CSC 433/533 Computer Graphics

Alon Efrat
Credit: Joshua Levine

Lecture 11 Ray Tracing 3

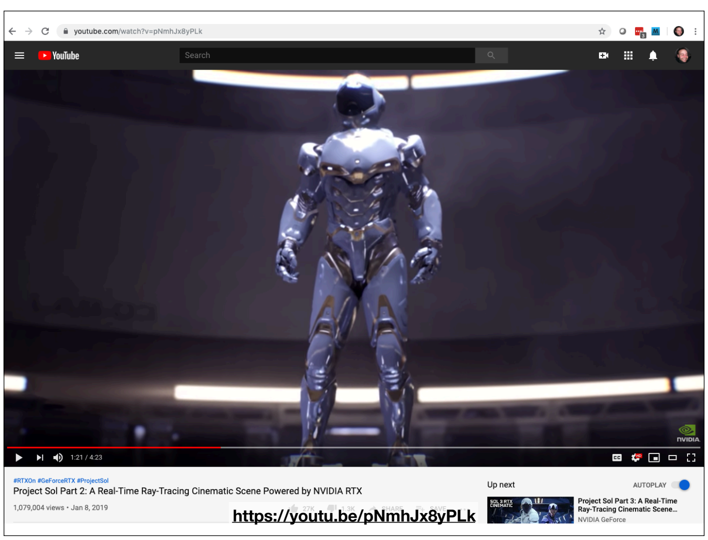
Oct. 2, 2019

Today's Agenda

- Reminders:
  - A03, questions?
- Goals for today:
  - Continue discussing Lighting and Shading

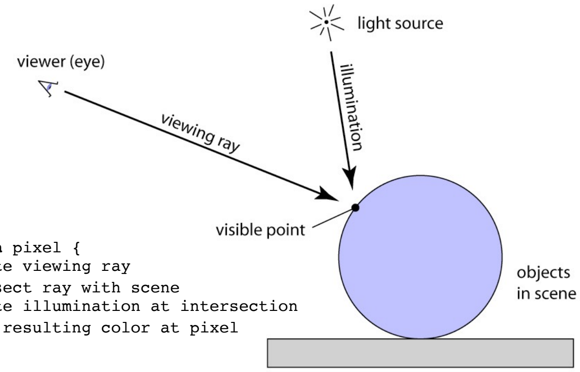
More Examples Of What Can Be Done With Ray Tracing

- https://developer.nvidia.com/optix
- https://embree.github.io/gallery.html



Last Time

Ray Tracing Algorithm




```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Ray-Sphere Intersection

- Two conditions must be satisfied:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be on a sphere: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0$
- Can substitute the equations and solve for t in $f(\mathbf{p}(t))$:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - R^2 = 0$$

- Solving for t is a quadratic equation

Ray-Plane Intersection

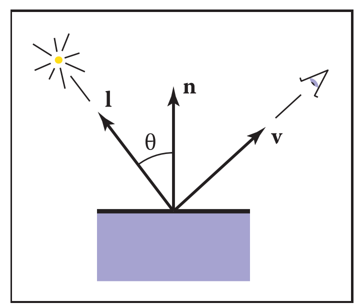
- Two conditions must be satisfied:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be on the plane: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$
- Can substitute the equations and solve for t in $f(\mathbf{p}(t))$:

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

- This means that $t = ((\mathbf{a} - \mathbf{o}) \cdot \mathbf{n}) / (\mathbf{d} \cdot \mathbf{n})$

Shading

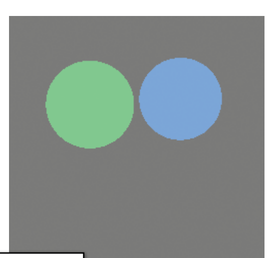
- Goal: Compute light reflected toward camera
- Inputs:
  - eye direction
  - light direction (for each of many lights)
  - surface normal
  - surface parameters (color, shininess, ...)



Images Without Shading

- With only eye-ray generation and scene intersection

```
for each pixel p in Image {
  let hit_surf = undefined;
  ...

