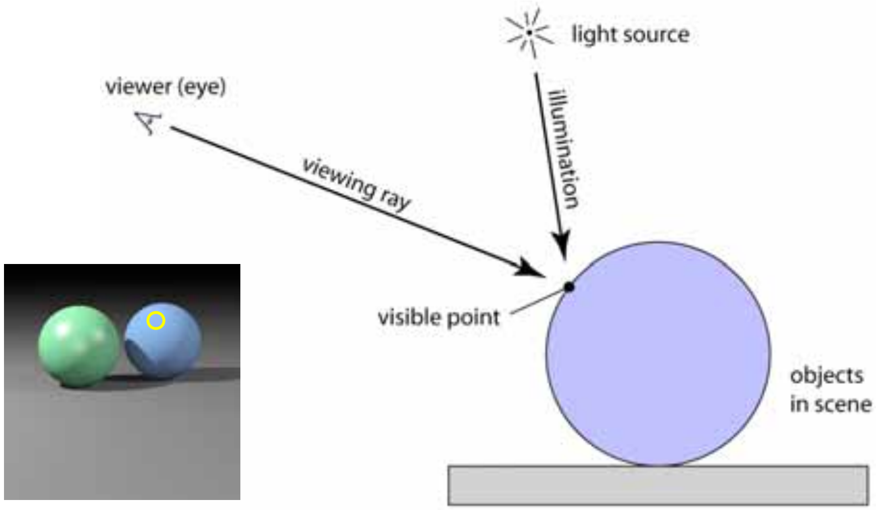
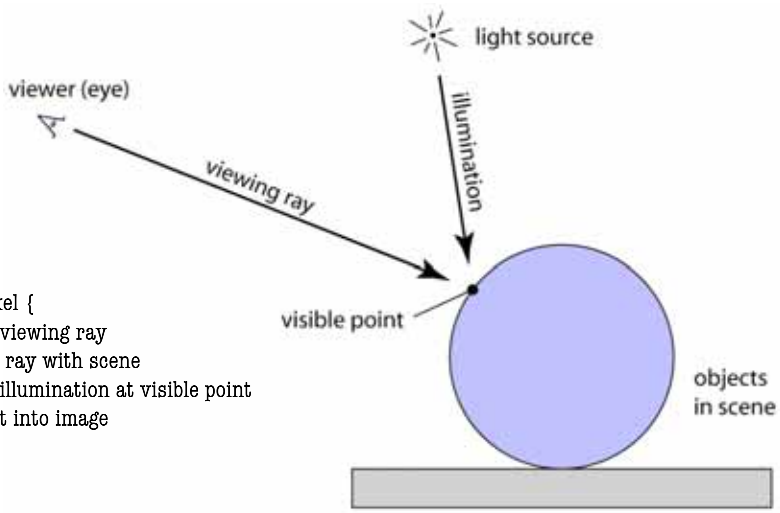
Ray Tracing

