Shading

Brian Curless
CSE 457
Autumn 2017

Reading

Optional:

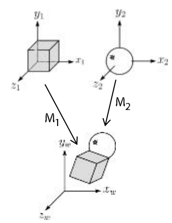
- Angel and Shreiner: chapter 5.
- Marschner and Shirley: chapter 10, chapter 17.

Further reading:

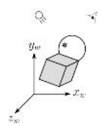
- OpenGL red book, chapter 5.

Basic 3D graphics

With affine matrices, we can now transform virtual 3D objects in their local coordinate systems into a global (world) coordinate system:

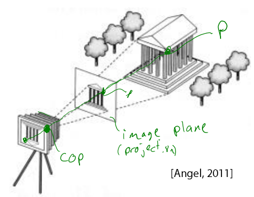


To synthesize an image of the scene, we also need to add light sources and a viewer/camera:



Pinhole camera

To create an image of a virtual scene, we need to define a camera, and we need to model lighting and shading. For the camera, we use a **pinhole camera**.



[Angel, 2011]

The image is rendered onto an **image plane** (usually in front of the camera).

Viewing rays emanate from the **center of projection** (COP) at the center of the pinhole.

The image of an object point **P** is at the intersection of the viewing ray through **P** and the image plane.

But is **P** visible? This the problem of **hidden surface removal** (a.k.a., **visible surface determination**). We'll consider this problem later.

Shading

Next, we'll need a model to describe how light interacts with surfaces.

Such a model is called a **shading model**.

Other names:

- Lighting model
- Light reflection model
- Local illumination model
- Reflectance model
- BRDF

An abundance of photons

Given the camera and shading model, properly determining the right color at each pixel is *extremely hard.*

Look around the room. Each light source has different characteristics. Trillions of photons are pouring out every second.

These photons can:

- interact with molecules and particles in the air ("participating media")
- strike a surface and
  - be absorbed
  - be reflected (scattered)
  - cause fluorescence or phosphorescence.
- interact in a wavelength-dependent manner
- generally bounce around and around

Our problem

We're going to build up to a *approximations* of reality called the **Phong and Blinn-Phong illumination models**.

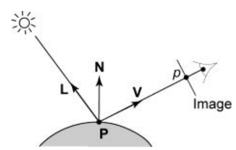
They have the following characteristics:

- *not* physically correct
- gives a "first-order" *approximation* to physical light reflection
- very fast
- widely used

In addition, we will assume **local illumination**, i.e., light goes: light source -> surface -> viewer.

No interreflections, no shadows.

Setup…



Given:

- a point **P** on a surface visible through pixel p
- The normal **N** at **P**
- The lighting direction, **L**, and (color) intensity, I_L, at **P**
- The viewing direction, **V**, at **P**
- The shading coefficients at **P**

Compute the color, I, of pixel p.

Assume that the direction vectors are normalized:

$$\|\mathbf{N}\| = \|\mathbf{L}\| = \|\mathbf{V}\| = 1$$

"Iteration zero"

The simplest thing you can do is…

Assign each polygon a single color:

$$I = k_e$$

where

- I is the resulting intensity
- k_e is the **emissivity** or intrinsic shade associated with the object

This has some special-purpose uses, but not really good for drawing a scene.

"Iteration one"

Let's make the color at least dependent on the overall quantity of light available in the scene:

$$I = k_e + k_a I_{La}$$

- k_a is the **ambient reflection coefficient.**
  - really the reflectance of ambient light
  - "ambient" light is assumed to be equal in all directions
- I_{La} is the **ambient light intensity**.

Physically, what is "ambient" light?

"poor person's interreflection"

Wavelength dependence

Really, k_e, k_a, and I_{La} are functions over all wavelengths λ.

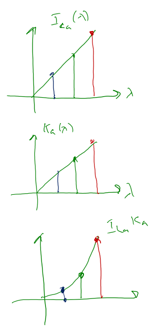
Ideally, we would do the calculation on these functions. For the ambient shading equation, we would start with:

$$I(\lambda) = k_a(\lambda) I_{La}(\lambda)$$

then we would find good RGB values to represent the spectrum $I(\lambda)$.

