CSC 433/533
Computer Graphics

Alon Efrat
Credit: Joshua Levine

Lecture 10
Ray Tracing 2

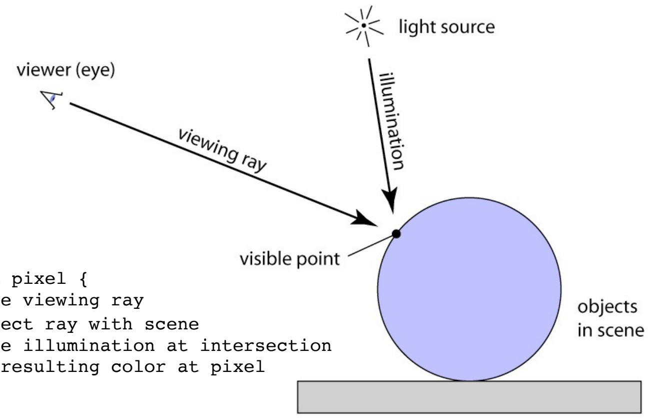
Sept. 30, 2019

Today's Agenda

- Reminders:
  - A03, questions?
- Goals for today:
  - Discuss shapes
  - Introduce lighting and shading

Last Time

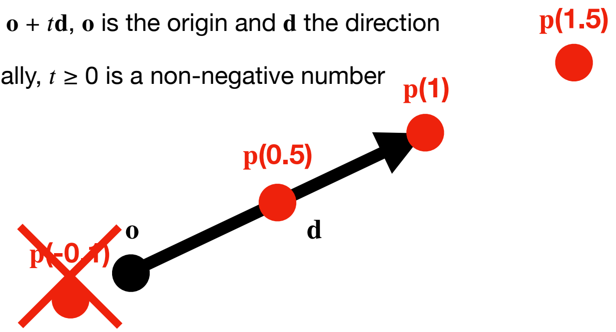
Ray Tracing Algorithm



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

Mathematical Description of a Ray

- Rays define a family of points, $\mathbf{p}(t)$, using a **parametric** definition
- $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$, $\mathbf{o}$ is the origin and $\mathbf{d}$ the direction
- Typically, $t \geq 0$ is a non-negative number



