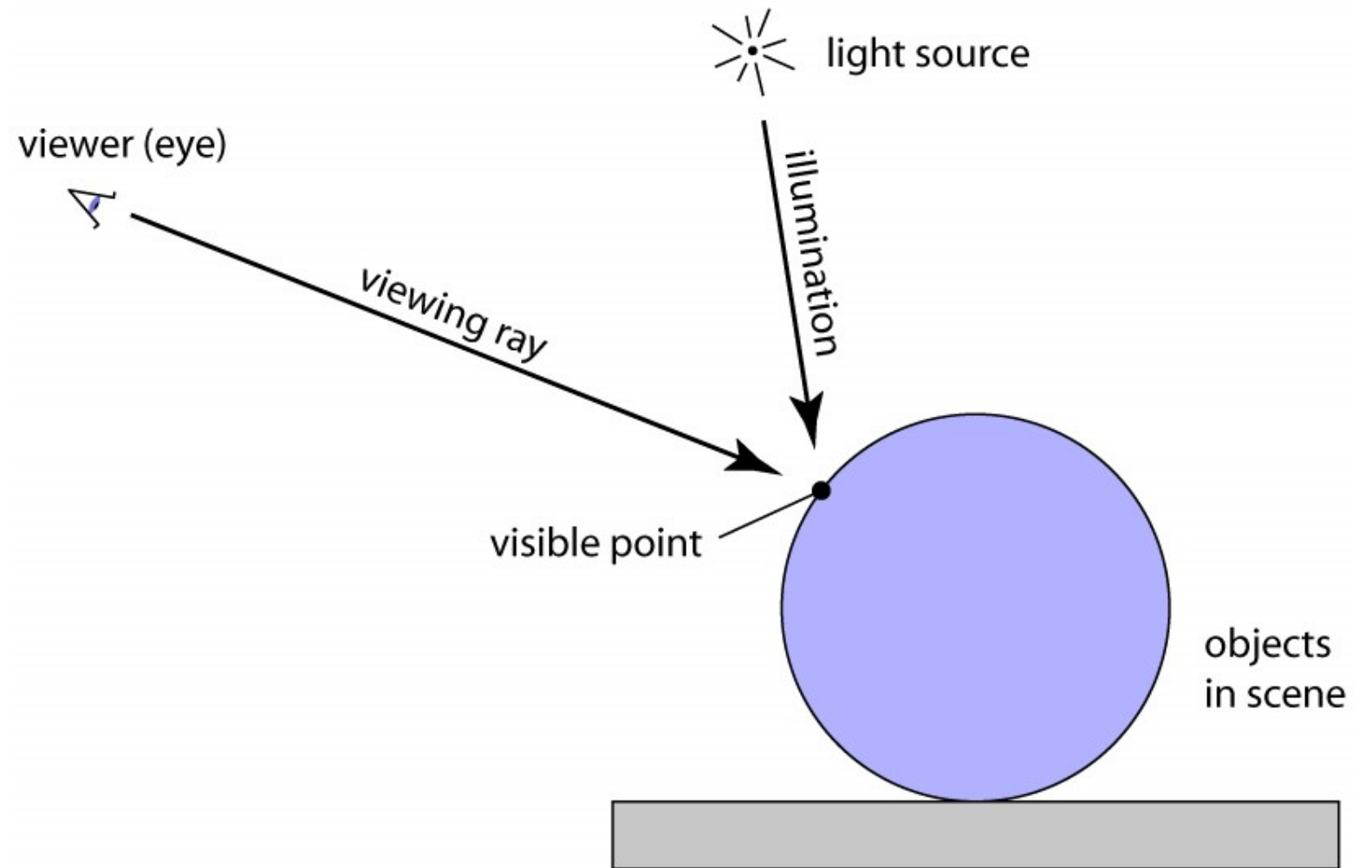
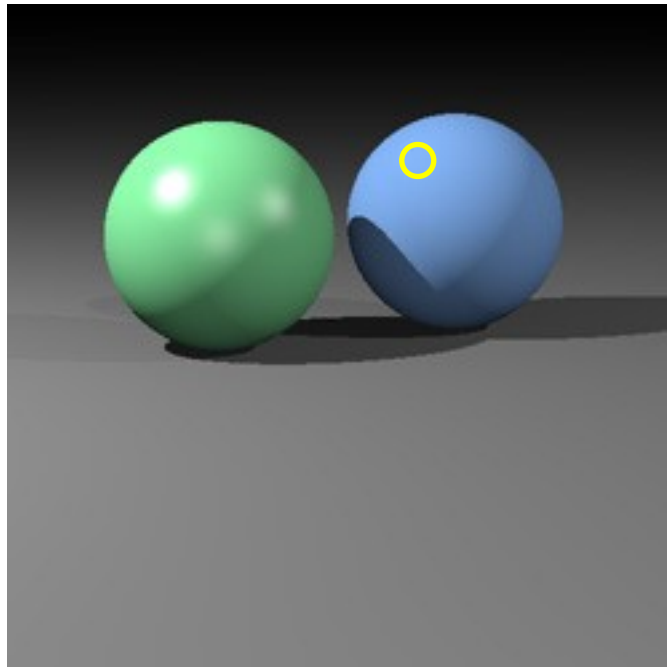


