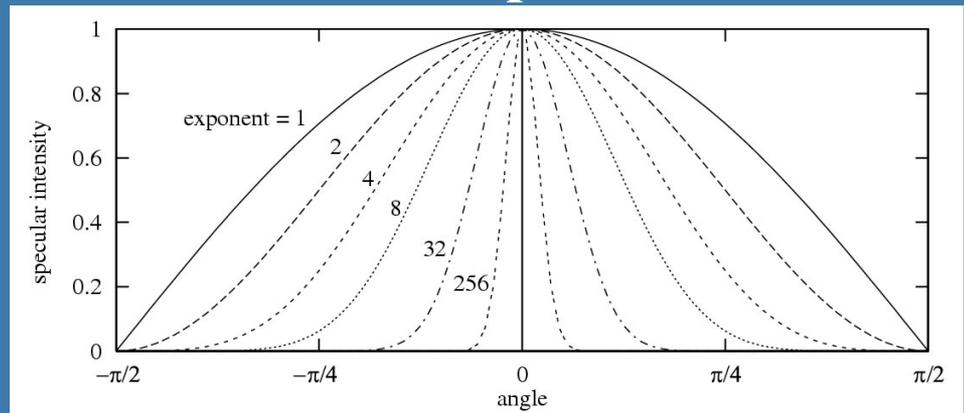
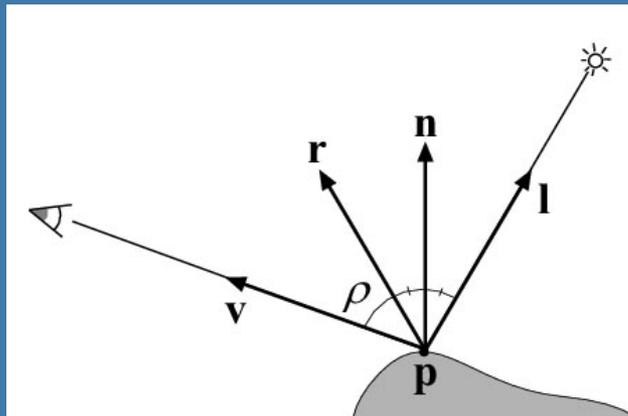
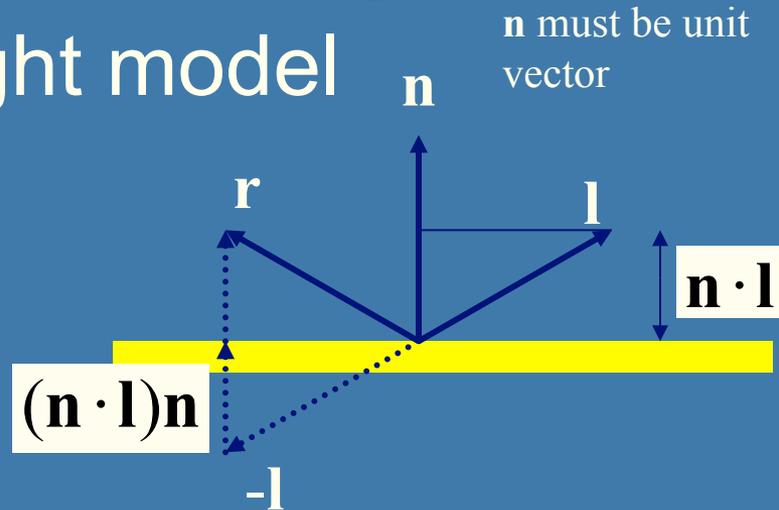


