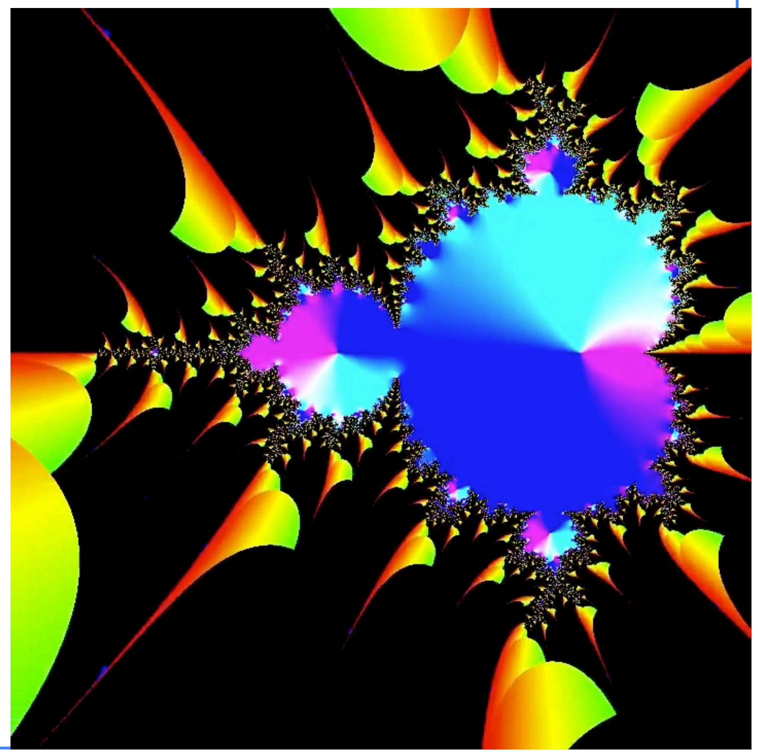
TSBK07/TSBK11

Computer Graphics

Martin Clason, ISY





Lecture 5
3D graphics part 3

Illumination

Illumination applied: Shading

Surface detail: Mappings
Texture mapping
...

Lecture questions

1. What is a Lambertian surface?

2. What vectors are used for calculating diffuse light?

3. …and what vectors are used for calculating specular light?

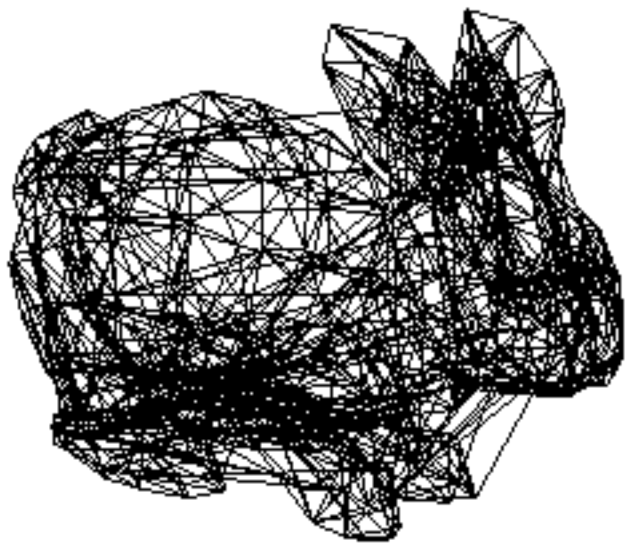
4. Why do we need a max() function for light?

5. What do you need to do with your normal vectors in Phong shading?

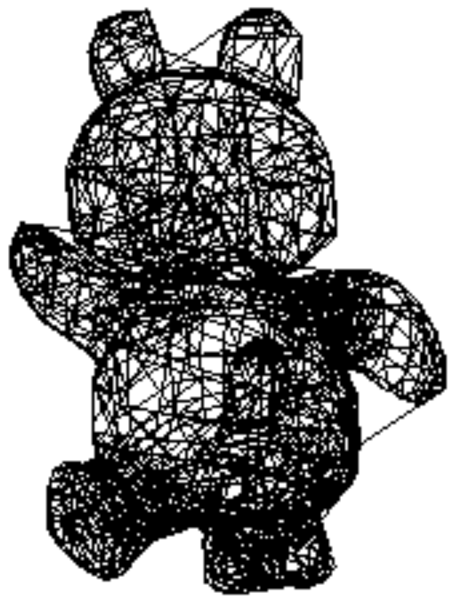
6. How can we create simple shadows?

7. Why do we have texture *units*?

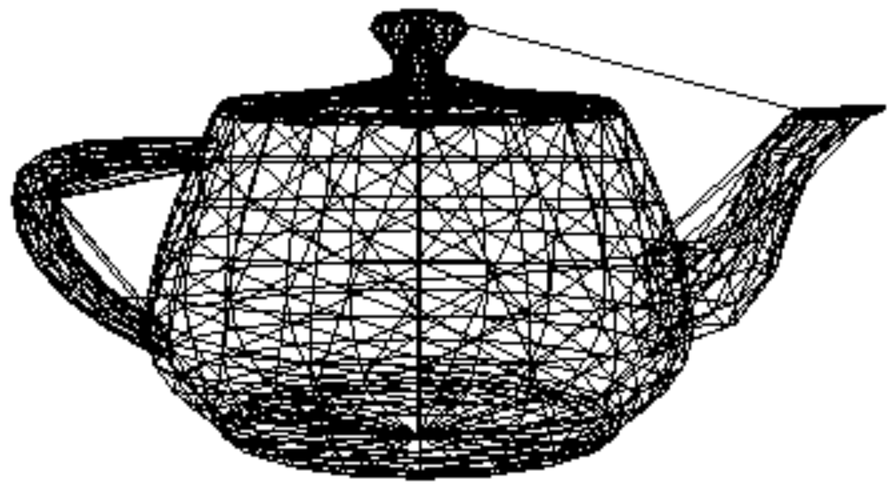
3D models



Bunny



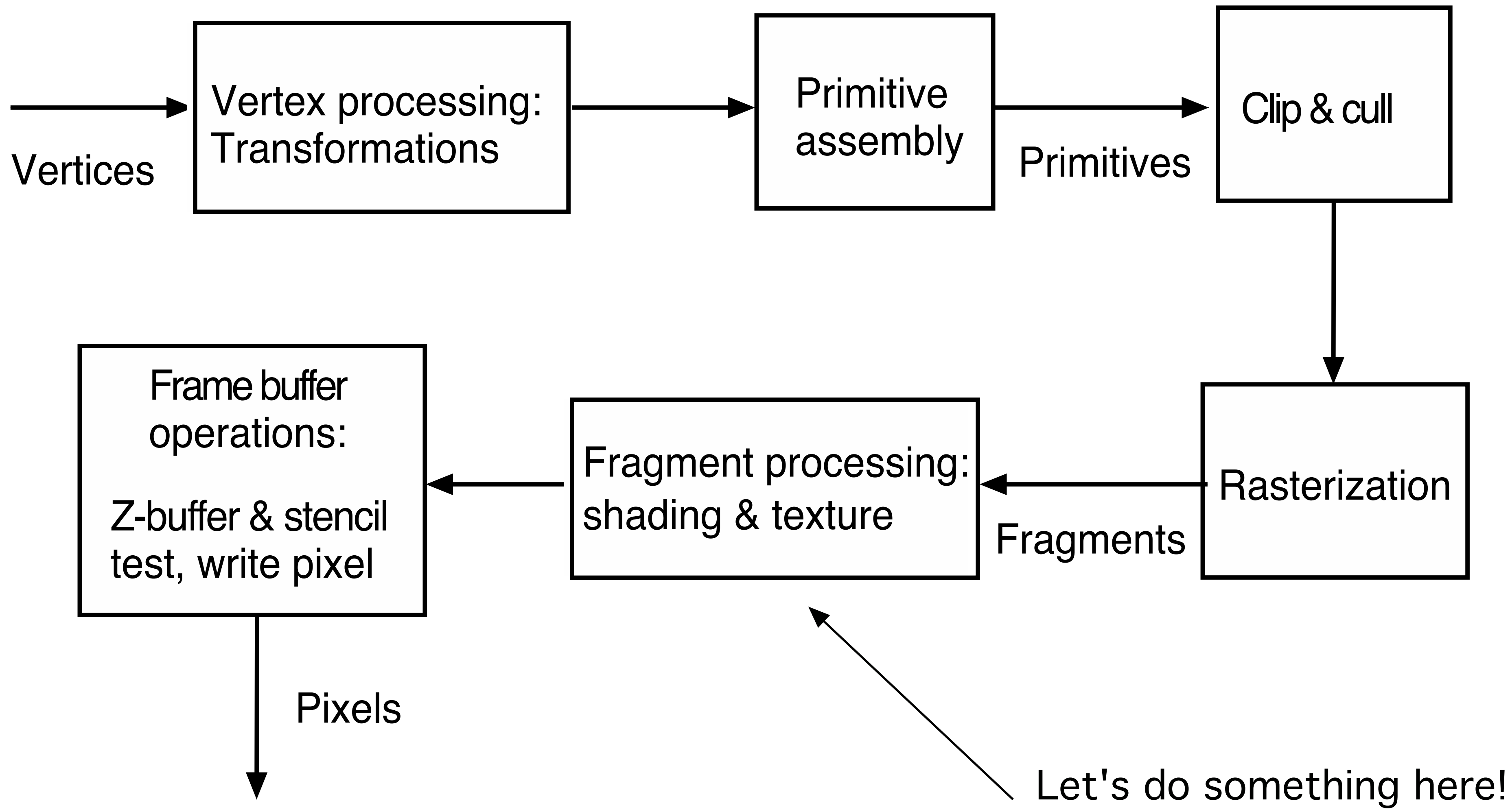
Teddy



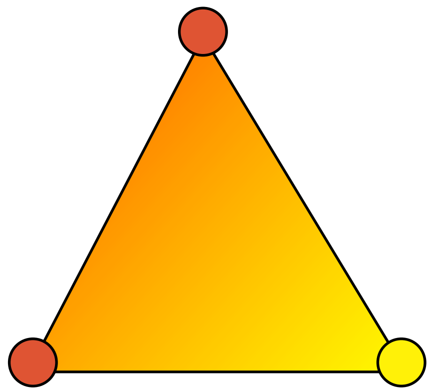
Utah Teapot

Find more on-line. Make your own with e.g. Blender
or TinkerCAD

The OpenGL pipeline



Varying



Values sent from vertex shaders are interpolated and sent to fragments

Key component for all fragment shaders!

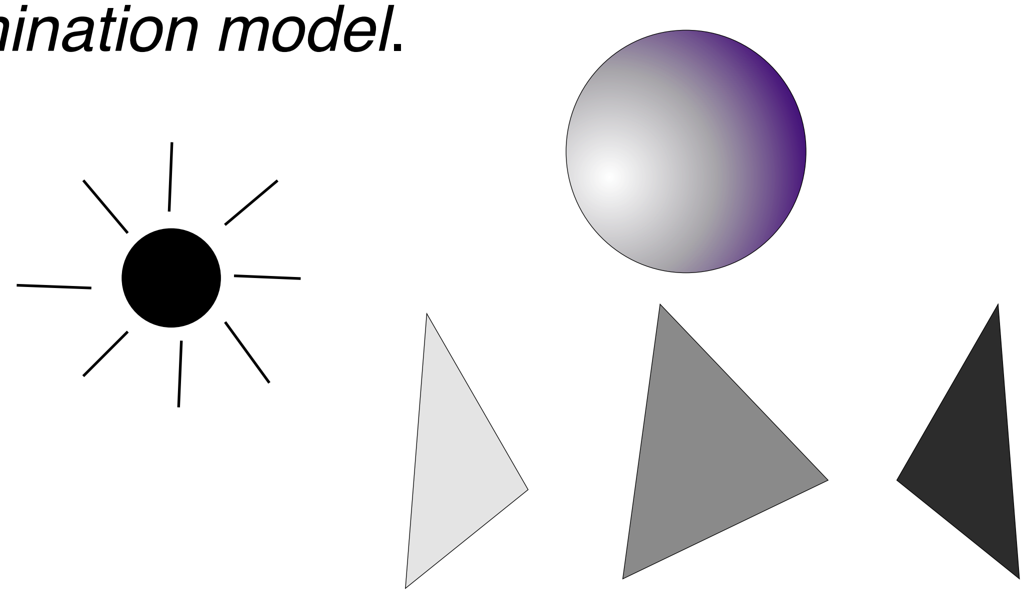
Illumination

We know where to put a polygon. Now, what should we fill it with? What pixel value should we choose?

Several factors to take into account. The most important ones:

• Shading, illumination.
• Texture mapping.

Illumination is determined according to an *illumination model*.



