Shaders

Apodaca and Gritz, Advanced RenderMan
The Lighting Equation

Multiplying the BRDF by an incoming irradiance gives a resulting outgoing radiance

\[ L_{o\text{ due to }i}(\omega_i, \omega_o) = BRDF(\omega_i, \omega_o) dE_i \]

For more complex lighting, we will bounce light all around the scene, and it is tedious to convert between irradiance and radiance (from two slides earlier) to obtain:

Finally, the outgoing radiance considering the light coming from all incoming directions is

\[ L_o = \int_{i \in \text{in}} BRDF(\omega_i, \omega_o)L_i \cos \theta_i \, d\omega_i \]

completing the derivation
Simplification: Point Lights

WELCOME TO

SIMPPLICITY

ENJOY YOUR JOURNEY
Point lights

• Assume that all the incoming light comes from a point light source incoming from direction \( \omega_i \), so that both the BRDF and the cosine term are approximately constant for all nonzero contributions to the integral.

• Then we can approximately write

\[
L_o = BRDF(\omega_i, \omega_o) \cos \theta_i \int L_i \, d\omega_i
\]

• Recall

\[
L = \frac{dI}{dA \cos \theta} \quad \text{and} \quad d\omega = \frac{dA \cos \theta}{r^2}
\]

• Thus for a point light, we may write

\[
\int_{i \in \text{in}} L_i \, d\omega_i = \frac{I}{r^2}
\]

• If all objects are approximately the same distance from the point light (e.g. the sun), we may approximate \( r \) as a constant and fold it into the radiant intensity to get \( \hat{I} \)
Point lights

- The lighting equation for a point light then becomes

\[ L_o = BRDF(\omega_i, \omega_o) \hat{I} \cos \theta_i \]

- Integrating a number of point lights separately in this same fashion results in

\[ L_o = \sum_{j=1}^{\text{#lights}} BRDF(\omega_i^j, \omega_o^j) \hat{I}_i^j \cos \theta_i^j \]

- Recall that this is done independently for each channel: Red, Green, and Blue
Point lights

• Understand what is lost in such an approximation, because it affects the image:
  • All the lighting from other objects in the scene has been turned off, so the scene gets darker overall
  • Surfaces that are not visible from the point light source are completely black (for the same reason)
  • Shadows will have a harsh boundary, unlike the soft shadows obtained from area lights
  • Indoor scenes will not be brighter near the light source, since the radius has been removed
  • Etc...

• The key to using strong/”hacky” approximations is to understand what is lost, so that one may try to “hack”/model it back in when necessary
Simplification: Diffuse Materials

Welcome to Simplicity

Enjoy Your Journey
Assume the object reflects incoming light equally/uniformly in all directions, and also that this doesn’t depend on the incoming direction.

Then the BRDF no longer depends on incoming or outgoing directions and is simply a constant value:

\[
BRDF(\omega_i, \omega_o) = k_d
\]

\[
L_o = BRDF(\omega_i, \omega_o) \hat{I}_i \cos \theta_i = k_d \hat{I}_i \cos \theta_i \quad \left(= k_d \hat{I}_i \max(0, -\omega_i \cdot \hat{N}) \right)
\]

(note how “max” prunes out a hemisphere)
Diffuse Materials

- Models a very rough surface with lots of very tiny microfacets that randomly reflect incoming light in every outgoing direction.

- The shading still depends on the position of the light, because of the cosine term.

- But it does not depend on the position of the camera/viewer.

- Good model for ideally diffuse/dull/matte surfaces, such as chalk.
Simplification: Ambient Shading

WELCOME TO SIMPLICITY

ENJOY YOUR JOURNEY
Ambient Shading

- Models a constant illumination independent of the incident light angle
- Thus, drop the cosine term to get $L_o = k_a \hat{I}_i$ where $k_a$ is the ambient reflectivity for each object

- Useful for adding some light in the shadowed regions that would otherwise potentially be completely black (without global illumination...)
Ambient Shading

