Illumination
Aliasing
Ray Tracing

Summer Course on Graphics & Vision

Subodh Kumar

subodh@cse.iitd.ac.in
Learning Objectives

• Ray tracing
• Raster pipeline
• Essentials of light (color), illumination, and shading
• Texture mapping
• OpenGL (Shaders, Geometry buffers, Texture buffers)
Image Generation

• Light
  ➡️ Intensity, Shape, Location, Filters

• Camera
  ➡️ Location, Direction, Orientation, Field of View, Focal plane, Depth of field

• Action
  ➡️ Shape of objects, Material properties and color, Microstructure, Movement specification
Animated films
Animated films
Insufficient Pixels
• Transformations
  ➡ World ➜ Camera ➜ Clip ➜ Image

• Vertex projections
  ➡ Vertex shader

• Rasterization
  ➡ Interpolation, Texture lookup, Illumination
  ➡ Fragment shader

• Output to frame-buffer
  ➡ Z-test, Alpha-test, Color (mask), MSAA..
Spaces & Transformations

• Modeling coordinates
• World coordinates
• Camera coordinates
• Clip coordinates
• Image coordinates
Spaces & Transformations

- Modeling coordinates
- World coordinates
- Camera coordinates
- Clip coordinates
- Image coordinates

Diagram showing the relationships between different spaces and transformations, including modeling coordinates, world coordinates, camera coordinates, clip coordinates, and image coordinates. The diagram includes nodes for various body parts like head, body, arm, leg, trunk, mesh, face, mouth, eye, sphere, and cylinder, connected with transformation matrices (T0, T1, T2, T3, T4).
Ray Tracing

(Center of projection)
Ray Tracing

View Direction

COP (Center of projection)
Ray Tracing

normal

View Direction

I (shadow ray)

Light

COP (Center of projection)
Ray Tracing

View Direction

normal

I (shadow ray)

Light

(Center of projection)
Ray-tracing

View Direction

normal

I (shadow ray)

Light

COP

(Center of projection)
Ray Tracing

Monte-carlo Ray-tracing

normal

View Direction

COP (Center of projection)

Light

I (shadow ray)
Ray-tracing

View Direction

normal

Light

I (shadow ray)

COP

(Center of projection)
Ray-tracing

-COP (Center of projection)

normal

View Direction

Light

I (shadow ray)
Ray-tracing

View Direction

normal

COP (Center of projection)

Light

I (shadow ray)
Ray-tracing

View Direction

normal

Light

r

r'

t

t'

l (shadow ray)

(Center of projection)

Subodh Kumar
Ray Tracing

Recursive Ray-tracing

View Direction

Light

(Center of projection)

normal

r

r'

t

t'

l

(Shadow ray)
Color Trace(const Ray &ray, int depth, Model &model)
{
  if(depth >= MAX_DEPTH) return backGround;
  hitPt = firstIntersection(ray, objects);
  if (hitPt == NULL) return backGround;

  Color color = 0;
  Ray reflectRay = Reflect(ray.Dir, hitPt.Normal);
  Color reflectColor = Trace(reflectRay, depth + 1, model);
  color += reflectColor * fractionReflect(hitPt.object.material, hitPt.Normal, reflectRay.Dir);

  Ray refractRay = Refract(hitPt.object->material, ray.Dir, hitPt.Normal);
  Color refractColor = Trace(refractRay, depth + 1, model);
  color += refractColor * fractionRefract(hitPt.object.material, hitPt.Normal, refractRay.Dir);

  Dir dir2Light = (lightPos - hitPt); dir2Light.normalize();
  Ray shadowRay(hitPt, dir2Light);
  shadowHit = firstIntersection(objects, shadowRay);
  if Dist(hitPt, shadowHit) > Dist(hitPt, lightPos)
    color += lightColor * fractionRefract(hitPt.object.material, hitPt.Normal, dir2Light);
  return color;
}
Color Trace(const Ray &ray, int depth, Model &model) {
    if (depth >= MAX_DEPTH) return backGround;
    hitPt = firstIntersection(ray, objects);
    if (hitPt == NULL) return backGround;

