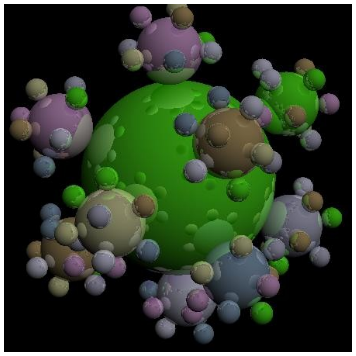
Basic Ray Tracing



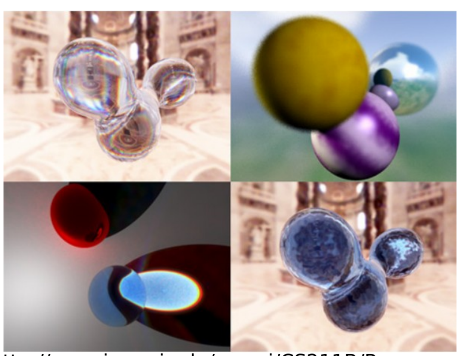
Readings: Chapter 4

Some slides courtesy of Steven Marschner

Announcements

- HW3 due on the 8th (this Wed.)
- Will have a ray-tracer exercise next class. Please check it out by tomorrow morning.
- Lecture by Yuval Boger (CEO), Sensics on 4/27 (Optional)

What is ray tracing?



http://www.ics.uci.edu/~gopi/CS211B/Ray Tracing%20tutorial.pdf

Render images with computers.

Physically correct images are composed by light and that light will usually come from a light source and bounce around as light rays in a scene before hitting our eyes or a camera. By being able to reproduce in computer simulation the path followed from a light source to our eye, we should be able to determine what our eyes see.

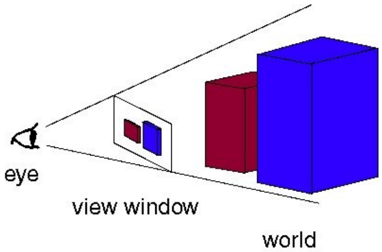
Objectives

- Learn the basic ray tracer
  - When to use it
  - How to do it in OpenGL
  - What are these techniques

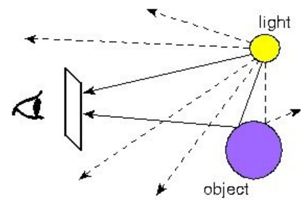
- Resources:
  - https://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm
  - (better in my opinion) http://www.ics.uci.edu/~gopi/CS211B/RayTracing%20tutorial.pdf

High-level idea

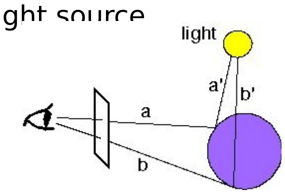
- Find the color of each pixel on the view window.
- E.g., if our image resolution is 640x480, we'd break up the view window into a grid of 640 squares across and 400 square down. Ray tracer is to assign colors to these points.



- Tracing rays from the light source to the eye. Lots of rays are wasted because they never reach the eye.

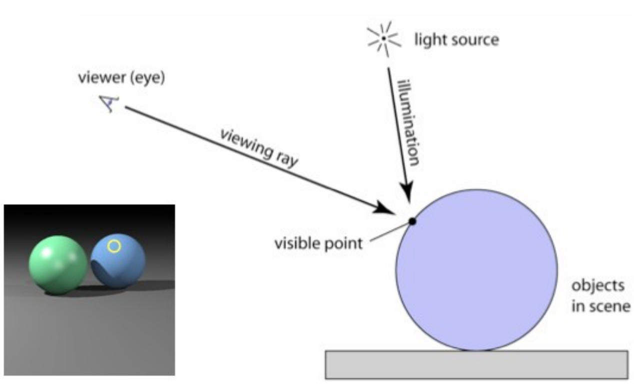


- We trace a new ray from each ray-object intersection directly towards the light source