Traditionally, though, k_a and I_{La} are represented as RGB triples, and the computation is performed on each color channel separately:

$$I^R = k_a^R \; I_{La}^R$$
$$I^G = k_a^G \; I_{La}^G$$
$$I^B = k_a^B \; I_{La}^B$$



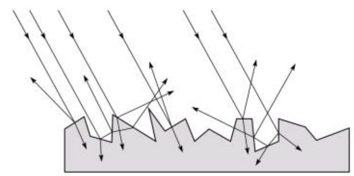
Diffuse reflectors

Emissive and ambient reflection don't model realistic lighting and reflection. To improve this, we will look at **diffuse** (a.k.a., **Lambertian**) reflection.

Diffuse reflection can occur from dull, matte surfaces, like latex paint, or chalk.

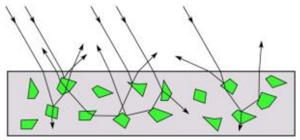
These diffuse reflectors reradiate light equally in all directions.

Picture a rough surface with lots of tiny **microfacets.**



Diffuse reflectors

…or picture a surface with little pigment particles embedded beneath the surface (neglect reflection at the surface for the moment):



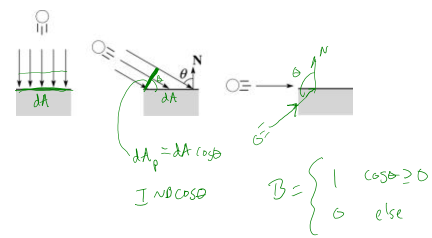
The microfacets and pigments distribute light rays in all directions.

Embedded pigments are responsible for the coloration of diffusely reflected light in plastics and paints.

Note: the figures in this and the previous slide are intuitive, but not strictly (physically) correct.

Diffuse reflectors, cont.

The reflected intensity from a diffuse surface does not depend on the direction of the viewer. The incoming light, though, does depend on the direction of the light source:



"Iteration two"

The incoming energy is proportional to cosθ, giving the diffuse reflection equations:

$$I = k_e + k_a I_{La} + k_d I_L B \, \cos\theta$$

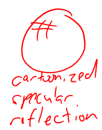
$$= k_e + k_a I_{La} + k_d I_L B(N \cdot L)$$

where:

- k_d is the **diffuse reflection coefficient**
- I_L is the (color) intensity of the light source
- **N** is the normal to the surface (unit vector)
- **L** is the direction to the light source (unit vector)
- B prevents contribution of light from below the surface:

$$B = \begin{cases} 1 & \text{if } N \cdot L > 0 \\ 0 & \text{if } N \cdot L \le 0 \end{cases}$$

Specular reflection



Specular reflection accounts for the highlight that you see on some objects.

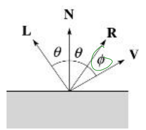
It is particularly important for *smooth, shiny* surfaces, such as:

- metal
- polished stone
- plastics
- apples
- skin

Properties:

- Specular reflection depends on the viewing direction **V**.
- For non-metals, the color is determined solely by the color of the light.
- For metals, the color may be altered (e.g., brass)

Specular reflection "derivation"



For a perfect mirror reflector, light is reflected about **N**, so

$$I = \begin{cases} I_L & \text{if } \mathbf{V} = \mathbf{R} \\ 0 & \text{otherwise} \end{cases}$$

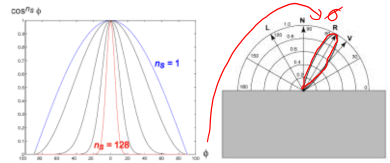
For a near-perfect reflector, you might expect the highlight to fall off quickly with increasing angle ϕ.

Also known as:

- **"rough specular" reflection**
- **"directional diffuse" reflection**
- **"glossy" reflection**

17

Phong specular reflection



One way to get this effect is to take (**R·V**), raised to a power n_s.

Phong specular reflection is proportional to:

$$I_{\text{specular}} \sim B(\mathbf{R} \cdot \mathbf{V})_+^{n_s}$$

where $(x)_+ \equiv \max(0, x)$.

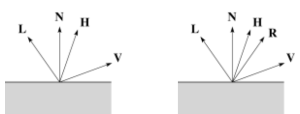
Q: As n_s gets larger, does the highlight on a curved surface get smaller or larger?

18

Blinn-Phong specular reflection

A common alternative for specular reflection is the **Blinn-Phong model** (sometimes called the **modified Phong model**.)

