Ray Tracing

Ray tracing idea

viewer (eye)

light source

viewing ray

illumination

visible point

objects in scene
Ray tracing algorithm

for each pixel {
    compute viewing ray
    intersect ray with scene
    compute illumination at visible point
    put result into image
}

Generating eye rays

- Use window analogy directly
Generating eye rays—orthographic

- Just need to compute the view plane point $s$:

$$p = s; \quad d = d_v$$

$$r(t) = p + td$$

- but where exactly is the view rectangle?
Generating eye rays—orthographic

- Camera basis $u, v, w$
- View rectangle in $u$-$v$ plane
  - $l, r, t, b$

Generating eye rays—orthographic

- Pixel $(i,j)$ located at $(u,v)$
  - $u = l + (r - l)(i + 0.5)/n_x$
  - $v = b + (t-b)(j + 0.5)/n_y$
Generating eye rays—orthographic

- Pixel \((i,j)\) located at \((u,v)\)
  - ray direction \(\rightarrow -w\)
  - ray origin \((s)\) \(\rightarrow e + uu + vv\)

Generating eye rays—perspective

- Camera basis \(u, v, w\)
- View rectangle in \(u-v\) plane
  - \(l, r, t, b\)
Generating eye rays—perspective

- Pixel \((i,j)\) located at \((u,v)\)
  - \(u = l + (r - l)(i + 0.5)/n_x\)
  - \(v = b + (t-b)(j + 0.5)/n_y\)

Generating eye rays—perspective

- Pixel \((i,j)\) located at \((u,v)\)
  - ray origin \(\rightarrow e\)
  - ray direction \(\rightarrow e + uu + vv - dw\)
Pixel-to-image mapping

- One last detail: \((u, v)\) coords of a pixel

\[
\begin{align*}
  u &= l + (r - l)(i + 0.5)/n_x \\
  v &= b + (t - b)(j + 0.5)/n_y
\end{align*}
\]

Ray intersection
Ray: a half line

- Standard representation: point $p$ direction $d$
  \[ r(t) = p + td \]
  
- directly generate the points on the line
- if we restrict to $t > 0$ then we have a ray
- note replacing $d$ with $ad$ doesn’t change ray ($a > 0$)

![Diagram of a ray with t-values](image)

Image so far

- With eye ray generation and sphere intersection

```java
Surface s = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= lx < nx
  for 0 <= ly < ny {
    ray = camera.getRay(lx, ly);
    hitSurface, t = s.intersect(ray, 0, +inf)
    if hitSurface is not null
      image.set(lx, ly, white);
  }
```
Image so far

- With eye ray generation and scene intersection

```java
for 0 <= ly < ny 
    for 0 <= lx < nx { 
        ray = camera.getRay(lx, ly); 
        o = scene.trace(ray, 0, +inf); 
        image.set(lx, ly, o); 
    }
... 

Scene.trace(ray, tMin, tMax) {
    surface, t = surf.intersect(ray, tMin, tMax); 
    if (surface != null) return surface.color(); 
    else return black; 
}
```

Shading model

- Compute light reflected toward camera / eye

  - **v** - view direction
  - **l** - light direction
  - **n** - surface normal

  - surface parameters
    - color, shininess, etc.
Diffuse reflection

- Light is scattered uniformly in all directions
- the surface color is the same for all viewing directions

Top face of cube receives a certain amount of light
Top face of 60° rotated cube intercepts half the light
In general, light per unit area is proportional to $\cos \theta = l \cdot n$

Diffuse

- Lambertian shading
- Shading independent of view direction

$$L_d = k_d I \max(0, n \cdot l)$$

- $L_d$ - diffusely reflected light
- $k_d$ - diffuse coefficient
- $I$ - illumination from source
Lambertian shading

- $P_1$

$k_d$ →

Diffuse shading
Scene.trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if surface is not null {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point, normal, light);
    }
    else return backgroundColor;
}
...

Surface.shade(ray, point, normal, light) {
    v = -normalize(ray.direction);
    l = normalize(light.pos - point);
    // compute shading
}

---

Shadows

- Surface is only illuminated if nothing blocks its view of the light.
- With ray tracing it’s easy to check
  - just intersect a ray with the scene!
Image so far

```
Surface.shade(ray, point, normal, light) {
    shadRay = (point, light.pos – point);
    if (shadRay not blocked) {
        v = normalize(ray.direction);
        l = normalize(light.pos – point);
        // compute shading
    }
    return black;
}
```

Multiple lights

- Important to fill in black shadows
- Just loop over lights, add contributions
- Ambient shading
  - black shadows are not really right
  - one solution: dim light at camera
  - alternative: add a constant “ambient” color to the shading...
Image so far

```
shade(ray, point, normal, lights) {
    result = ambient;
    for light in lights {
        if (shadow ray not blocked) {
            result += shading contribution;
        }
    }
    return result;
}
```

Specular

- Blinn-Phong
  - Intensity depends on view direction
  - Highlights
Specular

- Blinn-Phong
  - reflection brightest when \( \mathbf{v} \) and \( \mathbf{i} \) are symmetric across surface normal

\[
\mathbf{h} = \text{bisector}(\mathbf{v}, \mathbf{l}) = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}
\]

\[
L_s = k_s I \max(0, \cos \alpha)^p
= k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p
\]
Specular

- $p$ - Phong exponent > 1

Fig. 16.9 Different values of $\cos^n \alpha$ used in the Phong illumination model.
Diffuse + Phong shading

- Independent of everything
- add constant color
- fill in black shadows

\[ L_a = k_a I_a \]

- \( L_a \) - reflected ambient light
- \( k_a \) - ambient coefficient
- Ambient + diffuse + specular

\[ L = L_a + L_d + L_s \]
\[ = k_a I_a + k_d I \max(0, n \cdot l) + k_s I \max(0, n \cdot h)^p \]