Specular component: Phong

- 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}

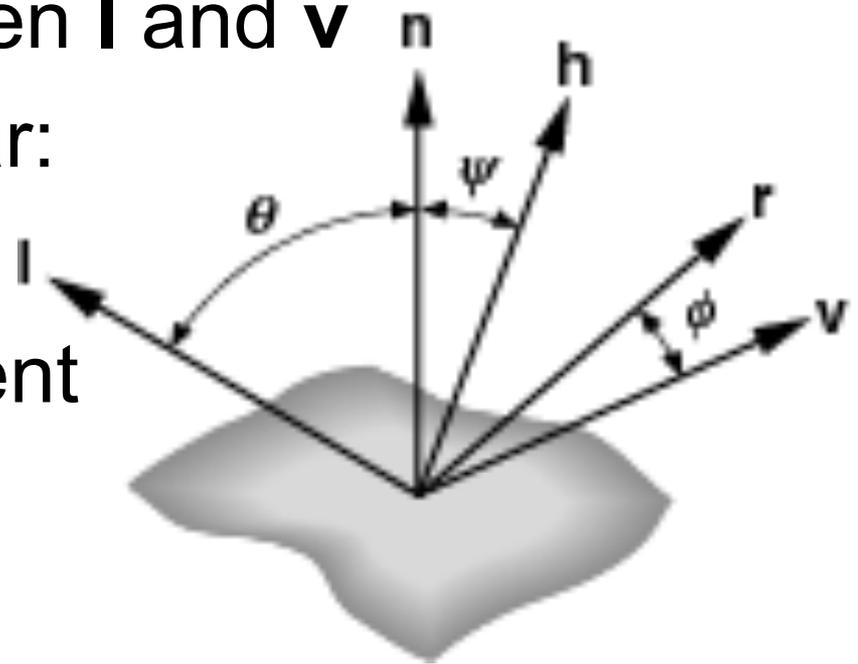
If \mathbf{n} , \mathbf{l} , and \mathbf{v} are coplanar:

$$\psi = \phi/2$$

Must then adjust exponent

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

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



Lighting

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



+



+



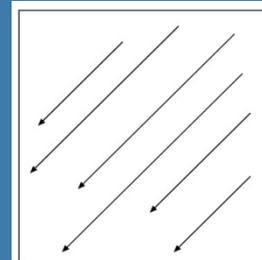
=



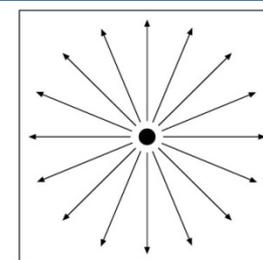
- This is just a hack!
- Has little to do with how reality works!

Additions to the lighting equation

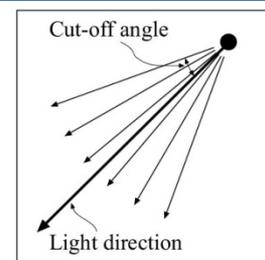
- Depends on distance: $1/(a+bt+ct^2)$
- Can have more lights: just sum their respective contributions
- Different light types:



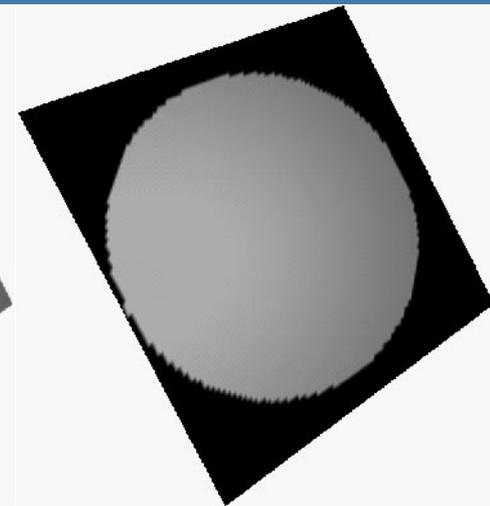
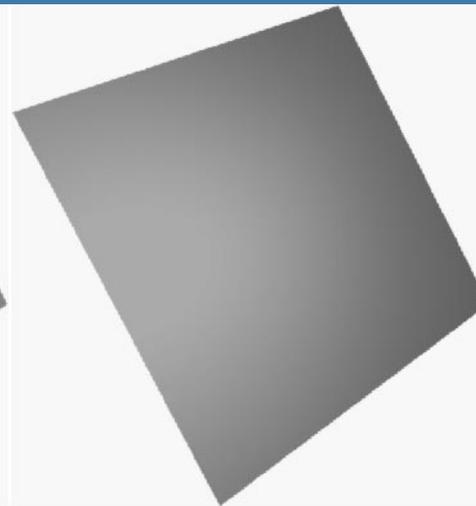
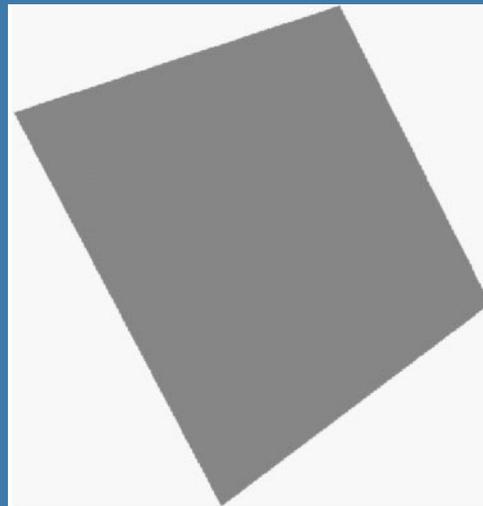
Directional Light