We compute the vector halfway between **L** and **V** as:



Analogous to Phong specular reflection, we can compute the specular contribution in terms of (**N·H**), raised to a power n_s:

$$I_{\text{specular}} \sim B(\mathbf{N} \cdot \mathbf{H})_+^{n_s}$$

where, again, $(x)_+ \equiv \max(0, x)$.

19

"Iteration three"

The next update to the Blinn-Phong shading model is then:

$$I = k_e + k_a I_{La} + k_d I_L B(\mathbf{N} \cdot \mathbf{L}) + k_s I_L B(\mathbf{N} \cdot \mathbf{H})_+^{n_s}$$

$$= k_e + k_a I_{La} + I_L B \left[k_d (\mathbf{N} \cdot \mathbf{L}) + k_s (\mathbf{N} \cdot \mathbf{H})_+^{n_s} \right]$$

where:

- k_s is the **specular reflection coefficient**
- n_s is the **specular exponent** or **shininess**
- **H** is the unit halfway vector between **L** and **V**, where **V** is the viewing direction.

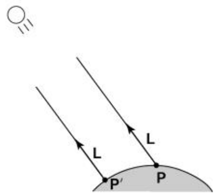
20

Directional lights

The simplest form of lights supported by renderers are ambient, directional, and point. Spotlights are also supported often as a special form of point light.

We've seen ambient light sources, which are not really geometric.

Directional light sources have a single direction and intensity associated with them.

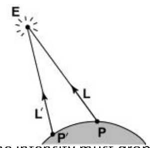


Using affine notation, what is the homogeneous coordinate for a directional light? 0, it's a vector!

Point lights

The direction of a **point light** sources is determined by the vector from the light position to the surface point.

$L = \frac{E-P}{\|E-P\|}$



Physics tells us the intensity must drop off inversely with the square of the distance:

$$f_{atten} = \frac{1}{r^2}$$

Sometimes, this distance-squared dropoff is considered too "harsh." A common alternative is:

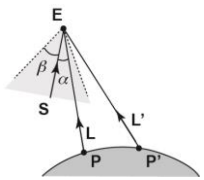
$$f_{atten} = \frac{1}{a + br + cr^2}$$

with user-supplied constants for a, b, and c.

Using affine notation, what is the homogeneous coordinate for a point light? 1, it's a point!

Spotlights

We can also apply a *directional attenuation* of a point light source, giving a **spotlight** effect.



A common choice for the spotlight intensity is:

$$f_{spot} = \begin{cases} \dfrac{(\mathbf{L}\cdot\mathbf{S})^e}{a + br + cr^2} & a \le b \\ 0 & \text{otherwise} \end{cases}$$

where

- $\mathbf{L}$ is the direction to the point light.
- $\mathbf{S}$ is the center direction of the spotlight.
- α is the angle between $\mathbf{L}$ and $\mathbf{S}$
- β is the cutoff angle for the spotlight
- e is the angular falloff coefficient similar to how n_s works

Note: $\alpha \le \beta \;\Leftrightarrow\; \cos^{-1}(\mathbf{L}\cdot\mathbf{S}) \le \beta \;\Leftrightarrow\; \mathbf{L}\cdot\mathbf{S} \ge \cos\beta$.

"Iteration four"

Since light is additive, we can handle multiple lights by taking the sum over every light.

Our equation is now (for spotlight lighting):

property of material

$$I = k_e + \sum_j k_a I_{La,j} + \frac{(\mathbf{L}_j\cdot\mathbf{S}_j)^{e_j}_{b_j}}{a_j + b_j r_j + c_j r_j^2} I_{L,j} B_j \left[k_d (\mathbf{N}\cdot\mathbf{L}_j) + k_s (\mathbf{N}\cdot\mathbf{H}_j)_+^{n_s} \right]$$

This is the Blinn-Phong illumination model (for spotlights). Note that, in practice, we usually set $k_a = k_d$.

Which quantities are spatial vectors?

N, L, S, H

Which are RGB triples?

k, I

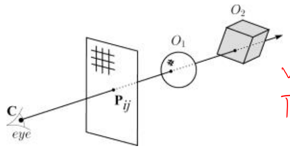
Which are scalars?

a, b, c, r, e, β, n_s

Going back to the pinhole camera…

Recall that the Trace project uses, by default, the pinhole camera model.