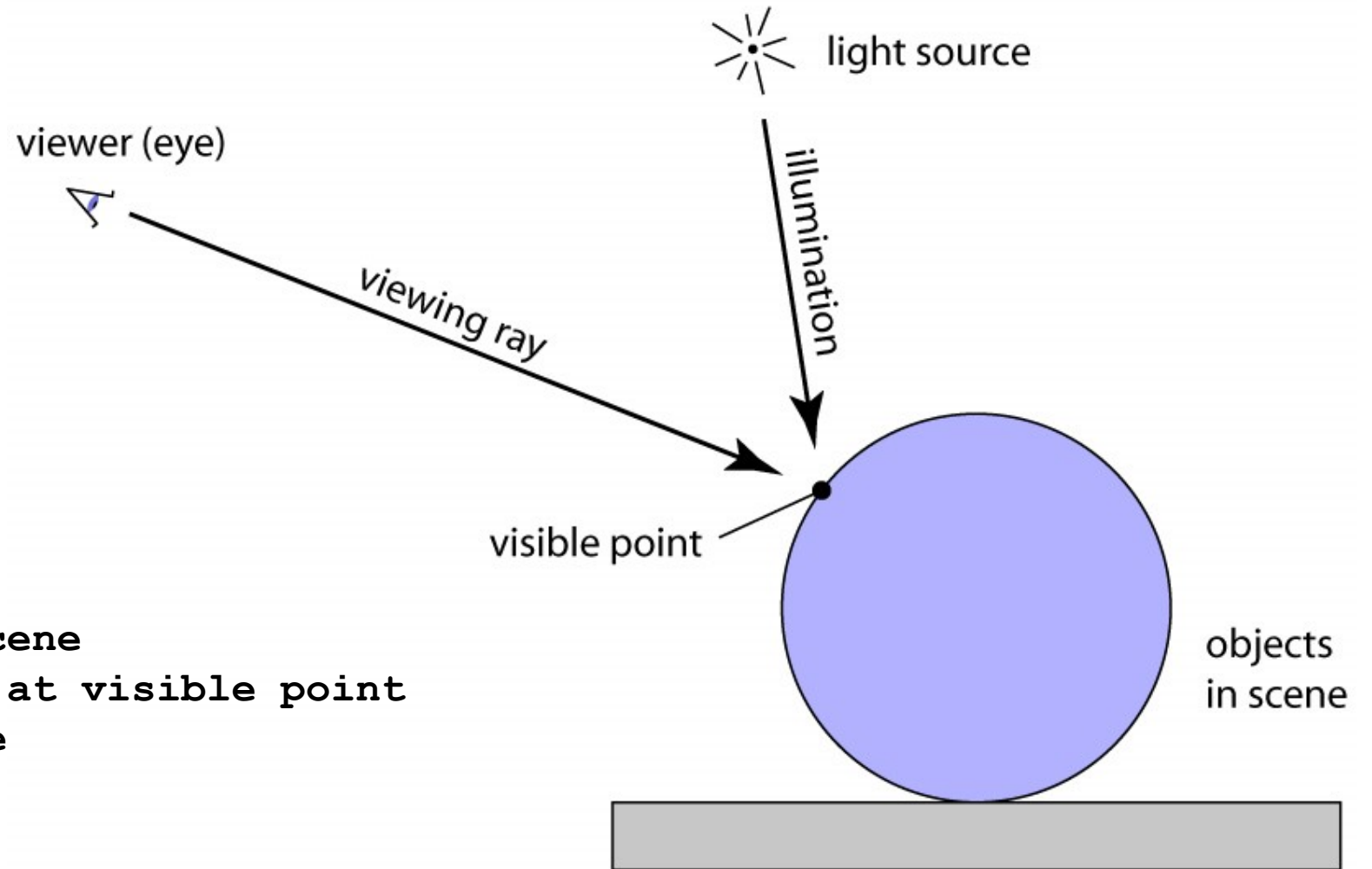
Ray Tracing

Ray tracing idea



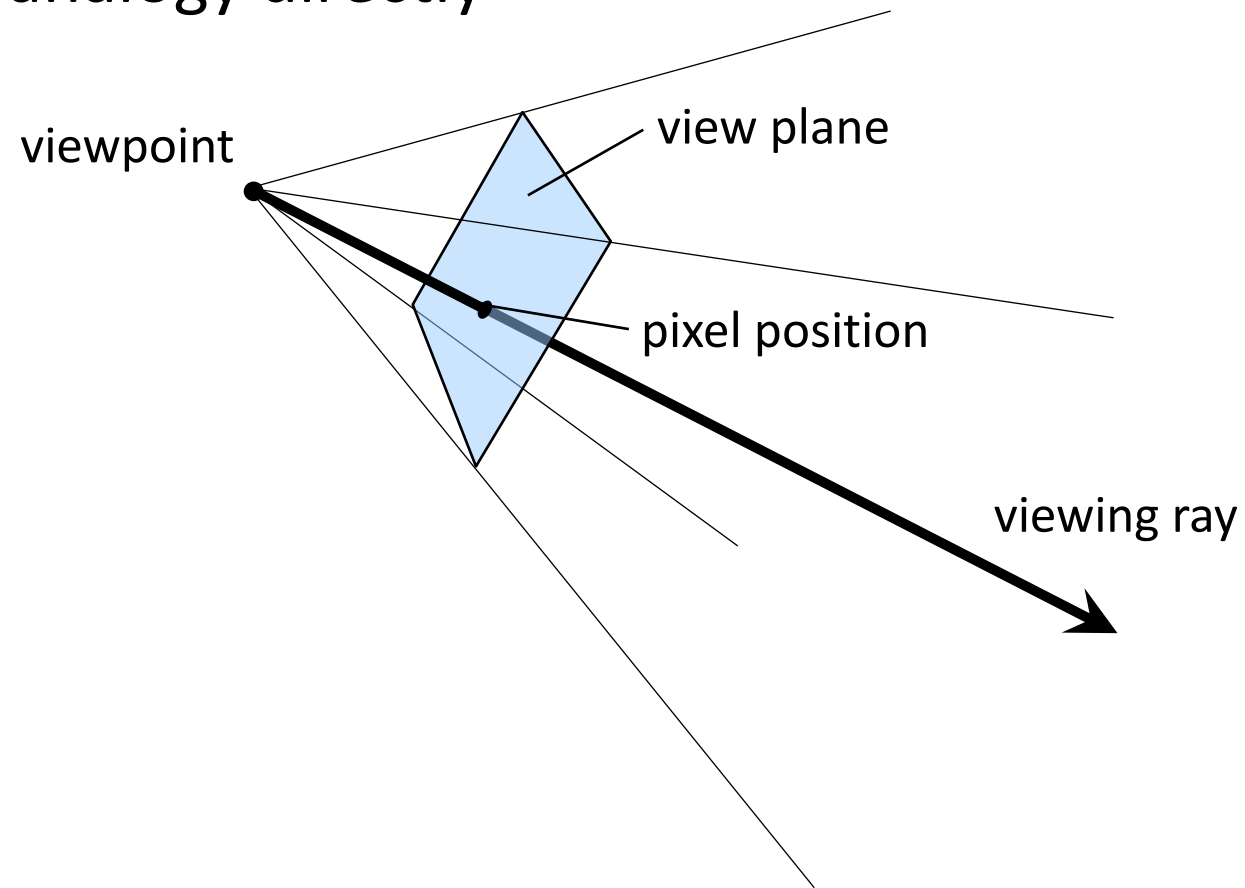
Ray tracing algorithm

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



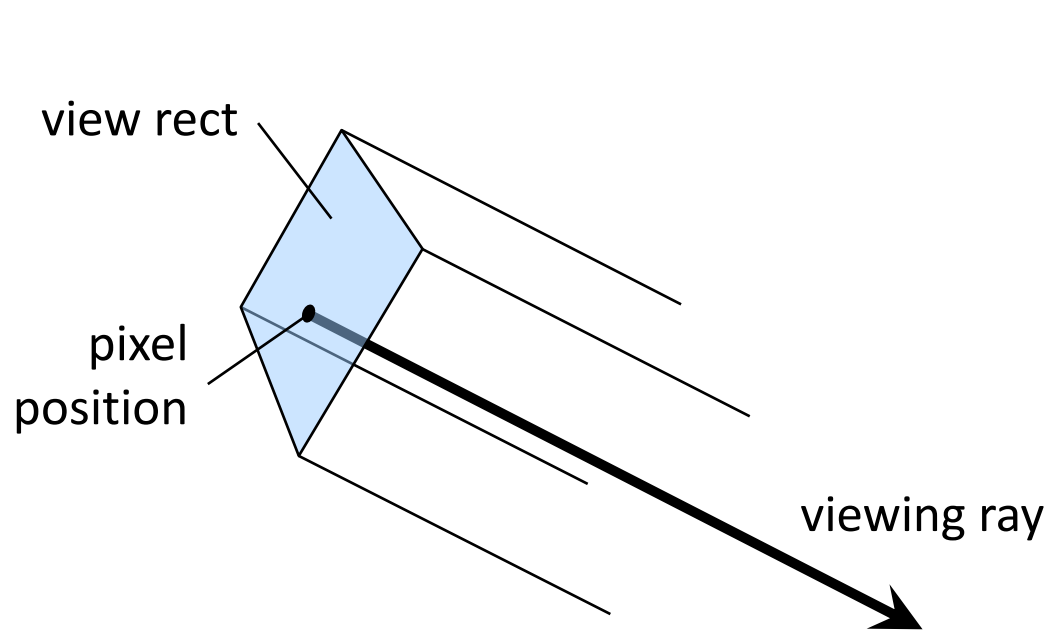
Generating eye rays

- Use window analogy directly

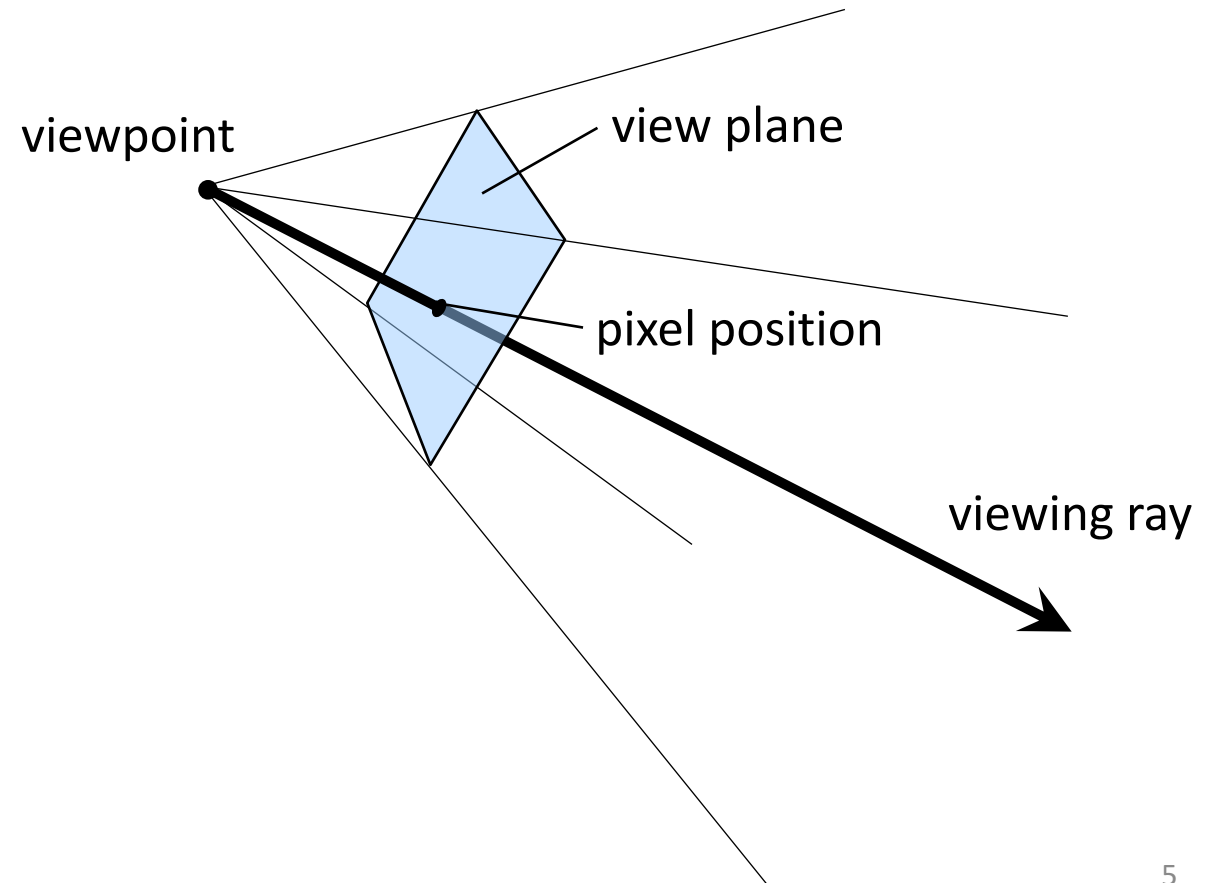


Generating eye rays

ORTHOGRAPHIC

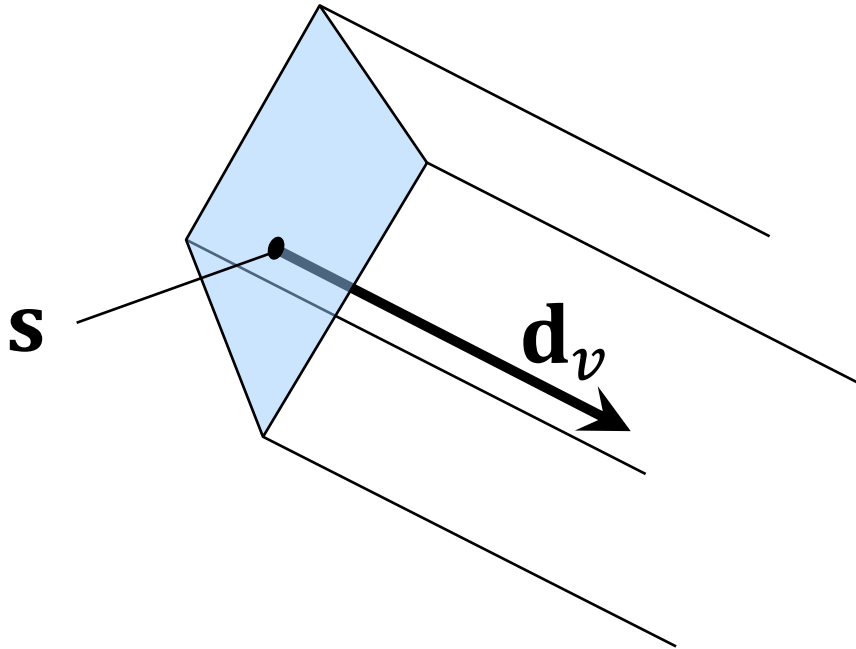


PERSPECTIVE



Generating eye rays—orthographic

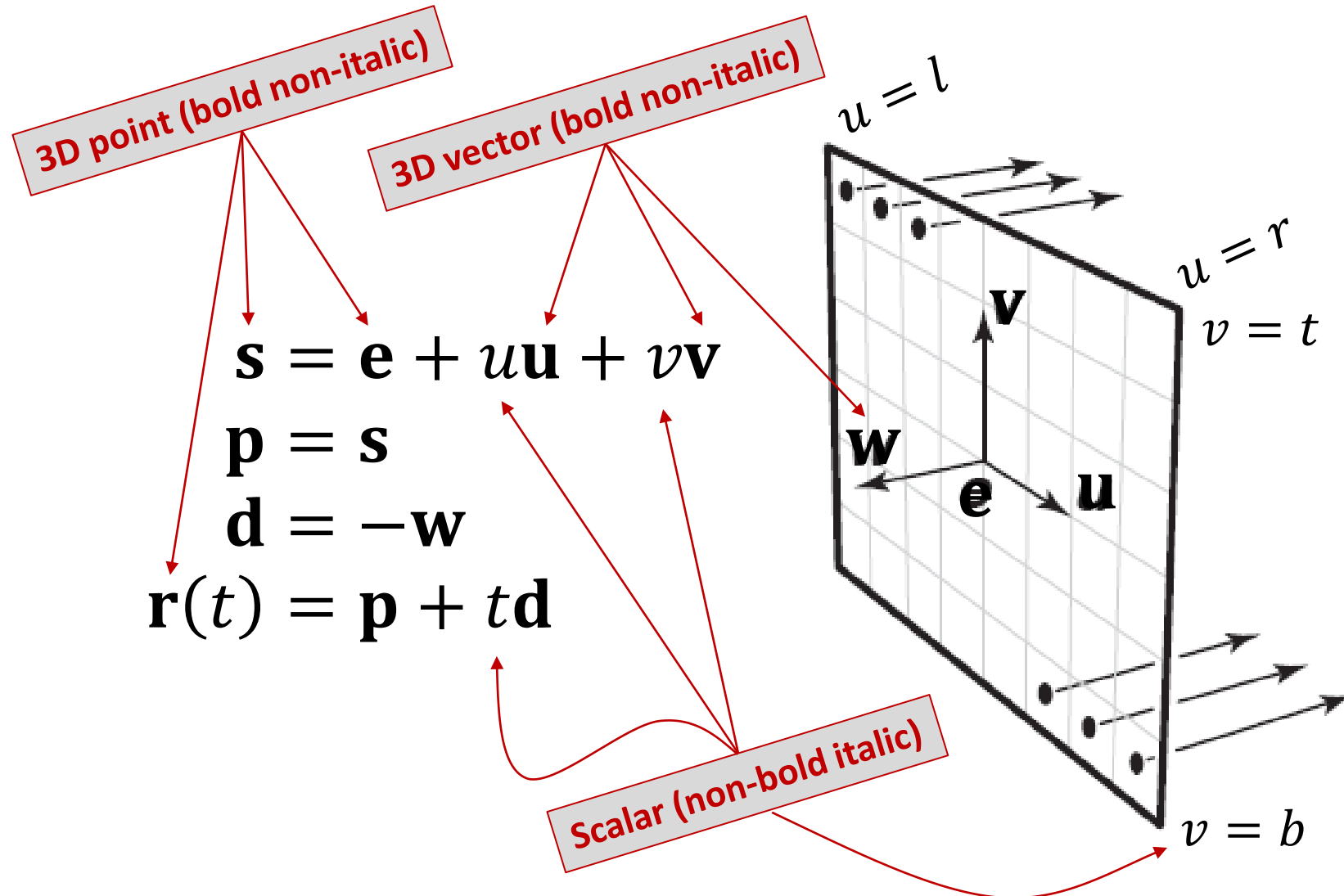
- Just need to compute the view plane point \mathbf{s}



$$\begin{aligned}\mathbf{p} &= \mathbf{s} \\ \mathbf{d} &= \mathbf{d}_v \\ \mathbf{r}(t) &= \mathbf{p} + t\mathbf{d}\end{aligned}$$