  scene.surfaces.forEach( function(surf) {
    if (surf.intersect(eye, dir, ...)) {
      hit_surf = surf;
      ...
    }
  });

  c = hit_surf.ambient;
  Image.update(p, c);
}
```



Each surface storing a single ambient color

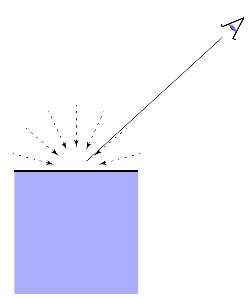
Ambient Shading

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

$$L_a = k_a I_a$$

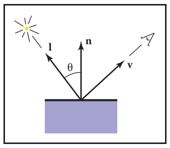
ambient coefficient (of surface)

ambient light intensity



Lambertian (Diffuse) Shading

- Simple model: amount of energy from a light source depends on the direction at which the light ray hits the surface
- Results in shading that is *view independent*

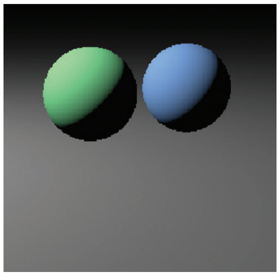


$$L_d = k_d I \max(0, \mathbf{n} \cdot \mathbf{l})$$

diffuse coefficient

intensity/color of light

cos θ



Blinn-Phong (Specular) Shading

- Symmetric arrangement captured by examining the half vector $\mathbf{h}$ between $\mathbf{v}$ and $\mathbf{l}$

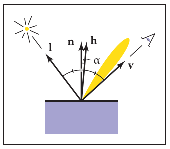
$$\mathbf{h} = (\mathbf{v} + \mathbf{l}) / \|\mathbf{v} + \mathbf{l}\|$$

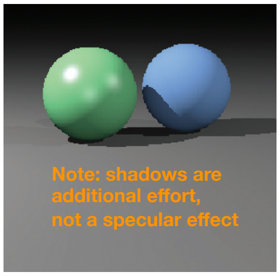
- When $\mathbf{n} \cdot \mathbf{h}$ is maximal, most reflection

$$L_s = k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

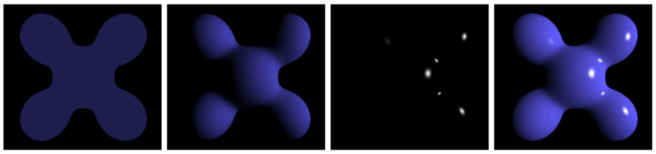
specular coefficient

Phong exponent





Blinn-Phong Decomposed



Ambient + Diffuse + Specular = Phong Reflection

https://en.wikipedia.org/wiki/Phong_shading

Simple Ray Tracer

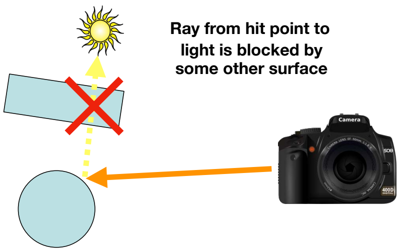
```
function ray_cast(eye, dir, near, far) {
  let hit_surf = undefined;   let hit_rec = undefined;
  let t_min = 0;   let hit_t = Infinity;
  let color = background;    //default background color

  scene.surfaces.forEach( function(surf) {
    let intersect_rec = surf.hit(eye, dir, t_min, hit_t);
    if (intersect_rec.hit) {
      hit_surf = surf;
      hit_t = intersect_rec.t;
      hit_rec = intersect_rec;
    }
  });

  if (hit_surf !== undefined) {
    color = hit_surf.kA * Ia;
    scene.lights.forEach( function(light) {
      //compute li, hi
      color = color + hit_surf.kD*Ii*max(0,n · li) + hit_surf.kS*Ii*max(0,n · hi)^p;
    });
  }

  return color;
}
```

```
for each pixel p in Image {
  let [eye, dir] = camera.compute_ray(p);
  let c = ray_cast(eye, dir, 0, Infinity);
  image.update(p, c);
}
```

Recursive Ray Tracing

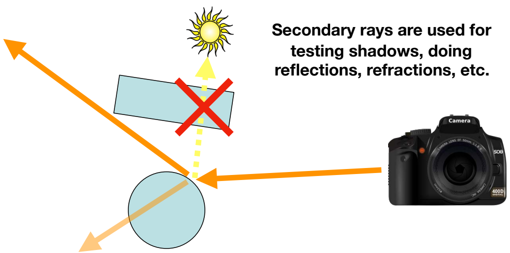
Shadows

- Surface should only be illuminated if nothing blocks the light from hitting the surface
- This can be easily checked by intersecting a new ray with the scene!



Ray Casting vs Ray Tracing

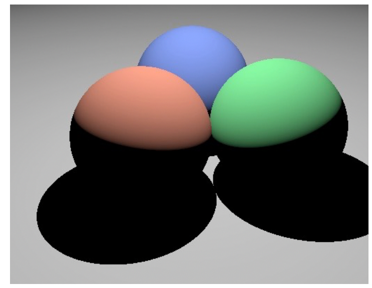
- Ray casting: tracing rays from eyes only
- Ray tracing: tracing secondary rays



Secondary rays are used for testing shadows, doing reflections, refractions, etc.

Shadows

- Idea: after finding the closest hit, cast a ray to each light source to determine if it is visible
- Be careful not to intersect with the object itself. Two solutions:
  - Only check for hits against all other surfaces
  - Start shadow rays a tiny distance away from the hit point by adjusting t_{min}



Ray Tracer w/ Shadows