CS 465 Lecture 3

Ray tracing idea

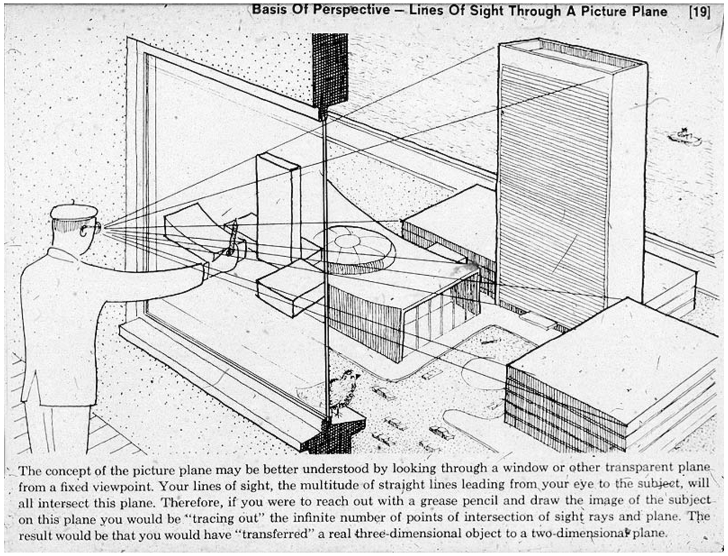


Ray tracing algorithm




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

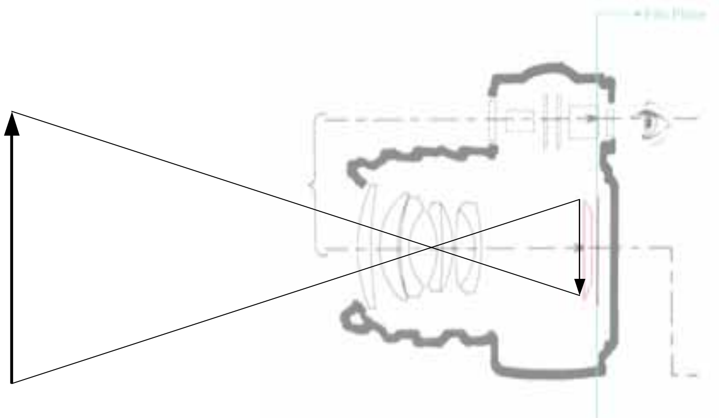

Plane projection in drawing



[CS 417 Spring 2002]

Plane projection in photography

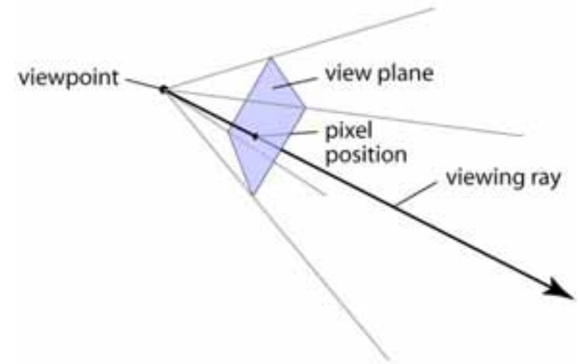
- This is another model for what we are doing
  - applies more directly in realistic rendering



[CS 417 Spring 2002]

Generating eye rays

- Use window analogy directly



Vector math review

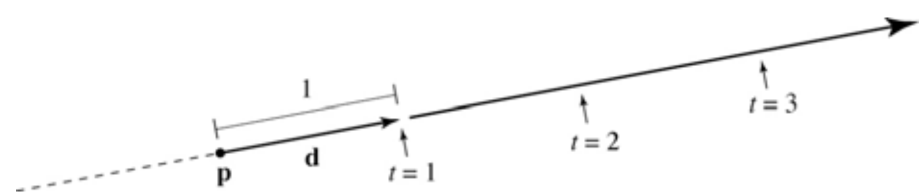
- Vectors and points
- Vector operations
  - addition
  - scalar product
- More products
  - dot product
  - cross product

Ray: a half line

- Standard representation: point **p** and direction **d**

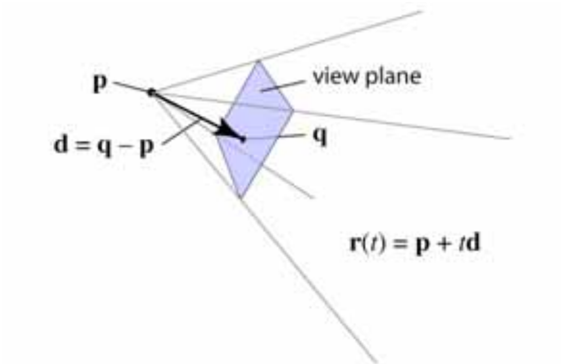
  $$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

  - 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 **d** with a**d** doesn't change ray ($a > 0$)



Generating eye rays

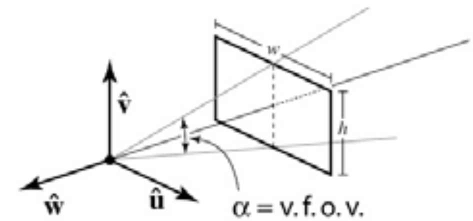
- Just need to compute the view plane point **q**:



  - but where exactly is the view rectangle?

Generating eye rays

- Positioning the view rectangle
  - lots of ways to do this; here is one
  - center is 1 unit away in the forward direction
  - size is *w* by *h* (more on *w* and *h* in a moment)
  - orientation?
  - establish three vectors to be *camera basis*



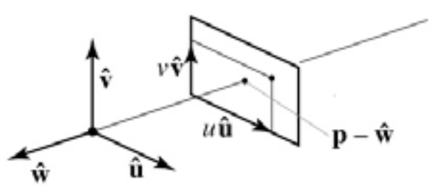
Generating rays in camera basis

- Compute image plane points using **u**, **v**, **w**
  - View rect. center is $\mathbf{p} - \mathbf{w}$
  - Lower left of view rect:

$$\mathbf{p} - \mathbf{w} - \frac{1}{2} w\,\mathbf{u} - \frac{1}{2} h\,\mathbf{v}$$

  - Upper right of view rect:

$$\mathbf{p} - \mathbf{w} + \frac{1}{2} w\,\mathbf{u} + \frac{1}{2} h\,\mathbf{v}$$