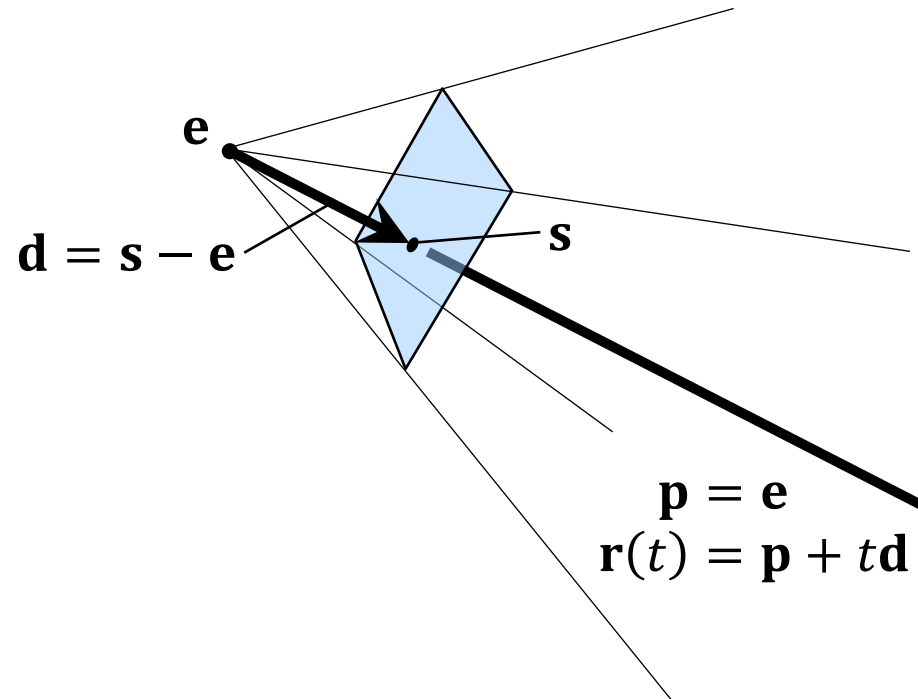
But where exactly is the view rectangle?

Generating eye rays—orthographic



Generating eye rays—perspective

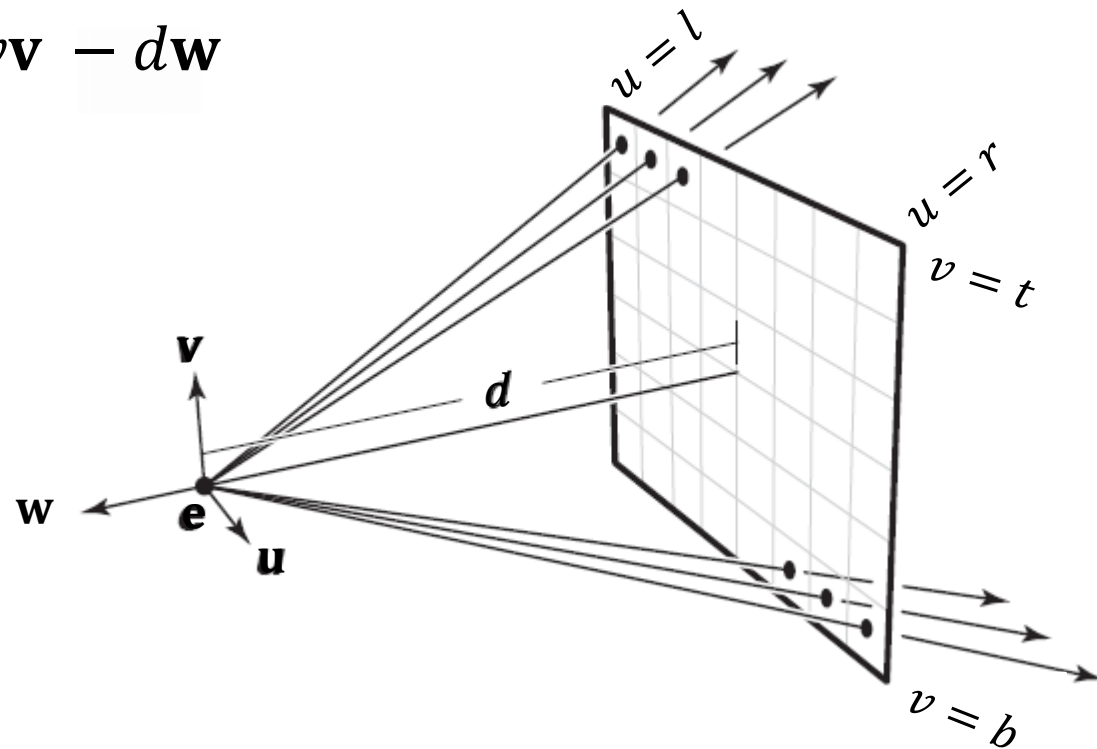
- View rectangle needs to be away from viewpoint
- Distance is important: “focal length” of camera
 - still use camera frame but position view rectangle away from viewpoint
 - ray origin always \mathbf{e}
 - ray direction now controlled by \mathbf{s}



Generating eye rays—perspective

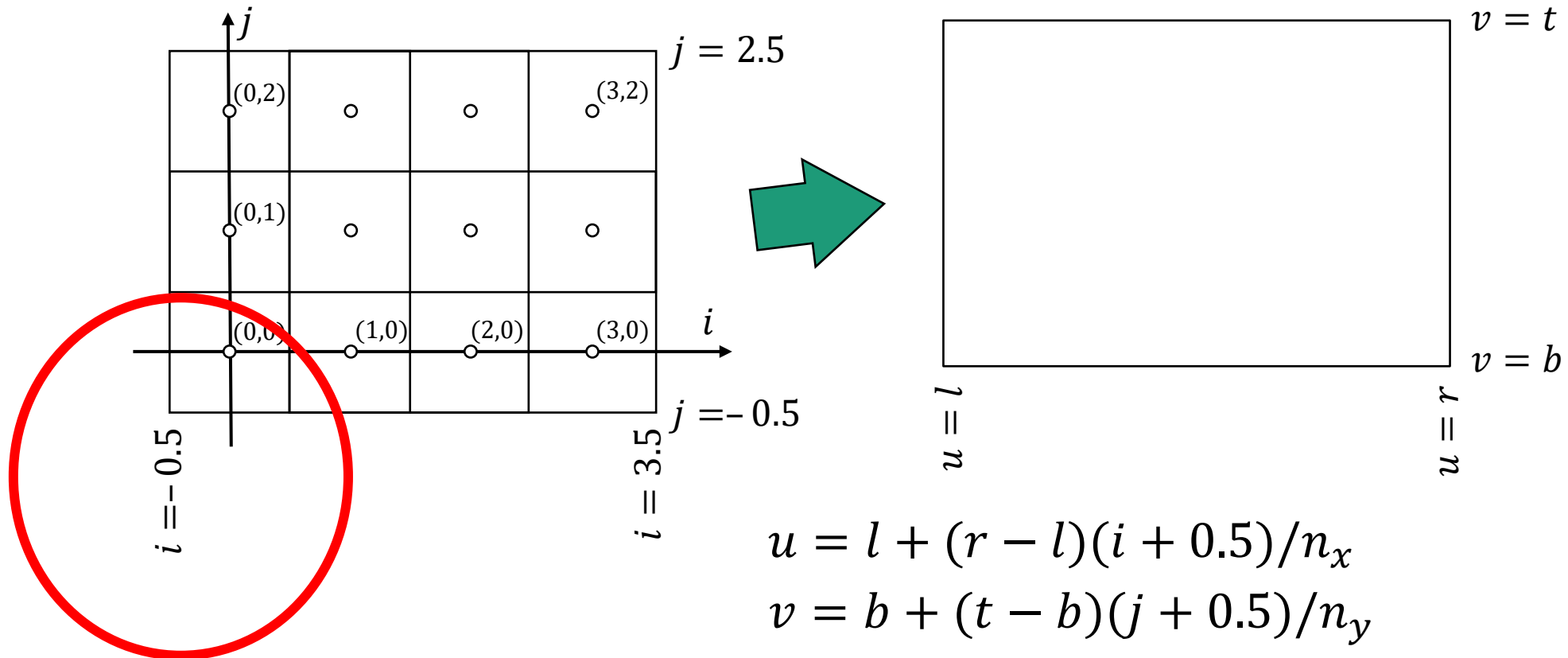
- Compute \mathbf{s} in the same way; just subtract $d\mathbf{w}$
 - coordinates of \mathbf{s} are $(u, v, -d)$

$$\begin{aligned}\mathbf{s} &= \mathbf{e} + u\mathbf{u} + v\mathbf{v} - d\mathbf{w} \\ \mathbf{p} &= \mathbf{e} \\ \mathbf{d} &= \mathbf{s} - \mathbf{e} \\ \mathbf{r}(t) &= \mathbf{p} + t\mathbf{d}\end{aligned}$$