Intersecting Objects

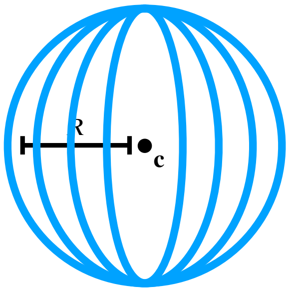
```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Defining a Sphere

- We can define a sphere of radius R, centered at position $\mathbf{c}$, using the implicit form

$$f(\mathbf{p}) = (\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0$$

- Any point $\mathbf{p}$ that satisfies the above lives on the sphere



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-Sphere Intersection

- Solve $(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - R^2 = 0$ for t:
- Rearrange terms:

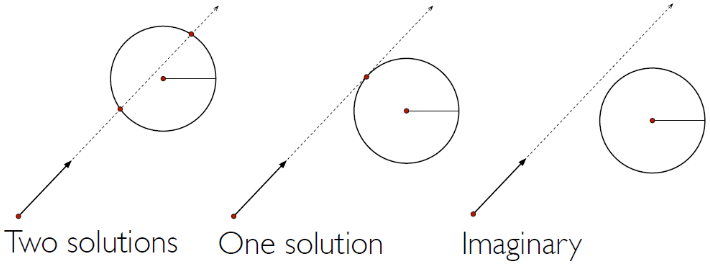
$$(\mathbf{d} \cdot \mathbf{d})t^2 + (2\mathbf{d} \cdot (\mathbf{o} - \mathbf{c}))t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - R^2 = 0$$

- Solve the quadratic equation $At^2 + Bt + C = 0$ where
  - $A = (\mathbf{d} \cdot \mathbf{d})$
  - $B = 2{*}\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})$
  - $C = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - R^2$

Discriminant, D = B^2-4*A*C
Solutions must satisfy:
t = (-B ± √(D)) / 2A

Ray-Sphere Intersection

- Number of intersections dictated by the discriminant
- In the case of two solutions, prefer the one with lower t



Geometric Method (instead of Algebraic)

Ray: $P = P_0 + tV$
Sphere: $IP - OI^2 - r^2 = 0$

Geometric Method

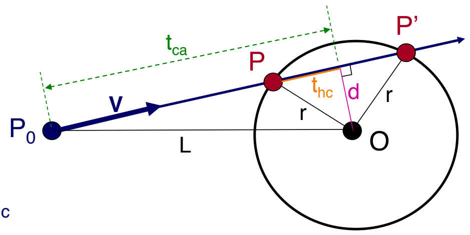
$L = O - P_0$

$t_{ca} = L \bullet V$
if ($t_{ca} < 0$) return 0

$d^2 = L \bullet L - t_{ca}^2$
if ($d^2 > r^2$) return 0

$t_{hc} = \mathrm{sqrt}(r^2 - d^2)$
$t = t_{ca} - t_{hc}$ and $t_{ca} + t_{hc}$

$P = P_0 + tV$

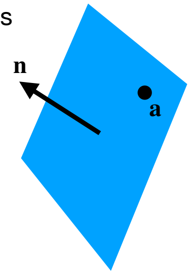


Defining a Plane

- A point $\mathbf{p}$ that satisfies the following implicit form lives on a plane through point $\mathbf{a}$ that has normal $\mathbf{n}$

$$f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$$

- $f(\mathbf{p}) > 0$ lives on the “front” side of the plane (in the direction pointed to by the normal
- $f(\mathbf{p}) < 0$ lives on the “back” side



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

From Planes to Triangles

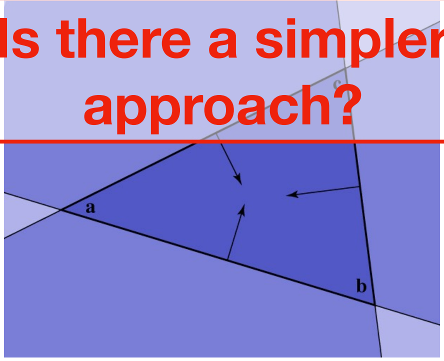
- Given 3 points $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ on the triangle, can we define the plane of it?
- Recall: a plane is defined by a point $\mathbf{a}$ and a normal $\mathbf{n}$
- How to define the normal?
  - $\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$

Ray-Triangle Intersection

- One approach is to satisfy 3 conditions:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be on the plane: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$
  - Must be inside the triangle! How?

Point In Triangle

- In plane, triangle is the intersection of 3 half spaces
- Can check that the point is on the same side of these half spaces (perhaps after a transformation)



Is there a simpler approach?

Barycentric Coordinates

- A coordinate system to write all points $\mathbf{p}$ as a weighted sum of the vertices

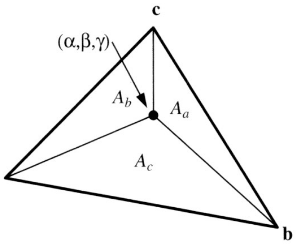
$$\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$$
$$\alpha + \beta + \gamma = 1$$

- Equivalently, α, β, γ are the proportions of area of subtriangles relative total area, A

$$A_a / A = \alpha$$
$$A_b / A = \beta$$
$$A_c / A = \gamma$$

- Triangle interior test:

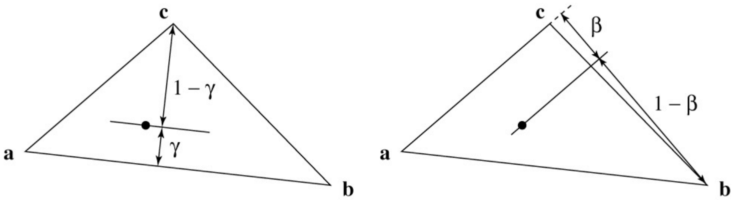
$$\alpha > 0,\ \beta > 0, \text{ and } \gamma > 0$$



[Shirley 2000]

Barycentric Coordinates

- Also related to distances



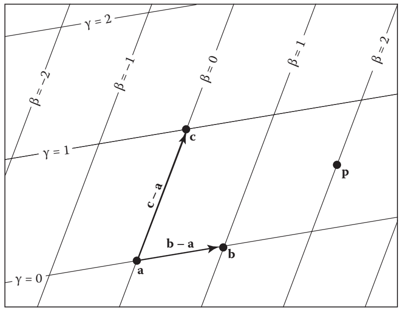
- And, they provide a basis relative to the edge vectors

$$\alpha = 1 - \beta - \gamma$$

$$\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

Barycentric Coordinates

- This basis defines the plane of the triangle



- In this view, the triangle interior test becomes:

$$\beta > 0,\ \gamma > 0,\ \beta + \gamma \leq 1$$

Barycentric Ray-Triangle Intersection

- Two conditions must be satisfied:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be in the triangle: $\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$
- So, set them equal and solve for t, β, γ:

$$\mathbf{o} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

- This is possible to solve because you have 3 equations and 3 unknowns

Barycentric Ray-Triangle Intersection

$$\mathbf{o} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

$$\beta(\mathbf{a} - \mathbf{b}) + \gamma(\mathbf{a} - \mathbf{c}) + t\mathbf{d} = \mathbf{a} - \mathbf{o}$$

$$\begin{bmatrix} \mathbf{a} - \mathbf{b} & \mathbf{a} - \mathbf{c} & \mathbf{d} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} \mathbf{a} - \mathbf{o} \end{bmatrix}$$

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_{\mathbf{o}} \\ y_a - y_{\mathbf{o}} \\ z_a - z_{\mathbf{o}} \end{bmatrix}$$

- Cramer's rule good fast way to solve this system
  (see Ch. 4 for details and the closed form expressions)

Generic Shapes

- Helpful to consider all types of objects from an abstract parent class of surfaces:

