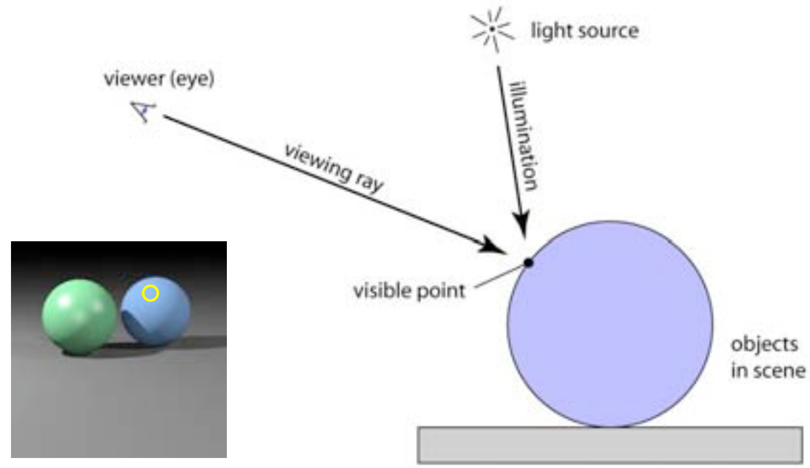
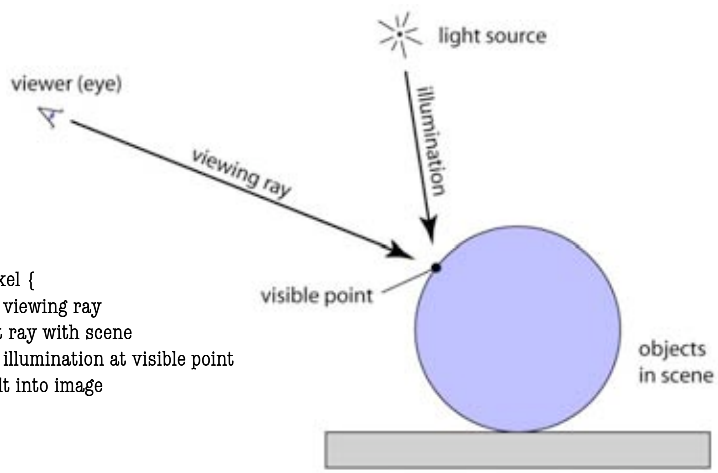
Ray Tracing

CS 465 Lecture 3

Ray tracing idea

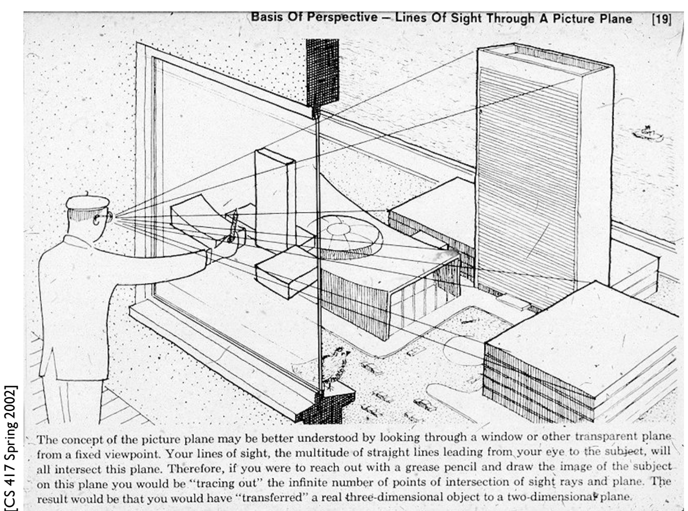


Ray tracing algorithm




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

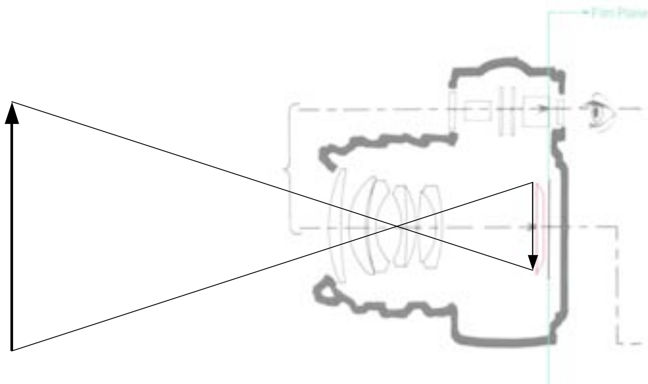

Plane projection in drawing



[CS 417 Spring 2002]