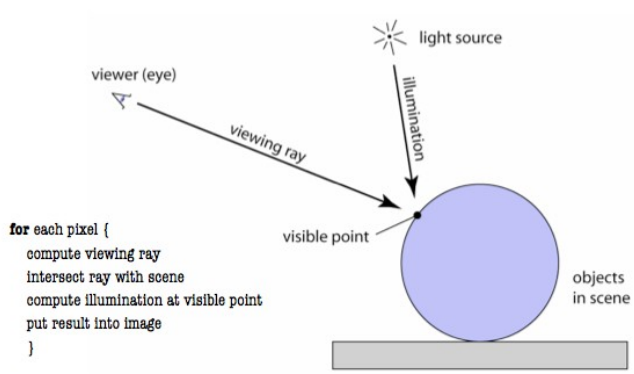


http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html#glas90

Ray tracing idea

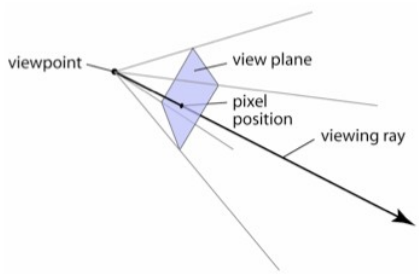


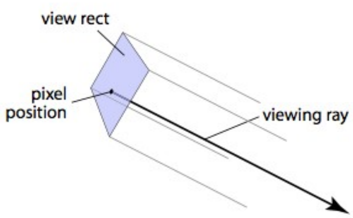
Ray tracing algorithm



Generate eye rays

- Use window analogy directly



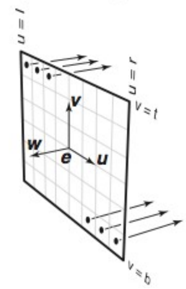


Generate eye rays - orthographic

- Positioning the view rectangle
  - Establish three vectors to be camera basis: u, v, w
  - View rectangle is in u-v plane, specified by l, r, t, b

– now ray generation is easy:

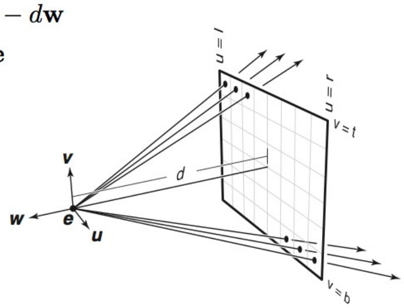
$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v}$$
$$\mathbf{p} = \mathbf{s};\ \mathbf{d} = -\mathbf{w}$$
$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$



Generating eye rays - perspective

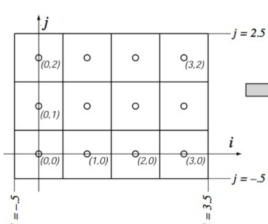
- Compute s in the same way; just substract dw
  - Coordinates of s are (u, v, -d)

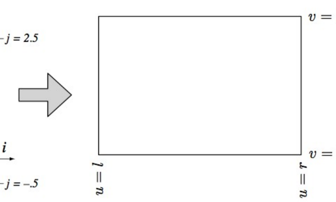
$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v} - d\mathbf{w}$$
$$\mathbf{p} = \mathbf{e};\ \mathbf{d} = \mathbf{s} - \mathbf{e}$$
$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$



Pixel-to-image mapping

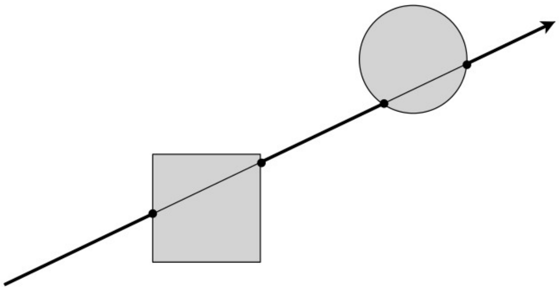
- One last detail: (u, v) coords of a pixel





$$u = l + (r - l)(i + 0.5)/n_x$$
$$v = b + (t - b)(j + 0.5)/n_y$$

Ray intersection

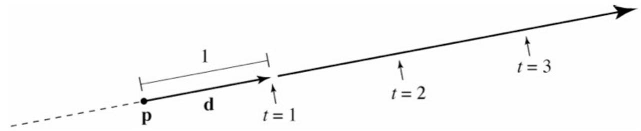


Ray: a half line

- Standard representation: point p and direction d

$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

- this is a *parametric equation* for the line
- lets us directly generate the points on the line
- if we restrict to $t > 0$ then we have a ray
- note replacing $\mathbf{d}$ with $a\mathbf{d}$ doesn't change ray ($a > 0$)