If we just consider finding out which surface point is visible at each image pixel, then we are **ray casting**.



talk more about later when we get to Trace!

For each pixel center $\boldsymbol{P}_{ij}$

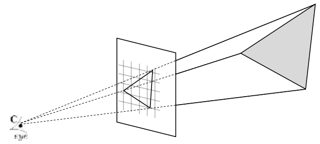
- Send ray from eye point (COP), **C**, through $\boldsymbol{P}_{ij}$ into scene.
- For each object, intersect with the ray
- Select nearest intersection.

Alternative Approach

We could also flip the order of the loops:

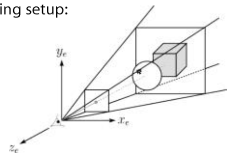
For each triangle in the scene,

- For each pixel, determine if the triangle projects onto it
- Update pixel if this triangle is the closest one so far

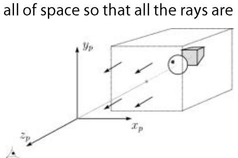


Warping space

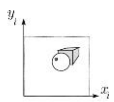
To determine which pixels a triangle projects onto, take this imaging setup:



then warp all of space so that all the rays are parallel:



then just drop the z-coordinate to get pixel coordinates:

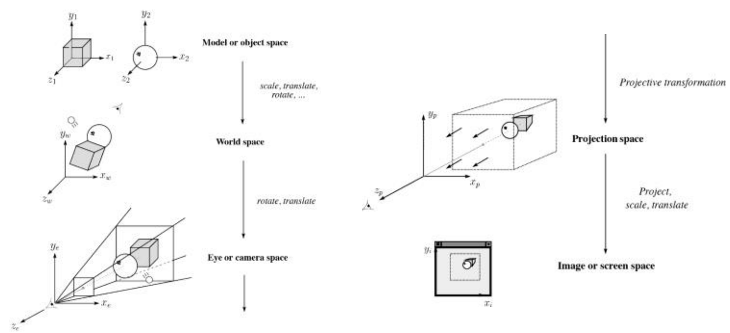


In practice, we keep track of the z-coordinate during drawing to determine visibility. Z-buffering

3D Geometry Pipeline

Graphics hardware follows the "warping space" approach.

Before being turned into pixels, a piece of geometry goes through a number of transformations...



Z-buffer

The **Z-buffer** or **depth buffer** algorithm [Straßer, 1974][Catmull, 1974] can be used to determine which surface point is visible at each pixel.

Here is pseudocode for the Z-buffer hidden surface algorithm, for a viewer looking down the $-z$ axis (bigger – i.e., more positive – z's are closer):

```
for each pixel (i, j) do
    Z-buffer [i, j] ← FAR
    Framebuffer [i, j] ← <background color>
end for
for each triangle A do
    for each pixel (i, j) in A do
        Compute depth z of A at (i, j)
        color ← shader(A, i, j)
        if z > Z-buffer [i, j] then
            Z-buffer [i, j] ← z
            Framebuffer [i, j] ← color
        end if
    end for
end for
```

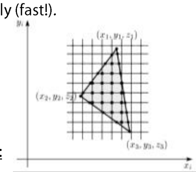
Q: What should *FAR* be set to? *−∞*

Rasterization

We only need to compute the pixel coordinates of the vertices of the triangle – the interior pixels can be determined via interpolation.

This process called **rasterization**.

During rasterization, the z value can be computed incrementally (fast!).



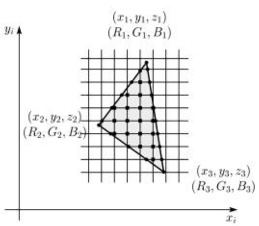
Curious fact:

- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B of [SSS] as a point of comparison for huge memories, but written off as totally impractical.

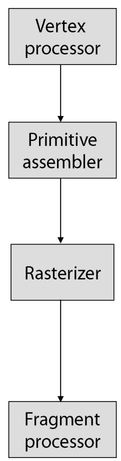
Today, Z-buffers are commonly implemented in hardware.

Rasterization with color

During rasterization, colors can be smeared across a triangle as well:



Hardware Pipeline

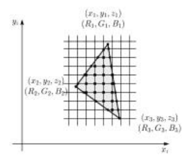


A vertex shader is run for each vertex, and outputs values to be interpolated across the triangle.