Fake light

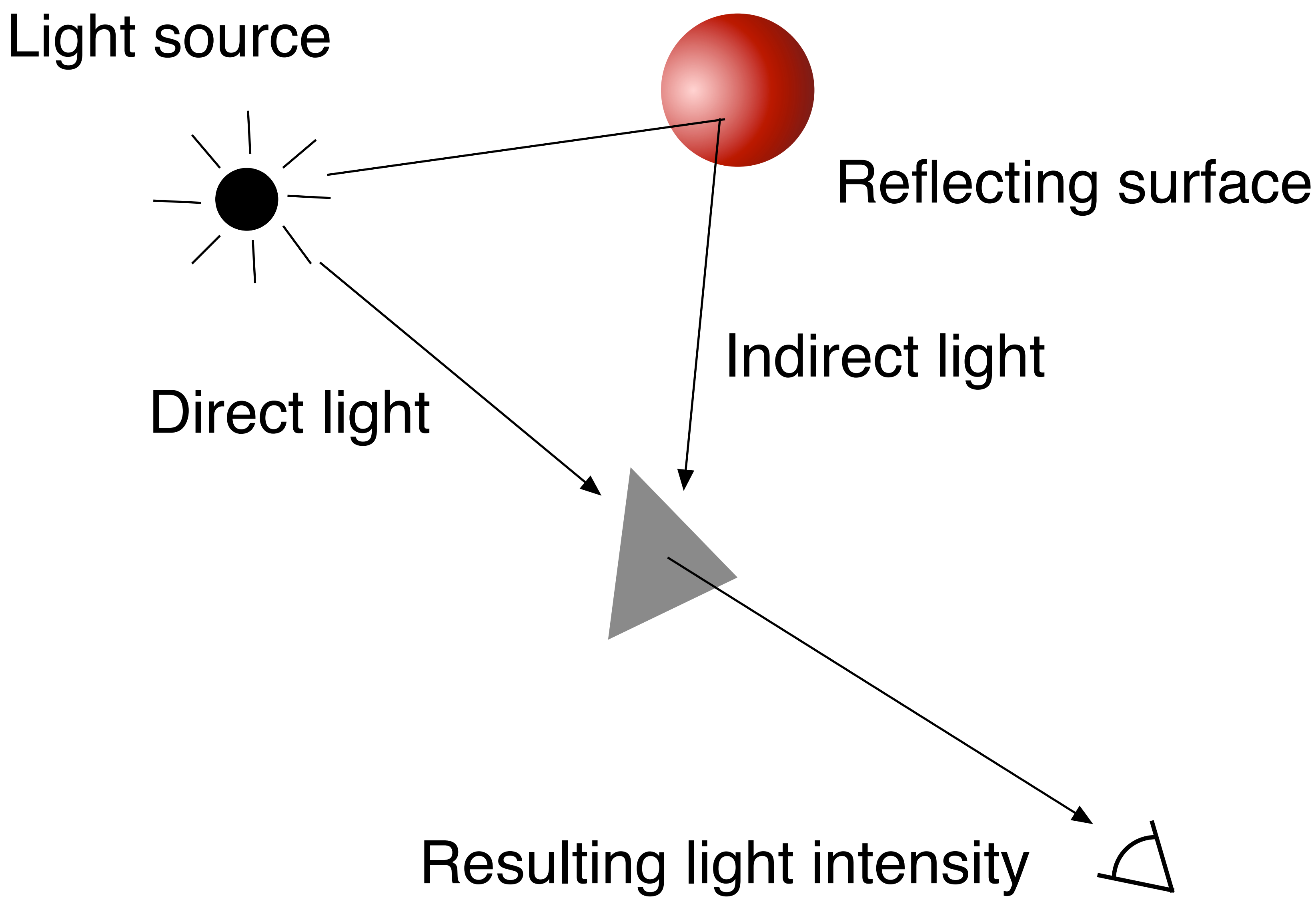
Did you realize that you did light in lab 1?

Mapping normals onto model = fake light.

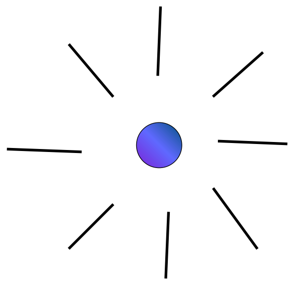
Try this: ex_Normal.zzz

Looks like the model is lit from the camera!

Light sources



Light sources



Small sources can be modelled as point light source

Others can be modelled as distributed light sources

Reflections

Diffuse reflection

Specular reflection

Paper Plastic Metal Mirror

3-component illumination model

A common simple illumination model is built from three components:

Ambient light

Diffuse reflections

Specular reflections

Ambient light

Same everywhere!

$$I_{amb} = k_d * I_a$$

Light emitted from surface

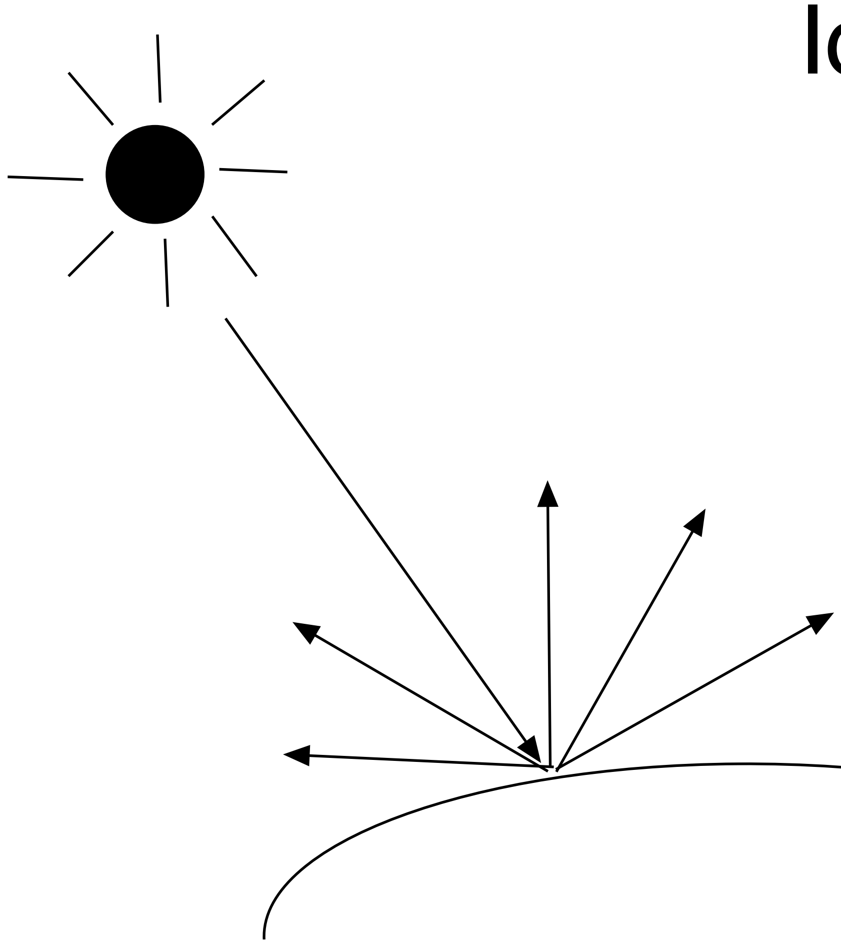
Diffuse reflectivity

Ambient light level of scene

Diffuse reflections

Ideal diffuse reflector =
Lambertian surface

Incoming light produces same
intensity in all directions!



Lambert's cosine law

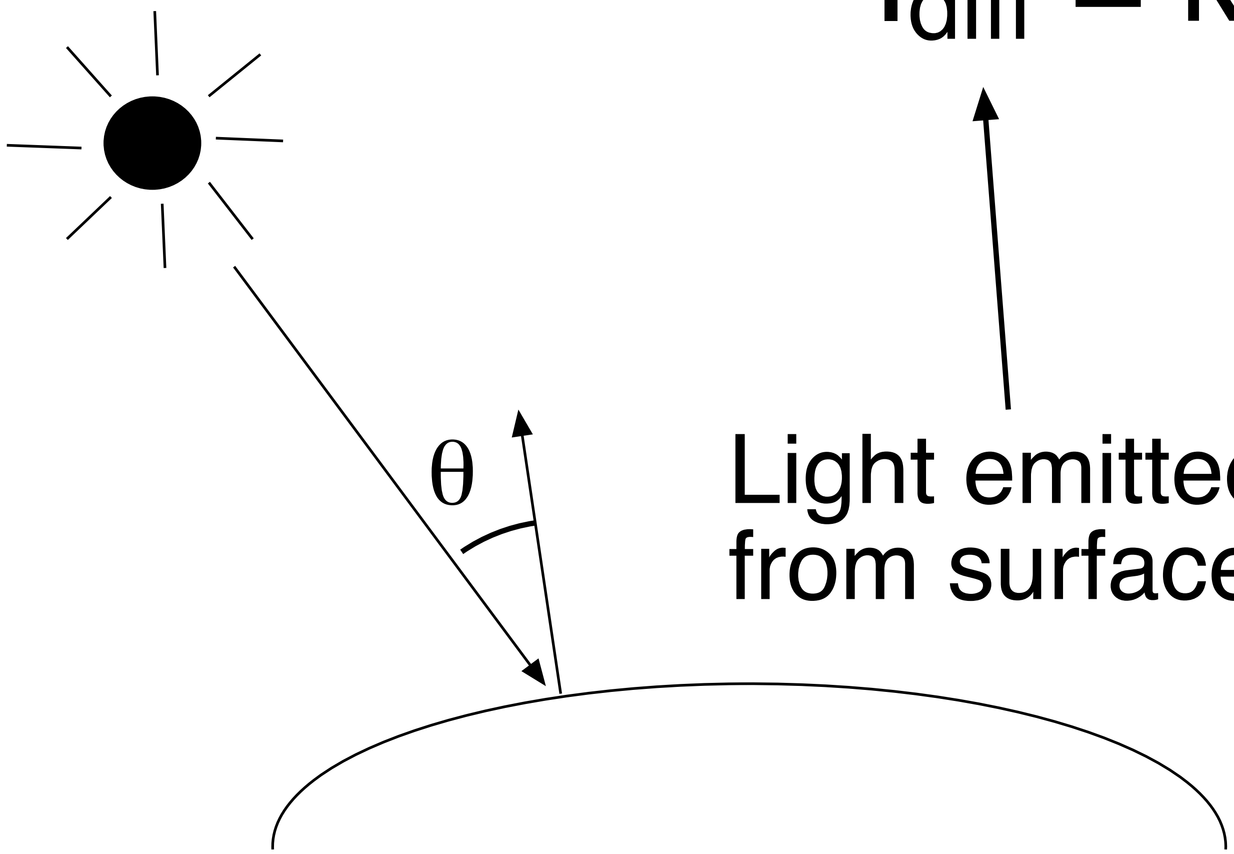
Angle between normal vector of surface and the direction towards the light source

$$I_{diff} = k_d * I_l * \cos\theta$$

Light emitted from surface

Diffuse reflectivity

Light level of light source



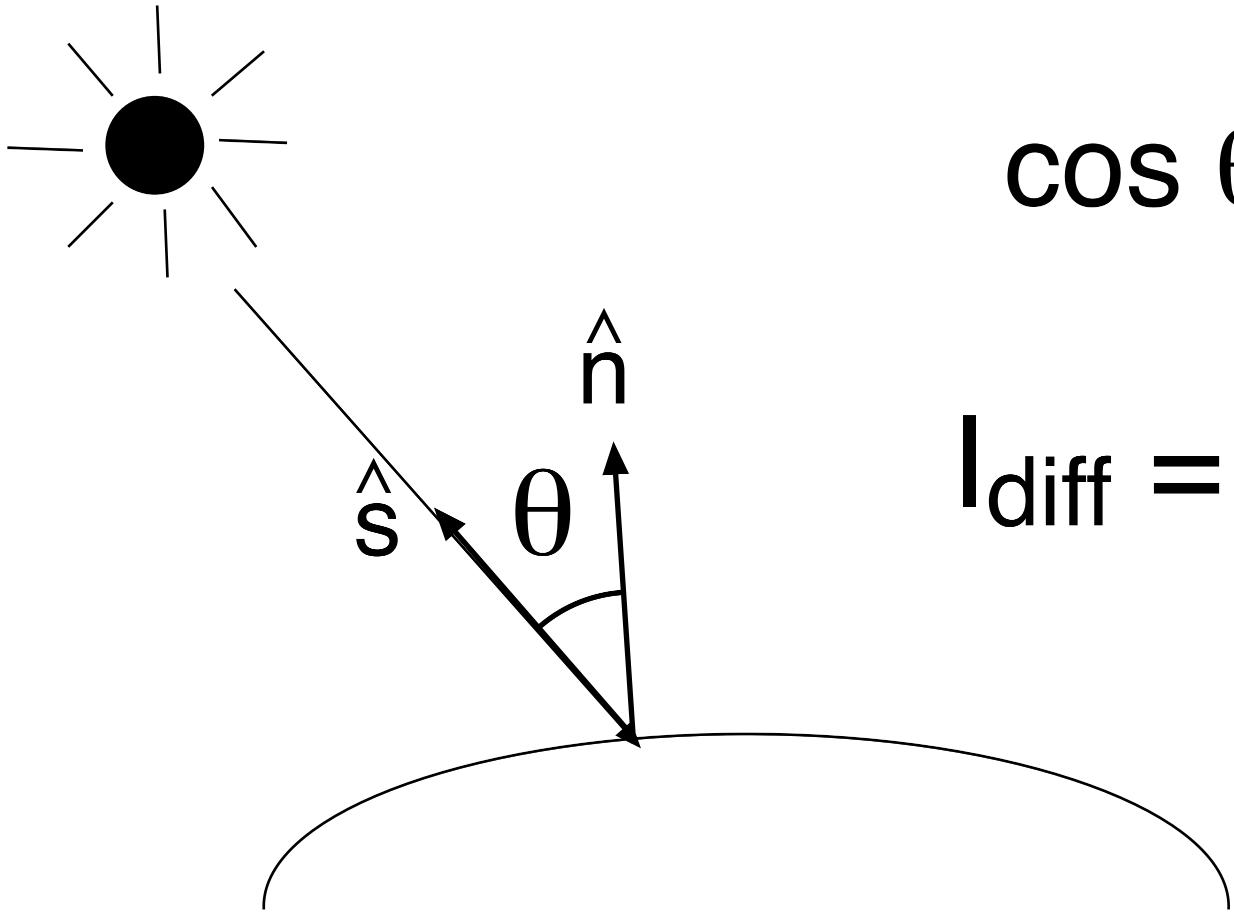
Example: Diffuse sphere

$k_d = 1$

$I_l = 0.9$

$I_a = 0.1$

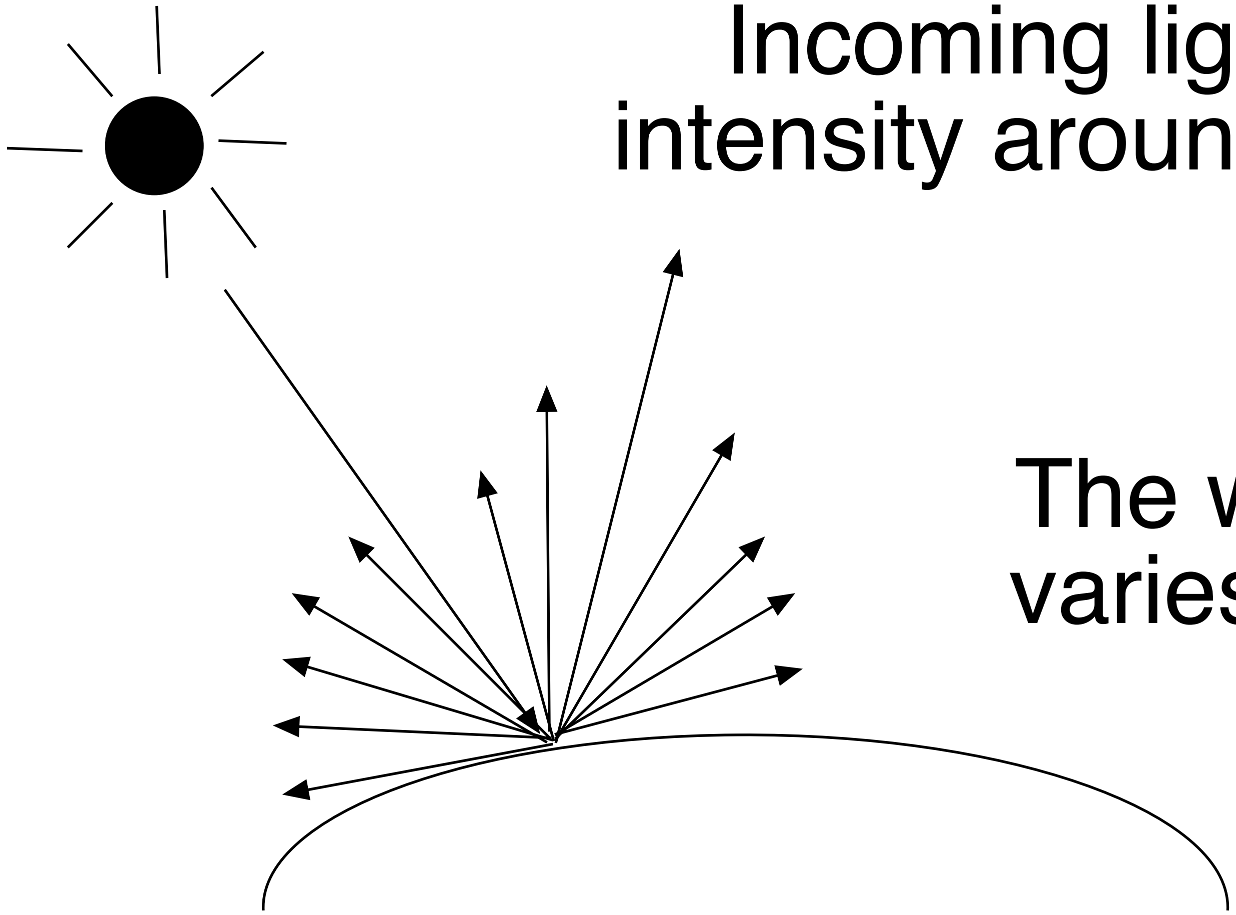
Dot product is nicer than angles!



