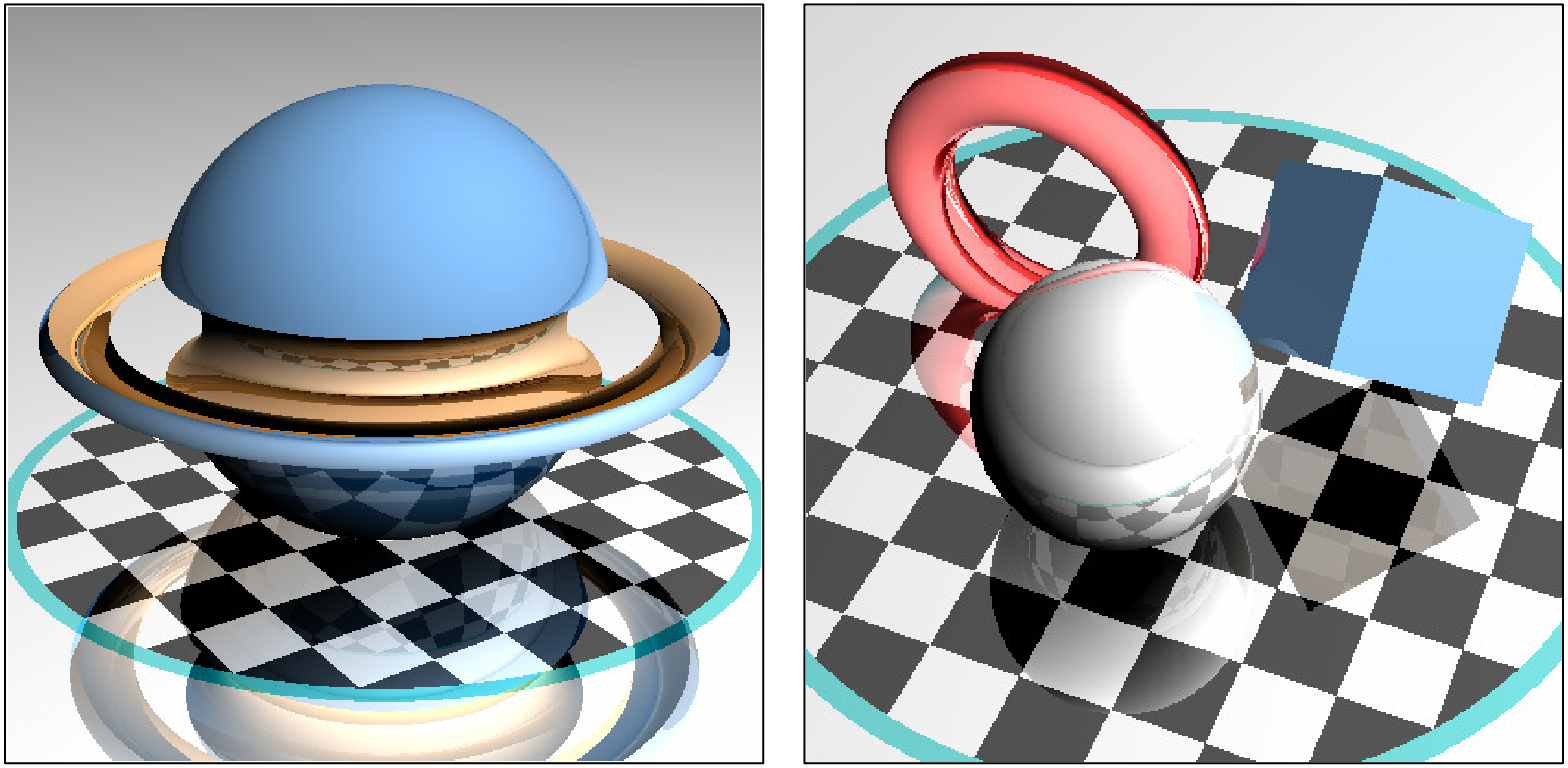
Advanced Graphics

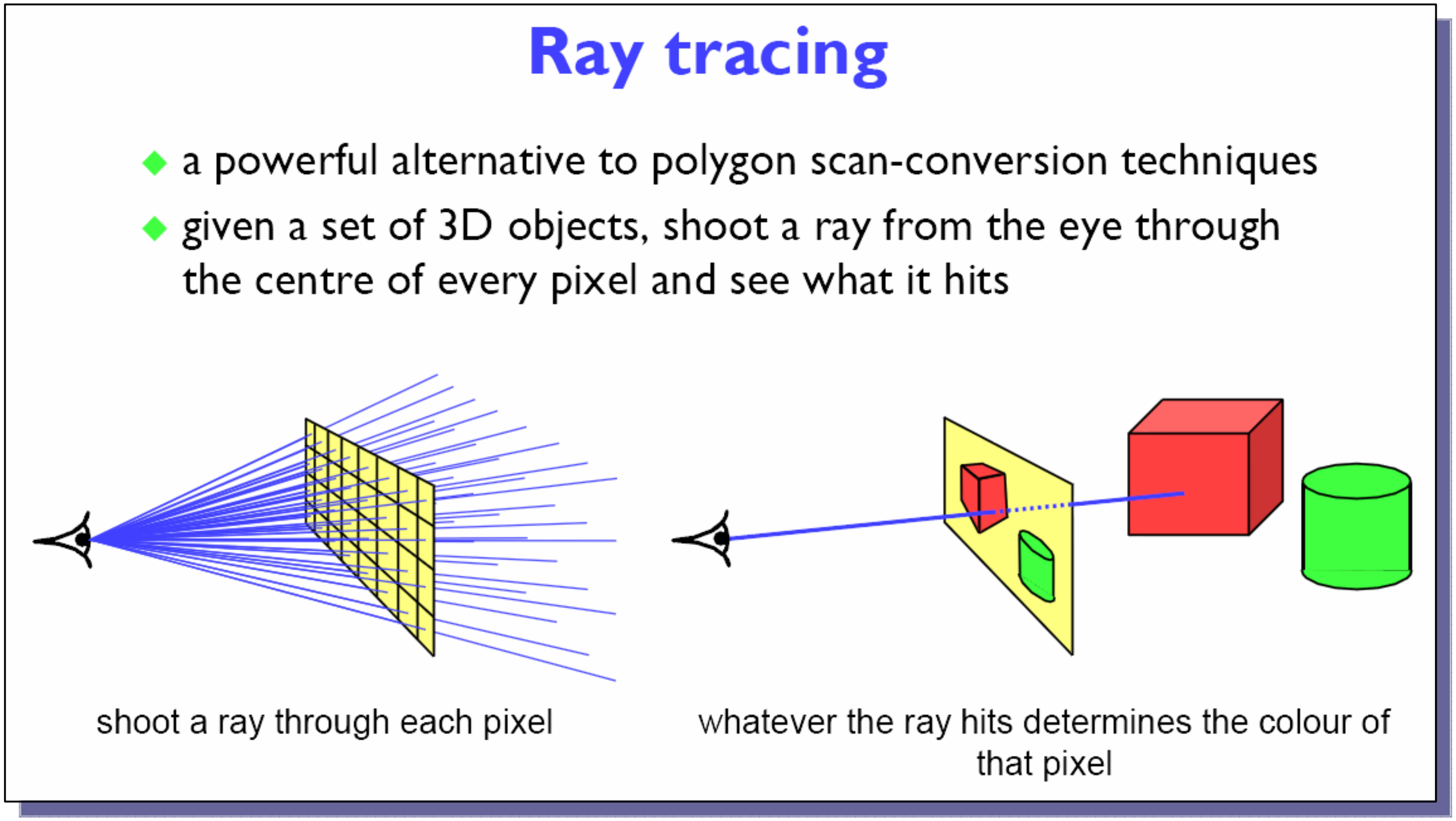


Ray Tracing: Geometry and Lighting

Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk

Supported in part by Google UK, Ltd

Ray tracing revisited



Ray tracing

- a powerful alternative to polygon scan-conversion techniques
- given a set of 3D objects, shoot a ray from the eye through the centre of every pixel and see what it hits

shoot a ray through each pixel

whatever the ray hits determines the colour of that pixel

(Slide from Neil Dodgson's *Computer Graphics and Image Processing* notes, Cambridge University.)