    Color color = 0;
    Ray reflectRay = Reflect(ray.Dir, hitPt.Normal);
    Color reflectColor = Trace(reflectRay, depth + 1, model);
    color += reflectColor * fractionReflect(hitPt.object.material, hitPt.Normal, reflectRay.Dir);

    Ray refractRay = Refract(hitPt.object->material, ray.Dir, hitPt.Normal);
    Color refractColor = Trace(refractRay, depth + 1, model);
    color += refractColor * fractionRefract(hitPt.object.material, hitPt.Normal, refractRay.Dir);

    Dir dir2Light = (lightPos - hitPt); dir2Light.normalize();
    Ray shadowRay(hitPt, dir2Light);
    shadowHit = firstIntersection(objects, shadowRay);
    if (Dist(hitPt, shadowHit) > Dist(hitPt, lightPos))
        color += lightColor * fractionRefract(hitPt.object.material, hitPt.Normal, dir2Light);
    return color;
}
Recursive Ray Tracing

Color Trace(const Ray &ray, int depth, Model &model)
{
    if(depth >= MAX_DEPTH) return backGround;
    hitPt = firstIntersection(ray, objects);
    if (hitPt == NULL) return backGround;

    Color color = 0;
    Ray reflectRay = Reflect(ray.Dir, hitPt.Normal);
    Color reflectColor = Trace(reflectRay, depth + 1, model);
    color += reflectColor * fractionReflect(hitPt.object.material, hitPt.Normal, reflectRay.Dir);

    Ray refractRay = Refract(hitPt.object->material, ray.Dir, hitPt.Normal);
    Color refractColor = Trace(refractRay, depth + 1, model);
    color += refractColor * fractionRefract(hitPt.object.material, hitPt.Normal, refractRay.Dir);

    Dir dir2Light = (lightPos - hitPt); dir2Light.normalize();
    Ray shadowRay(hitPt, dir2Light);
    shadowHit = firstIntersection(objects, shadowRay);
    if Dist(hitPt, shadowHit) > Dist(hitPt, lightPos)
        color += lightColor * fractionRefract(hitPt.object.material, hitPt.Normal, dir2Light);
    return color;
}
Recursive Ray Tracing

Color Trace(const Ray &ray, int depth, Model &model)
{
    if(depth >= MAX_DEPTH) return backGround;
    hitPt = firstIntersection(ray, objects);
    if (hitPt == NULL) return backGround;

    Color color = 0;
    Ray reflectRay = Reflect(ray.Dir, hitPt.Normal);
    Color reflectColor = Trace(reflectRay, depth + 1, model);
    color += reflectColor * fractionReflect(hitPt.object.material, hitPt.Normal, reflectRay.Dir);

    Ray refractRay = Refract(hitPt.object->material, ray.Dir, hitPt.Normal);
    Color refractColor = Trace(refractRay, depth + 1, model);
    color += refractColor * fractionRefract(hitPt.object.material, hitPt.Normal, refractRay.Dir);

    Dir dir2Light = (lightPos - hitPt); dir2Light.normalize();
    Ray shadowRay(hitPt, dir2Light);
    shadowHit = firstIntersection(objects, shadowRay);
    if Dist(hitPt, shadowHit) > Dist(hitPt, lightPos)
        color += lightColor * fractionRefract(hitPt.object.material, hitPt.Normal, dir2Light);
    return color;
}
Color Trace(const Ray &ray, int depth, Model &model) {
    if (depth >= MAX_DEPTH) return backGround;
    hitPt = firstIntersection(ray, objects);
    if (hitPt == NULL) return backGround;