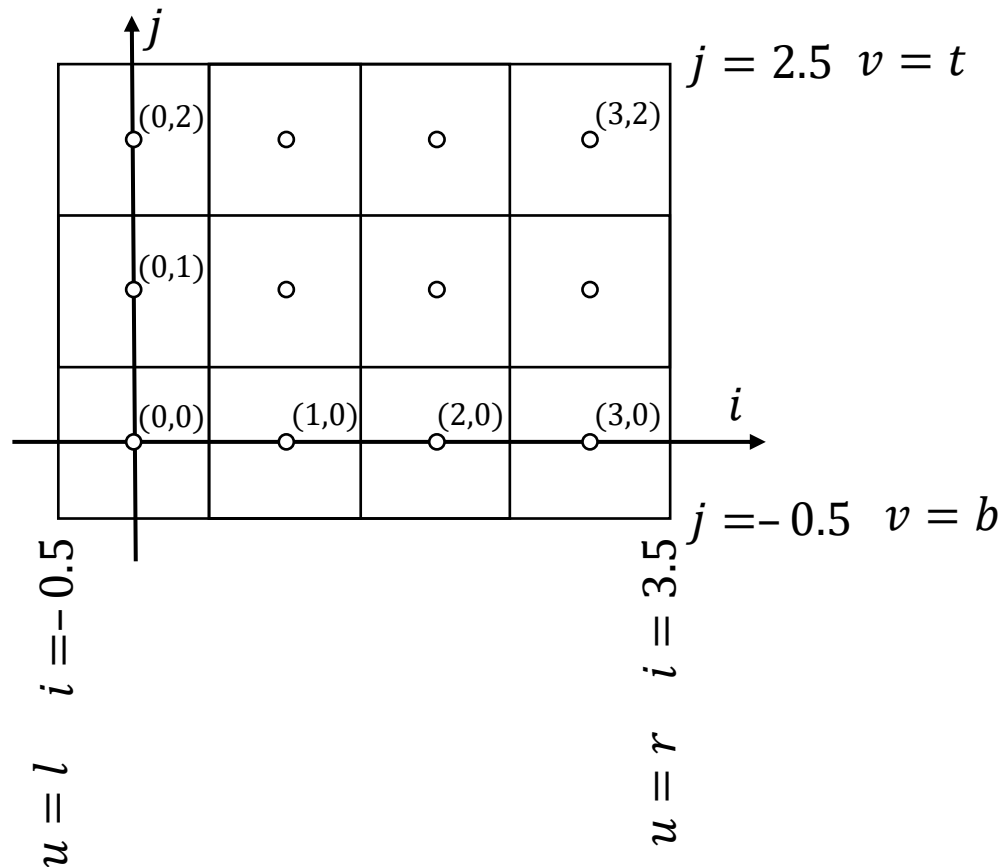
Pixel-to-image mapping

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



Pixel-to-image mapping

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



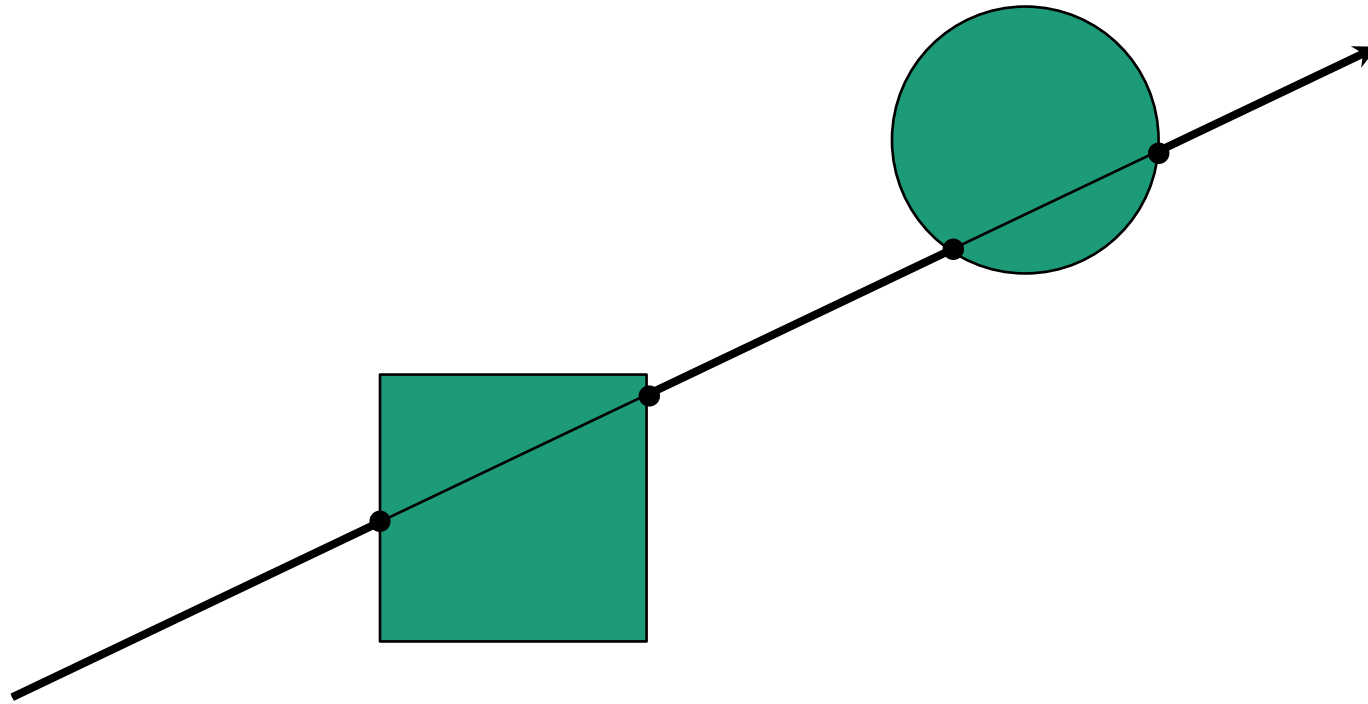
$$u = l + (r - l)(i + 0.5)/n_x$$
$$v = b + (t - b)(j + 0.5)/n_y$$

Generating eye rays – camera specification

- Given
 - Field of view in y direction,
 - eye point, look at, up direction
 - Image resolution $n_x \times n_y$
- How do you generate eye rays?
 - Do you need to compute a viewing transformation matrix?
 - Do you need to invert a matrix?
 - What is the aspect ratio of the image?
 - What is the left right top and bottom?
 - If the focal length is unspecified, what should d be?



Ray intersection

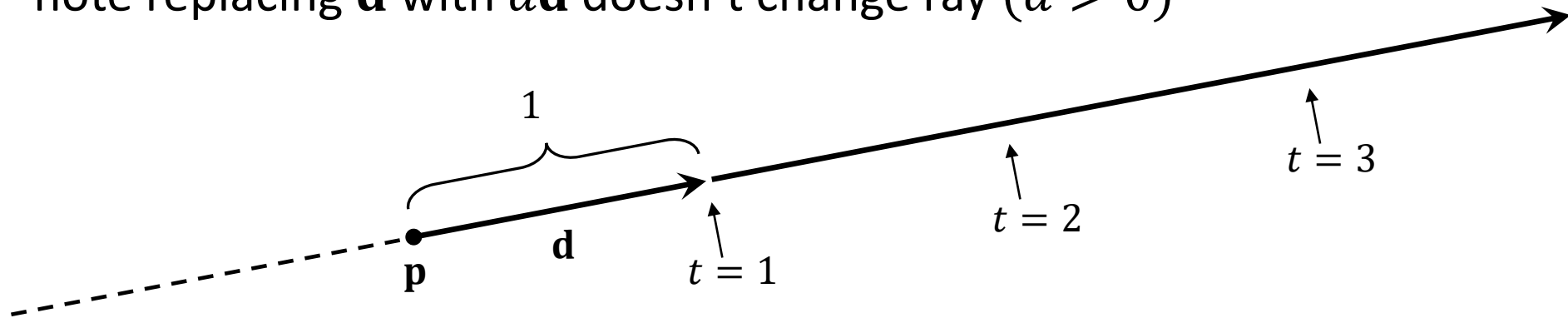


Ray: a half line

- 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 **d** with $a\mathbf{d}$ 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

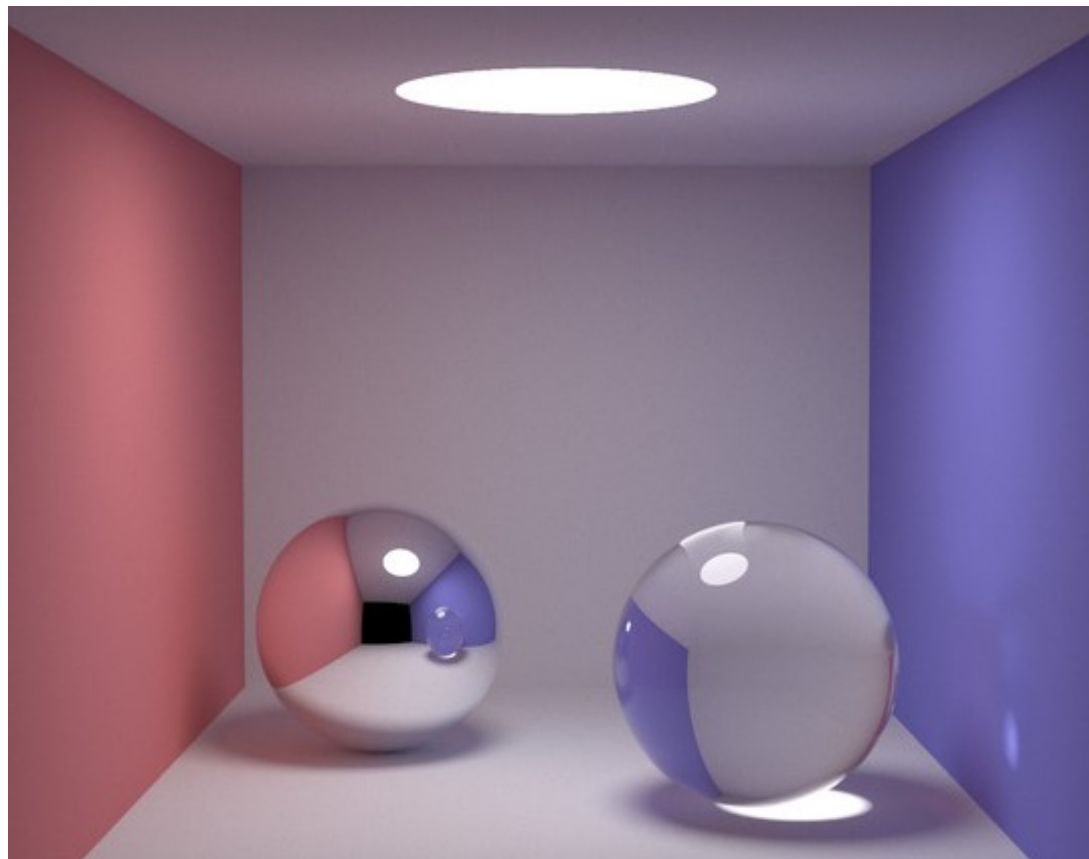
- Solution for t by quadratic formula:

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

What can you do with Spheres? (ASIDE)