$$\cos\theta = \hat{s}\cdot\hat{n}$$

$$I_{diff} = k_d * I_l * \hat{s} \bullet \hat{n}$$

Specular reflections



Incoming light produces higher intensity around the mirroring angle!

The width of the highlight varies with surface types!

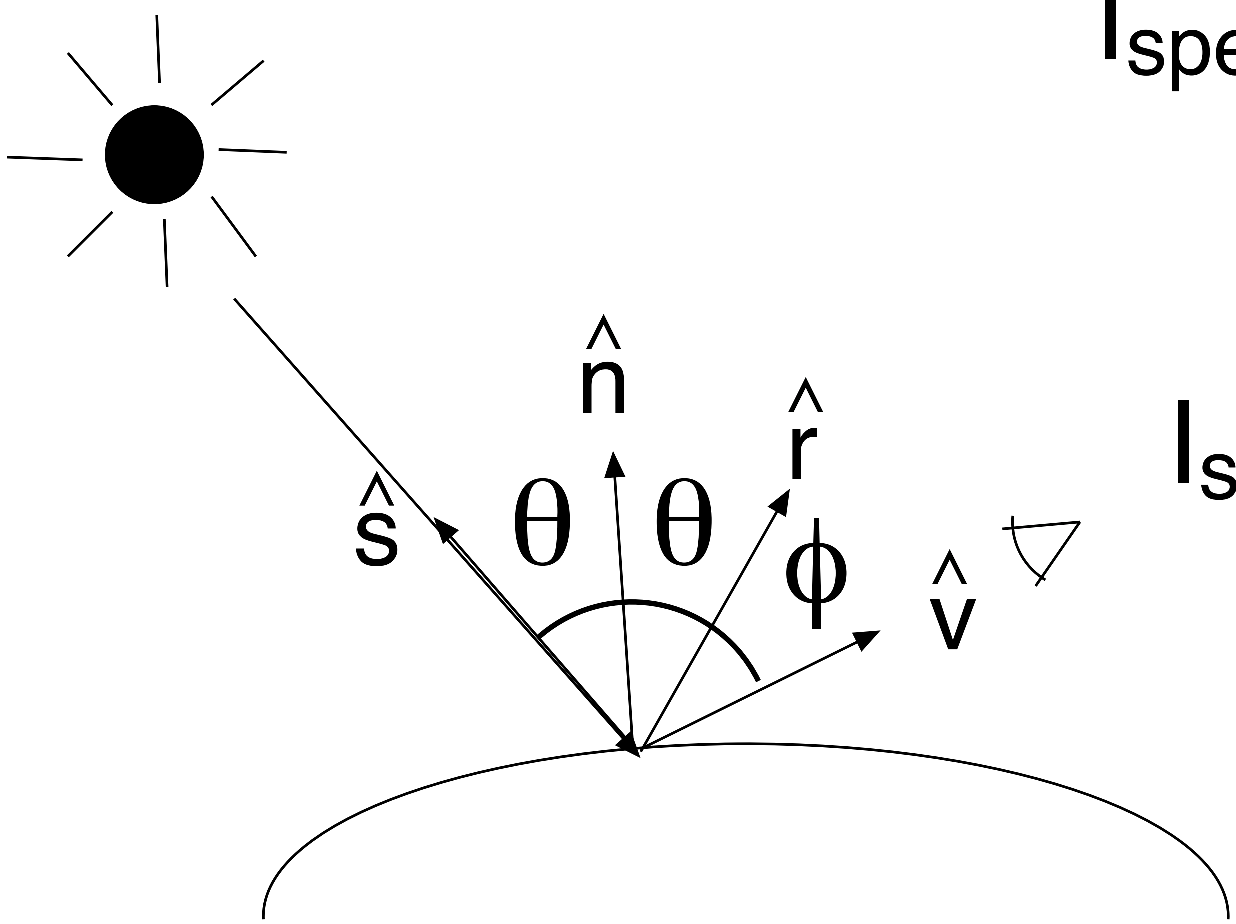
The Phong model

Angle between mirrored light source vector and direction towards camera

$$I_{spec} = W(\theta) * I_l * \cos^{\alpha} \phi$$

or

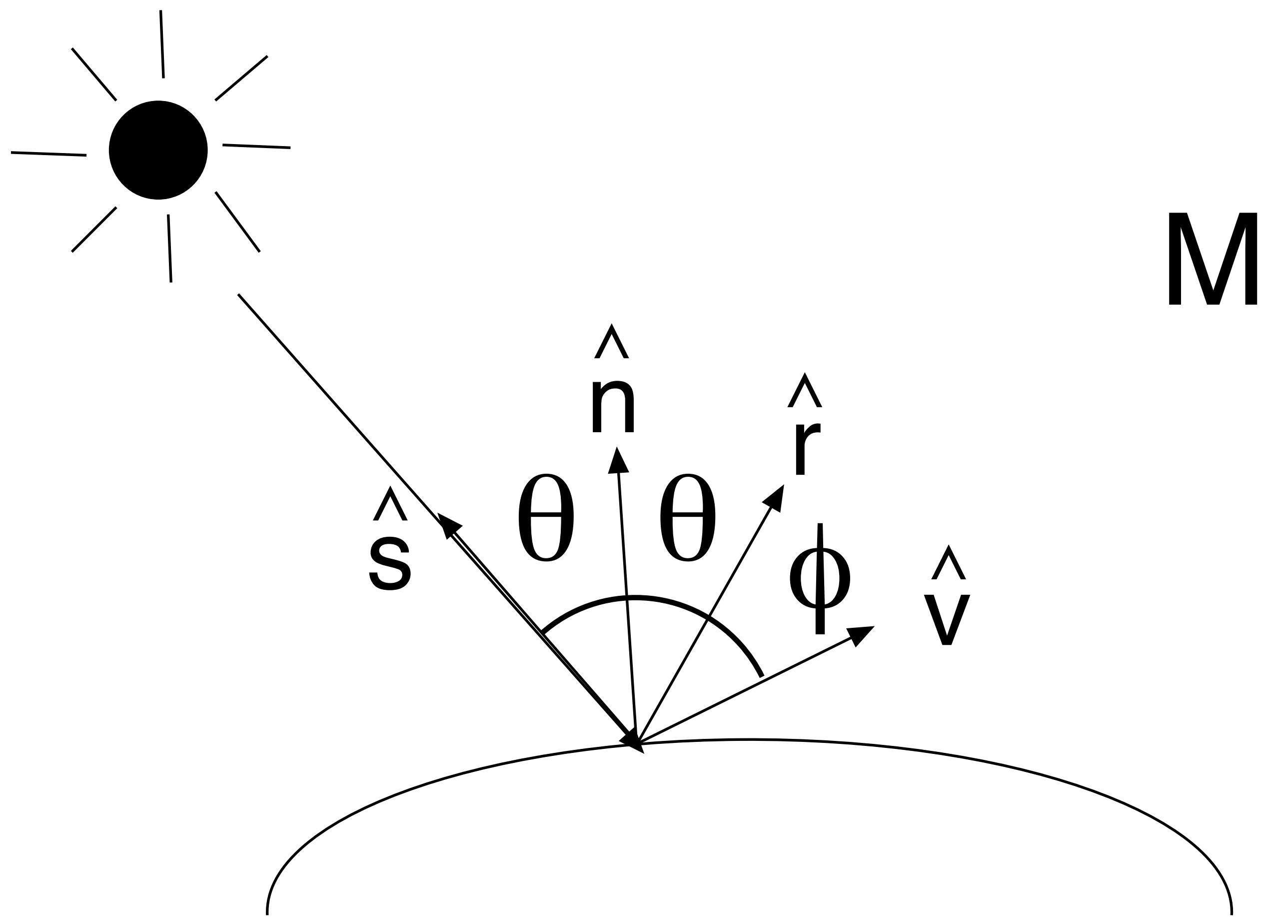
$$I_{spec} = k_s * I_l * \cos^{\alpha} \phi$$



$$\cos \phi = \hat{r} \cdot \hat{v}$$

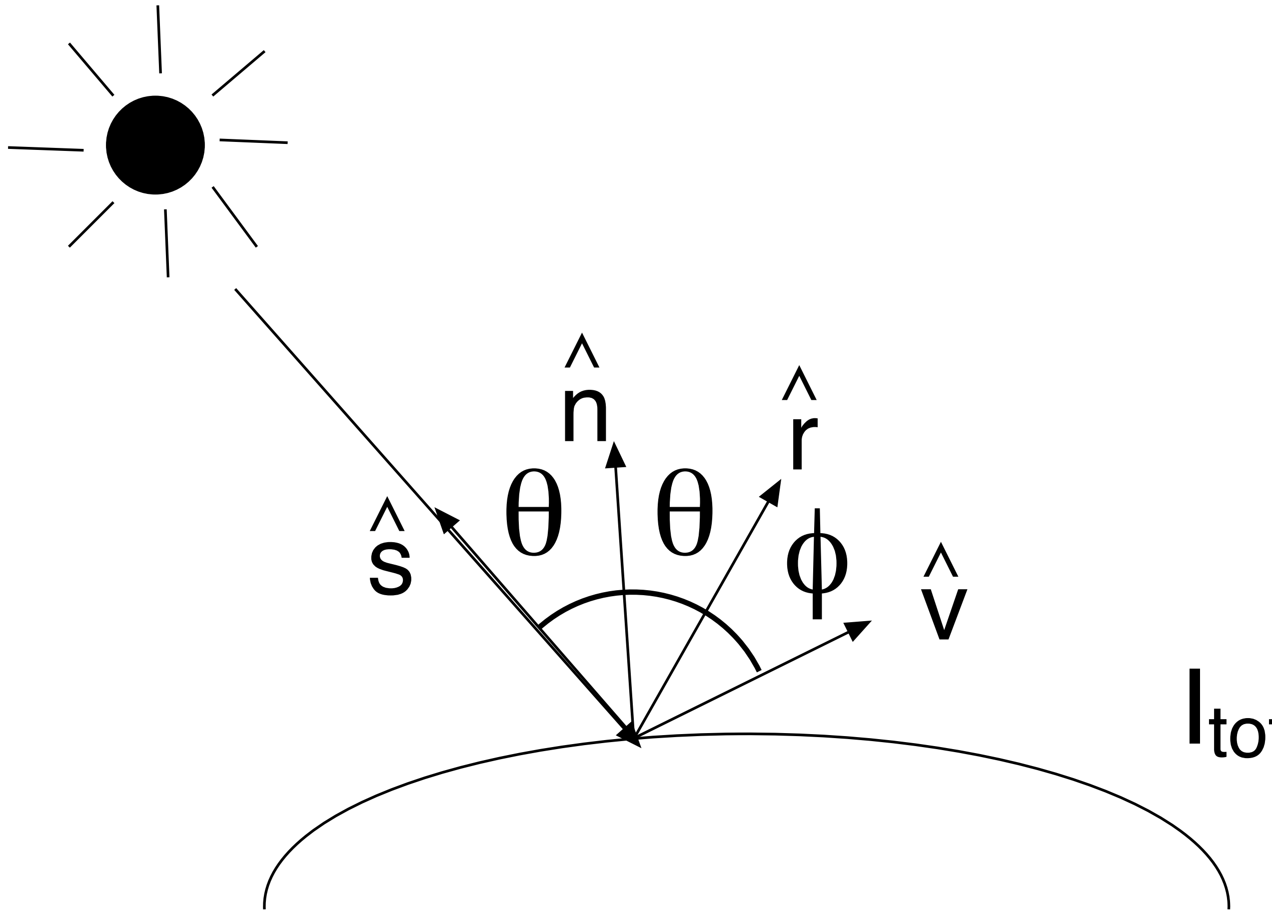
α = "specularity"

Calculation of $\hat{r}$



Mirror $\hat{s}$ by $\hat{n}$!