    Color color = 0;
    Ray reflectRay = Reflect(ray.Dir, hitPt.Normal);
    Color reflectColor = Trace(reflectRay, depth + 1, model);
    color += reflectColor * fractionReflect(hitPt.object.material, hitPt.Normal, reflectRay.Dir);

    Ray refractRay = Refract(hitPt.object->material, ray.Dir, hitPt.Normal);
    Color refractColor = Trace(refractRay, depth + 1, model);
    color += refractColor * fractionRefract(hitPt.object.material, hitPt.Normal, reflectRay.Dir);

    Dir dir2Light = (lightPos - hitPt); dir2Light.normalize();
    Ray shadowRay(hitPt, dir2Light);
    shadowHit = firstIntersection(objects, shadowRay);
    if (Dist(hitPt, shadowHit) > Dist(hitPt, lightPos))
        color += lightColor * fractionReflect(hitPt.object.material, hitPt.Normal, dir2Light);
    return color;
}
Recursive Ray Tracing

Color Trace(const Ray &ray, int depth, Model &model)
{
    if(depth >= MAX_DEPTH) return backGround;
    hitPt = firstIntersection(ray, objects);
    if (hitPt == NULL) return backGround;

    Color color = 0;
    Ray reflectRay = Reflect(ray.Dir, hitPt.Normal);
    Color reflectColor = Trace(reflectRay, depth + 1, model);
    color += reflectColor * fractionReflect(hitPt.object.material, hitPt.Normal, reflectRay.Dir);

    Ray refractRay = Refract(hitPt.object->material, ray.Dir, hitPt.Normal);
    Color refractColor = Trace(refractRay, depth + 1, model);
    color += refractColor * fractionRefraction(hitPt.object.material, hitPt.Normal, refractRay.Dir);

    Dir dir2Light = (lightPos - hitPt); dir2Light.normalize();
    Ray shadowRay(hitPt, dir2Light);
    shadowHit = firstIntersection(objects, shadowRay);
    if(Dist(hitPt, shadowHit) > Dist(hitPt, lightPos))
        color += lightColor * fractionReflect(hitPt.object.material, hitPt.Normal, dir2Light);
    return color;
}
Recursive Ray Tracing

Color Trace(const Ray &ray, int depth, Model &model)
{
    if(depth >= MAXDEPTH) return background;
    hitPt = firstIntersection(ray, objects);
    if (hitPt == NULL) return background;

    Color color = 0;
    Ray reflectRay = Reflect(ray.Dir, hitPt.Normal);
    Color reflectColor = Trace(reflectRay, depth + 1, model);
    color += reflectColor * fractionReflect(hitPt.object.material, hitPt.Normal, reflectRay.Dir);

    Ray refractRay = Refract(hitPt.object->material, ray.Dir, hitPt.Normal);
    Color refractColor = Trace(refractRay, depth + 1, model);
    color += refractColor * fractionRefract(hitPt.object.material, hitPt.Normal, reflectRay.Dir);

    Dir dir2Light = (lightPos - hitPt); dir2Light.normalize();
    Ray shadowRay(hitPt, dir2Light);
    shadowHit = firstIntersection(objects, shadowRay);
    if Dist(hitPt, shadowHit) > Dist(hitPt, lightPos)
        color += lightColor * fractionReflect(hitPt.object.material, hitPt.Normal, dir2Light);
    return color;
}

Or Refract
Color Trace(const Ray &ray, int depth, Model &model) {
    if (depth >= MAX_DEPTH) return backGround;
    hitPt = firstIntersection(ray, objects);
    if (hitPt == NULL) return backGround;
    Color color = 0;
    Ray reflectRay = Reflect(ray.Dir, hitPt.Normal);
    Color reflectColor = Trace(reflectRay, depth + 1, model);
    color += reflectColor * fractionReflect(hitPt.object.material, hitPt.Normal, reflectRay.Dir);
    Ray refractRay = Refract(hitPt.object->material, ray.Dir, hitPt.Normal);
    Color refractColor = Trace(refractRay, depth + 1, model);
    color += refractColor * fractionRefract(hitPt.object.material, hitPt.Normal, reflectRay.Dir);
    Dir dir2Light = (lightPos - hitPt); dir2Light.normalize();
    Ray shadowRay(hitPt, dir2Light);
    shadowHit = firstIntersection(objects, shadowRay);
    if Dist(hitPt, shadowHit) > Dist(hitPt, lightPos)
        color += lightColor * fractionReflect(hitPt.object.material, hitPt.Normal, dir2Light);
    return color;
}