- The incoming ambient light can be any color you wish:
  \[(\hat{I}_{i,R}, \hat{I}_{i,G}, \hat{I}_{i,B})\]
- And the reflection coefficients can vary based on color:
  \[(k_{a,R}, k_{a,G}, k_{a,B})\]
- Then we have:
  \[(L_{o,R}, L_{o,G}, L_{o,B}) = (k_{a,R} \hat{I}_{i,R}, k_{a,G} \hat{I}_{i,G}, k_{a,B} \hat{I}_{i,B})\]

- For example, if the incoming light is set to white, adjusting the reflection coefficients gives the object a certain ambient color

- But one could alternatively set the color of the incoming light independently for each object as well
Lights

WELCOME TO SIMPLICITY

ENJOY YOUR JOURNEY
Common Light Types

- **Ambient light**
  - Represents a fixed-intensity light source that hits all objects equally
  - Used to uniformly brighten/darken the entire scene
  - Useful as a “hacky” model for indirect lighting so shadows aren’t completely black

- **Directional light**
  - Lights all objects equally, but only from a specified direction
  - Infinitely large light & infinitely far away
  - Good approximation to sunlight, because the sun is far away and rays of sunlight are approximately parallel
Common Light Types

- **Point light**
  - Light emitted from a single point in space
  - Uniformly lights all directions

- **Spotlight**
  - Light emitted from a single point in space
  - Spreads outwards in a cone
  - Angular subset of a point light

- **Area light**
  - Light emitted from a surface
  - Objects behind the surface are not illuminated
  - Can be thought of (and treated) as a large collection of point lights on a surface
Vertex Data

Welcome to Simplicity

Enjoy your journey
Triangle Color

• Every pixel inside a triangle projected onto the image plane requires (R,G,B) values

• If the triangle has a uniform color, such blue (0,0,255), one can simply color every interior pixel accordingly

• However, if a triangle has multiple colors, we need a way of specifying the color for every pixel

• Since the pixel locations aren’t known ahead of time, we need a way of specifying color throughout the interior of the triangle
Vertex Data

- We store data (such as color, etc.) on the vertices of triangles, and subsequently use interpolation to compute values of this data on the interior of the triangle.

- For example:
  - Specify an (R,G,B) color on each vertex of a triangle.
  - For each pixel inside the triangle, compute the interpolation coordinates for that pixel.
  - Then use these interpolation coordinates to compute an interpolated (R,G,B) color value for that pixel.
Interpolation

- Convert discrete values into a continuous function by filling in all the “in between” values

**Linear Interpolation**

\[ y(t) = (1 - t) y_1 + t y_2 \]
Barycentric Interpolation

2D/3D Edges

\[ p = \alpha_0 p_0 + \alpha_1 p_1 \]
\[ \alpha_0 + \alpha_1 = 1 \]

2D/3D Triangles

\[ p = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2 \]
\[ \alpha_0 + \alpha_1 + \alpha_2 = 1 \]
Geometric Approach

\[ \alpha_0 = \frac{\text{area}(p_0 p_1 p_2)}{\text{area}(p_0 p_1 p_2)} \]

\[ \alpha_1 = \frac{\text{area}(p_0 p p_2)}{\text{area}(p_0 p_1 p_2)} \]

\[ \alpha_2 = \frac{\text{area}(p_0 p_1 p)}{\text{area}(p_0 p_1 p_2)} \]

\[ \text{area}(A, B, C) = \frac{|AB \times AC|}{2} \]
Algebraic Approach

\[ \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2 = p \]

\[ \alpha_0 \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \alpha_1 \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + (1 - \alpha_0 - \alpha_1) \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \]

Solve a 2x2 matrix equation:

\[ \begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \]

- Determinant is zero if the points are co-linear (multiple solutions – but a size zero triangle)
- Works in 3D as well: 3 equations, 2 unknowns, 3x2 matrix, convert to a 2x2 matrix via the normal equations
Review: Linear System Solver

\[ Ax = b \]
\[ x = A^{-1}b \]

\[ A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \Delta = \frac{ad - bc}{\Delta} \]
\[ A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & -\frac{b}{\Delta} \\ -\frac{c}{\Delta} & \frac{a}{\Delta} \end{pmatrix} \]

\[ A^T Ax = A^T b \]
\[ \hat{A}x = \hat{b} \]
Barycentric Coordinates vs. Basis Vectors