Total contribution from one light source



$$I_{amb} = k_d * I_a$$

$$I_{diff} = k_d * I_l * \hat{s} \cdot \hat{n}$$

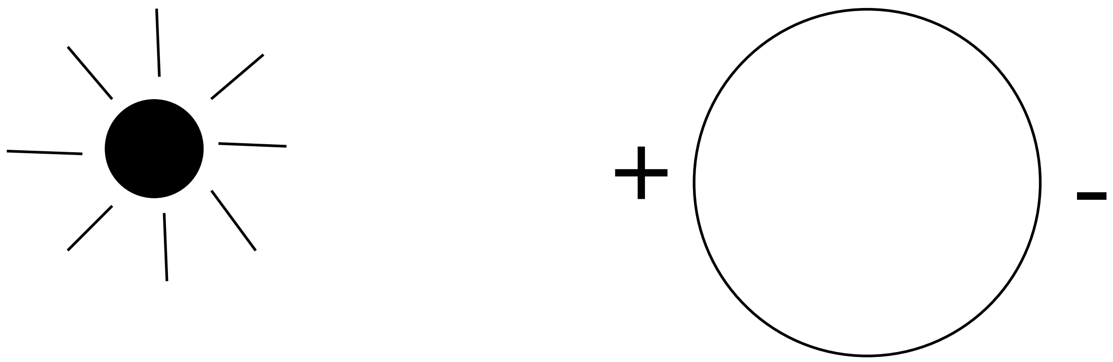
$$I_{spec} = k_s * I_l * (\hat{r} \cdot \hat{v})^{\alpha}$$

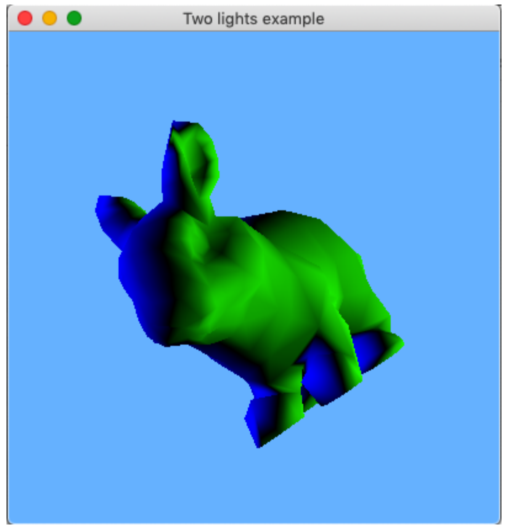
$$I_{total} = I_{amb} + I_{diff} + I_{spec}$$

Clamping shading

Note that the three-component light model will produce negative light!

This will cause problems in scenes with several light sources!





To avoid problems, clamp the resulting value, e.g. max(0, light)

Complete formula

$$I = I_{amb} + I_{diff} + I_{spec} =$$

$$= k_d * I_a + \sum_l (k_d * I_l * \max(0, \hat{s} \cdot \hat{n}) + k_s * I_l * \max(0, \hat{r} \cdot \hat{v})^\alpha)$$

where $\sum$ sums over all light sources

Examples

Diffuse surface, $k_d = 0.9$

Specular surface, $\alpha = 1$

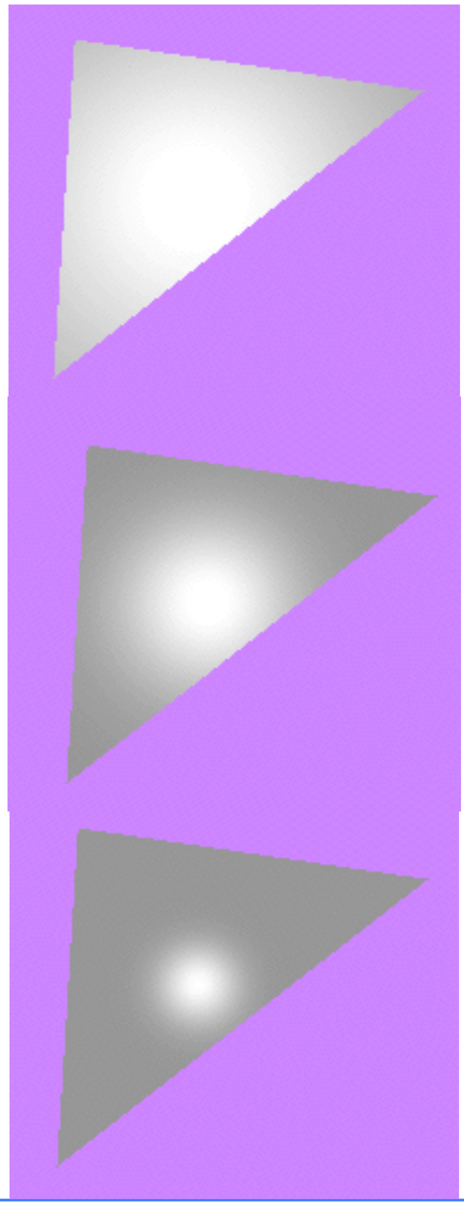
Specular surface, $\alpha = 5$

Specular surface, $\alpha = 25$

Specular surface, $\alpha = 125$

$k_d = 0.45$
$k_s = 0.5$
$I_l = 1.0$
$I_a = 0.1$

Examples



Specular surface, $\alpha = 5$

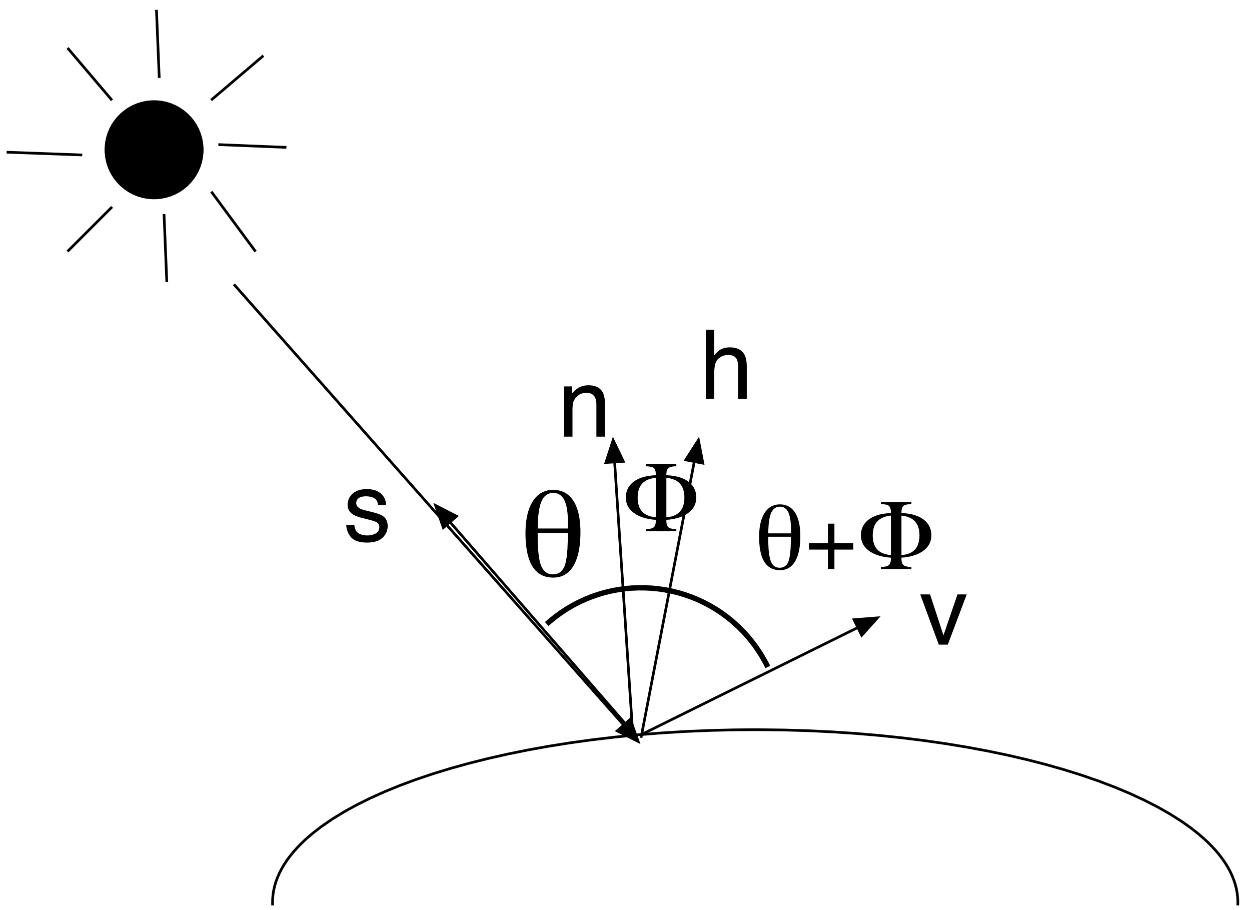
Specular surface, $\alpha = 25$

Specular surface, $\alpha = 125$

$k_d = 0.45$
$k_s = 0.5$
$I_l = 1.0$
$I_a = 0.1$

Alternative formulation

Blinn-Phong



Halfway vector

$$\hat{h} = (s+v) / |s + v|$$

$$I_{spec} = k_s * I_l * \cos^{\alpha} \Phi =$$

$$= I_{spec} = k_s * I_l * (\hat{n} \cdot \hat{h})^{\alpha}$$

Transform

Transform to the proper coordinate system

Surface = model coords

Light = World coords

Camera = View coords

Transform all to e.g. view!

Light source types

Positional - located in a specified place

E.g. lamps

Directional - infinitie distance

E.g. the sun

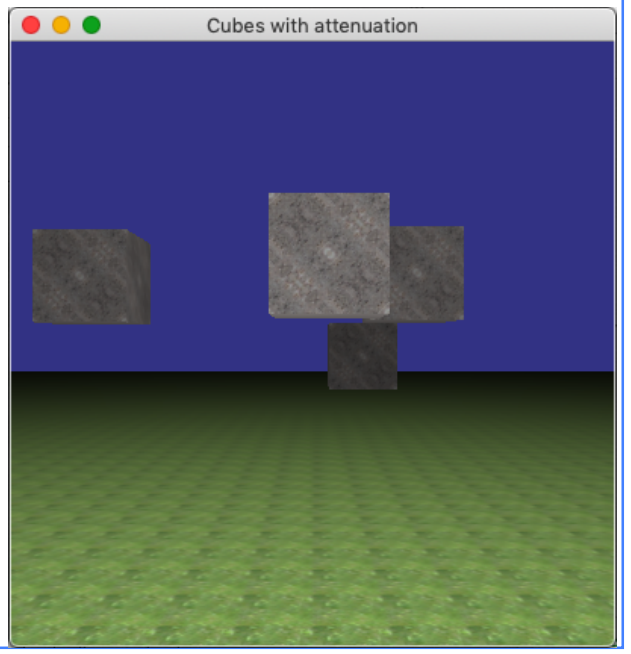
Light attenuation

Light levels depend on distance.

Point light = light level proportional to $1/d^2$

Linear part often a more realistic approximation:

$$f(d) = \frac{1}{a + bd + cd^2}$$



Advanced illumination models

Make k_s a function of the viewing angle - better modelling of glass and paper

BRDF - highly general multi-dimensional function

Global illumination models

Radiosity: Models light exchange recursively

Ray-tracing, trace viewing rays.

Path tracing, extended ray-tracing.

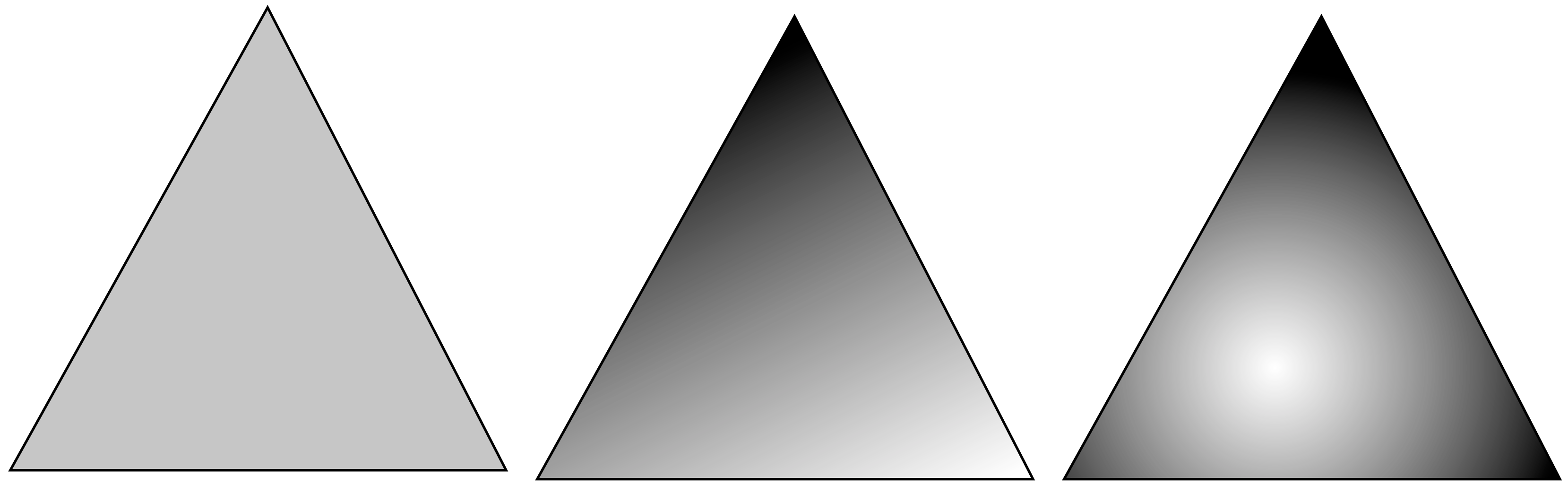
Photon mapping, “backwards ray-tracing”, trace light rays.