for (int y = 0; y < imageHeight; y++) {
    for (int x = 0; x < imageWidth; x++) {
        Ray ray = computeRay(x, y);
        out[y][x] = Trace(ray, 0, model);
    }
}
• Determine samples

• Form ray for each sample

• Recursively trace ray
  ➡ Find first intersection \( I \) (among intersections with all objects)
  ➡ Compute Reflection, Refraction, Shadow rays
  ➡ Compute their collected color recursively
  ➡ Compute illumination due to collected ray colors
  ➡ Send color back for the incoming ray at \( I \)
Rasterize & Render

COP
Camera Space
OpenGL Pipeline

- **Vertex Shader**
- **Assembly**
- **Hull Shader**
- **Tessellator**
- **Domain Shader**
- **Primitive Assembly**
- **Geometry Shader**
- **Clip & Setup**
- **Rasterizer**
- **Fragment Shader**

Flow:
- **Textures etc.**
- **Frame buffer**
- **Blend**

Output:
- **Picture**
OpenGL Pipeline

Vertex Shader → Assembly → Hull Shader → Tessellator → Domain Shader

Primitive Assembly → Geometry Shader → Clip & Setup → Rasterizer

Textures etc. → Frame buffer

Blend → Raster OP
OpenGL Pipeline

Vertex Shader → Assembly → Hull Shader → Tessellator → Domain Shader

Primitive Assembly → Geometry Shader → Clip & Setup → Rasterizer

Stream Out

Textures etc. → Raster OP

Frame buffer

Read out, Set Texture

Blend

Picture
OpenGL Pipeline

- Vertex Shader
- Assembly
- Hull Shader
- Tessellator
- Domain Shader
- Primitive Assembly
- Geometry Shader
- Clip & Setup
- Rasterizer
- Fragment Shader
- Stream Out
- Read out, Set Texture
- Textures etc.
- Frame buffer
- Blend
- Frame buffer
- Textures etc.
Similarly, 3D graphics models are almost always represented in a high-level form: Though in 3D we don't refer to this as "vector graphics"!
VERTICES
A: (1, 1, 1) E: (1, 1, -1)
B: (-1, 1, 1) F: (-1, 1, -1)
C: (1, -1, 1) G: (1, -1, -1)
D: (-1, -1, 1) H: (-1, -1, -1)

TRIANGLES
EHF, GFH, FGB, CBG, GHC, DCH, ABD, CDB, HED, ADE, EFA, BAF

Similarly, 3D graphics models are almost always represented in a high-level form: Though in 3D we don't refer to this as "vector graphics"!
EM Spectrum
• Wave
• Photon
• Energy
• Power
• Wavelength
• Photometry vs Radiometry
• Energy

\[ Q = \frac{hC}{\lambda} \]
• Energy
  ➤ $Q = hC/\lambda$

• Some wavelengths appear brighter (photometry)
  ➤ We measure energy in different bands
    ▸ Spectral power $\Phi$
• Energy
  ➤ $Q = \frac{hC}{\lambda}$

• Some wavelengths appear brighter (photometry)
  ➤ We measure energy in different bands
    ‣ Spectral power $\Phi$
• Energy
  ➤ $Q = \frac{hc}{\lambda}$

• Some wavelengths appear brighter (photometry)
  ➤ We measure energy in different bands
    ‣ Spectral power $\Phi$