Point Light



Spot Light



OpenGL Example – (fixed pipeline)

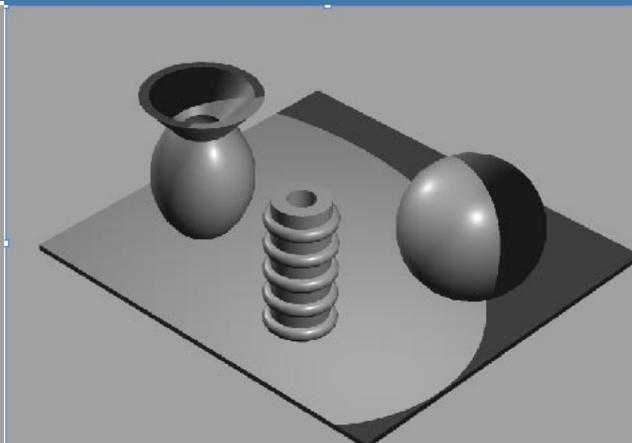
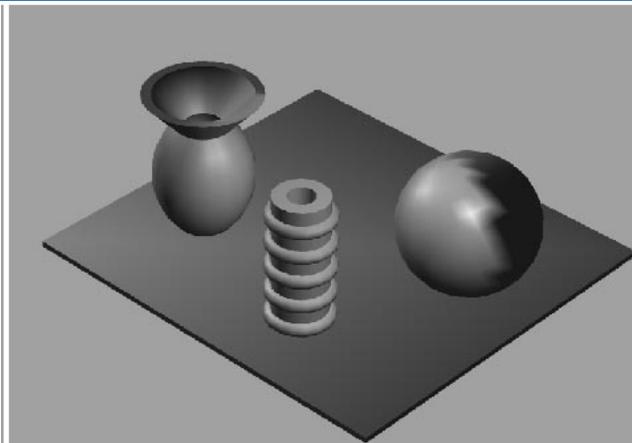
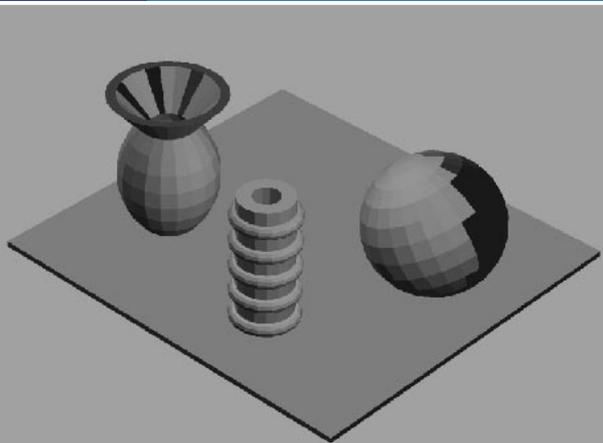
Sets constant (no) attenuation:

```
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

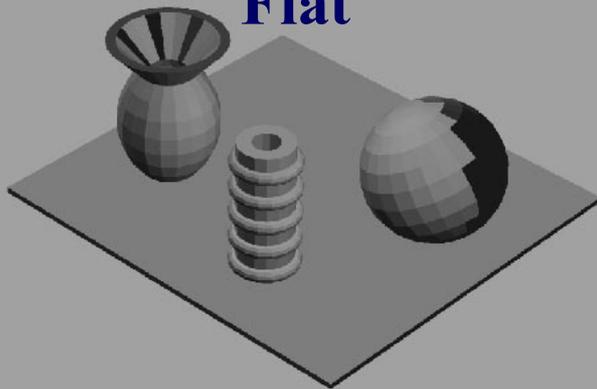
- Shading: do lighting (at e.g. vertices) and determine pixel's colors from these
- Three common types of shading:
 - Flat, Goraud, and Phong
 - Old: `glShadeModel(GL_FLAT / GL_SMOOTH);`



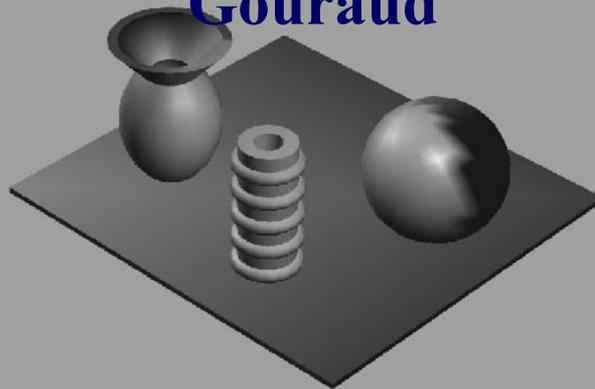
Shading

- 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 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 vec3 mtrlAmb, mtrlDiffuse, mtrlSpec, mtrlEmission;
uniform vec3 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);
}
```

```
// 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