Ray-sphere intersection: algebraic

- Condition I: point is on ray

$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

- Condition 2: point is on sphere
  - Assume unit sphere:

$$\|\mathbf{x}\| = 1 \Leftrightarrow \|\mathbf{x}\|^2 = 1$$
$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{x} - 1 = 0$$

- Substitute

$$(\mathbf{p} + t\mathbf{d}) \cdot (\mathbf{p} + t\mathbf{d}) - 1 = 0$$

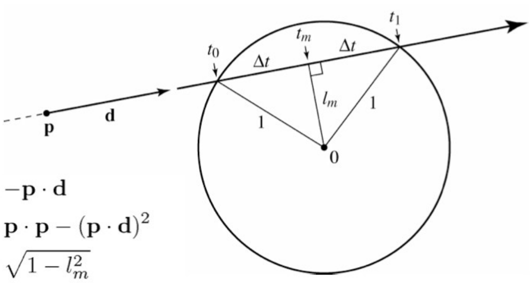
  - This is a quadratic equation in t

Ray-sphere intersection: algebraic

- Solution for t by quadratic formula:
  - Simpler from holds when d is a unit vector
  - But we won't assume this in practice
  - I will use the unit-vector form to make the geometric interpolation

$$t = \frac{-\mathbf{d} \cdot \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \mathbf{p})^2 - (\mathbf{d} \cdot \mathbf{d})(\mathbf{p} \cdot \mathbf{p} - 1)}}{\mathbf{d} \cdot \mathbf{d}}$$
$$t = -\mathbf{d} \cdot \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \mathbf{p})^2 - \mathbf{p} \cdot \mathbf{p} + 1}$$

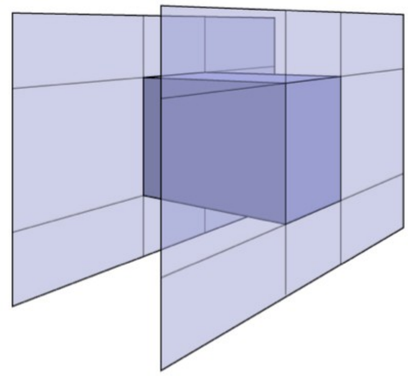
Ray-sphere intersection: geometric



$$t_m = -\mathbf{p}\cdot\mathbf{d}$$
$$l_m^2 = \mathbf{p}\cdot\mathbf{p} - (\mathbf{p}\cdot\mathbf{d})^2$$
$$\Delta t = \sqrt{1 - l_m^2}$$
$$= \sqrt{(\mathbf{p}\cdot\mathbf{d})^2 - \mathbf{p}\cdot\mathbf{p} + 1}$$
$$t_{0,1} = t_m \pm \Delta t = -\mathbf{p}\cdot\mathbf{d} \pm \sqrt{(\mathbf{p}\cdot\mathbf{d})^2 - \mathbf{p}\cdot\mathbf{p} + 1}$$

Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs

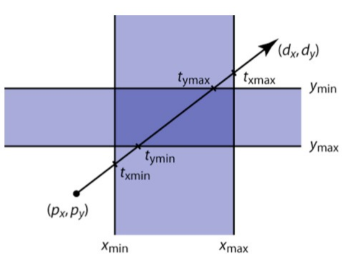


Ray-slab intersection

- 2D example
- 3D is the same!

$$p_x + t_{x\min}\, d_x = x_{\min}$$
$$t_{x\min} = (x_{\min} - p_x)/d_x$$

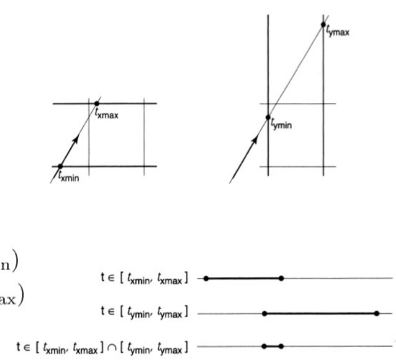
$$p_y + t_{y\min}\, d_y = y_{\min}$$
$$t_{y\min} = (y_{\min} - p_y)/d_y$$



Intersection intersection

- Each intersection is an interval
- Want last entry point and first exist point

$$t_{\min} = \max(t_{x\min}, t_{y\min})$$
$$t_{\max} = \min(t_{x\max}, t_{y\max})$$