Plane projection in photography

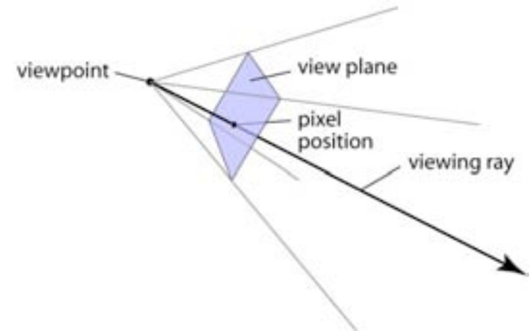
- This is another model for what we are doing
  - applies more directly in realistic rendering



[CS 417 Spring 2002]

Generating eye rays

- Use window analogy directly



Vector math review

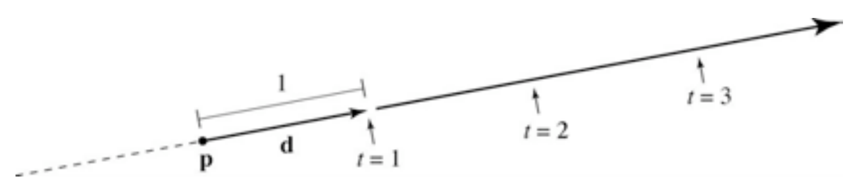
- Vectors and points
- Vector operations
  - addition
  - scalar product
- More products
  - dot product
  - cross product
- Bases and orthogonality

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 ($a > 0$)



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*

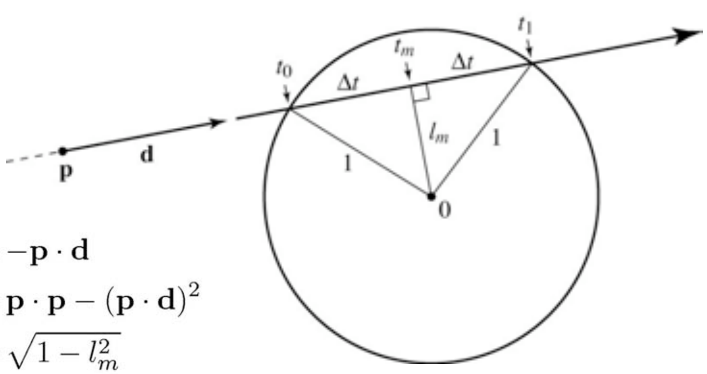
Ray-sphere intersection: algebraic

- 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

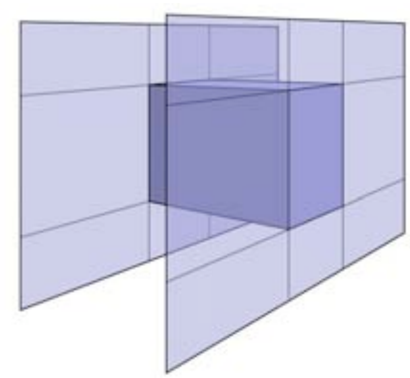
Ray-sphere intersection: geometric



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

Ray-box intersection

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

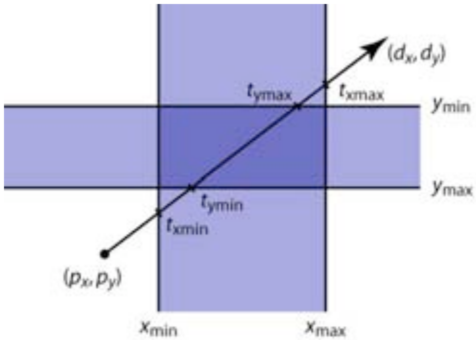


Ray-slab intersection

- 2D example
- 3D is the same!

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

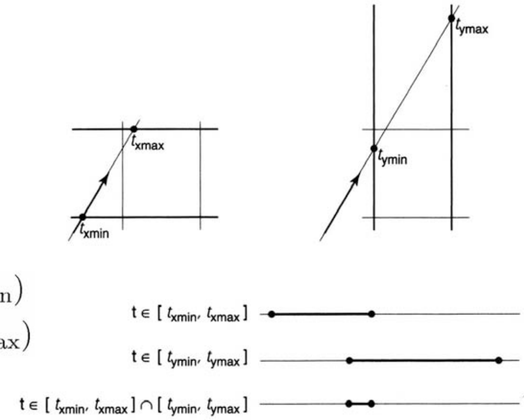
$$p_y + t_{y\min}\, d_y = y_{\min}$$
$$t_{y\min} = (y_{\min} - p_y)/d_y$$



Intersecting intersections

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

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

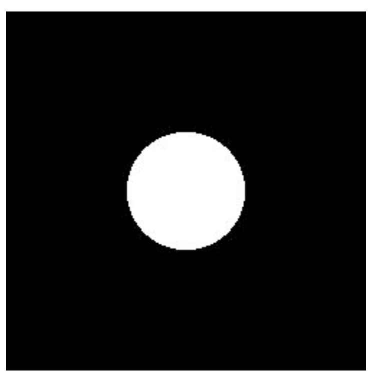


Shirley fig. 10.16

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)