Physically based rendering with 100 lines of code

<http://www.kevinbeason.com/smallpt/>



```
1. #include <math.h> // smallpt, a Path Tracer by Kevin Beason, 2008
2. #include <stdlib.h> // Make : g++ -O3 -fopenmp smallpt.cpp -o smallpt
3. #include <stdio.h> // Remove "-fopenmp" for g++ version < 4.2
4. struct Vec { // usage: time ./smallpt 5000 400 400 image.ppm
5.     double x, y, z; // position, also color (x,y,b)
6.     Vec(double x=0, double y=0, double z=0) { x=x; y=y; z=z; }
7.     Vec operator+(const Vec &b) const { return Vec(x+b.x, y+b.y, z+b.z); }
8.     Vec operator-(const Vec &b) const { return Vec(x-b.x, y-b.y, z-b.z); }
9.     Vec operator*(double s) const { return Vec(x*s, y*s, z*s); }
10.    Vec mult(const Vec &b) const { return Vec(x*b.x, y*b.y, z*b.z); }
11.    Vec& norm() { return *this = *this * (1/sqrt(x*x+y*y+z*z)); }
12.    double dot(const Vec &b) const { return x*b.x+y*b.y+z*b.z; } // cross:
13.    Vec operator^(const Vec &b) const { return Vec(y*b.z-z*b.y, z*b.x-x*b.y, x*b.y-y*b.x); }
14. };
15. struct Ray { Vec o, d; Ray(Vec o, Vec d) : o(o), d(d) {} };
16. enum Refl_t { DIFF, SPEC, REFR }; // material types, used in radiance()
17. struct Sphere {
18.     double rad; // radius
19.     Vec p, e, c; // position, emission, color
20.     Refl_t refl; // reflection type (DIFFuse, SPECular, REFRactive)
21.     Sphere(double rad, Vec p, Vec e, Vec c, Refl_t refl) :
22.         rad(rad), p(p), e(e), c(c), refl(refl) {}
23.     double intersect(const Ray &r) const { // intersection if nohit
24.         Vec op = p-r.o; // Solve t^2*d.d + 2*t*(o-p).d + (o-p).(o-p) - R^2 = 0
25.         double t, eps=1e-4, b=op.dot(r.d), det=b*b-op.dot(op)+rad*rad;
26.         if (det<0) return 0; else det=sqrt(det);
27.         return (t=b-det)>eps ? t : ((t=b+det)>eps ? t : 0);
28.     }
29. };
30. Sphere spheres[] = { //Scene: radius, position, emission, color, material
31.     Sphere(1e5, Vec(1e5+1, 40.8, 81.6), Vec(1, Vec(.75, .25, .25), DIFF), //Left
32.     Sphere(1e5, Vec(-1e5+99, 40.8, 81.6), Vec(1, Vec(.25, .25, .75), DIFF), //Right
33.     Sphere(1e5, Vec(50, 40.8, 1e5), Vec(1, Vec(.75, .75, .75), DIFF), //Back
34.     Sphere(1e5, Vec(50, 40.8, -1e5+100), Vec(1, Vec(1, 1, 1), DIFF), //Front
35.     Sphere(1e5, Vec(-1e5, -1e5+11.6, 61.6), Vec(1, Vec(.5, .5, .5), DIFF), //Bottom
36.     Sphere(1e5, Vec(50, -1e5+11.6, 61.6), Vec(1, Vec(.5, .5, .5), DIFF), //Top
37.     Sphere(16.5, Vec(27, 16.5, 47), Vec(1, Vec(1, 1, 1)*.999, SPEC), //Mirror
38.     Sphere(16.5, Vec(73, 16.5, 78), Vec(1, Vec(1, 1, 1)*.999, REFR), //Glass
39.     Sphere(600, Vec(50, 681.6 - .27, 81.6), Vec(12, 12, 12), Vec(1, DIFF) //Lite
40. };
41. inline double clamp(double x) { return x<0 ? 0 : x>1 ? 1 : x; }
42. inline int toInt(double x) { return int(pow(clamp(x), 1/2.2)*255+.5); }
43. inline bool intersect(const Ray &r, double &t, int &id) {
44.     double n=sizeof(spheres)/sizeof(Sphere), d, int t=le20;
45.     for(int i=0; i<n; i++) if((d=spheres[i].intersect(r))<=d) {t=d;id=i;}
46.     return t<inf;
47. }
48. Vec radiance(const Ray &r, int depth, unsigned short *Xi) {
49.     double t; // distance to intersection
50.     int id=0; // id of intersected object
51.     if (!intersect(r, t, id)) return Vec(); // if miss, return black
52.     const Sphere &obj = spheres[id]; // the hit object
53.     Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?-1:1, f=obj.c;
54.     double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
55.     if (++depth>5) if (erand48(Xi)<p) f=f*(1/p); else return obj.e; //R.R.
56.     if (obj.refl == DIFF) { // Ideal DIFFUSE reflection
57.         double r1=2*M_PI*erand48(Xi), r2=erand48(Xi), r2s=sqrt(r2);
58.         Vec w=nl, u=((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(), v=w%u;
59.         Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
60.         return obj.e + f.mult(radiance(Ray(x, d), depth, Xi));
61.     } else if (obj.refl == SPECULAR reflection)
62.         return obj.e + f.mult(radiance(Ray(x, r.d-n*2), depth, Xi));
63.     Ray reflRay(x, r.d-n*2*n.dot(r.d)); // Ideal dielectric REFRACTION
64.     bool into = n.dot(nl)>0; // Ray from outside going in?
65.     double nc=1, nt=1.5, nnt=nt*nt/nc, ddn=r.d.dot(nl), ddnr=d.dot(dn);
66.     if ((cos2t=1-nnt*(1-ddn*ddn))<0) // Total internal reflection
67.         return obj.e + f.mult(radiance(reflRay, depth, Xi));
68.     Vec tdir = (r.d*nt - n*(into?-1:1)*(ddn*nnt+sqrt(cos2t))).norm();
69.     double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
70.     double Re=R0+(1-R0)*c*c*c*c*c, Tr=1-Re, P=.25+.5*Re, RP=Re/P, TP=Tr/(1-P);
71.     return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ? // Russian roulette
72.         radiance(reflRay, depth, Xi)*RP:radiance(Ray(x, tdir), depth, Xi)*TP) :
73.         radiance(reflRay, depth, Xi)*Re+radiance(Ray(x, tdir), depth, Xi)*Tr);
74. }
75. int main(int argc, char *argv[]) {
76.     int w=1024, h=768, samps = argc==2 ? atoi(argv[1])/4 : 1; // # samples
77.     Ray cam(Vec(50, 32, 295.6), Vec(0, -0.042612, -1).norm()); // cam pos, dir
78.     Vec cx=Vec(.5135/h), cy=(cx*cam.d).norm()*-.5135, z, *c=new Vec(w*h);
79.     #pragma omp parallel for schedule(dynamic, 1) private(x) // OpenMP
80.     for (int y=0; y<h; y++) { // Loop over image rows
81.         fprintf(stderr, "\tRendering (%d smp) %5.2f%%, samps*4, 100.y/(h-1));
82.         for (unsigned short x=0, Xi[3]=(0,0,y*y*y); x<w; x++) // Loop cols
83.             for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++) // 2x2 subpixel rows
84.                 for (int sx=0; sx<2; sx++) // 2x2 subpixel cols
85.                     for (int s=0; s<samps; s++) {
86.                         double r1=2*erand48(Xi), x=r1<.1 ? sqrt(r1)-.1 : 1-sqrt(2-r1);
87.                         double r2=2*erand48(Xi), dy=r2<.1 ? sqrt(r2)-.1 : 1-sqrt(2-r2);
88.                         Vec d = cx*( ( (sx+.5+dx)/2 + x)/w - .5) +
89.                             cy*( ( (sy+.5+dy)/2 + y)/h - .5) + cam.d;
90.                         r = r + radiance(Ray(cam.o+d*140,d.norm()), 0, Xi)*(1./samps);
91.                     } // Camera rays are pushed ^^^^^ forward to start in interior
92.                     c[i] = c[i] + Vec(clamp(x), clamp(y), clamp(z))*25;
93.             }
94.     }
95.     FILE *f = fopen("image.ppm", "w"); // Write image to PPM file.
96.     fprintf(f, "P3\n%d %d\n255\n", w, h);
97.     for (int i=0; i<w*h; i++) {
98.         fprintf(f, "%d %d %d ", toInt(c[i].x), toInt(c[i].y), toInt(c[i].z));
99.     }
100. }
```

vector & ray classes

sphere intersection

scene description

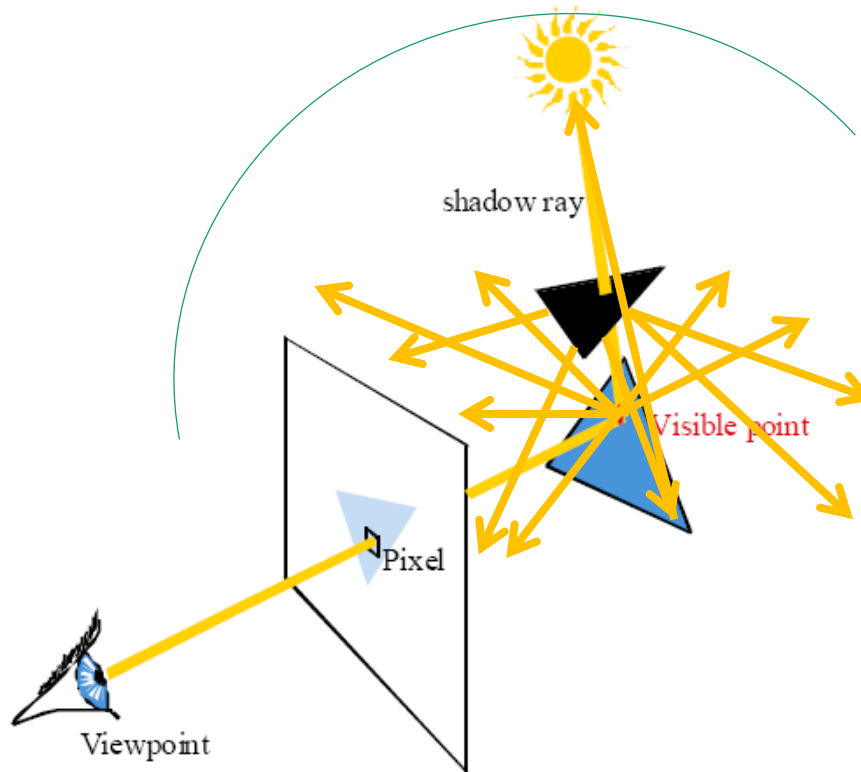
raytracing

path tracer

raytracing

image output

Global Illumination (ASIDE)



- Where does the light come from?
- Can still solve this problem for special cases, or with careful sampling, in both object-order and image-order implementations

Global Illumination (ASIDE)

- Light can bounce off walls to illuminate different surfaces
 - Red reflected light illuminates left side of left object
 - Green reflected light illuminates right side of cube on the right
- At right: which is a photo, and which is a rendering?



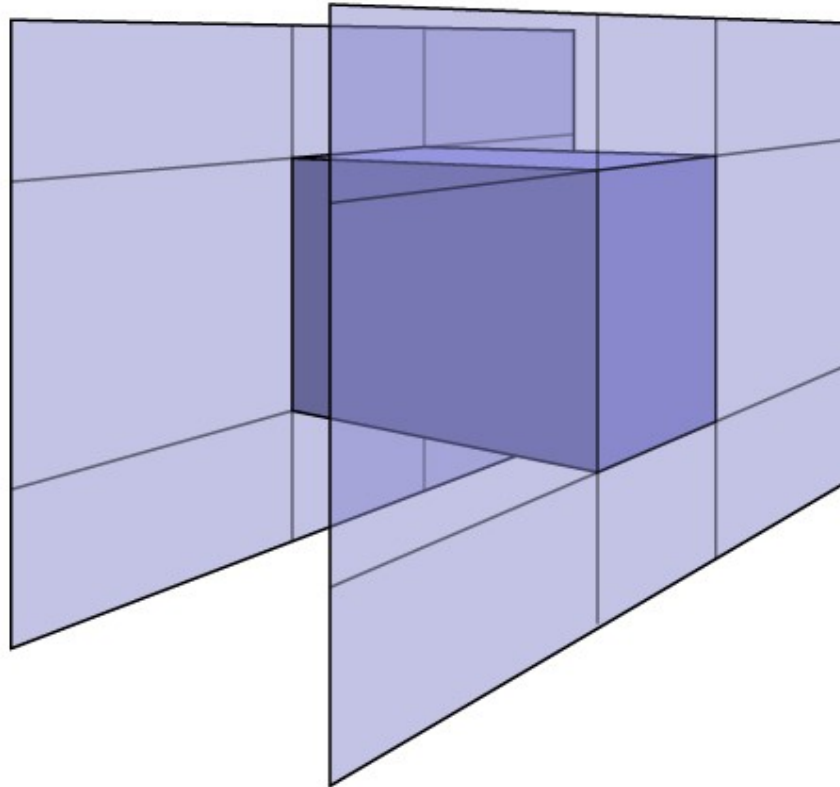
Subsurface Scattering (ASIDE)

- The full problem is even more complex!!!
 - Light is *everywhere* and can even be *scattered beneath* the surface of materials
 - Devising a global illumination method that is fast and that correctly models all *light transport* is the ultimate problem in rendering.



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!

$$p_x + t_{x\min} d_x = x_{\min}$$

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

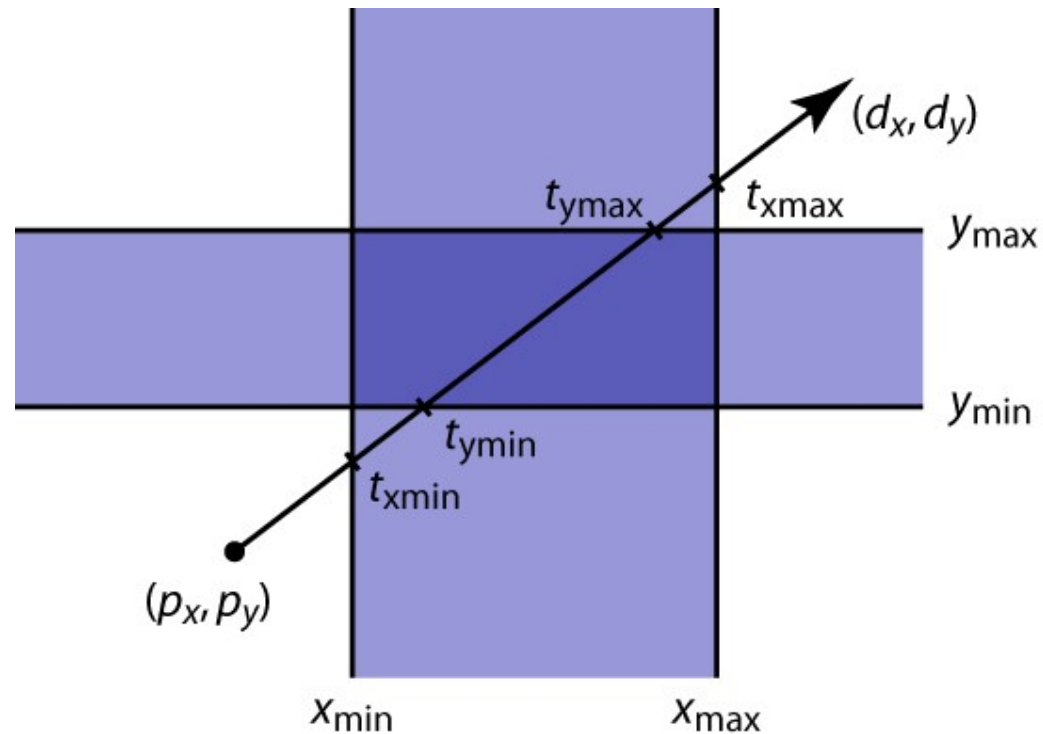
$$p_y + t_{y\min} d_y = y_{\min}$$

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

Compute t values where
ray crosses each plane

What if $d_x = 0$?