Polygon shading

Using the illumination models in high-speed polygon rendering

Three ways to render a shaded polygon:

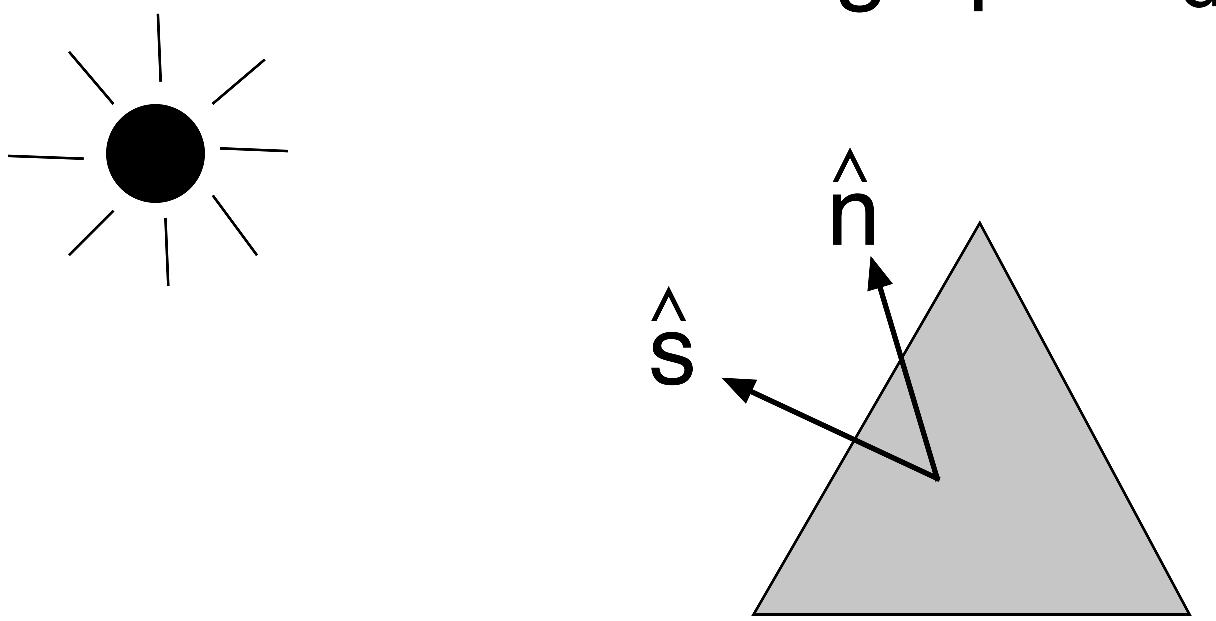
Flat shading
Gouraud shading
Phong shading



Flat shading

Intensity calculated once and for all for the whole polygon

E.g. $I_p = k_d \cdot \hat{n} \cdot \hat{s}$



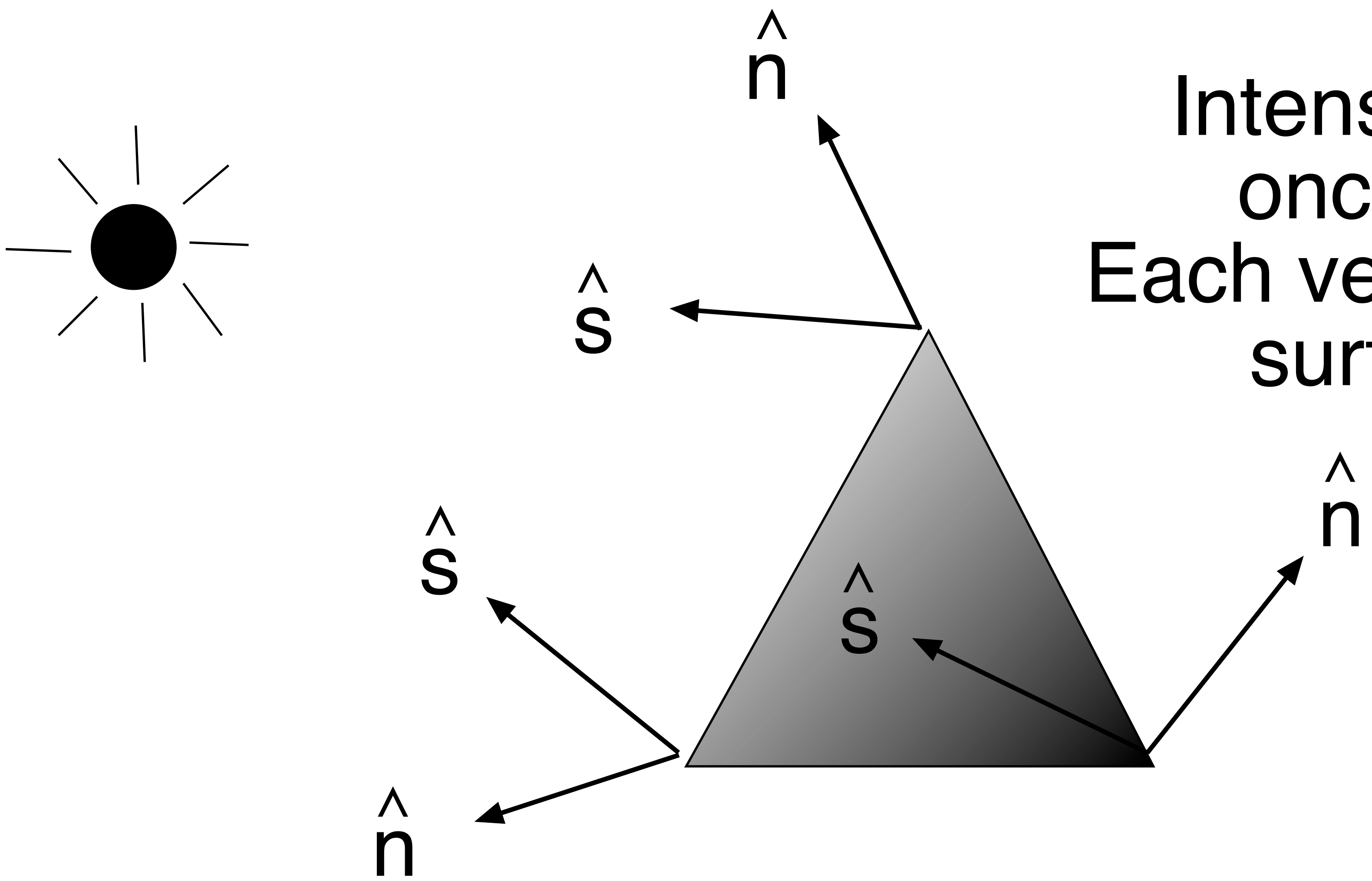
Flat shading is “correct” when:

1) The surfaces should be flat, not approximating a curved surface
2) Distance to light source high => n·s constant
3) Distance to camera high => v·r constant

and in particular

4) When the problem is not lighting, but something else! (Rendering surface identifications)

Gouraud shading



Intensity calculated once per vertex
Each vertex has its own surface normal

Interpolate intensities!

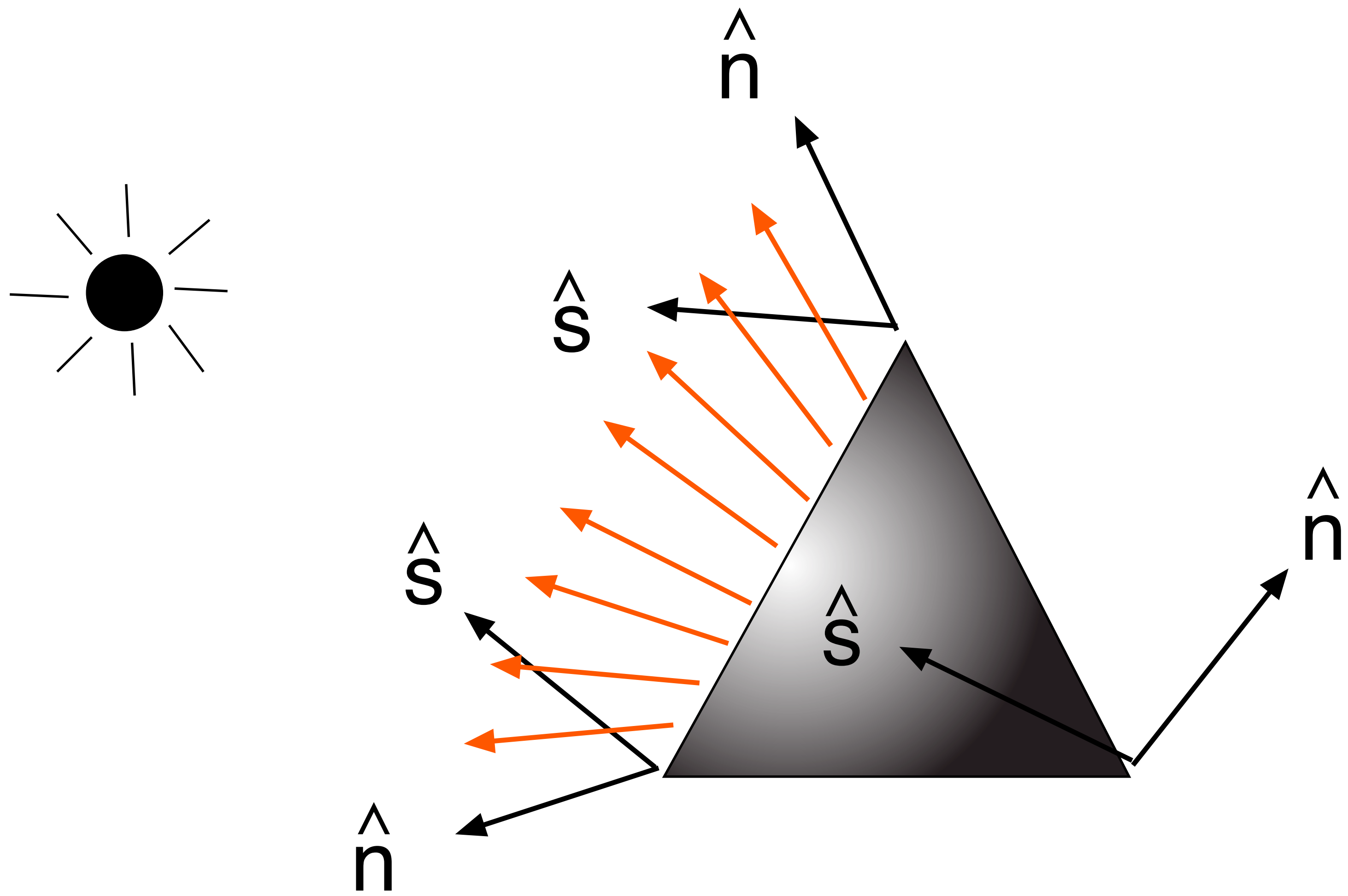
Gouraud shading

can simulate curved surfaces to some extent, but many polygons may be needed, and edges remain visible

OK for diffuse surfaces

Calculations in vertex shader - extremely fast!

Phong shading



Each vertex has its own surface normal

Normal vectors are interpolated

Calculate each pixel individually!

Normalize the normals!

Phong shading

can simulate curved surfaces very well, even with low polygon counts

Good for specular surfaces!

Calculate the light in the fragment shader

Computationally heavier

Phong shading
≠
The Phong model

Phong Shading doesn't necessarily use specular reflections.

Phong Shading = normal-vector interpolation shading

Example:
Gouraud shader

- Transform normal vectors
- Calculate shading value per vertex, (here using diffuse only), by dot product with light direction
- Interpolate light level between vertices

Gouraud shader - vertex shader

```
#version 150
in  vec3 inPosition;
in  vec3 inNormal;
out vec3 exColor;

void main(void)
{
   const vec3 light = vec3(0.58, 0.58, 0.58);
   float shade;
   shade = dot(normalize(inNormal), light);
   shade = max(shade, 0);
   exColor = vec3(shade);
   gl_Position = vec4(inPosition, 1.0);
}
```

Gouraud shader - fragment shader

```
#version 150
in  vec3 exColor;
out vec4 outColor;
void main(void)
{
   outColor = vec4(exColor,1.0);
}
```

Gouraud shader

Note:

The variable “exColor” is interpolated between vertices!

dot() och normalize() do what you expect.

inNormal is the normal vector in model coordinates
(Should be transformed in a real program!)

The constant vector ”light” is here hard coded

Typical Gouraud shaded bunny

Version 2: Add specular lighting to vertex shader

```
    // Specular
    vec3 reflectedLightDirection = reflect(-light, norm);
    vec3 eyeDirection = vec3(normalize(-inPosition));

    float specularStrength = 0.0;
    specularStrength = dot(reflectedLightDirection, eyeDirection);
    float exponent = 8.0;
    specularStrength = max(specularStrength, 0.01);
    specularStrength = pow(specularStrength, exponent);

    shade = (0.3*diffuseshade + 0.9*specularStrength);
}
```

(Again some transformations skipped.)

Specular Gouraud shaded bunny

A bit polygonal...

Example:
Phong shader

Better shading!

- Interpolate normal vectors between vertices
- Calculate shading value per fragment

Practically the same operations, but the light calculation are done in the fragment shader

Phong shader

Vertex shader

```
#version 150
in  vec3 inPosition;
in  vec3 inNormal;
out vec3 exNormal;
out vec3 surf;
void main(void)
{
   exNormal = inNormal;
   surf = inPosition; // For specular
   gl_Position = vec4(inPosition, 1.0);
}
```

Phong shader

Fragment shader

```
#version 150
out vec4 outColor;
in vec3 exNormal;
in vec3 surf;
void main(void)
{
   const vec3 light = vec3(0.58, 0.58, 0.58);
   float shade;
   shade = dot(normalize(exNormal), light);
   shade = clamp(shade, 0, 1);
   outColor = vec4(shade, shade, shade, 1.0);
}
```

..and add specular part

```
// Specular
	vec3 reflectedLightDirection = reflect(-lightDirection, n);
	vec3 eyeDirection = normalize(-surf);

	float specularStrength = 0.0;
	if (dot(lightDirection, n) > 0.0)
	{
		specularStrength = dot(reflectedLightDirection, eyeDirection);
		float exponent = 200.0;
		specularStrength = max(specularStrength, 0.01);
		specularStrength = pow(specularStrength, exponent);
	}

	outColor = vec4(diffuseStrength*0.5 + specularStrength*0.5);
}
```

Specular Phong shaded bunny

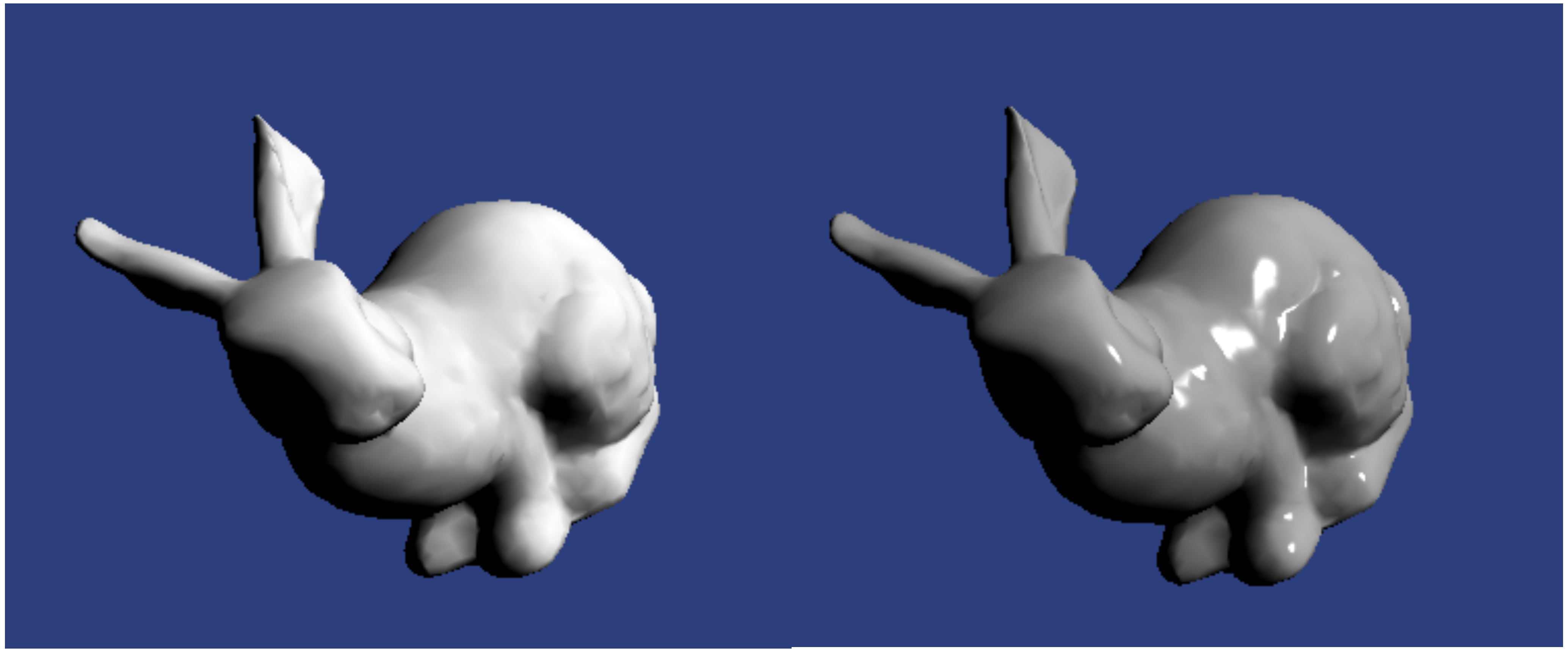
Now we're talking!