  - Point at position (*u*, *v*):

$$\mathbf{p} - \mathbf{w} + (u - \frac{1}{2}) w\,\mathbf{u} + (v - \frac{1}{2}) h\,\mathbf{v}$$

Ray-sphere intersection: algebraic

- Condition 1: point is on ray

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

- Condition 2: point is on sphere
  - assume unit sphere; see Shirley or notes for general

$$\|\mathbf{x}\| = 1 \Leftrightarrow \|\mathbf{x}\|^2 = 1$$

$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{x} - 1 = 0$$

- Substitute:

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

  - this is a quadratic equation in *t*

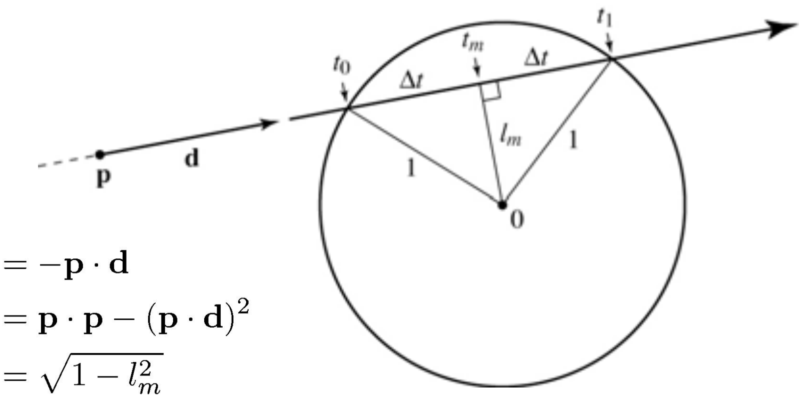
Ray-sphere intersection: algebraic

- Solution for *t* by quadratic formula:

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

  - simpler form holds when **d** is a unit vector but we won't assume this in practice (reason later)
  - I'll use the unit-vector form to make the geometric interpretation

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-triangle intersection

- Condition 1: point is on ray

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

- Condition 2: point is on plane

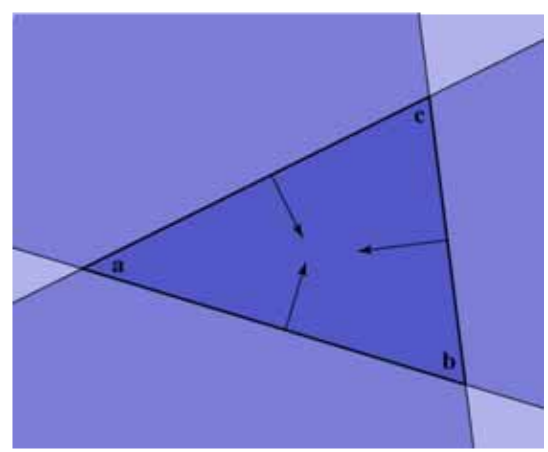
$$(\mathbf{x} - \mathbf{a})\cdot\mathbf{n} = 0$$

- Condition 3: point is on the inside of all three edges
- First solve 1&2 (ray–plane intersection)
  - substitute and solve for *t*:

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

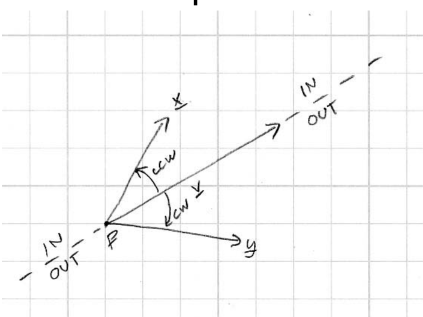
Ray-triangle intersection

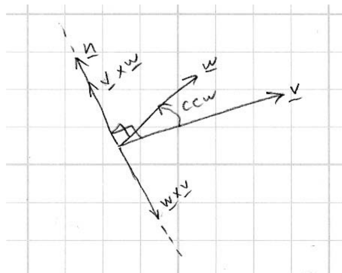
- In plane, triangle is the intersection of 3 half spaces



Inside-edge test

- Need outside vs. inside
- Reduce to clockwise vs. counterclockwise
  – vector of edge to vector to **x**
- Use cross product to decide