Shirley fig. 10.16

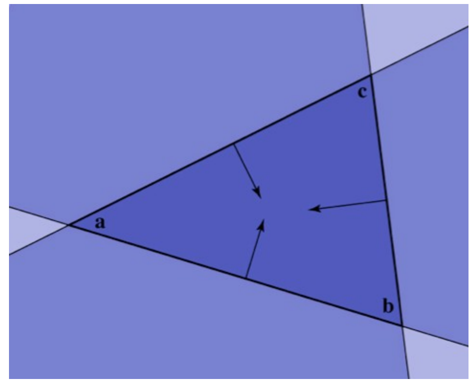
Ray-triangle intersection

- Condition I: point is on ray
  - R(t) = p + t d
- Condition 2: point is on plane
  - (x-a) . n = 0
- Condition 3: point is on the inside of all three edges
- First solve 1 & 2 (ray-plane intersection)
  - Substitute and solve for t:

$$(\mathbf{p} + t\mathbf{d} - \mathbf{a}) \cdot \mathbf{n} = 0$$
$$t = \frac{(\mathbf{a} - \mathbf{p}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

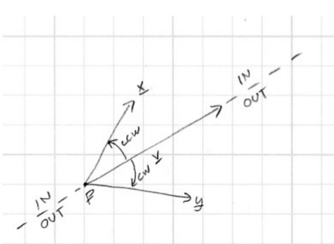
Ray-triangle intersection

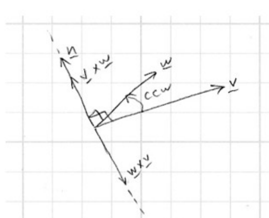
- In plane, triangle is the intersection of 3 half spaces



Inside-edge test

- Need outside vs. inside
- Reduce to clockwise vs. counterclockwise
  - Vector of edge to vector to x
- User cross product to decide





Ray-triangle intersection

$$(\mathbf{b} - \mathbf{a}) \times (\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} > 0$$
$$(\mathbf{c} - \mathbf{b}) \times (\mathbf{x} - \mathbf{b}) \cdot \mathbf{n} > 0$$
$$(\mathbf{a} - \mathbf{c}) \times (\mathbf{x} - \mathbf{c}) \cdot \mathbf{n} > 0$$

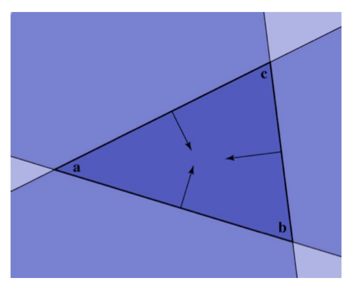
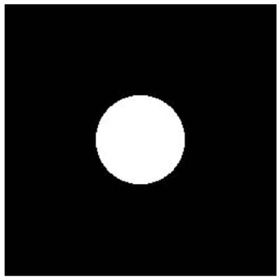


Image so far

- With eye ray generation and sphere intersection

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



Intersection against many shapes

- The basic idea is

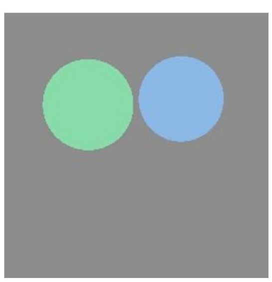
```
Group.intersect (ray, tMin, tMax) {
  tBest = +inf; firstSurface = null;
  for surface in surfaceList {
    hitSurface, t = surface.intersect(ray, tMin, tBest);
    if hitSurface is not null {
      tBest = t;
      firstSurface = hitSurface;
    }
  }
return hitSurface, tBest;
}
```

  - this is linear in the number of shapes but there are sublinear methods (acceleration structures)

Image so far

- With eye ray generation and scene intersection

```
for 0 <= iy < ny
  for 0 <= ix < nx {
    ray = camera.getRay(ix, iy);
    c = scene.trace(ray, 0, +inf);
    image.set(ix, iy, c);
  }
...
Scene.trace(ray, tMin, tMax) {
  surface, t = surfs.intersect(ray, tMin, tMax);
  if (surface != null) return surface.color();
  else return black;
}
```

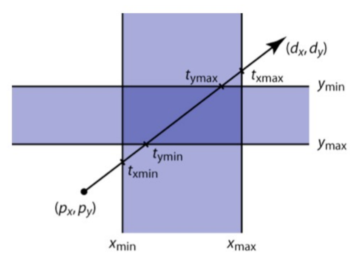


Shading

- 2D example
- 3D is the same!