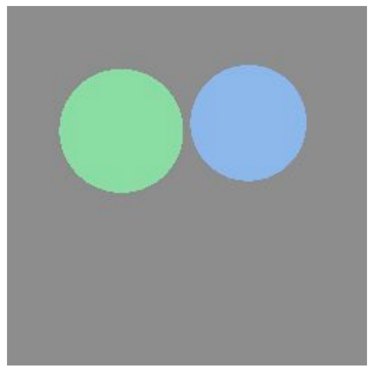
Image so far

- With eye ray generation and scene intersection

```
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

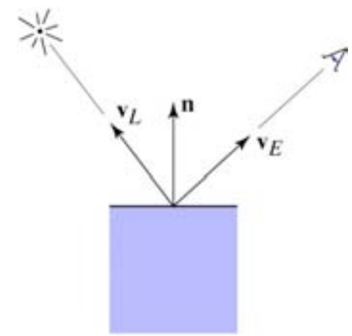
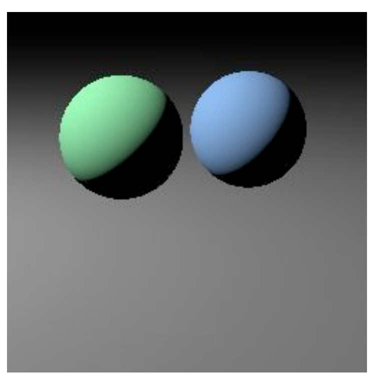


Image so far

```
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
}
```

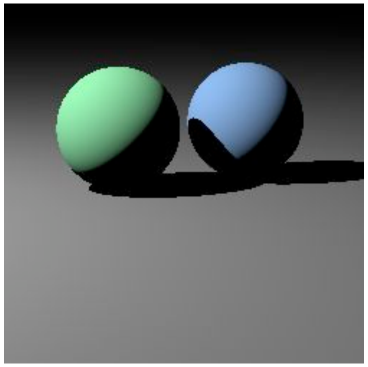


Shadows

- 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!

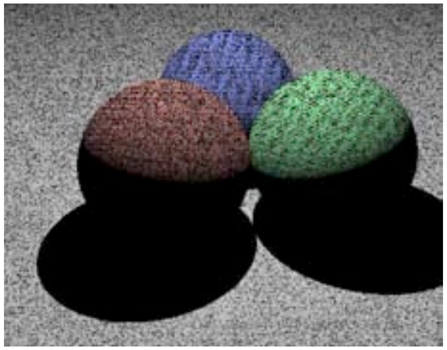
Image so far

```
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;
}
```



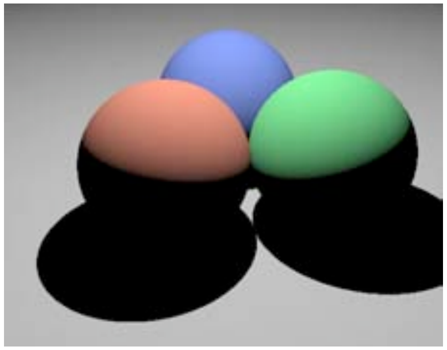
Shadow rounding errors

- Don't fall victim to one of the classic blunders:



- What's going on?
  - hint: at what *t* does the shadow ray intersect the surface you're shading?

- Solution: shadow rays start a tiny distance from the surface



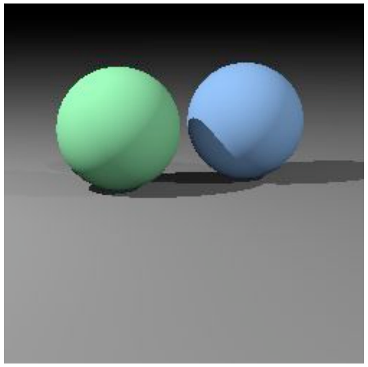
- Do this by moving the start point, or by limiting the *t* range

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: 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;
}
```



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
  - the whole scene
  - often need to be able to limit the range (*e.g.* shadow rays)
- Once you have the visible intersection, compute the color
  - this is an object that's associated with the object you hit
  - its job is to compute the color

Architectural practicalities

- Return values
  - surface intersection tends to want to return multiple values
    - *t*, surface or shader, normal vector, maybe surface point
  - in many programming languages (*e.g.* Java) this is a pain
  - 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 (which is very easy if you're new to Java)
- Efficiency
  - in Java the (or, a) key to being fast is to minimize creation of objects
  - 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, don't need closest)
  - but: "Get it Right, Then Make it Fast"