Gouraud not useful with specular light

Gouraud

Phong

Same for Stanford bunny



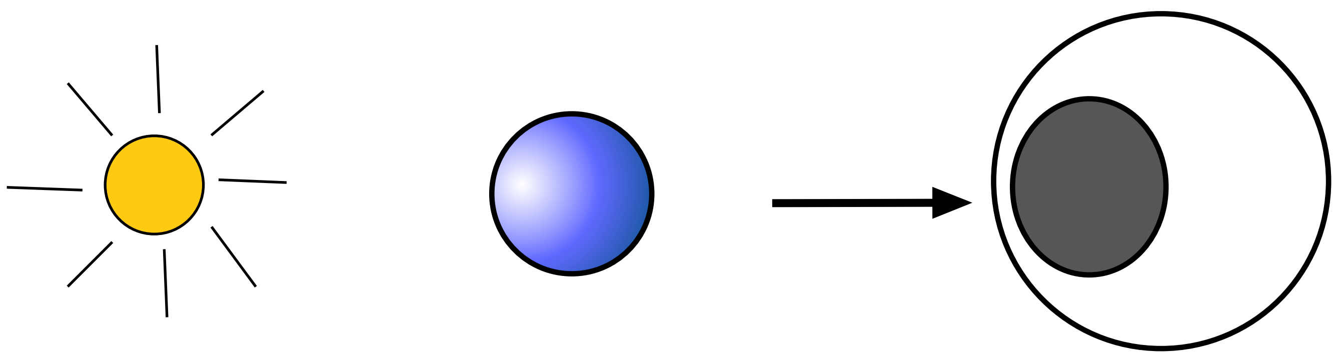
Gouraud Phong

Image removed
(Blues Brothers)

Shading and shadows

Shading will give you light variations due to shape, and the back side of objects will be shadowed.

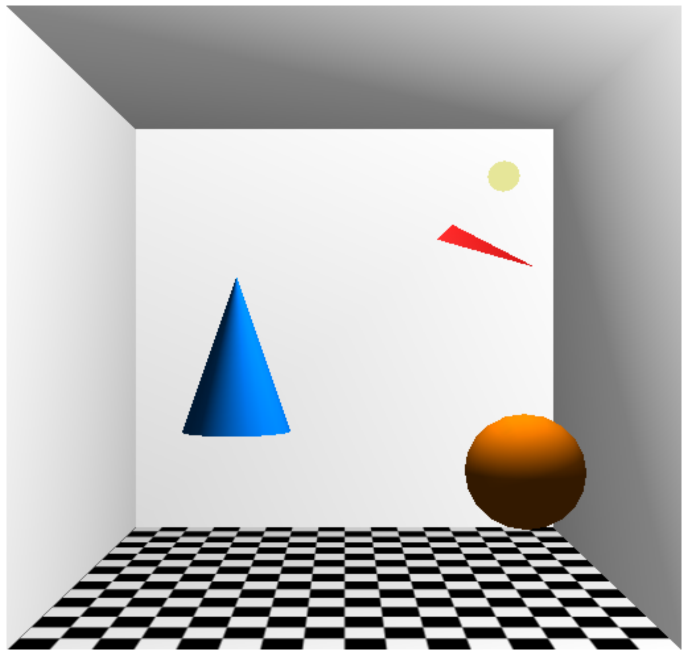
BUT, this will not produce shadows on one object cast by another object!

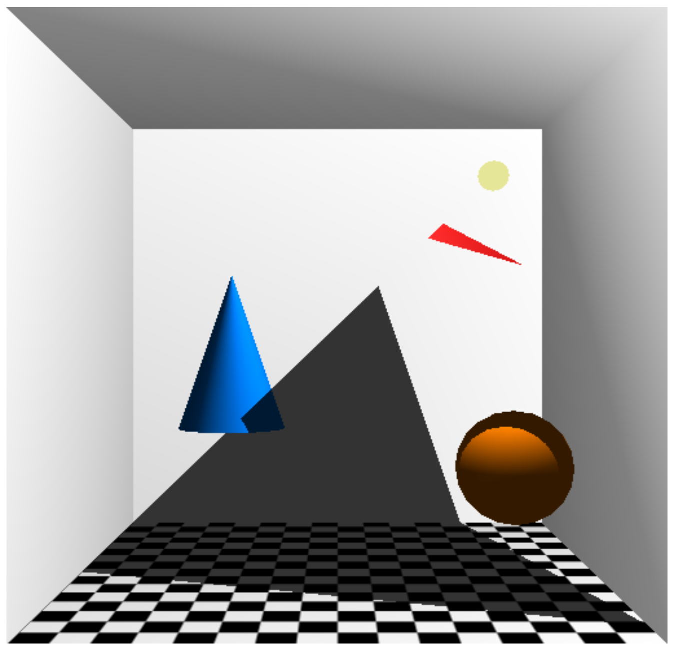


Shading and shadows

Local shading is easy (with simple light models)

Shadows are hard

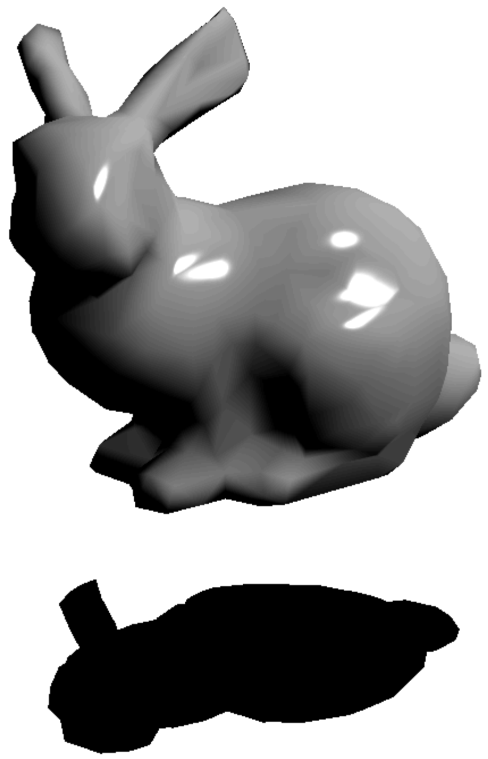




Simple shadows

Easiest: Linear planar shadow

Flatten object, paint black (optionally transparently),
rotate and translate to appropriate position



Example:

```
Translate to surface
Scale by (1, 0, 1)
Rotate as desired
Draw model in black
```

Position after flattening

Flatten

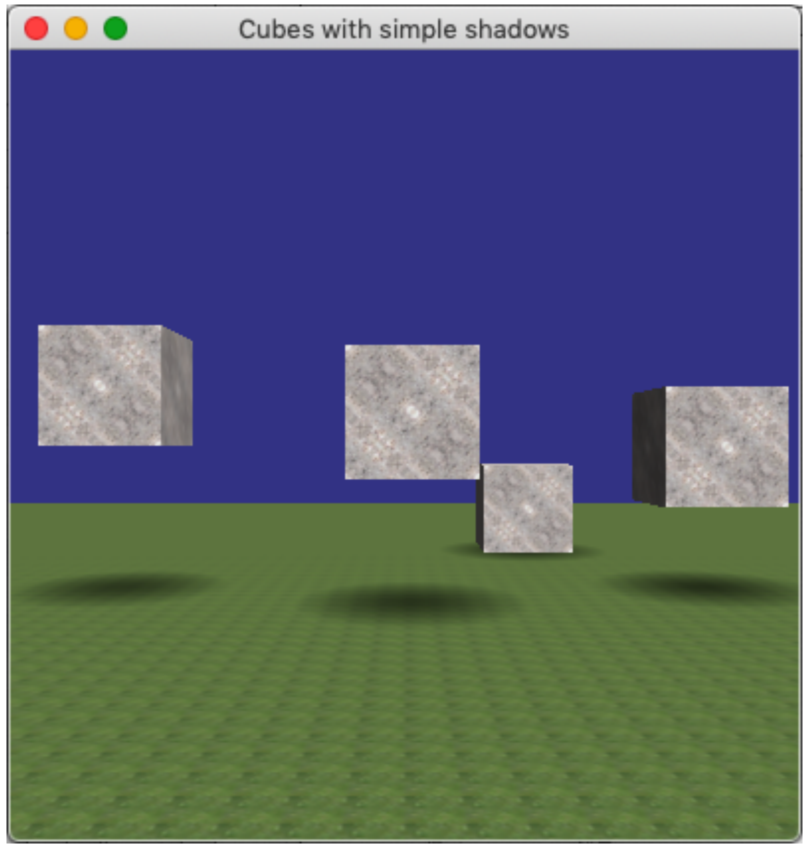
No light or texture

Simple shadows #2

Diffuse spot under the shape

Works with any surface

Requires the shader to know the positions of shadowing objects!



Advanced shadows

Planar projective shadows

Shadow volumes

Shadow mapping

Soft shadows

-> Advanced course (TSBK03)

Also: Natural part of ray-tracing and radiosity

Surface detail

Shading:
takes away the surface detail of the polygons

Texture mapping and other mappings:
add the surface detail that we really want

Surface mapping techniques

Texture mapping
Billboards
Bump mapping
Light mapping
Environment mapping

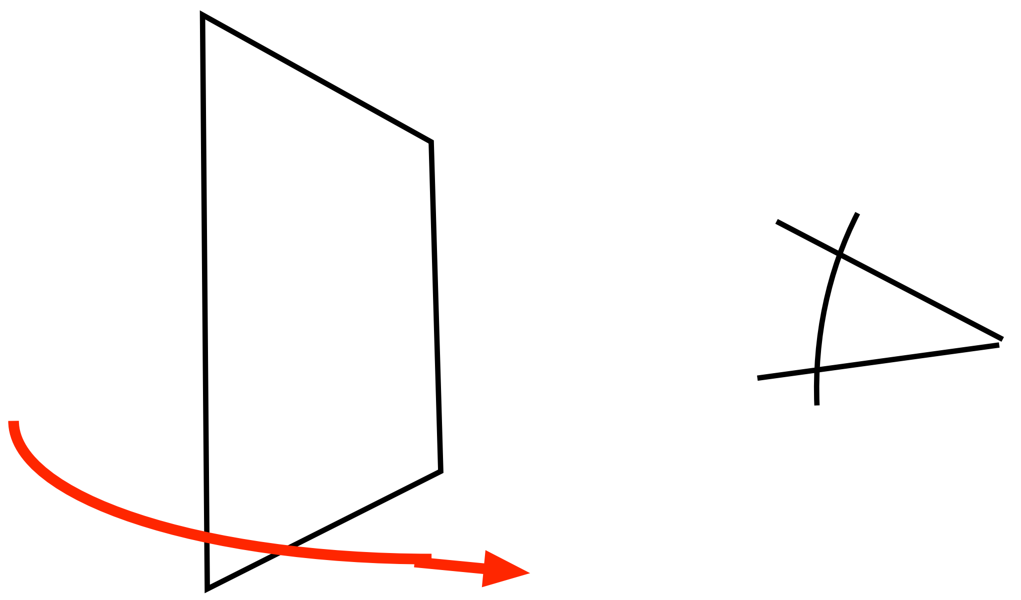
Texture mapping

In common use

Special support by the GPU hardware -
not just a memory access

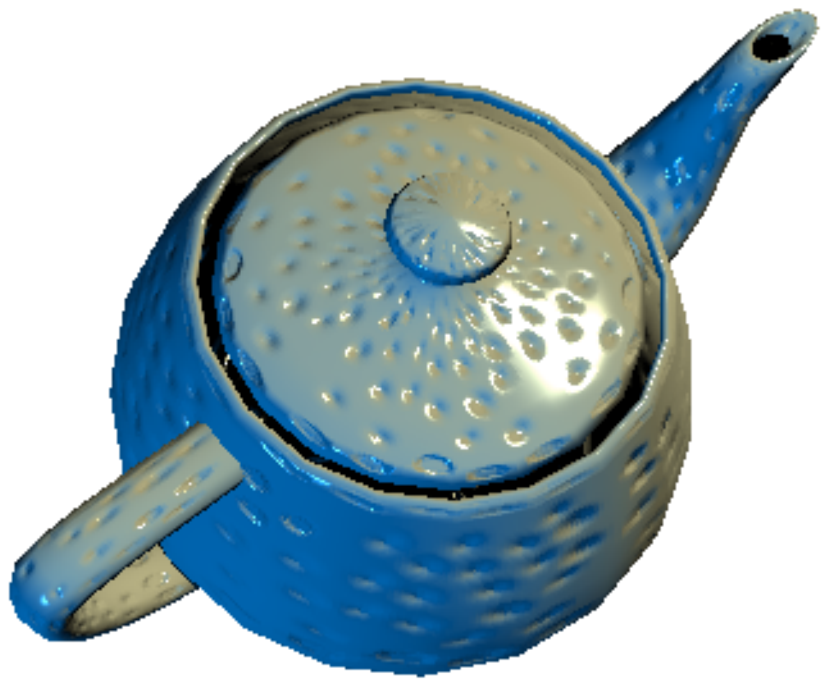
Billboards

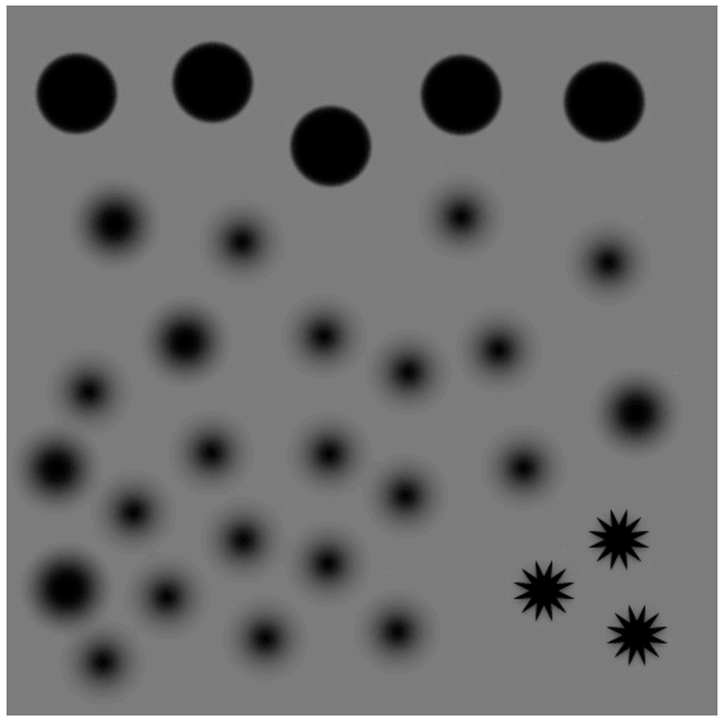
A billboard is a texture mapped polygon, which always faces the viewer