```
function ray_cast(eye, dir, near, far) {
  ...
  //initialize color c;  compute hit_surf, hit_position;
  ...

  if (hit_surf !== undefined) {
    color = hit_surf.kA * Ia;
    scene.lights.forEach( function(light) {
      //compute l_i, h_i
      //check if light is visible from hit point
      if (light is visible) {
        color += effect of light;
      }
   });
  }

  return color;
}
```

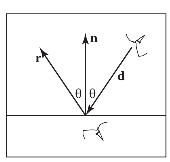
Ray Tracer w/ Shadows

```
function ray_cast(eye, dir, near, far) {
  ...
  //initialize color;  compute hit_surf, hit_pos;
  ...

  if (hit_surf !== undefined) {
    color = hit_surf.kA * Ia;
    scene.lights.forEach( function(light) {
      //compute l_i, h_i
      //check if light can is visible from hit point
      let shadow_hit = false;
      scene.surfaces.forEach( function(surf) {
        let intersect_rec = surf.hit(hit_pos, l_i, epsilon, ||hit_pos - light.pos||);
        if (intersect_rec.hit) {
          shadow_hit = true;
        }
      });

      if (shadow_hit == false) {
        color += hit_surf.kD*I_i*max(0,n · l_i) + hit_surf.kS*I_i*max(0,n · h_i)^p;
      }
   });
  }

  return color;
}
```

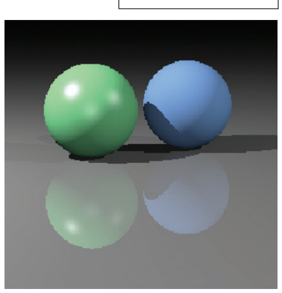
Offset shadow rays a small amount from surface

Cast ray no further than to the light position

Reflection



- Ideal **specular** reflection, or mirror reflection, can be modeled by casting another ray into the scene from the hit point
- Direction $\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$
- One can then recursively accumulate some amount of color from whatever object this hits
- color += k_m `*ray_cast()`



Recursive Ray Tracer

```
function ray_cast(eye, dir, near, far) {
  ...
  //initialize color;  compute hit_surf, hit_position;
  ...

  if (hit_surf is valid) {
    color = hit_surf.kA * Ia;
    scene.lights.forEach( function(light) {
      //compute l_i, h_i
      //check for shadow rays to decide if the light illuminates
      if (ray from hit_position in direction of l_i does not hit scene) {
        color += hit_surf.kD*I_i*max(0,n · l_i) + hit_surf.kS*I_i*max(0,n · h_i)^p;
      }
    });

    //compute reflect direction r_i
    //call ray_cast() recursively for mirror reflections
    color += hit_surf.kM * ray_cast(hit_position, r_i, epsilon, +inf);
  }