Ray tracing

- The basic algorithm is straightforward
- Much room for subtlety
  - Refraction
  - Reflection
  - Shadows
  - Anti-aliasing
  - Blurred edges, depth-of-field effects

```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};
struct sphere{ vec cen,color;double rad,kd,ks,kt,kl,ir}*s,
*best,sph[]={0.,6.,.5,1.,1.,1.,.9, .05,.2,.85,0.,1.7,-1.,8.,-
.5,1.,.5,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,
1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-
3.,-3.,12.,.8,1., 1.,5.,0.,0.,0.,.5,1.5,};yx;double
u,b,tmin,sqrt(),tan();double vdot(A,B)vec A ,B;{return
A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec
A,B;{B.x+=a* A.x;B.y+=a*A.y;B.z+=a*A.z;return B;}vec
vunit(A)vec A;{return vcomb(1./sqrt( vdot(A,A)),A,black);}
struct sphere*intersect(P,D)vec P,D;{best=0;tmin=1e30;s=
sph+5;while(s-->sph)b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-
vdot(U,U)+s->rad*s ->rad,u=u>0?sqrt(u):1e31,u=b-u>1e-7?b-
u:b+u,tmin=u>=1e-7&&u<tmin?best=s,u: tmin;return best;}vec
trace(level,P,D)vec P,D;{double d,eta,e;vec N,color; struct
sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else
return amb;color=amb;eta=s->ir;d= -vdot(D,N=vunit(vcomb(-
1.,P=vcomb(tmin,D,P),s->cen )));if(d<0)N=vcomb(-1.,N,black),
eta=1/eta,d= -d;l=sph+5;while(l-->sph)if((e=l ->kl* vdot(N,U=
vunit(vcomb(-1.,P,l->cen))))>0&& intersect(P,U)==l)
color=vcomb(e ,l->color,color);U=s->color; color.x*=U.x;
color.y*=U.y;color.z*=U.z;e=1-eta* eta*(1-d*d);return vcomb(s-
>kt,e>0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt
(e),N,black))):black,vcomb(s->ks,trace(level, P,vcomb(2*d,
N,D)),vcomb(s->kd, color,vcomb(s->kl,U,black))));
}main(){printf("%d %d\n",32,32);while(yx<32*32) U.x=yx%32-
32/2,U.z=32/2-yx++/32,U.y=32/2/tan(25/114.5915590261),
U=vcomb(255., trace(3,black,vunit(U)),black),printf("%.0f %.0f
%.0f\n",U);}/*minray!*/
```

Paul Heckbert's 'minray' ray tracer, which fit on the back of his business card. (circa 1983)

Ray tracing

- The ray tracing time for a scene is a function of
  (num rays cast) x
  (num lights) x
  (num objects in scene) x
  (num reflective surfaces) x
  (ray reflection depth) x …
- Contrast to polygon rasterization: time is a function of the number of elements in the scene times the number of lights.

(Scene from the realtime ray traced *Quake 4*)

The algorithm

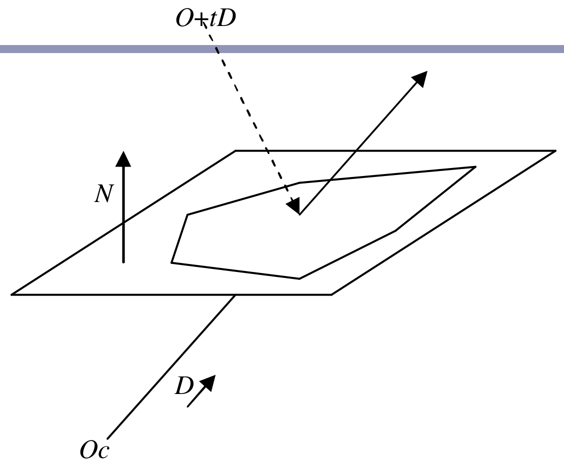
For each pixel on the screen, do:

1. Calculate ray from eye (O) through pixel (X)

   Set D = $(X-O) / |(X-O)|$

   Ray: $R=O+tD$
2. Find ray/primitive hit point (P) and normal (N)
3. Compute shadow, reflection, transparency rays; recursively call steps 2—4
4. Calculate lighting of surface at point