• Irradiance, $H$
  ➤ Incoming power/unit area on surface: $\frac{d\Phi}{dA_s}$
• Energy
  ➤ $Q = \frac{hC}{\lambda}$

• Some wavelengths appear brighter (photometry)
  ➤ We measure energy in different bands
    ▸ Spectral power $\Phi$

• Irradiance, $H$
  ➤ Incoming power/unit area on surface: $d\Phi/dA_s$
• Energy
  - \( Q = hC/\lambda \)

• Some wavelengths appear brighter (photometry)
  - We measure energy in different bands
    - Spectral power \( \Phi \)

• Irradiance, \( H \)
  - Incoming power/unit area on surface: \( d\Phi/dA_s \)
• Energy
  - Q = hC/λ

• Some wavelengths appear brighter (photometry)
  - We measure energy in different bands
    - Spectral power Φ

• Irradiance, H
  - Incoming power/unit area on surface: dΦ/dA_s
• Energy
  ➤ $Q = \frac{hC}{\lambda}$

• Some wavelengths appear brighter (photometry)
  ➤ We measure energy in different bands
    ▶ Spectral power $\Phi$

• Irradiance, $H$
  ➤ Incoming power/unit area on surface: $\frac{d\Phi}{dA_s}$

• Radiance, $L$
  ➤ $\frac{dH}{d\omega}$, per unit solid angle: $\frac{d\Phi}{dA_\omega}d\omega$

Radiometry

Power (Flux) vs. $\lambda$
• Energy
  - $Q = \frac{hc}{\lambda}$

• Some wavelengths appear brighter (photometry)
  - We measure energy in different bands
    - Spectral power $\Phi$

• Irradiance, $H$
  - Incoming power/unit area on surface: $d\Phi/dA_s$

• Radiance, $L$
  - $dH/d\omega$, per unit solid angle: $d\Phi/dA_\omega d\omega$
Surface Radiance

\[
\frac{dA}{\cos \theta}
\]

Flux density Fall-off

\[
\Phi = \frac{\Phi}{4\pi r^2}
\]

\[
\Phi = \frac{\Phi}{4\pi r_1^2}
\]
• Behavior depends on
  ➡ Material of surface, Wavelength of light, $\lambda$

• Light can be absorbed

• Light can be scattered
  ➡ Changes direction on ‘collision’

• Light can be reflected/refracted
  ➡ Can change direction
  ➡ Can re-emerge or be absorbed in the medium
    ▷ Depends on thickness
• Behavior depends on
  ➡ Material of surface, Wavelength of light, \( \lambda \)
  
• Light can be absorbed

• Light can be scattered
  ➡ Changes direction on ‘collision’

• Light can be reflected/refracted
  ➡ Can change direction
  ➡ Can re-emerge or be absorbed in the medium
    ➤ Depends on thickness
Light-Surface Interaction

- Behavior depends on
  - Material of surface, Wavelength of light, $\lambda$

- Light can be absorbed

- Light can be scattered
  - Changes direction on ‘collision’

- Light can be reflected/refracted
  - Can change direction
  - Can re-emerge or be absorbed in the medium
    - Depends on thickness
• Behavior depends on
  ➡ Material of surface, Wavelength of light, $\lambda$
• Light can be absorbed
• Light can be scattered
  ➡ Changes direction on ‘collision’
• Light can be reflected/refracted
  ➡ Can change direction
  ➡ Can re-emerge or be absorbed in the medium
    ▶ Depends on thickness
- Behavior depends on
  - Material of surface, Wavelength of light, $\lambda$
- Light can be absorbed
- Light can be scattered
  - Changes direction on ‘collision’
- Light can be reflected/refracted
  - Can change direction
  - Can re-emerge or be absorbed in the medium
    - Depends on thickness

\[ dL(\omega_o) = f(\omega_i, \omega_o) dH(\omega_i) \]

