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.

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:

\[ |N| = |L| = |V| = 1 \]
“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:

\[ I = k_e + k_a I_{La} \]

- \( k_e \) 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?

Wavelength dependence

Really, \( k_e \), \( k_a \), and \( I_{La} \) are functions over all wavelengths \( \lambda \).

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, \( k_e \) and \( I_{La} \) are represented as RGB triples, and the computation is performed on each color channel separately:

\[
\begin{align*}
I_R &= k_a R I_{La} R \\
I_G &= k_a G I_{La} G \\
I_B &= k_a B I_{La} B
\end{align*}
\]

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 \( I_L \), giving the diffuse reflection equations:

\[
I = k_e + k_a I_L + k_d I_L B \\
= k_e + k_a I_L + k_d I_L B( B )
\]

where:

- \( k_e \) 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:

\[
B = \begin{cases} 
1 & \text{if } N \cdot L > 0 \\
0 & \text{if } N \cdot L \leq 0 
\end{cases}
\]

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

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

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

Also known as:
- “rough specular” reflection
- “directional diffuse” reflection
- “glossy” reflection

I = $I_L$ if $V = R$ otherwise

Phong specular reflection

One way to get this effect is to take $(R \cdot V)$, raised to a power $n_s$.

Phong specular reflection is proportional to:

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

where $(x) = \max(0, x)$.  

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

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 $(N \cdot H)$, raised to a power $n_s$:

$$I_{\text{specular}} = B(N \cdot H)^{n_s}$$

where, again, $(x) = \max(0, x)$.

“Iteration three”

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

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

$$= k_s I_L + I_f B[k_f (N \cdot L) + k_d (N \cdot H)^{n_s}]$$

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.
**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?

---

**Point lights**

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

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

\[
J_{\text{ambient}} = \frac{1}{r^2}
\]

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

\[
J_{\text{ambient}} = \frac{1}{a + br + cr^2}
\]

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

Using affine notation, what is the homogeneous coordinate for a point light?

---

**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:

\[
J_{\text{spot}} = \begin{cases} \frac{(L \cdot S)}{a + br + cr^2} & \alpha \leq \beta \\ 0 & \text{otherwise} \end{cases}
\]

where

- \(L\) is the direction to the point light,
- \(S\) is the center direction of the spotlight,
- \(\alpha\) is the angle between \(L\) and \(S\)
- \(\beta\) is the cutoff angle for the spotlight
- \(\epsilon\) is the angular falloff coefficient

Note: \(\alpha \leq \beta \iff \cos^{-1}(L \cdot S) \leq \beta \iff L \cdot S \cos \beta.\)

---

**“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):

\[
I = k_e + \sum_j \left[ k_a I_{La, j} + J_{\text{spot}} (L_j \times S_j)(\beta_j) \left[ k_s (N \cdot L_j) + k_d (N \cdot H_j)^+ \right] \right]
\]

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

Which quantities are spatial vectors?

Which are RGB triples?

Which are scalars?
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.

For each pixel center $P_{ij}$:
- Send ray from eye point (COP), $C$, through $P_{ij}$ into scene.
- For each object, intersect with the ray.
- Select nearest intersection.

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.

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.

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):

```plaintext
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.

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

```glsl
#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);
}
```

GLSL: Anatomy of a Fragment Shader

```glsl
#version 400

in vec3 color;
out vec4 frag_color;

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

GLSL: Storage Qualifiers

- `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...

- `in`: Per-vertex attributes (that were sent to the GPU)
- `out`: Values to be interpolated at each fragment shader
- `inout`: Interpolated values of `in` attribute
- `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:

For a distant viewer and a distant light source and constant material properties over the surface, how will the color of each triangle vary?
Faceted shading (cont’d)

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.

Faceted shading vs. Gouraud interpolation

Gouraud interpolation artifacts

Gouraud interpolation has significant limitations.
1. If the polygonal approximation is too coarse, we can miss specular highlights.

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…
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.

Gouraud vs. Phong interpolation

Old pipeline: Gouraud interpolation

Programmable pipeline: Phong-interpolated normals!
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 $k_d + k_s + k_p < 1$
- Use a small $k_p$ (~0.1)

<table>
<thead>
<tr>
<th>Material</th>
<th>$n_s$</th>
<th>$k_d$</th>
<th>$k_p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metal</td>
<td>large</td>
<td>Small, color of metal</td>
<td>Large, color of metal</td>
</tr>
<tr>
<td>Plastic</td>
<td>medium</td>
<td>Medium, color of plastic</td>
<td>Medium, white</td>
</tr>
<tr>
<td>Planet</td>
<td>0</td>
<td>varying</td>
<td>0</td>
</tr>
</tbody>
</table>

BRDF

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

$$I = I_B \left[ k_d (N \cdot L) + k_s (N \cdot L) \left( \frac{L + V}{\|L + V\|} \right) \right]$$

$$= I_B N \cdot L \left[ k_d + k_s \left( \frac{L + V}{\|L + V\|} \right) \right]$$

$$= I_B N \cdot L f_r (L, V)$$

The function $f_r$ maps incoming (light) directions $\omega_i$ to outgoing (viewing) directions $\omega_o$:

$$f_r (\omega_i, \omega_o)$$

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

Here’s a plot with $\omega_i$ held constant:

BRDF’s can be quite sophisticated…
More sophisticated BRDF's (cont'd)

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

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

BSSRDFs for subsurface scattering

[Jensen et al., 2001]

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.