```
// 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

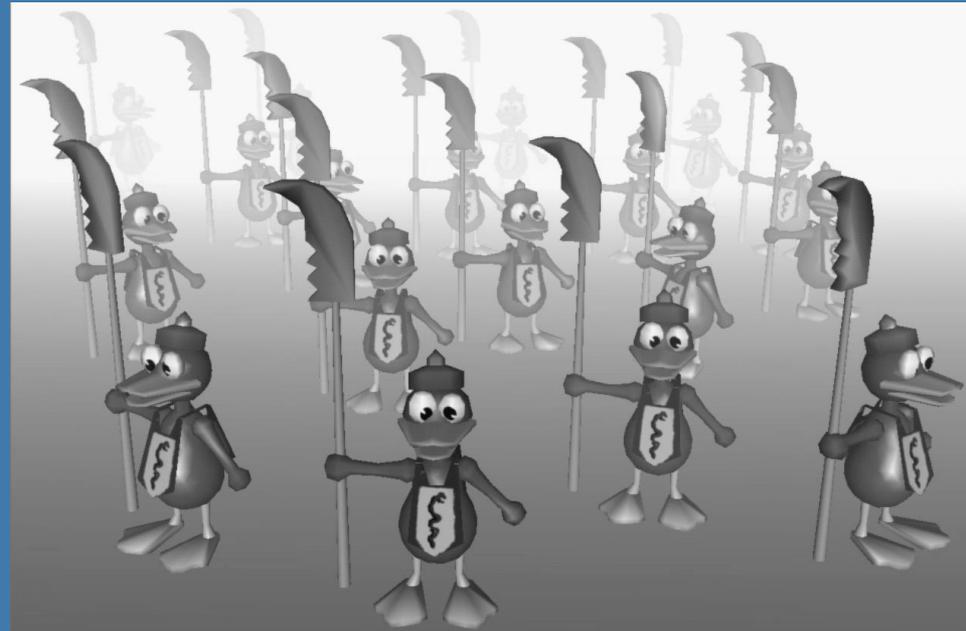
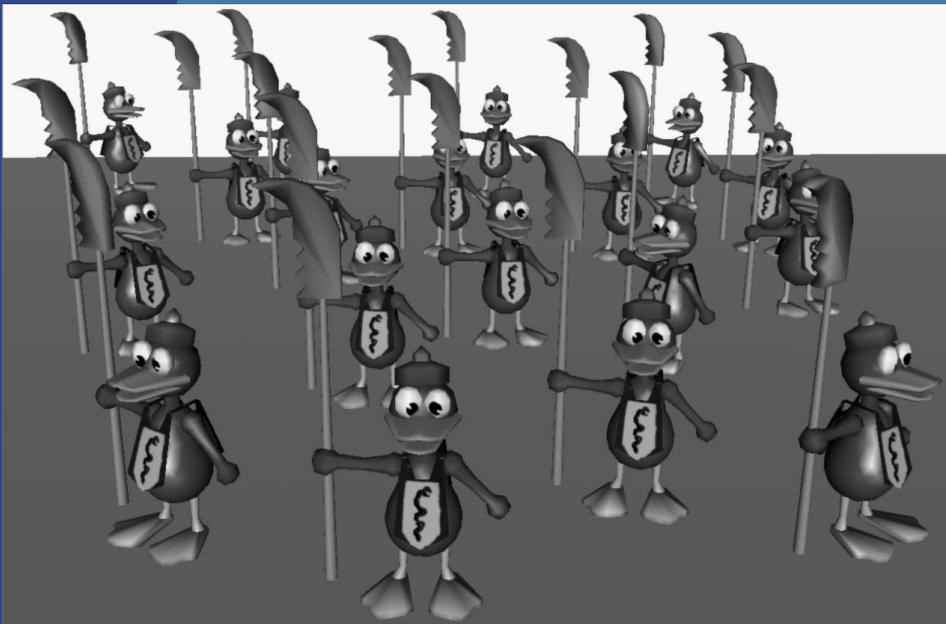
in vec3 outColor, lightDirection, N, pos;
in vec3 viewVec;
uniform vec3 mtrlDiffuse, mtrlSpec, mtrlEmission;
uniform vec3 lightDiffuse, lightSpec, lightEmission;
uniform float shininess;
out vec4 fragColor;