```
class Surface {
  ...
  intersect(eye, dir) {
    return {
      "t": t_min,
      "normal": undefined,
      "hit": false,
    };
  }
};
```

Ray to be intersected

Information about first intersection

Was there an intersection?

Note: Polymorphism in Javascript

- Similar to abstract base classes in Java except done at the function level:

```
class Surface {
  constructor(ambient) { ... }
  intersect(eye, dir) { ... }
};

class Sphere extends Surface {
  constructor(center, radius, ambient) {
    super(ambient);
    ...
  }
  intersect(eye, dir) {
    let hitrec = super.intersect(eye, dir);
    ...
  }
}
```

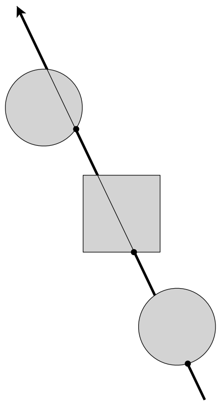
super keyword calls the function from the parent class

Generic Shapes

- Multiple subclasses can then extend and implement the same interface, filling in the details for the `intersect()` function

```
class Sphere extends Surface {
  ...
  intersect(eye, dir);
  ...
};


class Triangle extends Surface {
  ...
  intersect(eye, dir);
  ...
};
```

Intersection with Many Types of Shapes



- In a given scene, we also need to track which shape had the nearest hit point along the ray.
- This is easy to do by augmenting our interface to track a range of possible values for t, $[t_{\min}, t_{\max}]$:

```
intersect(eye, dir, t_min, t_max);
```

- After each intersection, we can then update the range

Intersection with Many Types of Shapes

```
for each pixel p in Image {
  let [eye, dir] = camera.compute_ray(p);
  let hit_surf = undefined;   let hit_rec = undefined;
  let t_min = 0;   let hit_t = Infinity;

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

  //Compute a color c
  image.update(p, c);
}
```

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

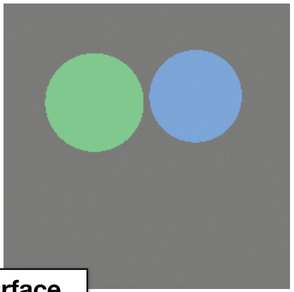
Illumination

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

Our images so far

- With only eye-ray generation and scene intersection

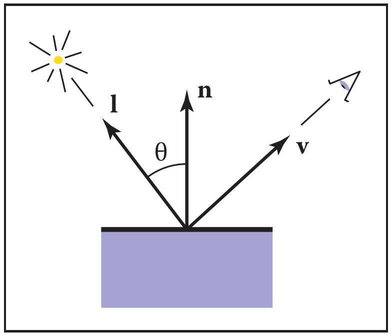
```
for each pixel p in Image {
  let hit_surf = undefined;
  ...

  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

Shading

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

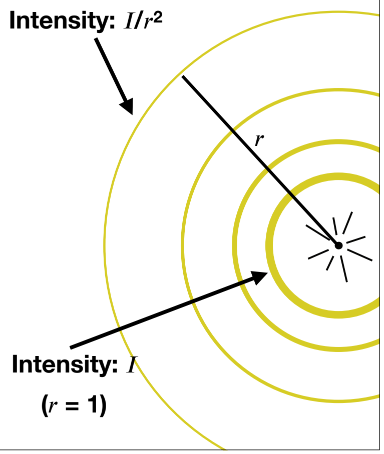


Normals

- The amount of light that reflects from a surface towards the eye depends on orientation of the surface at that point
- A **normal vector** describes the direction that is orthogonal to the surface at that point
- What are normal vectors for planes and triangles?
  - $\mathbf{n}$, the vector we already were storing!
- What are normal vectors for spheres?
  - Given a point $\mathbf{p}$ on the sphere $\mathbf{n} = (\mathbf{p} - \mathbf{c}) / \|\mathbf{p} - \mathbf{c}\|$

Light Sources

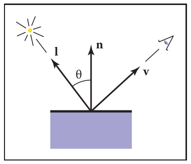
- There are many types of possible ways to model light, but for now we'll focus on **point lights**
- Point lights are defined by a position $\mathbf{p}$ that irradiates equally in all directions
- Technically, illumination from real point sources falls off relative to distance squared, but **<u>we will ignore this for now.</u>**