Ray-triangle intersection

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

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

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

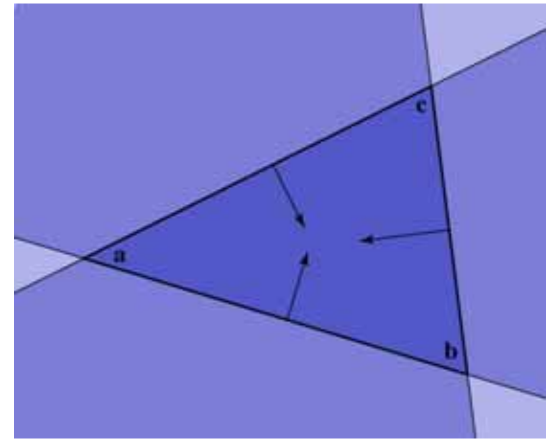
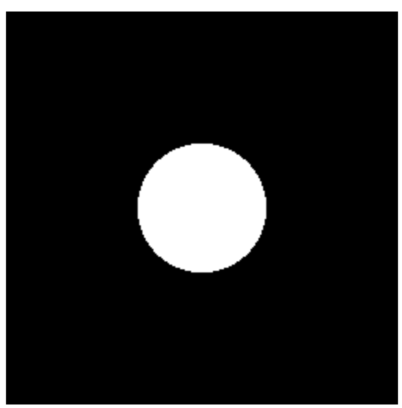


Image so far

- 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);
    if (s.intersect(ray, 0, +inf) < +inf)
      image.set(ix, iy, white);
  }
```



Intersection against many shapes

- The basic idea is:

```
hit (ray, tMin, tMax) {
  tBest = +inf; hitSurface = null;
  for surface in surfaceList {
    t = surface.intersect(ray, tMin, tMax);
    if t < tBest {
      tBest = t;
      hitSurface = surface;
    }
  }
return hitSurface, t;
}
```

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

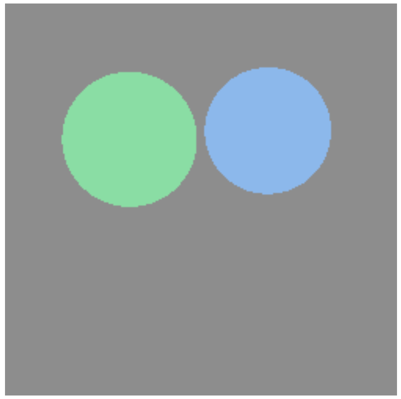
Image so far

- With eye ray generation and scene intersection

```
Geometry g = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        c = scene.trace(ray, 0, +inf);
        image.set(ix, iy, c);
    }

...

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



Shading

- Compute light reflected toward camera
- Inputs:
  - eye direction
  - light direction (for each of many lights)
  - surface normal
  - surface parameters (color, shininess, …)
- More on this in the next lecture

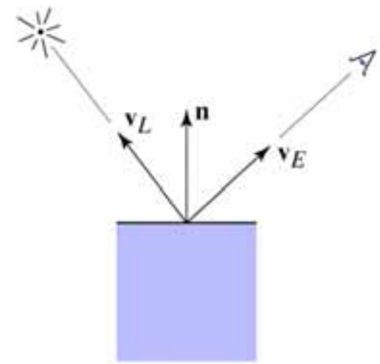
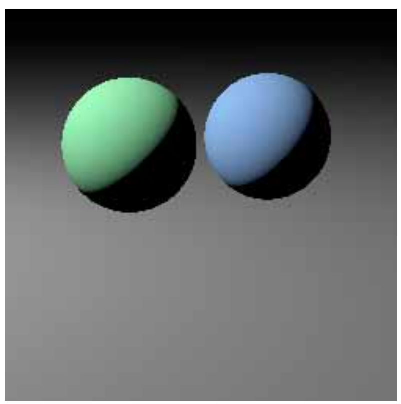


Image so far

```
trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if (surface != null) {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point,
            normal, light);
    }
    else return black;
}

...

shade(ray, point, normal, light) {
    v_E = –normalize(ray.direction);
    v_L = normalize(light.pos - point);
    // compute shading
}
```

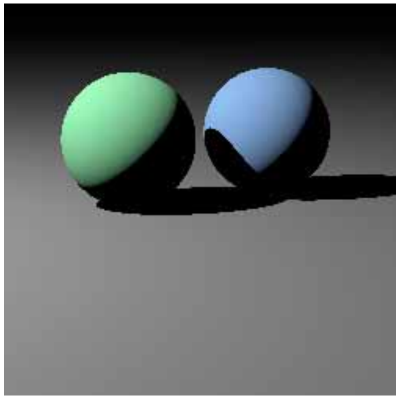


Shadows

- 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

```
shade(ray, point, normal, light) {
    shadRay = (point, light.pos - point);
    if (shadRay not blocked) {
        v_E = −normalize(ray.direction);
        v_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: all surface receive a bit more light
    - just 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;
}
```