"outs" are interpolated

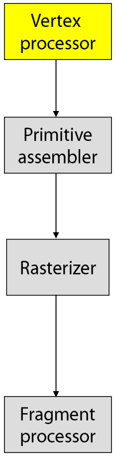
The vertices are grouped into triangles (or other primitives, e.g. lines) to be rasterized. A geometry shader is possibly run to generate more primitives.

We iterate through scanlines, interpolating outputs from the vertex shader at each pixel.



A fragment shader (or pixel shader) is called at each pixel in the primitive, which gets the interpolated values and outputs a final color to the framebuffer.

GLSL: Anatomy of a Vertex Shader

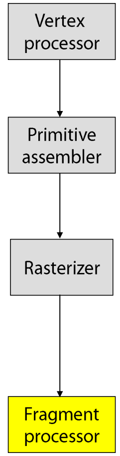



```
#version 400

in vec3 position;
in vec3 vertex_color;

out vec3 color;

uniform mat4 modelview;
uniform mat4 projection;

void main() {
    color = vertex_color;
    gl_Position = projection * modelview * vec4(position, 1.0);
    // color = vec3(1.0, 0.0, 0.0);
    // gl_Position = vec4(1.0, -1.0, 0.0, -1.0);
}
```

interpolated by rasterization

can use in other shaders too

GLSL: Anatomy of a Fragment Shader




```
#version 400

in vec3 color;

out vec4 frag_color;

void main() {
    frag_color = color;
}
```

your fragment shader will do the actual computation of color using normals interpolated from vertices

GLSL: Storage Qualifiers

important things to know about shaders!

uniform: Global value that is the same across all vertices and fragments (for this draw call).

- Model/view/projection matrices, light parameters, material parameters (maybe), textures…

Vertex shader **in**: Per-vertex attributes (that were sent to the GPU)

Vertex shader **out**: Values to be interpolated at each fragment shader

Fragment shader **in**: Interpolated values of Vertex shader **out**'s

Fragment shader **out**: Value to be written to frame buffer

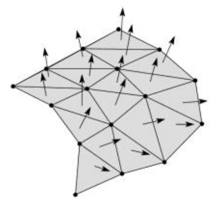
- Normals, positions, colors, material parameters (maybe), texture coordinates…

Shading with per-face normals

Assume each face has a constant normal:

L

V



For a distant viewer and a distant light source and constant material properties over the surface, how will the color of each triangle vary?

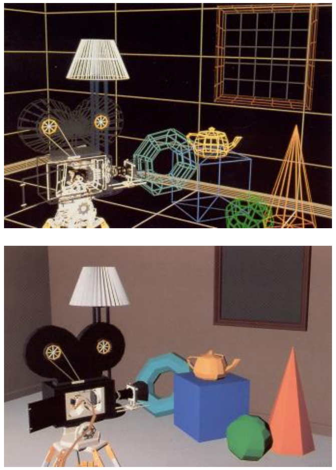
V = constant L = const (because of distance)

No variation, sad! faceted appearance

$I = I_L B(k_d N\cdot L + k_s (N\cdot H)^{n_s}_+)$

$H = \frac{L+V}{\|L+V\|}$

Faceted shading (cont'd)



[Williams and Siegel 1990]

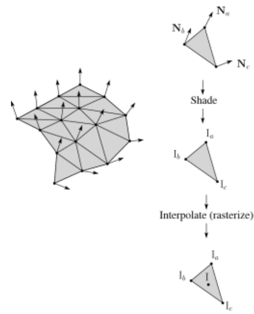
we can do WAY better

37

Gouraud interpolation

Rendering with per triangle normals leads to a faceted appearance. An improvement is to compute per-vertex normals and use graphics hardware to do **Gouraud interpolation**:

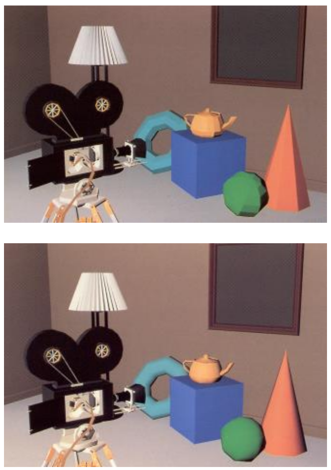
1. Compute normals at the vertices.
2. Shade only the vertices.
3. Interpolate the resulting vertex colors.



removes faceted appearence

38

Facted shading vs. Gouraud interpolation



[Williams and Siegel 1990]

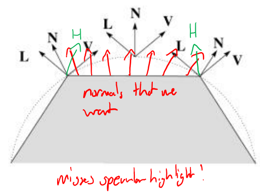
a little better...