Bump mapping

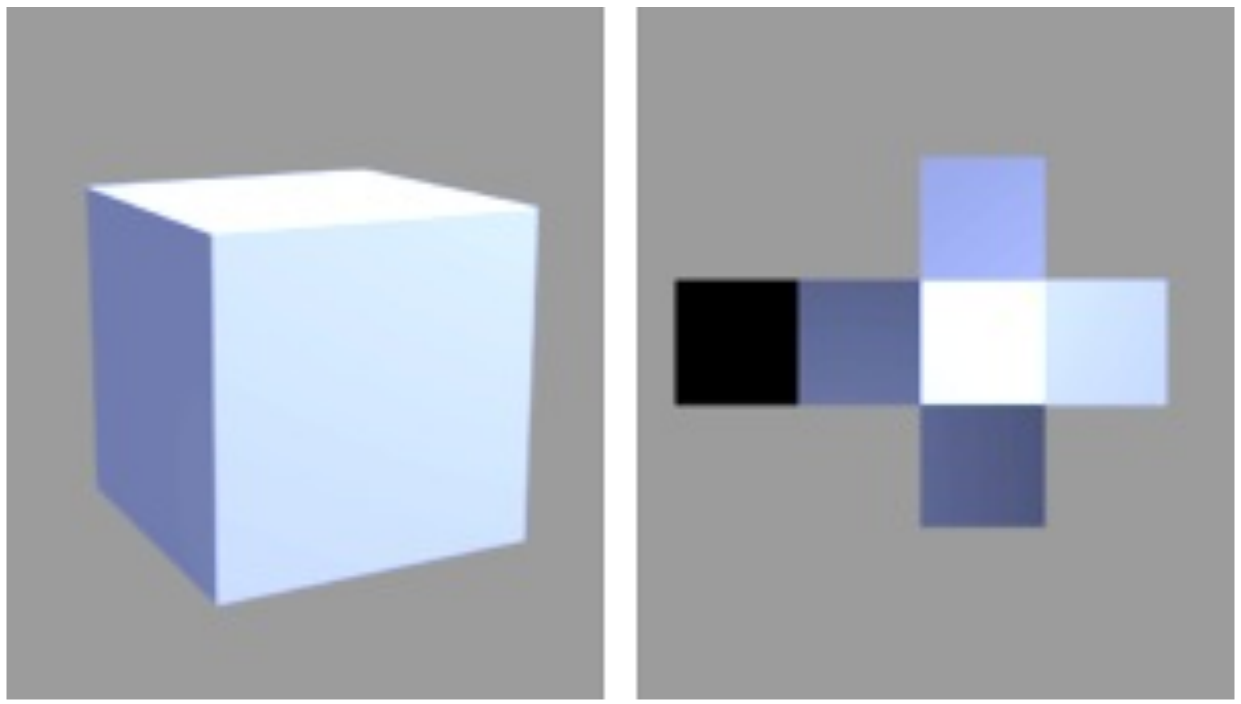
Simulates surface structure by manipulating the normal vector





Light mapping

Applies pre-calculated light to surfaces



(Image from Wikipedia)

Environment mapping

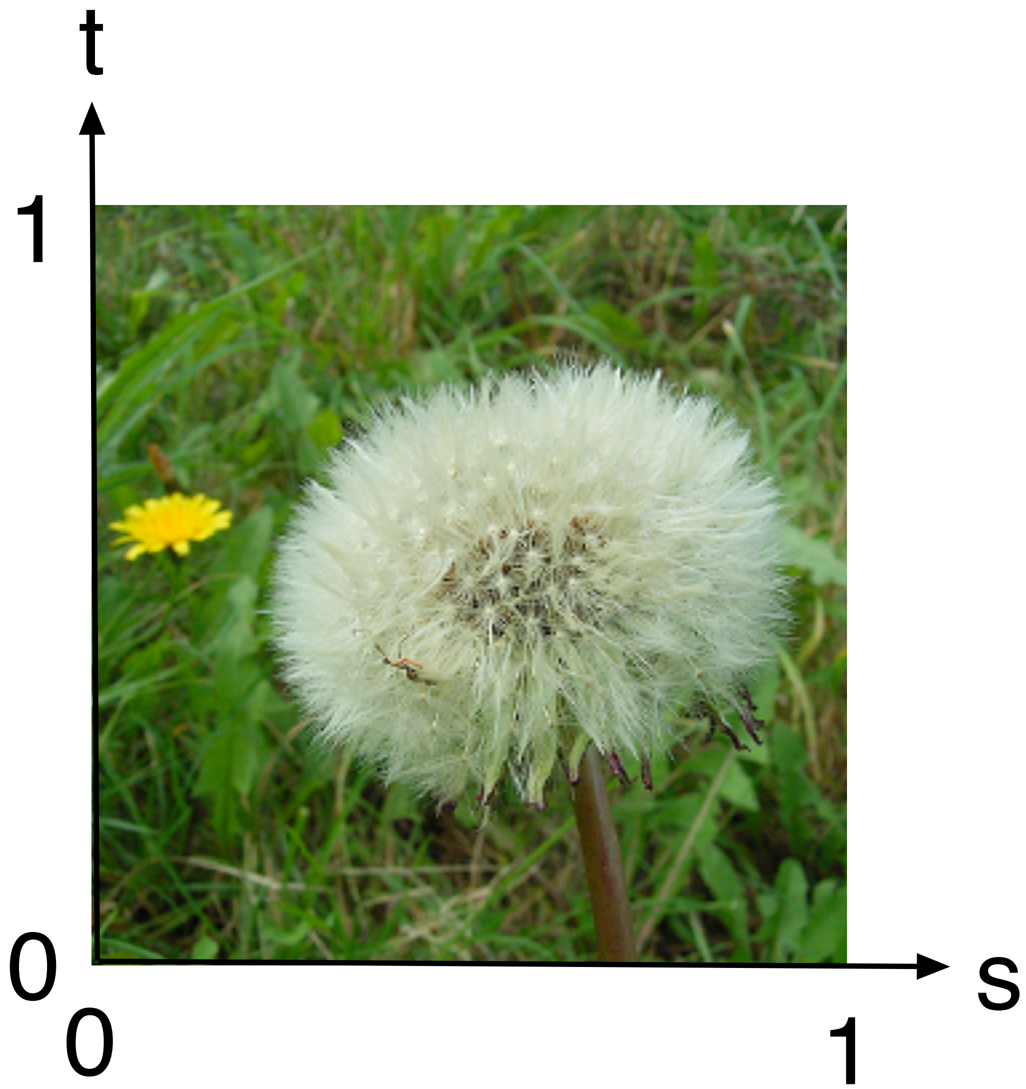
Maps an pre-rendered image as a reflection in the object

Texture mapping

“Wrap” a specified part of “texture space” onto an object

Consider the texture to be an elastic wrapping

Texture space



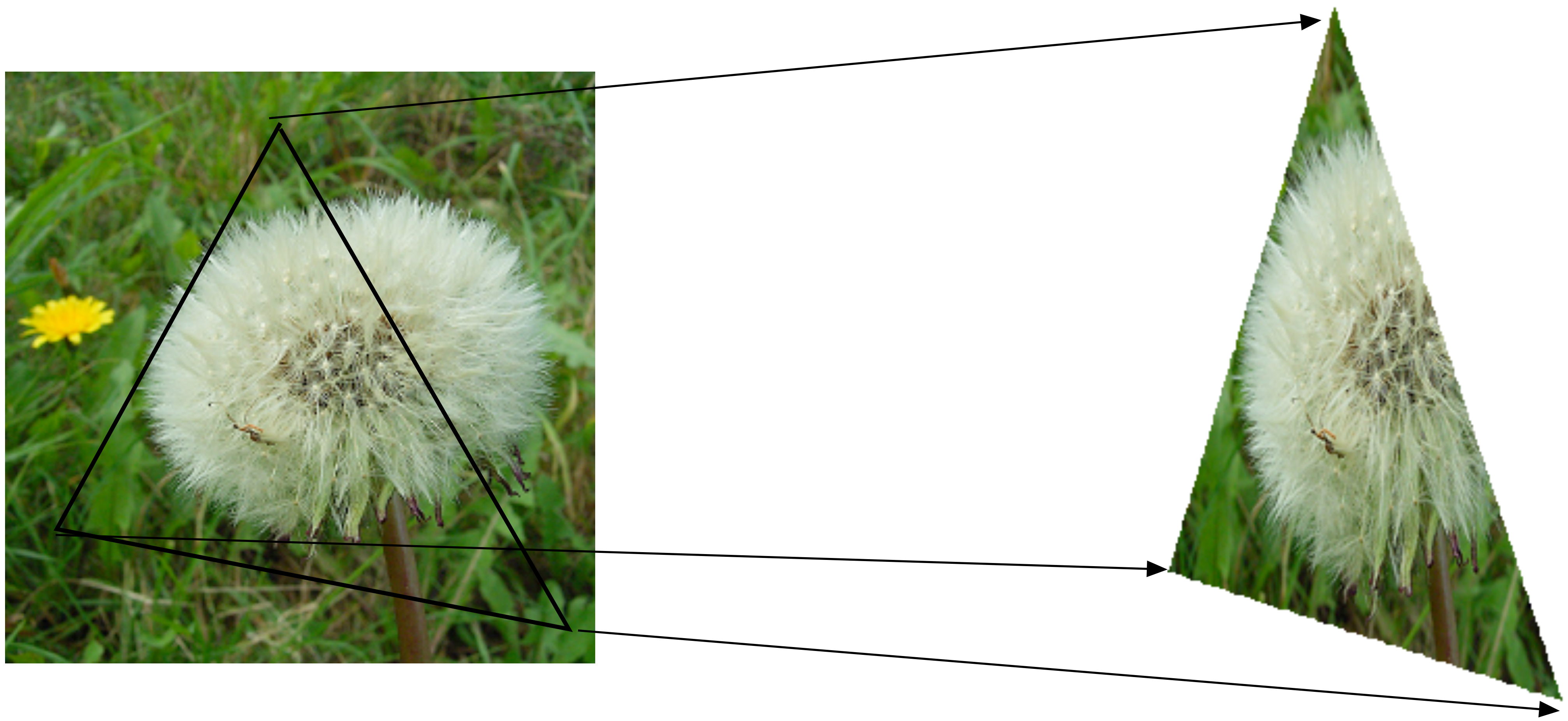
Texture = image used for texture mapping

Built from ”texels”

Texture space is usually 2-dimensional, (s, t), with textures defined in [0, 1]

Mapping from texture to surface

Each vertex has a texture coordinate; interpolate between, look up texture with interpolated coordinates.



Texture coordinates

Texture coordinates often included in models.

LittleOBJLoader.c/.h supports texture coordinates.

Pass as attribute array to vertex shader.

Interpolate from vertex to fragment shader.

Example: Procedural texture

Texture generated by fragment shader!

• Vertex shader passes on texture coordinates
• Texture coordinates are used in a texture generating function in the fragment shader

Simpler than you might think!