Radiance \quad BSDF \quad Irradiance
• Bi-directional Reflectance/Transmittance Distribution Function
  
  ➡ BSDF: Scattering Reflectance
  ➡ Reflection is symmetric
  ➡ No energy creation

\[ dL(p, \omega_o) = f(p, \omega_i, \omega_o)dH(p, \omega_i) \]

\[ = f(p, \omega_i, \omega_o)L(p, \omega_i) \cos \theta_i \, d\omega_i \]
Bi-directional Reflectance/Transmittance Distribution Function

- BSDF: Scattering Reflectance
- Reflection is symmetric
- No energy creation

$$dL(p, \omega_o) = f(p, \omega_i, \omega_o)dH(p, \omega_i)$$

$$= f(p, \omega_i, \omega_o)L(p, \omega_i) |\cos \theta_i| d\omega_i$$
• Bi-directional Reflectance/Transmittance Distribution Function

- BSDF: Scattering Reflectance
- Reflection is symmetric
- No energy creation

Rendering Equation

BSSRDF: Scattering Surface Reflectance

\[ L(p, \omega_o) = \int_A \int_\Omega S(p, p_i, \omega_i, \omega_o)L(p_i, \omega_i)|\cos \theta_i|d\omega_i dA \]
• Bi-directional Reflectance/Transmittance Distribution Function

⇒ BSDF: Scattering Reflectance

⇒ Reflection is symmetric

⇒ No energy creation

Rendering Equation

BSSRDF: Scattering Surface Reflectance

\[
L(p, \omega_o) = \int_A \int_\Omega S(p, p_i, \omega_i, \omega_o) L(p_i, \omega_i) |\cos \theta_i| \, d\omega \, dA
\]

\[
+ L_c(p, \omega_o)
\]
• Bi-directional Reflectance/Transmittance Distribution Function

- BSDF: Scattering Reflectance
- Reflection is symmetric
- No energy creation

Rendering Equation

BSSRDF: Scattering Surface Reflectance

\[ L(p, \omega_o) = \int_A \int_\Omega S(p, p_i, \omega_i, \omega_o)L(p_i, \omega_i) |\cos \theta_i| d\omega_i dA + L_c(p, \omega_o) \]
BRDF, BTDF

- Bi-directional Reflectance/Transmittance Distribution Function
  - BSDF: Scattering Reflectance
  - Reflection is symmetric
  - No energy creation

Rendering Equation

BSSRDF: Scattering Surface Reflectance

\[ L(p, \omega_o) = \int_A \int_\Omega S(p, p_i, \omega_i, \omega_o)L(p_i, \omega_i) |\cos \theta_i| d\omega dA \]

\[ + L_e(p, \omega_o) \]
Diffuse
BRDF

Diffuse

Glossy
BRDF

Diffuse  

Glossy  

Mirror
Do not depend on $\phi$
Only on $\theta$
Importance Sampling
- BRDF
- $\cos \theta$
- Source brightness

Do not depend on $\phi$
Only on $\theta$
Importance Sampling
- BRDF
- \( \cos \theta \)
- Source brightness

BTN space
Normal, Tangent 1, Tangent 2

BRDF
Do not depend on \( \phi \)
Only on \( \theta \)
CIE Color Standard

• Three components

→ X Y Z
  ▸ Y has luminance (perceived brightness)
  ▸ X and Z have brightness

→ C = X + Y + Z

→ Represented as
  ▸ \( x = \frac{X}{(X+Y+Z)} \), \( y = \frac{Y}{(X+Y+Z)} \), Y
  ▸ x and y have chromaticity, Y has luminance
Color Spaces

- HSV
- RGB

\[
\begin{bmatrix}
R \\
G \\
B
\end{bmatrix} =
\begin{bmatrix}
2.5623 & -1.1661 & -0.3962 \\
-1.0215 & 1.9778 & 0.0437 \\
0.0752 & -0.2562 & 1.1810
\end{bmatrix}
\begin{bmatrix}
X \\
Y \\
Z
\end{bmatrix}
\]