39

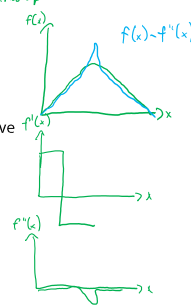
Gouraud interpolation artifacts

Gouraud interpolation has significant limitations.

1. If the polygonal approximation is too coarse, we can miss specular highlights.



Neither H lines up w/ N



2. We will encounter **Mach banding** (derivative discontinuity enhanced by human eye).

This is what graphics hardware does by default.

A substantial improvement is to do…

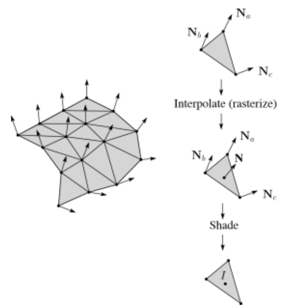
40

Phong interpolation

To get an even smoother result with fewer artifacts, we can perform **Phong *interpolation***.

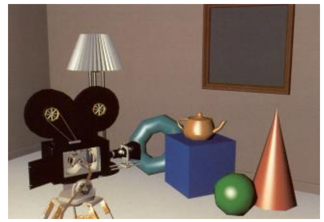
Here's how it works:

1. Compute normals at the vertices.
2. Interpolate normals and normalize.
3. Shade using the interpolated normals.



interpolate normal during rasterization instead of color!

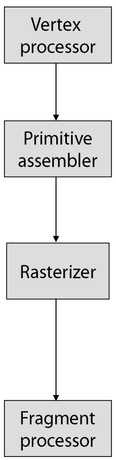
Gouraud vs. Phong interpolation



[Williams and Siegel 1990]

Good!

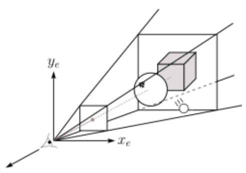
Old pipeline: Gouraud interpolation



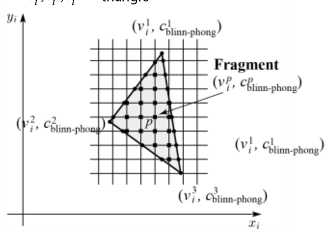
Default vertex processing:

$\mathbf{L} \leftarrow$ determine lighting direction
$\mathbf{V} \leftarrow$ determine viewing direction
$\mathbf{N} \leftarrow$ normalize($\mathbf{n}_e$)
$c_{\text{blinn-phong}}$ ¬ shade with $\mathbf{L}, \mathbf{V}, \mathbf{N}, k_d, k_s, n_s$
$v_i \leftarrow$ project v to image
out $c_{\text{blinn-phong}}$
out v_i

interpolated during rasterization



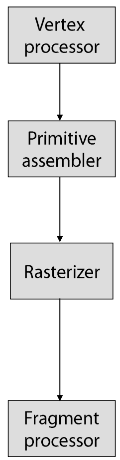
$v_i^1, v_i^2, v_i^3 \rightarrow$ triangle



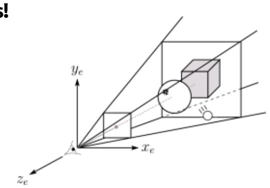
Default fragment processing:
color ¬ $c^p_{\text{blinn-phong}}$

no color calcs in frag shader

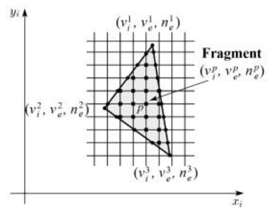
Programmable pipeline: Phong-interpolated normals!