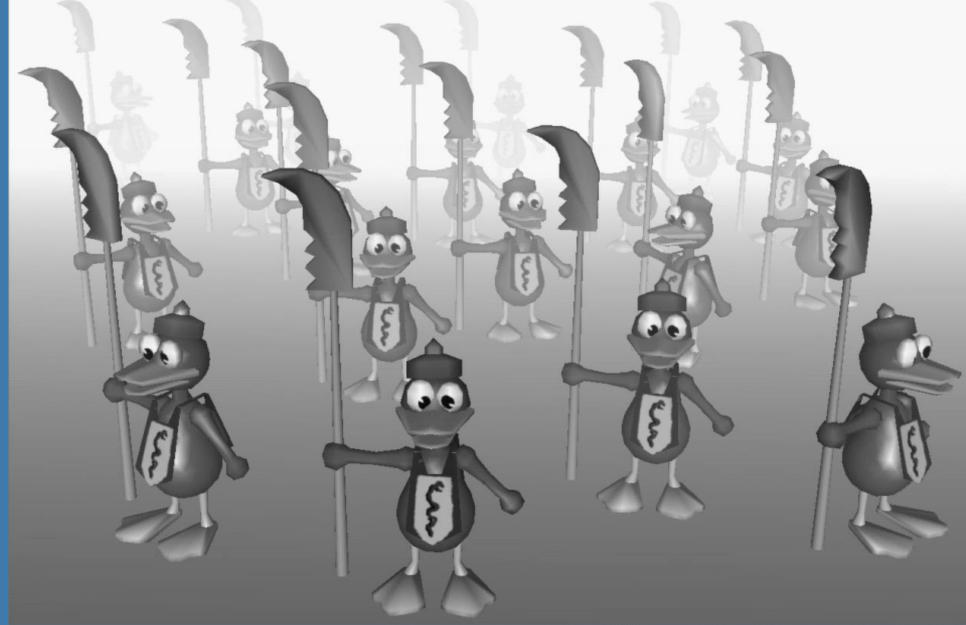
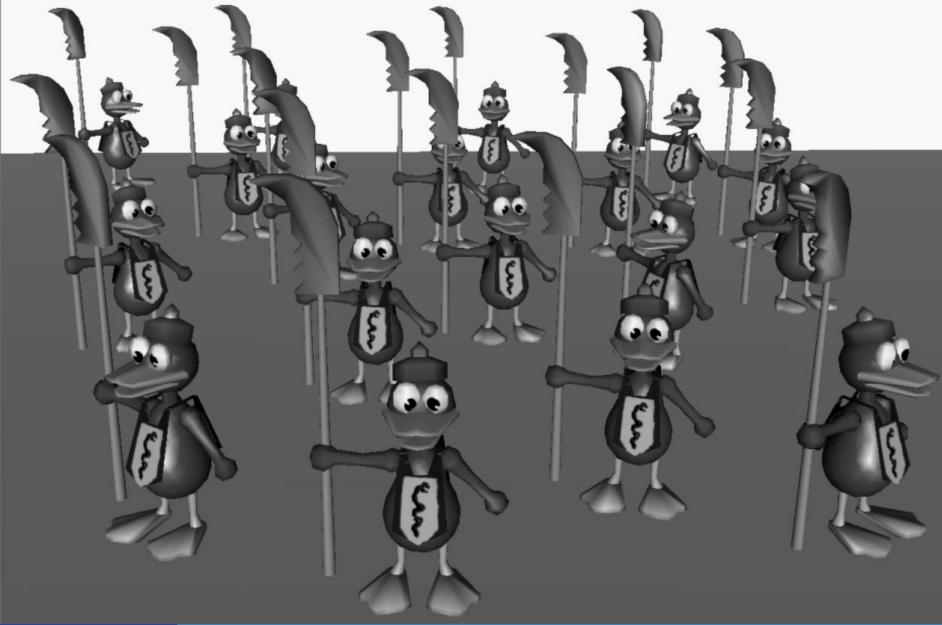
void main()
{
    // diffuse
    float intensity=max(0, dot(normal, lightDirection))
    outColor += lightDiffuse*mtrlDiffuse*intensity;