- CMYK
- HDR

➤ Tone Mapping
• Traditionally monitors display R, G, B
  ➔ Engineering convenience

• RED is not the same on every monitor
  ➔ Not even the same everytime on the same HW
     ▸ User knobs, Ambient lighting

• 0:1, in a normalized space
  ➔ Minimum:Maximum screen brightness

• Gamma correction
• Smooth normals make color “smooth”
  ➞ Interpolated from vertices

• All vectors must be in the same space
  ➞ Usually Camera coordinates
• Reflection uniformly in all directions
  - Matte (Non-shiny) appearance
  - Eg, chalk

• Most materials are not ideally diffuse
Diffuse Reflection

- Reflection uniformly in all directions
  - Matte (Non-shiny) appearance
  - E.g., chalk

- Most materials are not *ideally* diffuse
• Light reflects in a single direction
  ➡ Shiny
  ➡ Eg, silvered mirror

• Most materials are not ideally specular
Specular Reflection

- Light reflects in a single direction
  - Shiny
  - Eg, silvered mirror

- Most materials are not **ideally** specular
Most materials are a combination of diffuse and specular

Reflection distribution function

- Need not be in a plane
- Need not be isotropic
Most materials are a combination of diffuse and specular

Reflection distribution function

- Need not be in a plane
- Need not be isotropic
• Lambert’s law

- “Amount” of incident light per unit area is proportional to the cosine of the angle between the normal and the light rays: \( \hat{n} \cdot \hat{l} \)

- Related to surface irradiance
• Unit vector $l$ points to the light source $c_l$.

$$c_{\text{diff}} = f_{\text{diff}} \cdot c_l \cdot \hat{n} \cdot \hat{l}$$
• Distant light source

• A unit length direction vector $\mathbf{d}$ and a color $\mathbf{c}$

• $\mathbf{l} = -\mathbf{d}$

• Color shining on the surface $\mathbf{c}_l = \mathbf{c}$
Point Lights

- Radiates light equally in all directions

- Intensity from a point light source drops off proportionally to the inverse square of the distance from the light

\[ l = \frac{p - v}{|p - v|} \]
\[ c_l = \frac{c_{pnt}}{|p - v|^2} \]
• Sometimes, inverse square falloff behavior is approximated

• A common damping of “distance attenuation” is:

\[
\mathbf{c}_l = \frac{\mathbf{c}_{pnt}}{k_c + k_l d + k_q d^2}
\]

where \( d = \| \mathbf{p} - \mathbf{v} \| \)
• Simply add (in general)

• Interference does happen

→ E.g., soap bubbles

\[ c_{\text{diff}} = \sum_{i} f_{\text{diff}} c_{i} \hat{n} \cdot \hat{l}_{i} \]
• Poor man’s “global illumination”
• Same amount everywhere
• Often, \( f_{\text{amb}} \) is set to equal \( f_{\text{dif}} \)

\[
c = f_{\text{amb}} \ c_{\text{amb}} + \sum_i f_{\text{diff}} \ c_i \ \hat{n} \cdot \hat{l}_i
\]
Blinn’s Model

- Smooth => well defined small highlights,
- Rough => Blurred, larger
- Surface roughness modeled by microfacets
  - Distribution of microfacet normals

Polished: ____________________________
Smooth: ____________________________
Rough: ____________________________
Rougher: ____________________________
• To compute the highlight intensity, we start by finding the unit length ‘halfway’ vector $h$, which is halfway between the vector $l$ pointing to the light and the vector $e$ pointing to the eye (camera)

$$h = \frac{e + l}{|e + l|}$$
The halfway vector $h$ represents the direction that a mirror-like microfacet would have to be aligned in order to cause the maximum highlight intensity.
Specular Highlights

- The microfacet normals generally point in the direction of the macro surface normal.
  - The further $\mathbf{h}$ is from $\mathbf{n}$, fewer facets are likely to align with $\mathbf{h}$.