Barycentric coordinates:

\[ p = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2 \]

Basis vectors:

\[ p = p_1 + \alpha(p_0 - p_1) + \beta(p_2 - p_1) \]
\[ = \alpha p_0 + (1 - \alpha - \beta) p_1 + \beta p_2 \]

Therefore, we get the equivalence:

\[ \alpha_0 = \alpha, \quad \alpha_1 = 1 - \alpha - \beta, \quad \alpha_2 = \beta \]
Vertex Data

- Interpolate colors (for vertex based BRDFs)
  - \[ R = \alpha_0 R_0 + \alpha_1 R_1 + \alpha_2 R_2 \]
  - \[ G = \alpha_0 G_0 + \alpha_1 G_1 + \alpha_2 G_2 \]
  - \[ B = \alpha_0 B_0 + \alpha_1 B_1 + \alpha_2 B_2 \]

- Interpolate normal vectors (for pixel based BRDFs)
  - \[ N = \alpha_0 N_0 + \alpha_1 N_1 + \alpha_2 N_2 \]

- Interpolate \( z \)-buffer depth values (week 2)
  - \[ z = \alpha_0 z_0 + \alpha_1 z_1 + \alpha_2 z_2 \]

- Interpolate texture coordinates (week 5)
  - \[ u = \alpha_0 u_0 + \alpha_1 u_1 + \alpha_2 u_2 \]
  - \[ v = \alpha_0 v_0 + \alpha_1 v_1 + \alpha_2 v_2 \]
WELCOME TO SIMPLICITY
ENJOY YOUR JOURNEY
Faceted/Flat Shading

- If the BRDF uses the actual normal for each triangle, then one can see all the triangles (as expected)
- One remedy is to use lots and lots of triangles, but that can be overly computationally expensive
Vertex Normal (Average of Face Normals)

Can weight triangle normals by angle, area, etc.

\[ \overrightarrow{N_{V_i}} = \frac{\sum_{k=1}^{n} \overrightarrow{N_{f_{i,k}}}}{n} \]
Smooth Shading

- If the BRDF uses vertex normals interpolated to each pixel, then the triangles appear to be geometrically smoothed out.

Still a faceted silhouette!
Flat vs. Gouraud vs. Phong

- **Flat shading** (uses actual triangle normals)
  - One BRDF evaluation per triangle. Every pixel in the triangle gets the same color.

- **Gouraud shading** (uses vertex normals)
  - One BRDF evaluation per vertex. The resulting vertex colors are interpolated to the interior pixels of each triangle.

- **Phong shading** (uses vertex normals)
  - One BRDF evaluation per pixel. Each pixel uses a vertex normal interpolated to the pixel location.

* Don’t mix up Phong Shading with the Phong reflection model
Corners

- Normals are poorly defined and difficult to compute at corners
- If we use averaged vertex normals to shade a cube, the edges have unrealistic lighting
- Need to specify what type of shader to use for different parts of the object (the same triangle may need both flat and smooth shading!)

![Flat Shading](image1.png) ![Smooth Shading](image2.png)

What should the normal be at the corner?
Transforming Normals

- Can’t just multiply the normal by the model/view matrix
- If the model/view matrix is non-orthogonal, e.g. contains a non-uniform scaling, then the normal will be wrong

![Diagram showing transformed normal](image)

N’ is not the normal

- Want to preserve dot products $N \cdot V$ for arbitrary $V$
- Insert identity matrix $N \cdot V = N^T IV = N^T M^{-1} MV = N' \cdot V'$
- The transformed vectors are of the form $V' = MV$
- And so the transformed normal should be:

$$N' = M^{-T} N$$
Normals & Perspective Transformation

- Translation and rotation preserve lengths and angles.
- Scale, shear, etc. do not preserve lengths and angles, but we can preserve dot products $\mathbf{N} \cdot \mathbf{V}$ for arbitrary $\mathbf{V}$.