    // 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





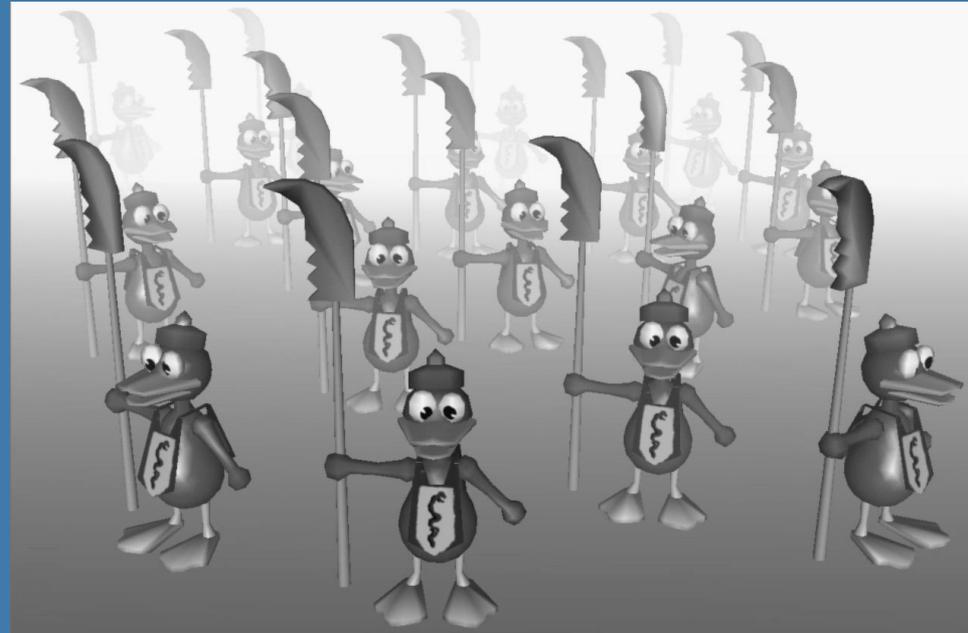
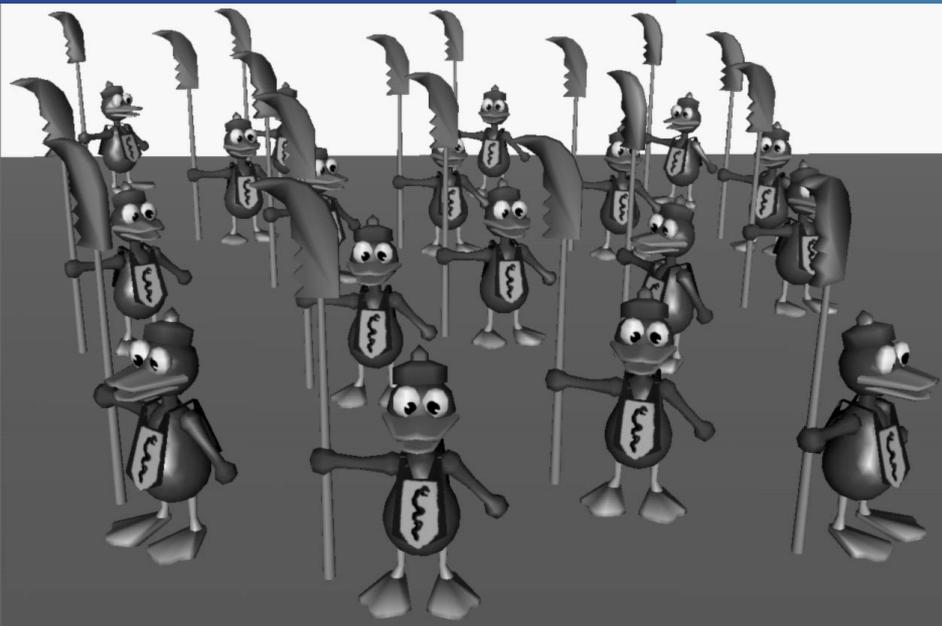
- Color of fog: \mathbf{c}_f color of surface: \mathbf{c}_s