- The Blinn lighting model:
  \[
  f_h = (\hat{h} \cdot \hat{n})^s
  \]
  - $s$ is shininess or specular exponent.
Specular Highlights

\[ f_h = (\hat{h} \cdot \hat{n})^s \]

- Higher exponent more narrow the highlight
To account for highlights, we simply add an additional contribution to our total lighting equation

$$c = f_{amb} \cdot c_{amb} + \sum_{i} c_{li} \left( f_{diff} \cdot \hat{n} \cdot \hat{l}_{i} + f_{spec}(\hat{n} \cdot \hat{h})^{s} \right)$$

- **Blinn lighting model.**
gl_Position =  MVP * vec4(vPos,1);
normal_cam = VnormM * vNorm;
Vec3 vPos_cam = VM *vPos;
viewDir_cam = vec3(0,0,0) - vPos_cam;
lightDir_cam = lightPos_cam - vPos_cam;
gl_Position = MVP * vec4(vPos, 1);
normal_cam = VnormM * vNorm;
Vec3 vPos_cam = VM * vPos;
viewDir_cam = vec3(0,0,0) - vPos_cam;
lightDir_cam = lightPos_cam - vPos_cam;

layout(location = 0) in vec3 vPos;
layout(location = 1) in vec3 vNorm;
out vec3 normal_cam;
out vec3 viewDir_cam;
out vec3 lightDir_cam;

uniform mat4 MVP, VM, VnormM;
uniform vec3 lightPos_cam;
vec3 n = normalize(normal_cam);
vec3 l = normalize(lightDir_cam);
vec3 v = normalize(viewDir_cam);

float cosTheta = clamp( dot( n, l ), 0, 1 );
vec3 r = reflect(-l, n);
float cosAlpha = clamp( dot( v, r ), 0, 1 );

color = ambient +
    diffuse* lightColor*cosTheta +
    specular * lightColor * pow(cosAlpha, specularity);
vec3 n = normalize(normal_cam);
vec3 l = normalize(lightDir_cam);
vec3 v = normalize(viewDir_cam);

float cosTheta = clamp( dot( n, l ), 0, 1 );
vec3 r = reflect(-l,n);
float cosAlpha = clamp( dot( v, r ), 0, 1 );

color = ambient +
  diffuse* lightColor*cosTheta +
  specular * lightColor * pow(cosAlpha,specularity);
Client Side Setup

Glint vbuf;
glGenBuffers(1, &vbuf);
glBindBuffer(GL_ARRAY_BUFFER, vbuf);
glBufferData(GL_ARRAY_BUFFER, nVerts*sizeof(Vertex), verts, GL_STATIC_DRAW);
GLuint stride = sizeof(verts[0]);
glEnableVertexAttribArray(0); // glGetAttribLocation(program, ”vPos”);
glVertexAttribPointer(0, 3, GL_FLOAT, false, stride, 0);
glEnableVertexAttribArray(1);  // glGetAttribLocation(program, ”vNorm”);
glVertexAttribPointer(1, 3, GL_FLOAT, false, stride, sizeof(verts[0].pos));
// glDrawArrays(GL_TRIANGLES, 0, 3*numTris);

GLuint elementbuf;
glGenBuffers(1, &elementbuf);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuf);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, numTris*3*sizeof(indices[0]), indices , GL_STATIC_DRAW);
glDrawElements(GL_TRIANGLES, 3*numTris,  GL_UNSIGNED_SHORT, 0); // ushort indices[]
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, vsSrc, NULL);
glCompileShader(vs);
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, fsSrc, NULL);
glCompileShader(fs);
GLuint program = glCreateProgram();
glAttachShader(program, vs);
glAttachShader(program, fs);
glLinkProgram(program);
glUseProgram(program);