Intersection everywhere if $p_x \in [x_{\min}, x_{\max}]$,
otherwise no intersection



Intersecting intersections

- Each intersection is an interval

$$t_{xlow} = \min(t_{xmin}, t_{xmax})$$

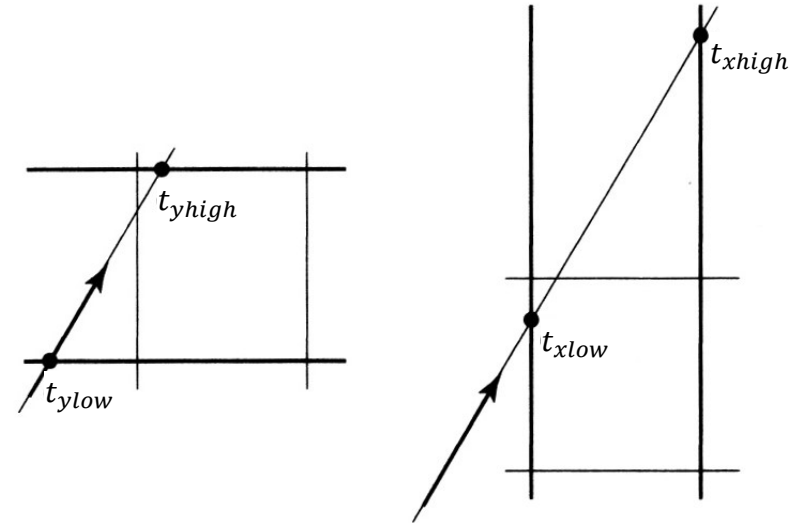
$$t_{xhigh} = \max(t_{xmin}, t_{xmax})$$

- Want last entry point and first exit point

$$t_{min} = \max(t_{xlow}, t_{ylo\text{w}})$$

$$t_{max} = \min(t_{xhigh}, t_{yhigh})$$

- If $t_{max} < t_{min}$ then empty interval, i.e., no intersection



$$t \in [t_{xlo\text{w}}, t_{xhigh}]$$

$$t \in [t_{ylo\text{w}}, t_{yhigh}]$$

$$t \in [t_{xlo\text{w}}, t_{xhigh}] \cap [t_{ylo\text{w}}, t_{yhigh}]$$

Ray-triangle intersection

- Condition 1: point is on ray

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

- Condition 2: point is on plane

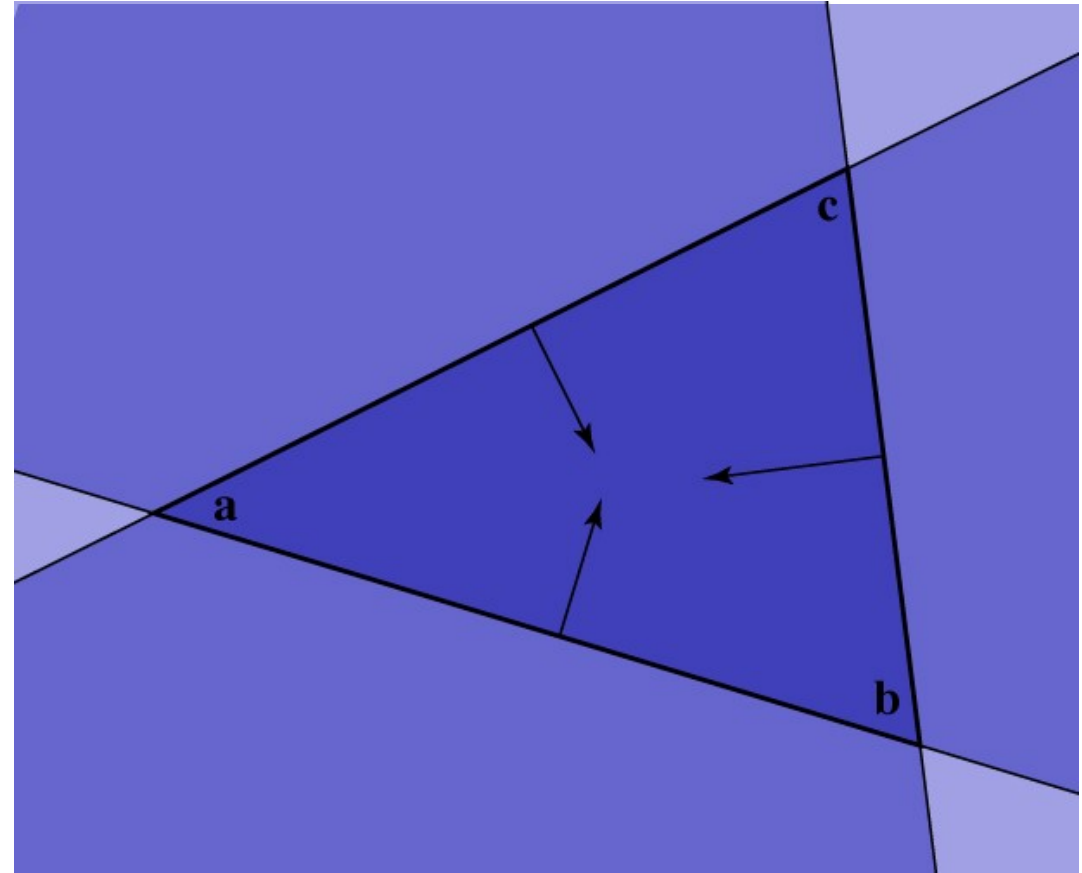
$$(\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} = 0$$

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

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

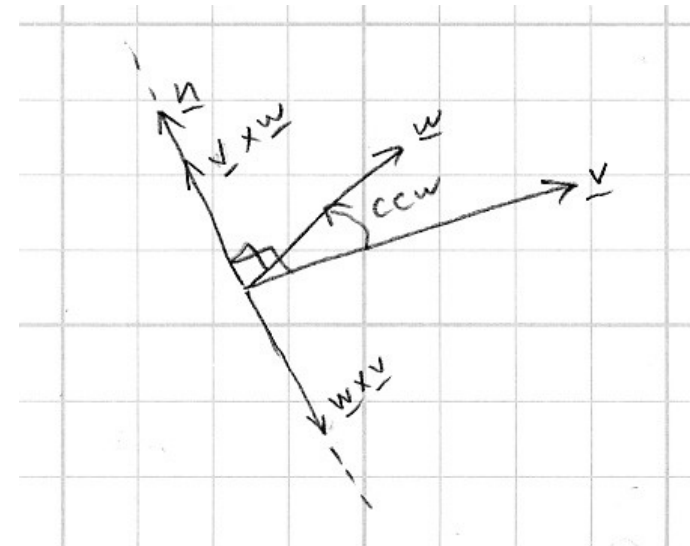
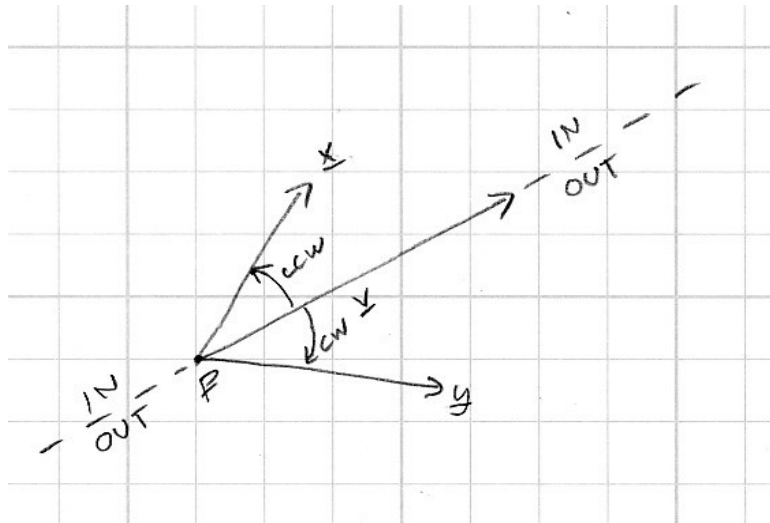
Ray-triangle intersection

- In plane, the 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 \mathbf{x}
- Use cross product to decide



Ray-triangle intersection

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

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

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

- Can see this as a step toward computing barycentric coordinates (useful as texture coordinates)
- Can also use a similar inside/outside test on non-convex n-gons, by computing the winding number (see CGPP 7.10)

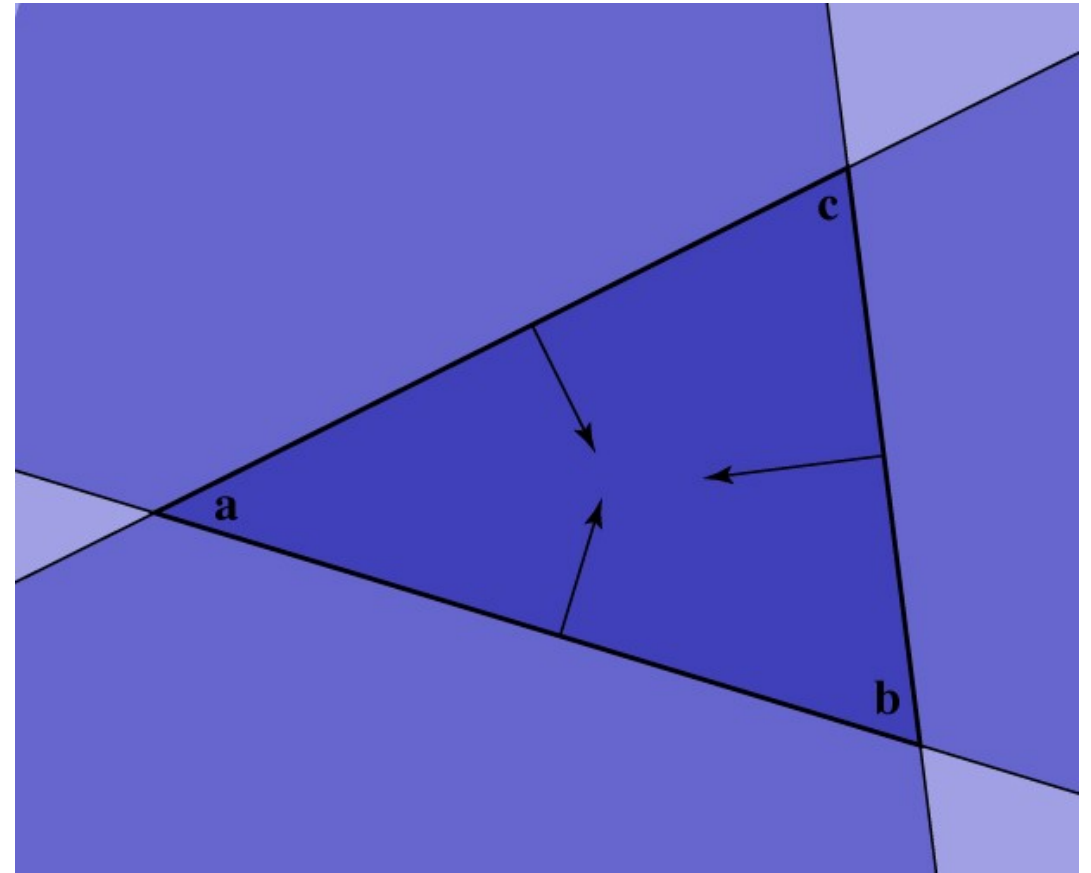
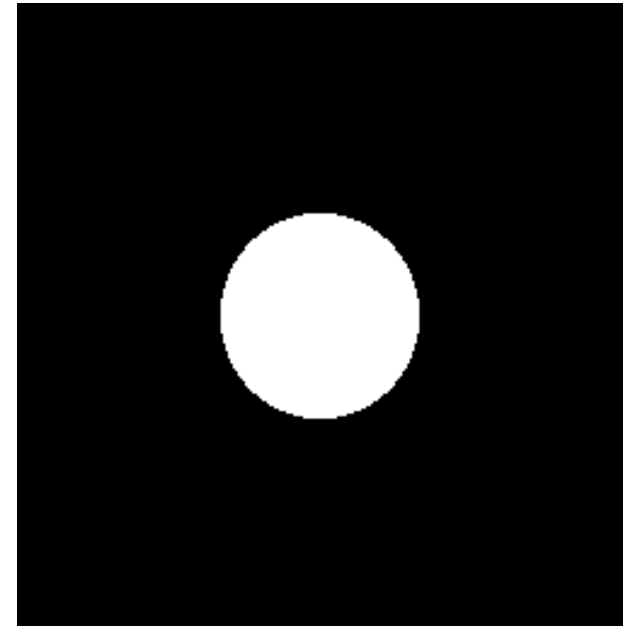