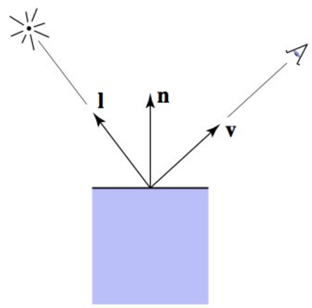
$$p_x + t_{x\min}\, d_x = x_{\min}$$
$$t_{x\min} = (x_{\min} - p_x)/d_x$$

$$p_y + t_{y\min}\, d_y = y_{\min}$$
$$t_{y\min} = (y_{\min} - p_y)/d_y$$



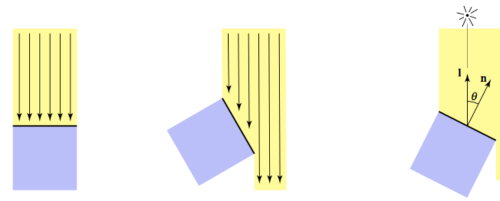
Shading

- Compute light reflected toward camera
- Inputs:
  - Eye direction
  - Light direction (for each of many lights)
  - Surface normal
  - Surface parameters (color, shininess,...)



Diffuse reflection

- Light is scattered uniformly in all directions
  - The surface color is the same for all viewing directions
- Lambert's cosine law



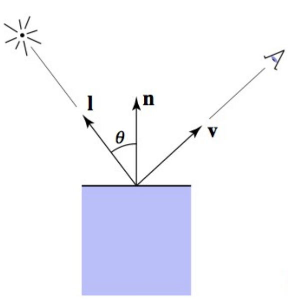
Top face of cube receives a certain amount of light

Top face of 60° rotated cube intercepts half the light

In general, light per unit area is proportional to cos θ = **l** • **n**

Lambertian shading

- Shading independent of view direction



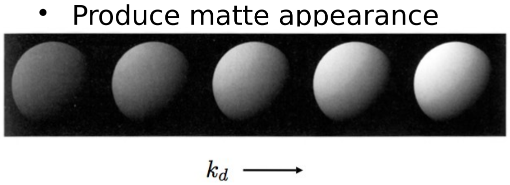
illumination from source

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

diffuse coefficient

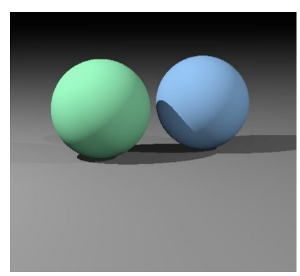
diffusely reflected light

- Produce matte appearance



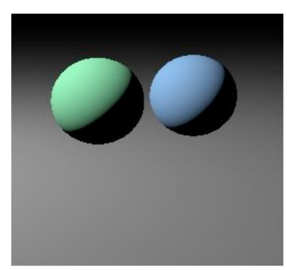
[Foley et al.]

Diffuse shading



- Image so far

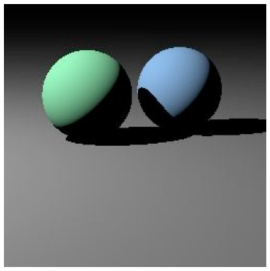
```
Scene.trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if surface is not null {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point,
            normal, light);
    }
    else return backgroundColor;
}

...

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



Shadows

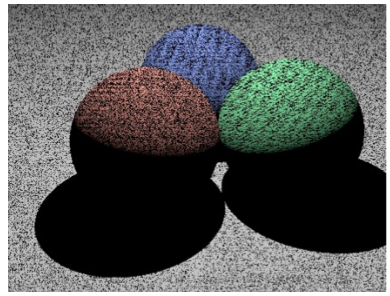
- Surface is only illuminated if nothing blocks its view of the light
- With ray tracing it is easy to check
  - Just intersect a ray with the scene
- Image so far