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

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

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

Fog example

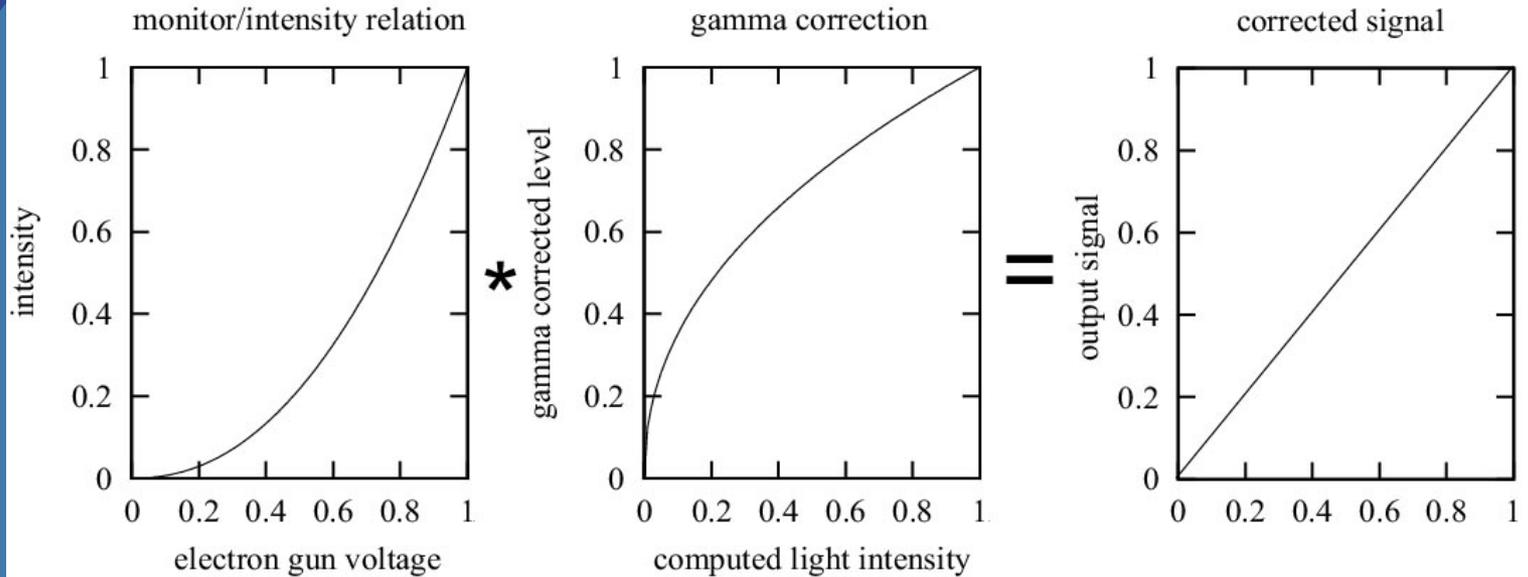


- Often just a matter of
 - Choosing fog color
 - Choosing fog model
 - Old OpenGL – just turn it on. New OpenGL – program it yourself in the fragment shader

Fog in up-direction



Gamma correction



- 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:

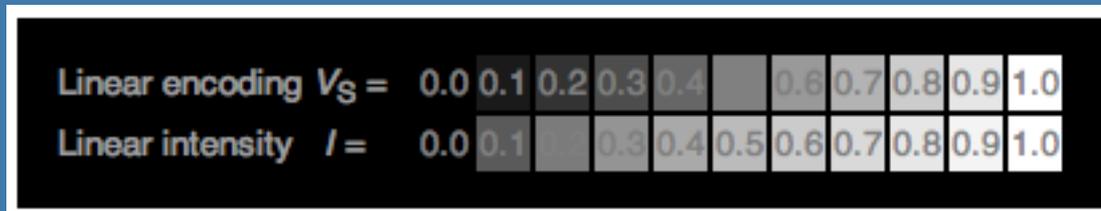
$$C = C_i^{(1/\gamma)}$$

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 somewhat motivated.