Ray/plane intersection



Ray $R=O+tD$

Poly $P=\{v^1,\dots,v^n\}$

$N=(v^n-v^1)\times(v^2-v^1)$

$N\bullet(O+tD-v^1)=0$

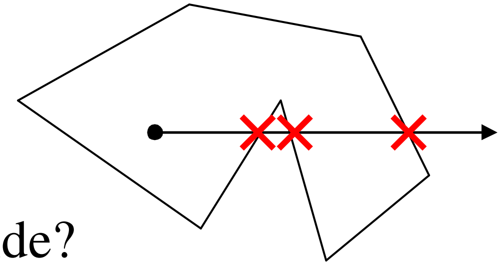
$N_x(O_x+tD_x-v_x^1)+N_y(O_y+tD_y-v_y^1)+N_z(O_z+tD_z-v_z^1)=0$

$N_xO_x+tN_xD_x-N_xv_x^1+N_yO_y+tN_yD_y-N_yv_y^1+N_zO_z+tN_zD_z-N_zv_z^1=0$

$tN_xD_x+tN_yD_y+tN_zD_z=N_xv_x^1+N_yv_y^1+N_zv_z^1-N_xO_x-N_yO_y-N_zO_z$

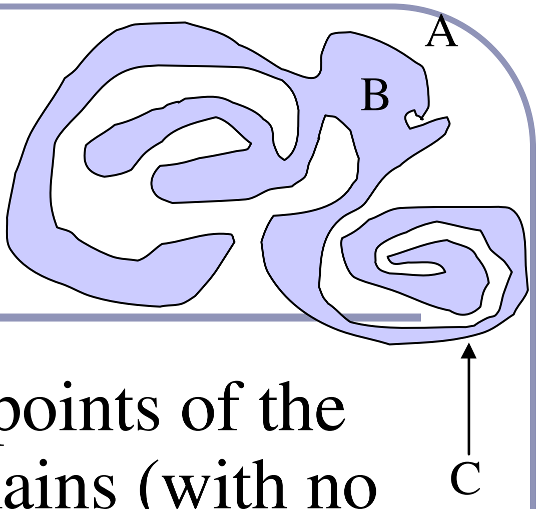
$t=((N\bullet v^1)-(N\bullet O))\,/\,(N\bullet D)$

Point-in-nonconvex-polygon



- *Ray casting* (1974)
  - Odd number of crossings = inside
  - Issues:
    - How to find a point that you *know* is inside?
    - What if the ray hits a vertex?
    - Best accelerated by working in 2D
      - You could transform all vertices such that the coordinate system of the polygon has normal = Z axis…
      - Or, you could observe that crossings are invariant under scaling transforms and just project along any axis by ignoring (for example) the Z component.
  - Validity proved by the *Jordan curve* theorem…

The *Jordan curve theorem*



"Any simple closed curve C divides the points of the plane not on C into two distinct domains (with no points in common) of which C is the common boundary."

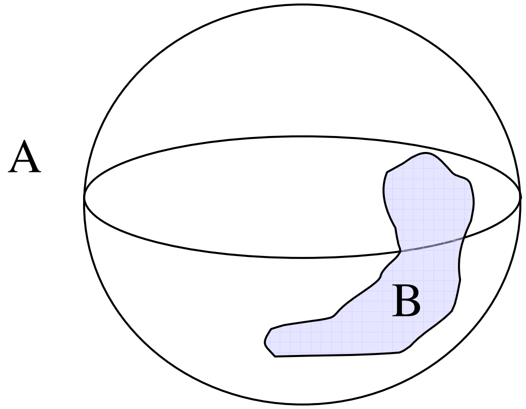
  - First stated (but proved incorrectly) by Camille Jordan (1838 -1922) in his *Cours d'Analyse*.

- Sketch of proof : (For full proof see Courant & Robbins, 1941.)
  - Show that any point in A can be joined to any other point in A by a path which does not cross C, and likewise for B.
  - Show that any path connecting a point in A to a point in B *must* cross C.

The Jordan curve theorem on a sphere

- Note that the Jordan curve theorem can be extended to a curve on a sphere, or anything which is topologically equivalent to a sphere.

“Any simple closed curve on a sphere separates the surface of the sphere into two distinct regions.”