```
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;
}
```



Shadow rounding errors

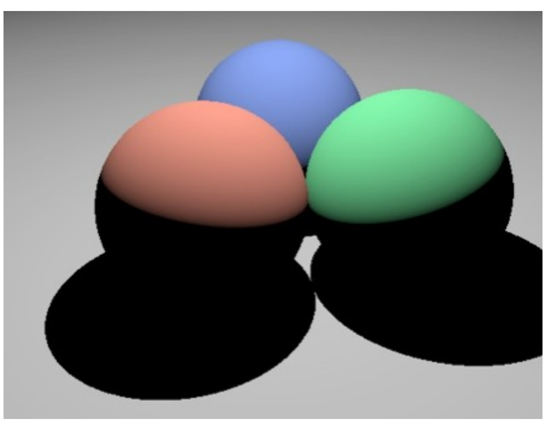
- Don’t fall victim to one of the classic blunders



- What is going on?
  - Hint: at what t does the shadow ray intersect the surface you’re shading

Shadow rounding errors

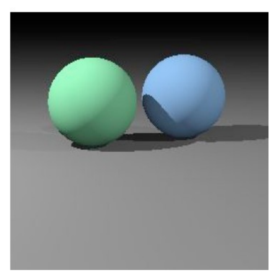
- Solution shadow rays start a tiny distance from the surface
- Do this by moving the start point, or by limiting the t range



Multiple lights

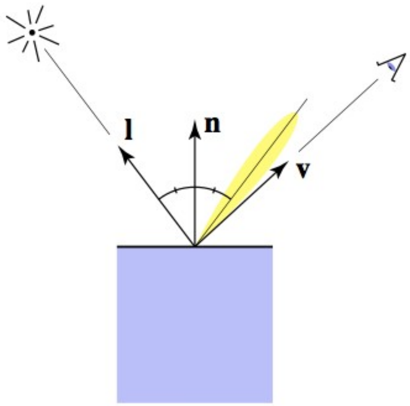
- Important to fill in black shadows
- Just loop over lights add contributions
- Ambient shading
  - Black shadows are not really right
  - One solution: dim light at camera
  - Alternative: add a constant “ambient” color to the shading…
- Image so far

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



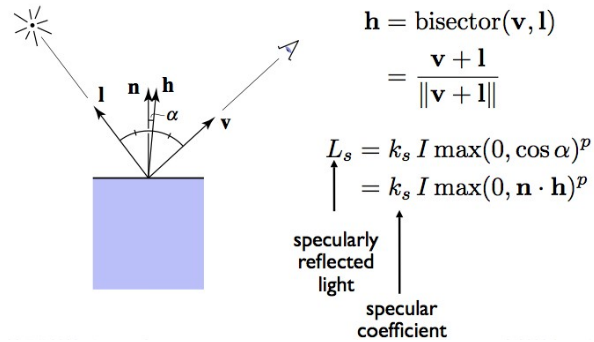
Specular shading (Blinn-Phong)

- Intensity depends on view direction
  - Bright near mirror configuration



Diffuse reflection

- Close to mirror ⇔ half vector near normal
  - Measure "near" by dot product of unit vectors



Phong model - plots

- Increasing n narrows the lobe

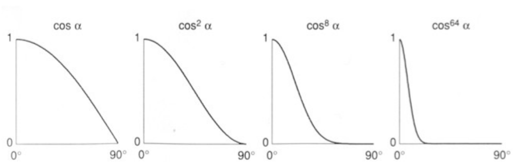
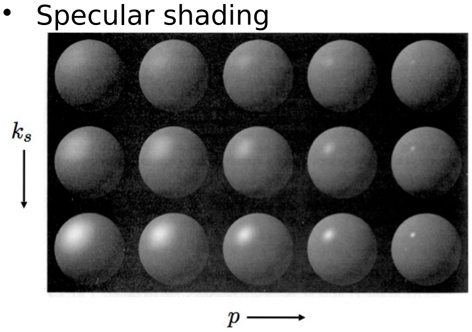


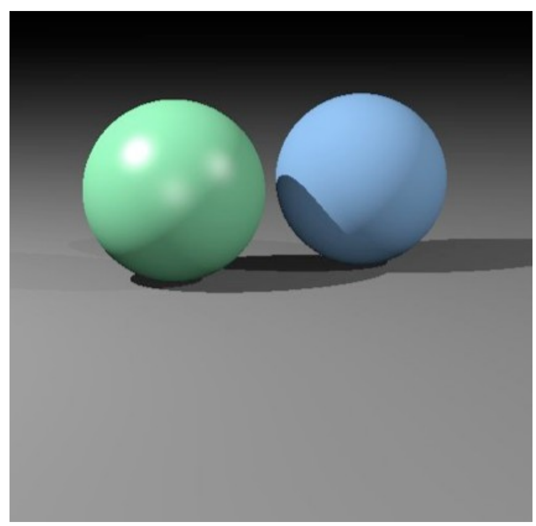
Fig. 16.9 Different values of $\cos^n\alpha$ used in the Phong illumination model.

[Foley et al.]

- Specular shading

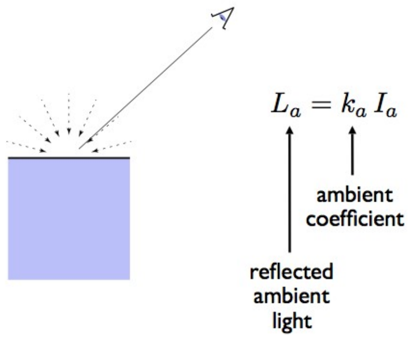


Diffuse + Phong shading



Ambient shading

- Shading that does not depend on anything
  - Add constant color to account for disregarded illumination and fill in black shadows



Putting it together

- Usually include ambient, diffuse, Phong in one model

$$\begin{aligned} L &= L_a + L_d + L_s \\ &= k_a I_a + k_d I \max(0, \mathbf{n}\cdot\mathbf{l}) + k_s I \max(0, \mathbf{n}\cdot\mathbf{h})^p \end{aligned}$$

- The final result is the sum over many lights

$$L = L_a + \sum_{i=1}^{N} [(L_d)_i + (L_s)_i]$$

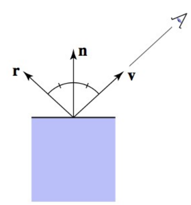
$$L = k_a I_a + \sum_{i=1}^{N} [k_d I_i \max(0, \mathbf{n}\cdot\mathbf{l}_i) + k_s I_i \max(0, \mathbf{n}\cdot\mathbf{h}_i)^p]$$