  return color;
}
```

Like w/ shadows, we need to offset a bit

Recursive!!

How much recursion? Typically, we use a max number of bounces

Mirror Reflection vs. Specular Reflection

- Consider perfectly shiny surface
  - Typically, no highlights! Usually, just reflections 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
- "Glazed" or "Glossy" materials often only have the mirror reflection + diffuse, which suggests

$$L = L_a + L_d + L_m$$

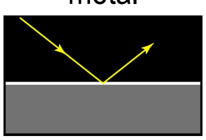
- Where L_m is evaluated by tracing a new ray and replaces L_s

Fancier Shading

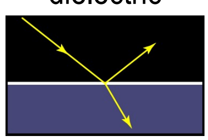
- While Phong has long been the heuristic baseline, newer methods are more based on physics:
  - When writing a shader, think like a bug standing on the surface
  - Bug sees an incident distribution of light that is arriving at the surface
  - Physics question: what is the outgoing distribution of light?

Simple materials

metal



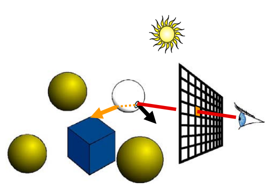
dielectric



Illuminating Dieletrics

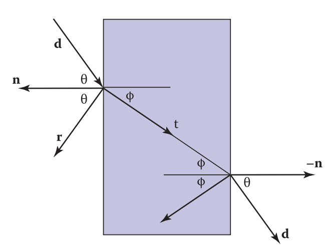
Translucency (Refraction)

- When a ray hits a dielectric surface, some portion of it transmits through the surface, but bends
- Color of the ray can be modulated by a refraction color



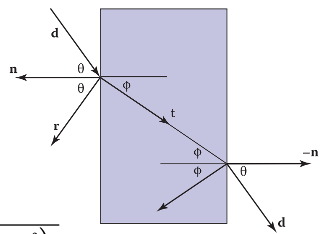
Snell's Law

- Governs the angle at which a refracted ray bends
- Computation based on refraction index of original medium, n, versus new index n_t
- $n_t \sin\theta = n \sin\phi$



Snell's Law

- Working with cosine's are easier because we can use dot products
- Can derive the vector for the refraction direction $\mathbf{t}$ as

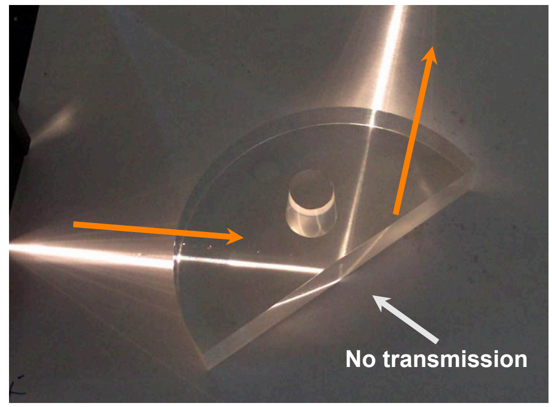


$$\mathbf{t} = \frac{n\left(\mathbf{d}+\mathbf{n}\cos\theta\right)}{n_t} - \mathbf{n}\cos\phi$$

$$= \frac{n\left(\mathbf{d}-\mathbf{n}(\mathbf{d}\cdot\mathbf{n})\right)}{n_t} - \mathbf{n}\sqrt{1 - \frac{n^2\left(1-(\mathbf{d}\cdot\mathbf{n})^2\right)}{n_t^2}}$$

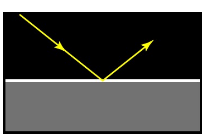
What happens if this is negative?

Total Internal Reflection

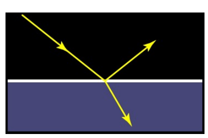


Specular reflection

- Smooth surfaces of pure materials have ideal specular reflection (said this before)
  - Metals (conductors) and dielectrics (insulators) behave differently
- Reflectance (fraction of light reflected) depends on angle



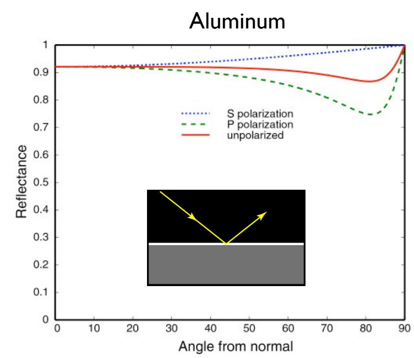
metal



dielectric

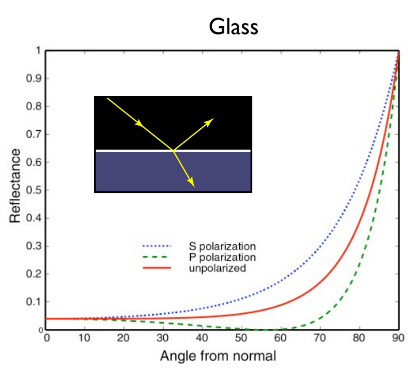
Specular Reflection from Metal

- Reflectance does depend on angle, but not much
- We typically ignore this for rendering



Specular Reflection from Dielectrics like Glass/Water

- Significant dependence on angle
- About 4% at normal incidence
- Nearly 100% at grazing (rest of the light is transmitted)
- Getting this right has a strong affect on appearance



Fresnel Equation Models the Reflectivity of Dielectrics

- Can be used to predict how much light reflects from a smooth interface (usually, one material is air/empty space)
- R is the fraction that is reflected
- (1-R) is the fraction that is transmitted

$$F_p = \frac{\eta_2\cos\theta_1 - \eta_1\cos\theta_2}{\eta_2\cos\theta_1 + \eta_1\cos\theta_2}$$

$$F_s = \frac{\eta_1\cos\theta_1 - \eta_2\cos\theta_2}{\eta_1\cos\theta_1 + \eta_2\cos\theta_2}$$

$$R = \frac{1}{2}\left(F_p^2 + F_s^2\right)$$

Lec12 Required Reading

- FOCG, Ch. 13