Point-in-nonconvex-polygon

- *Winding number* (1980s)
  - The *winding number* of a point P in a curve C is the number of times that the curve wraps around the point.
  - For a simple closed curve (as any well-behaved polygon should be) this will be zero if the point is outside the curve, non-zero of it's inside.
  - The winding number is the sum of the angles from v^i to P to v^{i+1}.
    - Caveat: This method is elegant but slow.

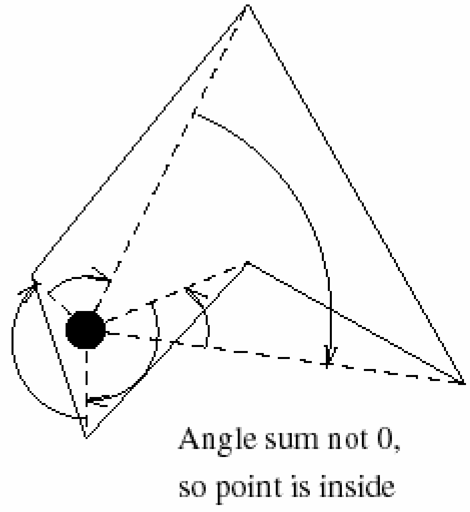
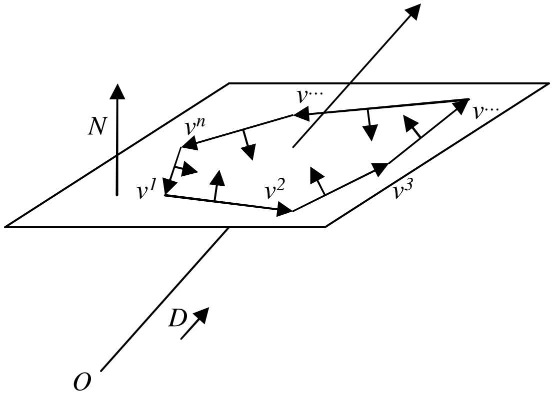
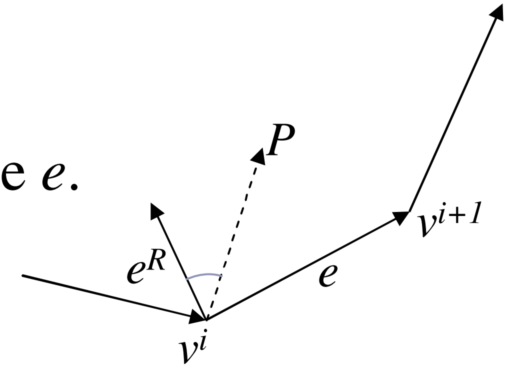


Figure from Eric Haines' "Point in Polygon Strategies", *Graphics Gems IV*, 1994

Point-in-convex-polygon

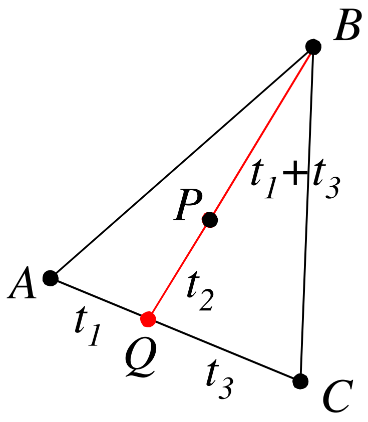
- Half-planes method
  - Each edge defines an infinite half-plane covering the polygon. If the point *P* lies in all of the half-planes then it must be in the polygon.
  - For each edge $e = v^i \rightarrow v^{i+1}$:
    - Rotate each edge 90˚ CCW around *N*.
    - If $e^R \bullet (P - v^i) < 0$ then the point is outside *e*.
  - Fastest known method.