Image so far

- 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

```
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;  
}
```

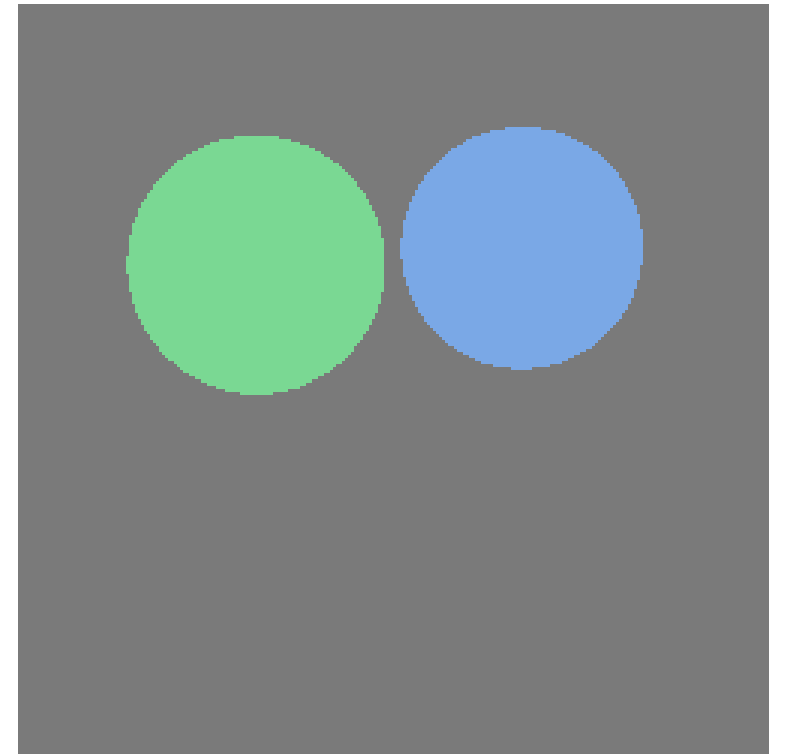
Image so far

- 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);
  }
```

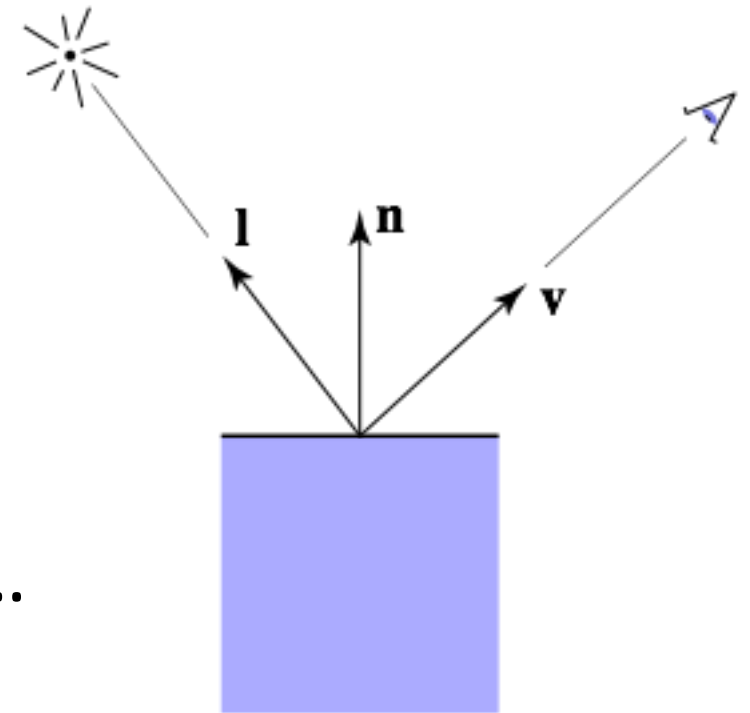
...

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



Shading

- Compute light reflected toward camera
- Inputs:
 - eye direction
 - light direction (for each of many lights)
 - surface normal
 - surface parameters (color, shininess, ...)
- Exact same equations as seen previously...

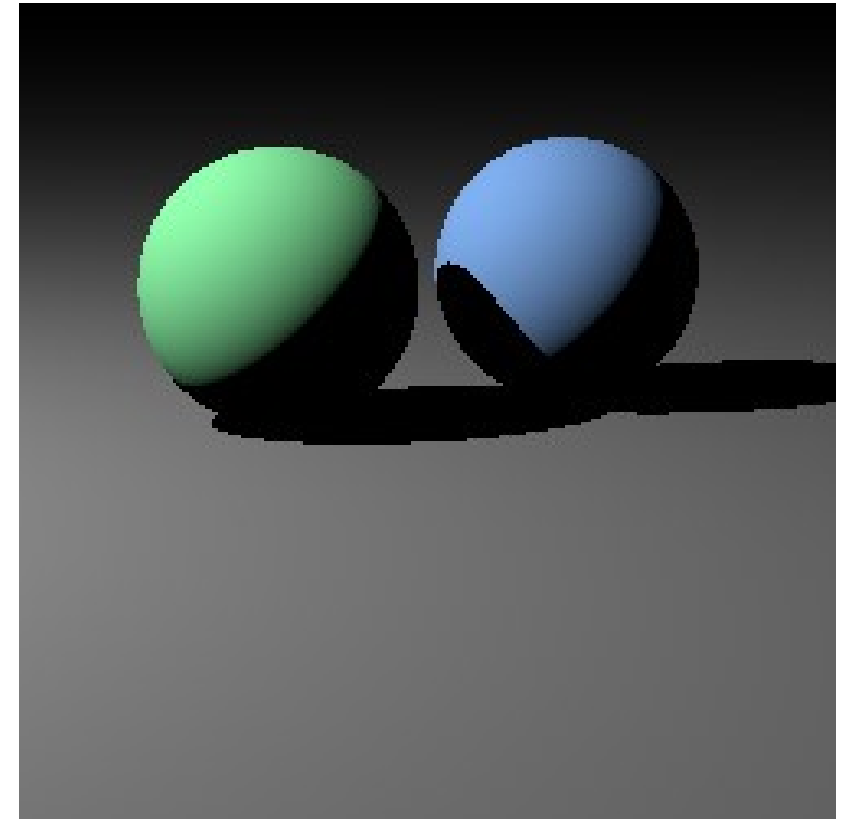


Shadows

- The 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!

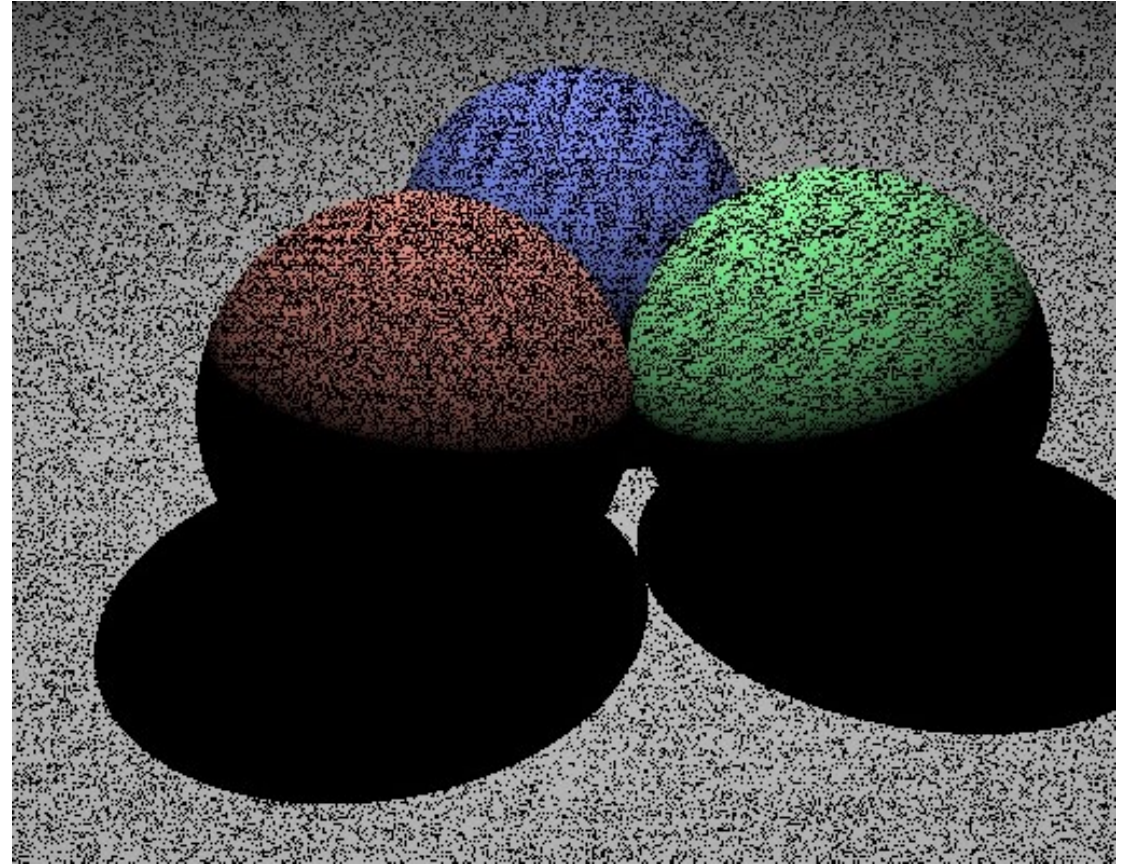
Image so far