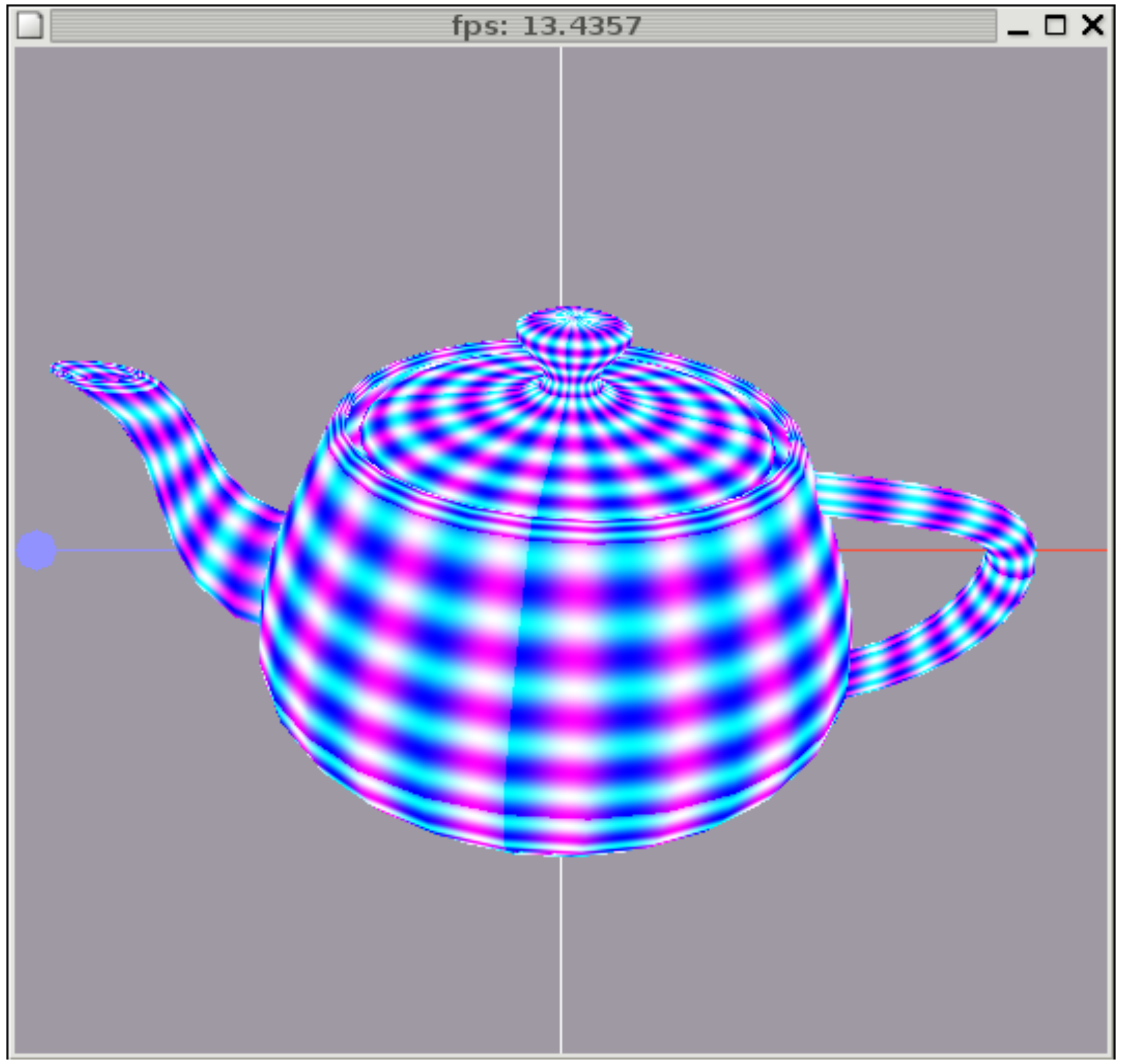
Procedural texture, Vertex shader

```
in  vec3 inPosition;
in vec2 inTexCoord;
out vec2 texCoord;

uniform mat4 mdlMatrix;
uniform mat4 camMatrix;
uniform mat4 projMatrix;

void main(void)
{
  texCoord = inTexCoord;

  gl_Position = projMatrix * camMatrix *
        mdlMatrix * vec4(inPosition, 1.0);
}
```

Procedural texture, Fragment shader

```
in vec2 texCoord;
out outColor;

void main()
{
   float a = sin(texCoord.s*30)/2+0.5;
   float b = sin(texCoord.t*30)/2+0.5;
   outColor = vec4(a, b, 1.0, 0.0);
}
```

Procedural texture
Result



Using pre-defined textures

Step 1: Load texture from file.

Step 2: Upload to texture object

Step 3: Bind to a texture *unit*

Step 4: Access in shader "sampler"

Load texture from file

Standard file functions to get data.

Unpack from file format to texture data. Hard for complex formats!

Packaged in lab code:

LoadTGA

but also LoadTexture (supports PNG and JPEG)

Texture objects

Referring to textures uploaded to the GPU

glGenTextures(...);
reserves texture numbers, making them available to use

glBindTexture(...);
makes a texture the current one

glTexImage2D(...);
loads a texture for the current texture number

Texture units

Texture units are hardware units in the GPU specialized on texture access.

Your texture must be bound to a texture unit to be accessible from shaders!

There are at least 16 units available.

Texture to OpenGL "samplers"

The texture is accessed from shaders using “samplers”
= texture units

Declare as uniforms.

Pass texture unit numbers as uniform variables.

When accessing a texture, the shader uses the texture unit ID, not the texture object!

Texture access sequence

Texture image → Texture object → Texture unit → Shader ”sampler”

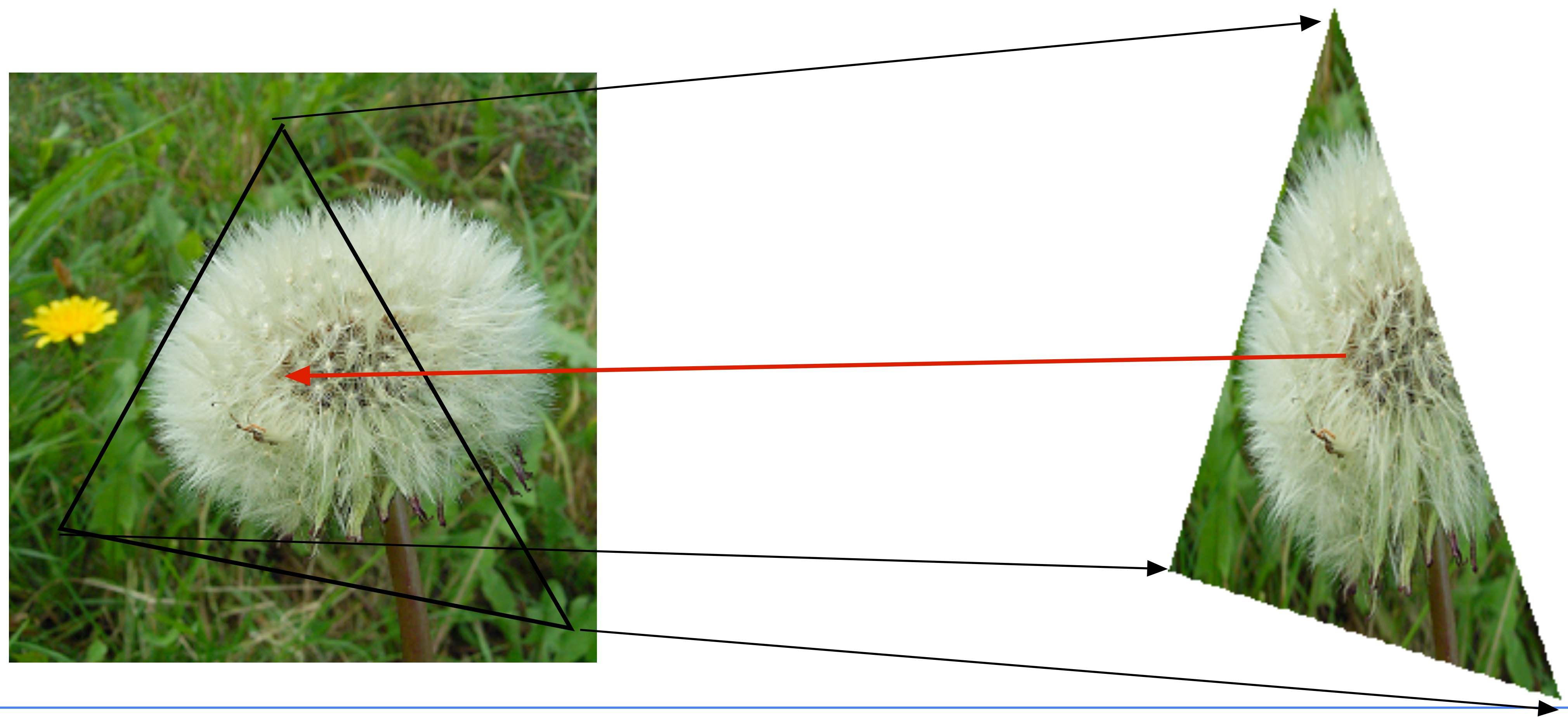
Shaders access texture units, not texture objects!

Why are we messing with texture units?

Texture coordinates are not integers!

We need to access between texels = interpolate

Texture units do this is hardware!



Texture access

Example:

```
uniform sampler2D tex;
out vec4 outColor;
in vec2 texCoord;

void main()
{
    outColor = texture(tex, texCoord);
}
```

texture() performs texture access

Example: texture, uniform sampler:

```
GLuint tex;

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, tex);
loc = glGetUniformLocation(PROG, "tex");
glUniform1i(loc, 0);
```

zero to glUniform1i = texture unit number!

Use in shader:

```
uniform sampler2D tex;

vec3 texval = vec3(texture(tex, gl_TexCoord[0].st));
```

VectorUtils likes to simplify things a bit:

Send a texture unit number to a specific shader:

```
void uploadUniformIntToShader(GLuint shader, char *nameInShader,
                              GLint i)
```

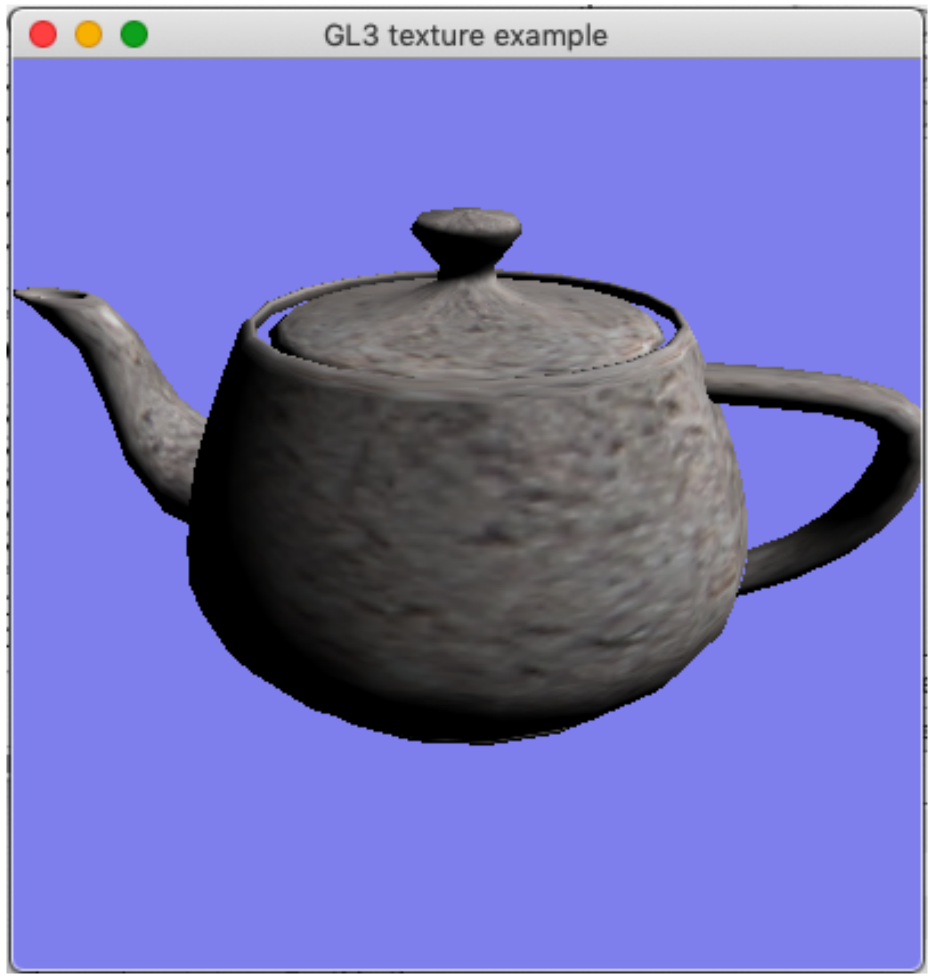
Bind a texture to a specific unit (really just two lines of code):

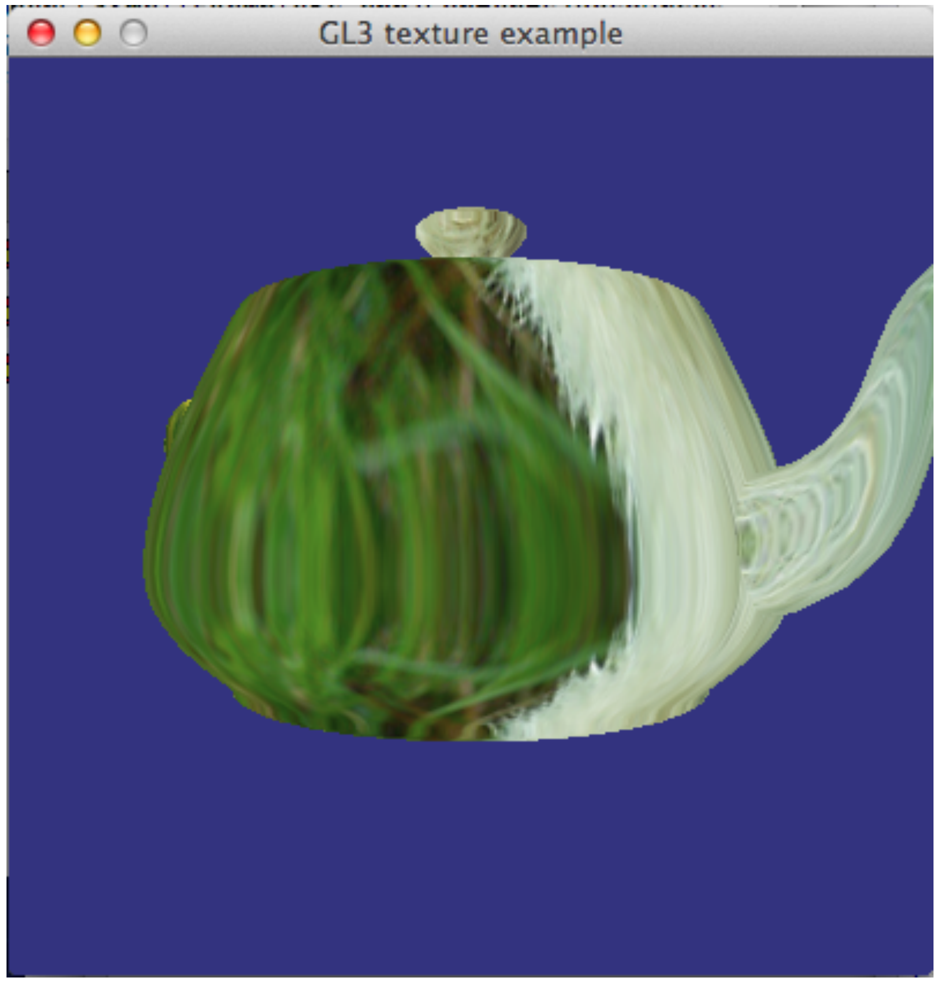
```
void bindTextureToTextureUnit(GLuint tex, int unit)
```

Non-mandatory, use these if it helps. (Costs some performance.)

Texture loaded from image

Result





Next lecture

More texture mapping

Sky boxes and skydomes

Environment mapping

Intro to bump mapping and others

More VSD