Gamma of 2.2

On most displays (those with gamma of about 2.2), one can observe that the linear-intensity scale 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. The linearly-encoded scale, which has a nonlinearly-increasing intensity, 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 on triangle
 - Represents the opacity
 - 1.0 is totally opaque
 - 0.0 is totally transparent
- The over operator:

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

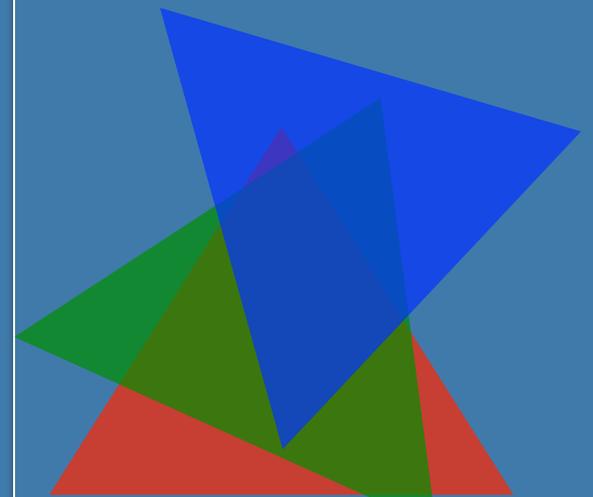
Rendered object

Transparency

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



So the texture rectangle
does not hide objects
behind



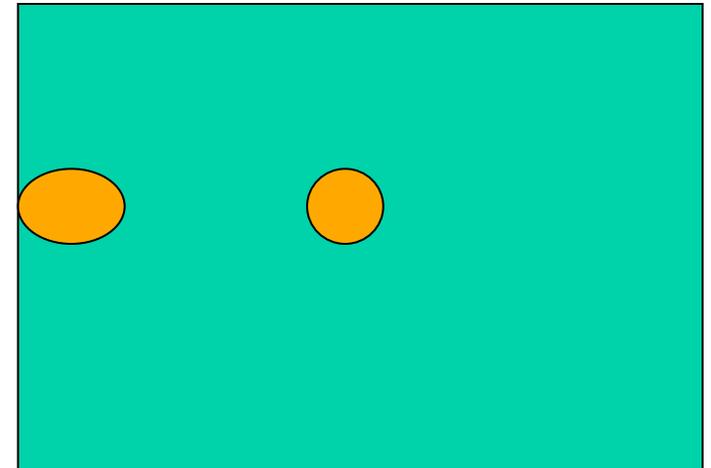
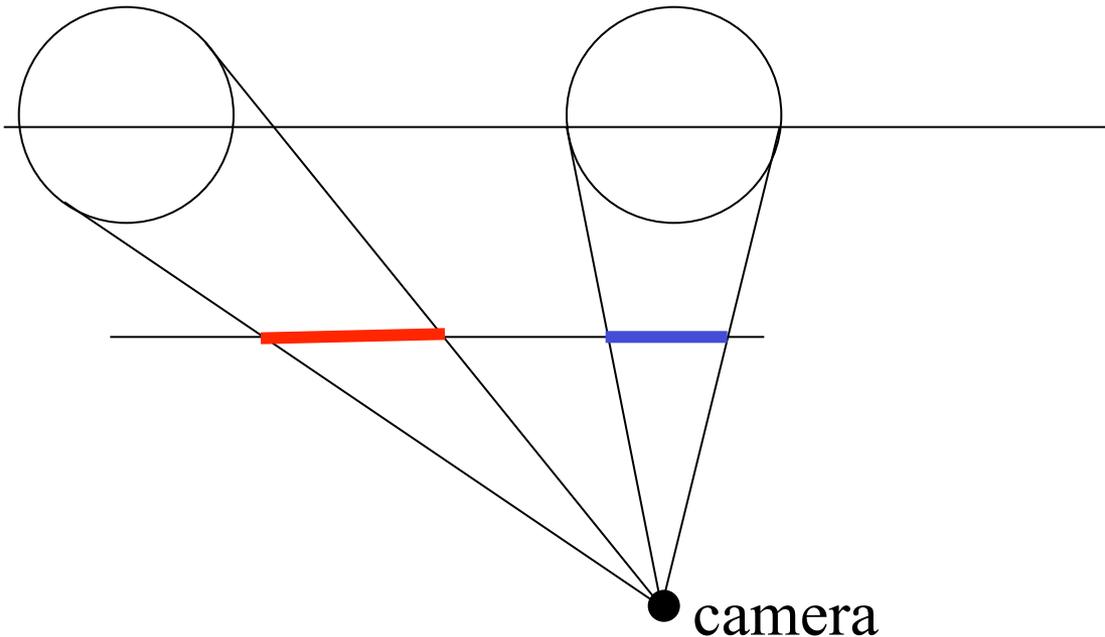
Blending

- 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)`



Perspective distortion

- Spheres often appear as ellipsoids when located in the periphery. Why?



Exaggerated example

If our eye was placed at the camera position, we would not see the distortion. We are often positioned way behind the camera.