Mirror reflection

- Consider perfectly shiny surface
  - There is not a highlight
  - Instead there's a reflection of other objects
- Can render this using recursive ray tracing
  - To find out mirror reflection color, ask what color is seen from surface point in reflection direction
    - Already computing reflection direction for Phong…
  - "Glazed" material has mirror reflection and diffuse
  - L = La + Ld + Lm
  - Where Lm is evaluated by tracing a new ray

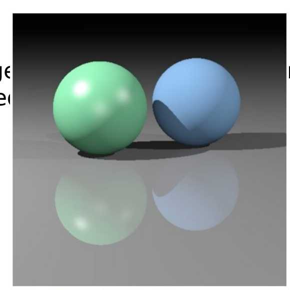
Mirror reflection

- Intensity depends on view direction
  - Reflects incident light from mirror direction



$$\begin{aligned} \mathbf{r} &= \mathbf{v} + 2((\mathbf{n}\cdot\mathbf{v})\mathbf{n} - \mathbf{v}) \\ &= 2(\mathbf{n}\cdot\mathbf{v})\mathbf{n} - \mathbf{v} \end{aligned}$$

- Image reflection – glaze



(glazed material on floor)

Ray tracer architecture 101

- You want a class called ray
  - Point and direction; evaluate (t)
  - Possible: tMin, tMax
- Some things can be intersected with rays
  - Individual surfaces
  - Groups of surfaces (acceleration goes here)
  - The whole scene
  - Make these all subclasses of surface
  - Limit the range of valid t values (e.g., shadow rays)
- Once you have the visible intersection, compute the color
  - may want to separate shading code from geometry
  - Separate class: material (each surface holds a reference to one)
  - Its job is to compute the color

Architectural practicalities

- Return values
  - Surface intersection tends to want to return multiple values
    - T, surface, normal vector; maybe surface point
  - Typical solution: an intersection record
    - A class with fields for all these things
    - Keep track of the intersection record for the closest intersection
    - Be careful of accidental aliasing
- Efficiency
  - What objects are created for every ray? Try to find a place for them where you can reuse them.
  - Shadow rays can be cheaper (any intersection will do, do not need closest)
  - But, “first get it right, then make it fast”