- Perspective transforms do not preserve lengths and angles and cannot preserve dot products $\mathbf{N} \cdot \mathbf{V}$.
  - Thus, BRDF calculations that depend on angles must be done before the perspective transformation.
  - Unless the vertex shader submits all information required for shading calculations into the fragment shader.
Phong Reflection Model
Phong Reflection Model

- Most basic shading model; fast, and widely used
- Models three types of reflection:
  - Ambient reflection
  - Diffuse reflection
  - Specular reflection – an approximation to glossy surfaces (highlights) that reflect most light out to the directions that are close to the mirror reflection direction
- Simply add the contributions from each of the three terms together
- OpenGL allows one to independently set the ambient, diffuse, and specular lights (for added modeling convenience)
- E.g. to increase the diffuse reflected light intensity of the whole scene, one only needs to increase the diffuse light intensity without changing the diffuse reflection coefficients of every object
Phong Reflection Model

\[
L_o = \sum_{j \in \text{lights}} \left( k_a \hat{I}^j_{i,a} + k_d \hat{I}^j_{i,d} \max(0, \omega_{i,d} \cdot \hat{N}) + k_s \hat{I}^j_{i,s} \max(V \cdot R^j, 0)^s \right)
\]

- One equation for each channel: R, G, B

* Don’t mix up Phong Shading with the Phong reflection model
Specular Highlights

- accounts for highlights on objects
- important for smooth glossy surfaces
Mirror Reflection

- Mirror – ideally smooth and specular surface
- Law of reflection

\[ \theta_i = \theta_o \]

\[ \mathbf{R} = -\mathbf{L} + 2(\mathbf{L} \cdot \mathbf{N})\mathbf{N} \]

- Incident light direction \( \mathbf{L} \), surface normal vector \( \mathbf{N} \), and reflected light direction \( \mathbf{R} \) are all coplanar
Specular Reflection

- For a glossy but not completely smooth surface, the microscopic spatial variation of normal directions smooths the reflection function into a lobe.
- The intensity falls off as the viewing direction $V$ differs from the mirror reflection direction $R$.
- The shininess coefficient $s$ determines the size of the lobe.

$$L_o = k_s \hat{I}_{i,s} \max(V \cdot R, 0)^s$$

$$\cos \Phi = V \cdot R$$
Shininess Coefficient

Larger $s$ gives a narrower highlight

Converges to mirror reflection as $s \to \infty$

$s = 1$

$s = 64$

$s=2$  $s=4$  $s=8$  $s=16$  $s=32$  $s=64$
Blinn–Phong Specular Model

- $H$ is the halfway vector between $L$ and $V$
- still viewing direction dependent, since $H$ depends on $V$
- If the viewer and the light source are very (infinitely) far away, $L$ and $V$ are unchanged for all pixels and $H$ only need be computed once for the entire image
- better than the Phong specular model for many types of surfaces

\[ L_o = k_s \hat{I}_{i,s} (N \cdot H)^s \]

\[ H = (L + V) / |L + V| \]
Specular Highlights

For a mirror, the outgoing radiance is

\[ L_o(\omega_o) = \begin{cases} k_s \hat{I}_i(\omega_i), & \text{if } \frac{\omega_i + \omega_o}{|\omega_i + \omega_o|} = \hat{N} \\ 0, & \text{otherwise} \end{cases} \]

For a glossy but rough surface, due to the microscopic spatial variation of normal directions, the impulse function is smoothed into a lobe:

\[ L_o(\omega_o) = k_s \left( \hat{N} \cdot \frac{\omega_i + \omega_o}{|\omega_i + \omega_o|} \right)^s \hat{I}_i(\omega_i) \]

Microfacet distribution\((\cos \theta_H)^s = (\hat{N} \cdot \hat{H})^s\)

\[ \text{BRDF}(\omega_i, \omega_o) = k_s (\cos \theta_H)^s / \cos \theta_i \]

\[ L_o = \text{BRDF}(\omega_i, \omega_o) \hat{I}_i \cos \theta_i = k_s (\cos \theta_H)^s \hat{I}_i \]

\[ \cos \theta_H = \hat{N} \cdot \hat{H} \]
Specular Highlights

- More detailed BRDFs for specular highlights can greatly improve the realism

http://content.gpwiki.org/index.php/D3DBook:(Lighting)_Ward#Results
Question 1 (short & long form)