```
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;  
}
```



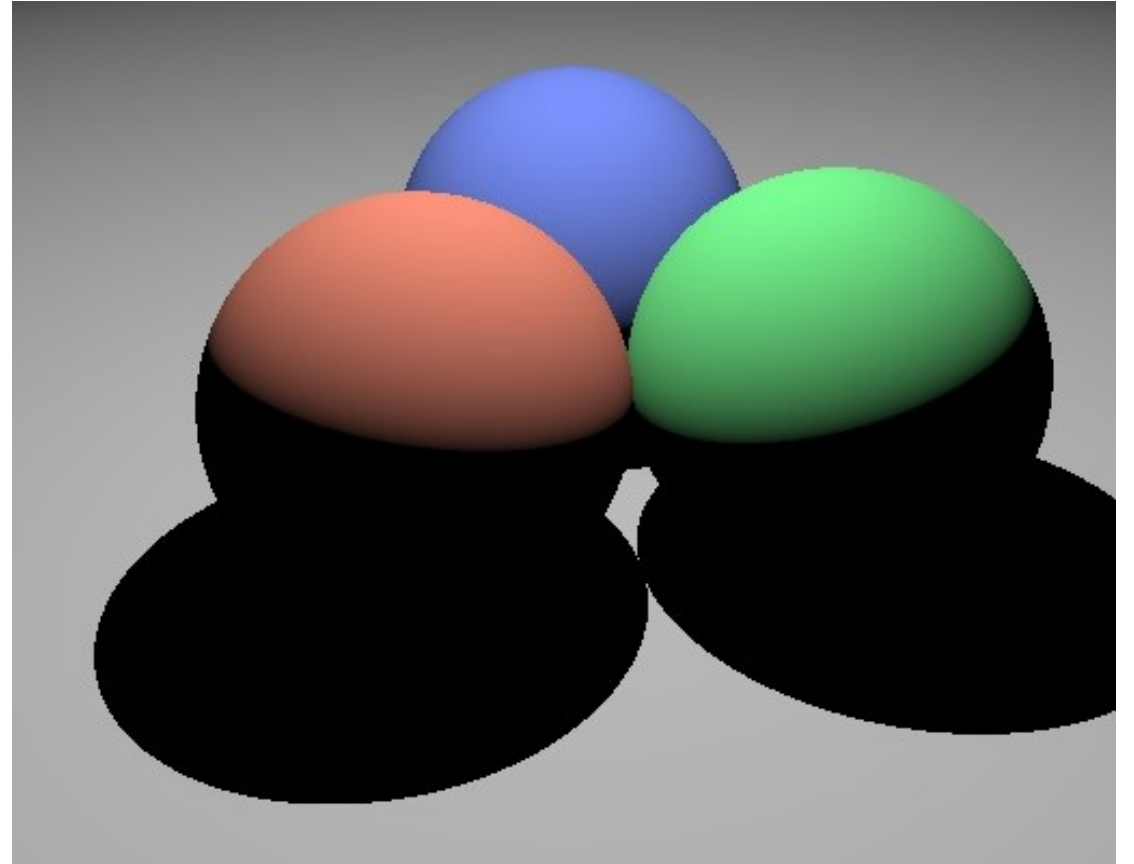
Shadow rounding errors

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

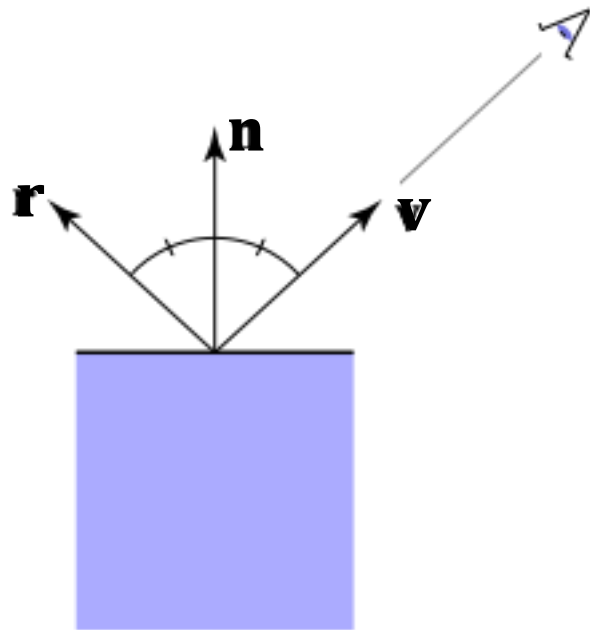


Mirror reflection

- Consider perfectly shiny surface
 - There isn't a highlight (i.e., no L_s)
 - 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
$$L = L_a + L_d + L_m$$
 - Here L_m is evaluated by tracing a new ray

Mirror reflection

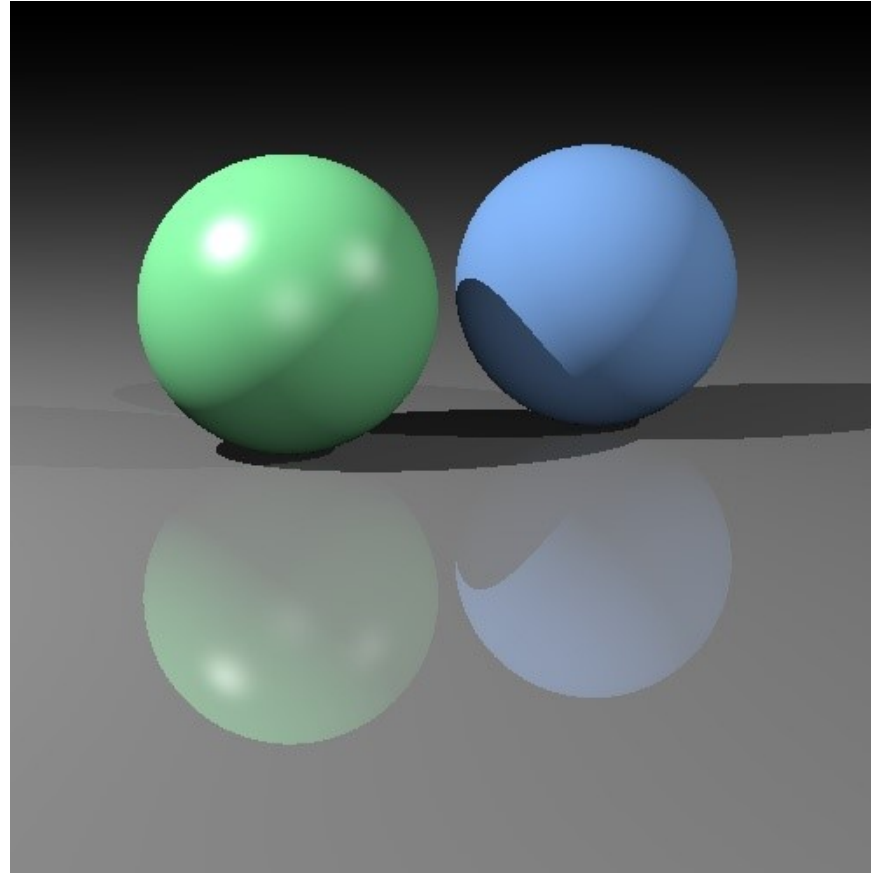
- Intensity depends on view direction
 - Reflects incident light from mirror direction



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

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

Diffuse + mirror reflection (glazed)



(glazed material on floor)

Ray tracer architecture 101

- You want a class called Ray
 - point and direction; evaluate(t)
 - possible: t_{Min} , t_{Max}
- 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

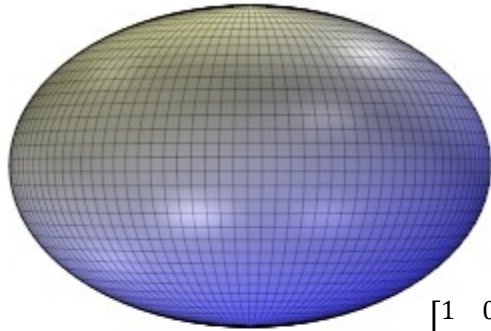
- Return values
 - Surface intersection tends to want to return multiple values
 - Scalar t , surface or shader, normal vector, maybe surface point
 - Typical solution: an *intersection record*
 - A class with fields for all these things
 - Keep track of the intersection record for the closest intersection
- Efficiency
 - In Java to be fast one should 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 that blocks the light will do, don't need closest)
 - But: “First Get it Right, Then Make it Fast”

Debugging strategies

- Test with small images
- Set breakpoints!!!
 - E.g., conditional on a specific pixel
- Make sure your rays are in the correct direction
 - For example, is the ray for the center of the image what you expect it to be?
- Watch out for other common mistakes...

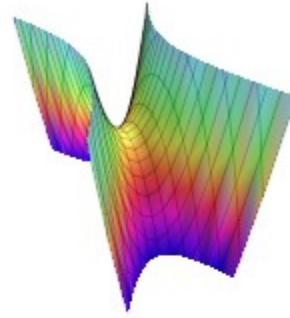
Quadrics

<http://en.wikipedia.org/wiki/Quadric>

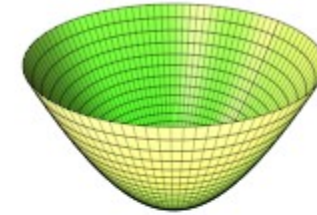


Ellipsoid

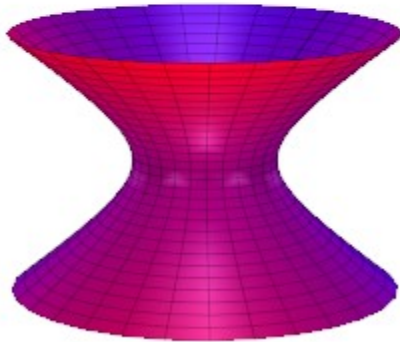
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$



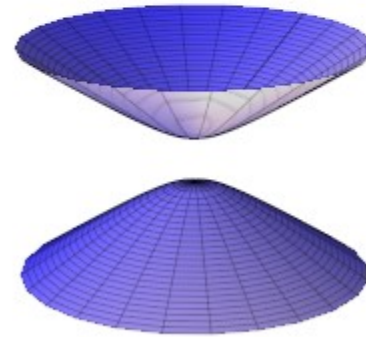
Hyperbolic paraboloid



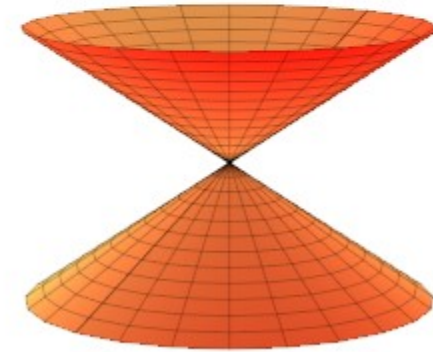
Elliptic paraboloid



Hyperboloid of one sheet



Hyperboloid of two sheets



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{Cone}$$

Quadrics

- In non-homogeneous coordinates we can write

$$[x \ y \ z] A \begin{bmatrix} x \\ y \\ z \end{bmatrix} - 2\mathbf{b}^T \begin{bmatrix} x \\ y \\ z \end{bmatrix} + c = 0 \quad A \in \mathbb{R}^{3 \times 3} \quad \mathbf{b} \in \mathbb{R}^3 \quad c \in \mathbb{R}$$

- In homogeneous coordinates, use $Q \in \mathbb{R}^{4 \times 4}$ matrix

$$Q = \begin{bmatrix} A & -\mathbf{b} \\ -\mathbf{b}^T & c \end{bmatrix} \quad \mathbf{x}^T Q \mathbf{x} = 0 \quad \mathbf{x} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

- Solution is same as ray sphere intersection.
 - Replace \mathbf{x} with ray equation, expand, solve for t
 - **Given intersection point \mathbf{x} , what is the normal?** $\nabla F(\mathbf{x}) = 2A\mathbf{x} - 2\mathbf{b}^T$

Review and More Information

- CGPP 15.4 ray tracing in general
- CGPP 14.5.2 ray tracing implicit surfaces
- Ray-Triangle intersection, see also FCG Section 4.4.2 for method based on linear systems and Cramer's rule, but see also Section 2.7 with respect to barycentric coordinates