Vertex shader:
$v_i \leftarrow$ project v to image
out $\mathbf{n}_e$
out v_e
out v_i



$v_i^1, v_i^2, v_i^3 \rightarrow$ triangle

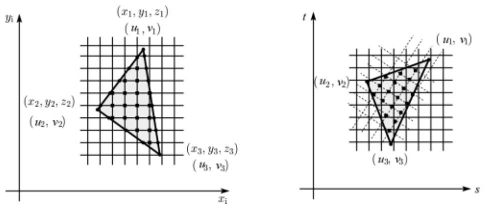


Fragment shader:
$\mathbf{L} \leftarrow$ determine lighting direction (using v_e^p)
$\mathbf{V} \leftarrow$ normalize$(-v_e^p)$
$\mathbf{N} \leftarrow$ normalize$(\mathbf{n}_e^p)$
color $\leftarrow$ shade with $\mathbf{L}, \mathbf{V}, \mathbf{N}, k_d, k_s, n_s$

Texture mapping and the z-buffer

Method:

- Supply per-vertex texture coordinates
- Scan conversion is done in screen space, as usual
- Texture coordinates are interpolated, as usual
- Supply a `uniform` with the texture data
- Each pixel is colored by looking up the texture at the interpolated coordinates



Note: Mapping is more complicated to handle perspective correctly! (OpenGL does this by default)

Choosing Blinn-Phong shading parameters

Experiment with different parameter settings. To get you started, here are a few suggestions:

- Try n_s in the range [0, 100]
- Try $k_a + k_d + k_s < 1$
- Use a small k_a (~0.1)

	n_s	k_d	k_s
Metal	large	Small, color color of metal	Large, color color of metal
Plastic	medium	Medium, color of plastic	Medium, white
Planet	0	varying	0

BRDF

For more physical correctness, we would also weight the specular part by $\mathbf{N}\cdot\mathbf{L}$:

$$I = I_L B\left[k_d(\mathbf{N}\cdot\mathbf{L}) + k_s(\mathbf{N}\cdot\mathbf{L})\left(\mathbf{N}\cdot\frac{\mathbf{L}+\mathbf{V}}{\|\mathbf{L}+\mathbf{V}\|}\right)_+^{n_s}\right]$$

$$= I_L B(\mathbf{N}\cdot\mathbf{L})\left[k_d + k_s\left(\mathbf{N}\cdot\frac{\mathbf{L}+\mathbf{V}}{\|\mathbf{L}+\mathbf{V}\|}\right)_+^{n_s}\right]$$

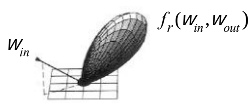
$$= I_L B(\mathbf{N}\cdot\mathbf{L}) f_r(\mathbf{L},\mathbf{V})$$

The function f_r maps incoming (light) directions ω_{in} to outgoing (viewing) directions ω_{out}:

$$f_r(w_{in}, w_{out}) \quad \text{or} \quad f_r(w_{in} \rightarrow w_{out})$$

This function is called the **Bi-directional Reflectance Distribution Function** (**BRDF**).

Here's a plot with ω_{in} held constant:

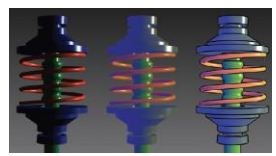


BRDF's can be quite sophisticated…

More sophisticated BRDF's

[Cook and Torrance, 1982]

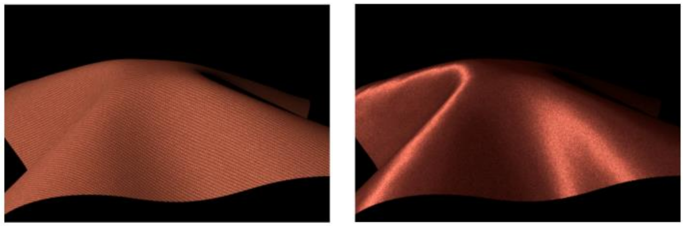
Anisotropic BRDFs [Westin, Arvo, Torrance 1992]



Artistics BRDFs [Gooch]

More sophisticated BRDF's (cont'd)

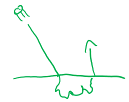
Hair illuminated from different angles [Marschner et al., 2003]



Wool cloth and silk cloth [Irawan and Marschner, 2012]

49

BSSRDFs for subsurface scattering



[Jensen et al., 2001]

50

Summary

You should understand the equation for the Blinn-Phong lighting model described in the "Iteration Four" slide:

- What is the physical meaning of each variable?
- How are the terms computed?
- What effect does each term contribute to the image?
- What does varying the parameters do?

You should also understand the differences between faceted, Gouraud, and Phong *interpolated* shading.

51