Shading Models

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*

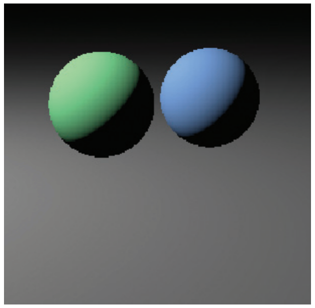


$$L_d = k_d I \max(0, \mathbf{n} \cdot \mathbf{l})$$

diffuse coefficient

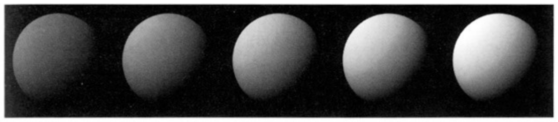
intensity/color of light

cos θ



Lambertian Shading

- k_d is a property of the surface itself
- Produces matte appearance of varying intensities

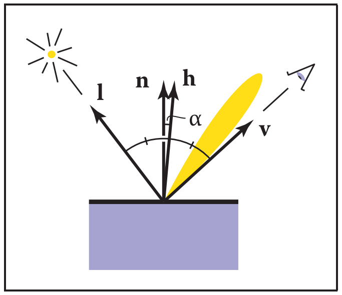


$k_d \longrightarrow$

[Foley et al.]

Blinn-Phong (Specular) Shading

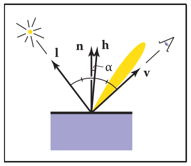
- Many real surfaces show some degree of shininess that produce specular reflections
- These effects move as the viewpoint changes
- Idea: produce reflection when $\mathbf{v}$ and $\mathbf{l}$ are symmetrically positioned across the surface normal



Blinn-Phong (Specular) Shading

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

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

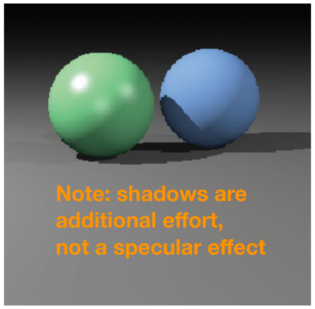


- When $\mathbf{n} \cdot \mathbf{h}$ is maximal, most reflection

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

specular coefficient

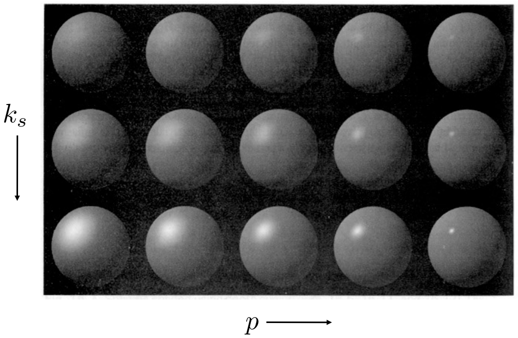
Phong exponent



Note: shadows are additional effort, not a specular effect

Blinn-Phong Shading

- Increasing p narrows the lobe
- This is kind of a hack, but it does look good



[Foley et al.]

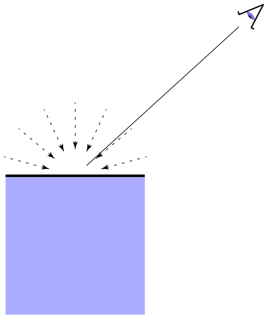
Ambient Shading

- 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



Putting it all together

- Usually include ambient, diffuse, and specular in one model

$$L = L_a + L_d + L_s$$

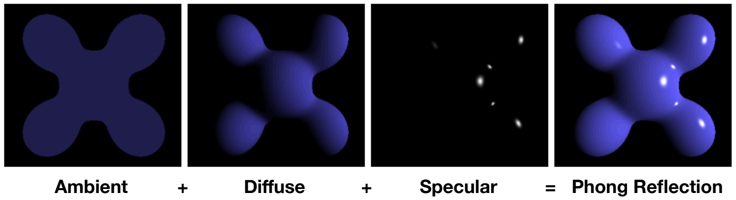
$$L = k_a I_a + k_d I \max(0,\mathbf{n} \cdot \mathbf{l}) + k_s I \max(0,\mathbf{n} \cdot \mathbf{h})^p$$

- And, the final result accumulates for all lights in the scene

$$L = k_a I_a + \Sigma_i \left(k_d I_i \max(0,\mathbf{n} \cdot \mathbf{l}_i) + k_s I_i \max(0,\mathbf{n} \cdot \mathbf{h}_i)^p \right)$$

- Be careful of overflowing! You may need to clamp colors, especially if there are many lights.

Blinn-Phong Decomposed



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 l_i, h_i
      color = color + hit_surf.kD*I_i*max(0,n · l_i) + hit_surf.kS*I_i*max(0,n · h_i)^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);
}
```

Lec11 Required Reading

- FOCG, Ch. 4, 10