Summarize...
(use class time for short; long form 1000 words)
Assignment 4

- Implement the Epic BRDF
- Implement point lights, directional lights, and hemisphere lights
- Specify material properties for your geometry
- Place lights in your scene
- Render your geometry using the Epic BRDF
- Find and fix problems in the geometry, material properties, lighting, and shader
- Ask others to look at your result and give you feedback
- Iterate!
OpenGL Overview: Pass Light to Shader

// Select proper subroutine for the light type (CPU)
GLint uniformLocation = glGetUniformLocation(
    shaderProgram, GL_FRAGMENT_SHADER,
    "inputLightSubroutine");
GLuint subroutineIndex = glGetUniformLocation(
    shaderProgram, GL_FRAGMENT_SHADER, "pointLightSubroutine");
glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1,
    &subroutineIndex);

// Pass [point] light information to shader (CPU)
glm::vec4 position{1, 1, 1, 0};
GLint uniformLocation = glGetUniformLocation(shaderProgram,
    "pointLight.pointPosition");
glUniform4fv(uniformLocation, 1, glm::value_ptr(position));

... The code to pass light color information to the shader is similar ...

// Use light in the shader (GPU)
struct PointLight {
    vec4 pointPosition;
};
uniform PointLight pointLight;
subroutine vec4 lightingSubroutine(vec4 worldPosition, vec3 worldNormal);
subroutine uniform lightingSubroutine inputLightSubroutine;

inputLightSubroutine(vertexWorldPosition, vertexWorldNormal);
OpenGL Overview: Pass Material to Shader

// Setup memory buffer for the uniform block (CPU)
GLuint blockLocation = glGetUniformLocation(shaderProgram, "InputMaterial");
std::array<const char*, 4> names = {"InputMaterial.matDiffuse", ...
std::array<GLuint, N>& indices;
std::array<GLint, N>& offsets;
std::vector<GLubyte>& data;
GLuint& blockLocation, &bufferLocation;
GLint& blockSize;

glGetActiveUniformBlockiv(shaderProgram, blockLocation,
    GL_UNIFORM_BLOCK_DATA_SIZE, &blockSize);
data.resize(blockSize);
glGetUniformIndices(shaderProgram, 4, names.data(), indices.data());
glGetActiveUniformsiv(shaderProgram, 4, indices.data(), GL_UNIFORM_OFFSET,
    offsets.data());
glGenBuffers(1, &bufferLocation);

// Copy data into the buffer (CPU) – repeat this step to update the material properties.
glm::vec4 diffuse{0.2f, 0.2f, 0.6f, 1.f};
memcpy((void*)(data.data() + offsets[0]), glm::value_ptr(diffuse), sizeof(glm::vec4));
... Do the same for specular, shininess, and ambient ...
glBindBuffer(GL_UNIFORM_BUFFER, bufferLocation);
glBufferData(GL_UNIFORM_BUFFER, blockSize, data.data(), GL_STATIC_DRAW);
glBindBufferBase(GL_UNIFORM_BUFFER, MATERIAL_BINDING_POINT, bufferLocation);
glUniformBlockBinding(shaderProgram, blockLocation, MATERIAL_BINDING_POINT);
Physically Based Materials and BRDF’s

Unreal Engine 4 Overview (Assignment 4)

• Material
  • Base Color $c$
  • Metallic $m$
  • Roughness $r$

• Diffuse (Lambertian)
  \[ \text{diffuse} = \frac{c^{\text{diff}}}{\pi} \]

• Specular (Cook–Torrance Microfacet)
  \[ \text{specular} = \frac{D(H)F(V,H)G(L,V,H)}{4(N \cdot L)(N \cdot V)} \]

• BRDF = $(1 - m) \text{ diffuse} + (m) \text{ specular}$

• More details in the assignment write-up! Start early!
Physically Based Materials and BRDF’s

More Physically Based Shading Resources:


Source Code to Learn From:

• Unreal Engine 4 (Free!): [https://www.unrealengine.com](https://www.unrealengine.com)

• Pixar Renderman (Free!): [https://renderman.pixar.com/](https://renderman.pixar.com/)