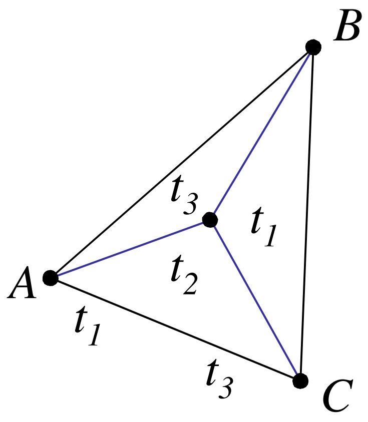




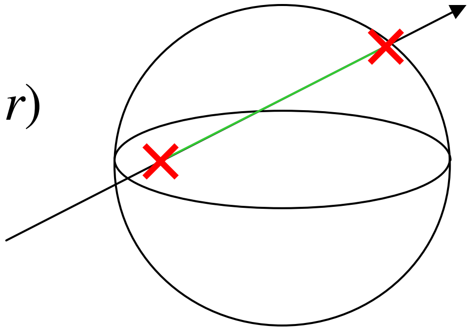
Barycentric coordinates

- *Barycentric coordinates* (t_1,t_2,t_3) are a coordinate system for describing the location of a point P inside a triangle (A,B,C).
- (t_1,t_2,t_3) are the ‘masses’ to be placed at (A,B,C) respectively so that the center of gravity of the triangle lies at P.
- Interestingly, (t_1,t_2,t_3) are also proportional to the subtriangle areas.





Ray/sphere intersection



Ray $R=O+tD$

Sphere $S=\{P \mid P \bullet P=r^2\}$ (centered at the origin; radius r)

$(O+tD) \bullet (O+tD) = r^2$

$(O_x+tD_x)^2 + (O_y+tD_y)^2 + (O_z+tD_z)^2 = r^2$

$(O_x{}^2+O_y{}^2+O_z{}^2) + 2t(O_xD_x+O_yD_y+O_zD_z) + t^2 (D_x{}^2+D_y{}^2+D_z{}^2) - r^2 = 0$

$t^2(D \bullet D) + 2t(O \bullet D) + (O \bullet O) - r^2 = 0$

Solve the quadratic at your leisure…

$t = (-(O \bullet D) \pm \sqrt{((O \bullet D)^2 - (D \bullet D)((O \bullet O) - r^2)))} / (D \bullet D)$

The normal on a sphere is easy: it's the point of intersection itself (normalized to unit length, of course.)

Primitives and world transforms

- Given a primitive P and its transform S, is it more efficient to find the intersection in screen space, world space or object space?
  - Not screen space: the transform from camera to screen co-ordinates is not affine, specifically it is not angle-preserving. This would prevent many nice optimizations, such as fast bounding box tests.
  - Our maths aren't optimized for world space; it would be nice to have each primitive encoded as statically as possible (ideally in assembler) with minimal parametrization.

Primitives and world transforms

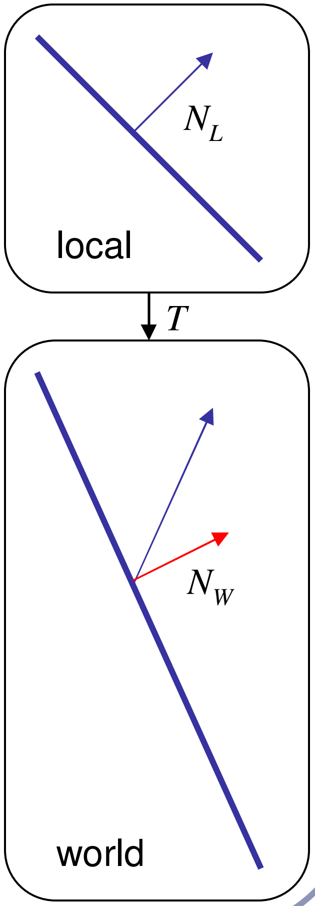
- Object space, then.
- Find $R = O+tD$ in object coords:
  - S is the local-to-world transform of P.
  - Invert S to find S^{-1}, the world-to-local transform.
  - Define $O_L=S^{-1}(O)$ and $D_L=S^{-1}(D)$.
  - The local ray: $R_L = O_\mathrm{L} + t'\mathrm{D}_\mathrm{L}$
  - Solve for t' and find the world hit point at $S(R_L(t'))$.
- Wyvill (1995) (Part 2, p.45) compares the floating-point ops required to hit a sphere with a ray in world or local coordinates. He found that it is actually 37% more efficient, per ray, to intersect in local space.

Primitives and world transforms

- What about the normal?
  - If S is just a concatenated sequence of rotates and translates then the normal can be transformed by S as above.
  - Scales make things trickier.
- To find the world-space normal, multiply the local normal by the *transpose of the inverse* of S:

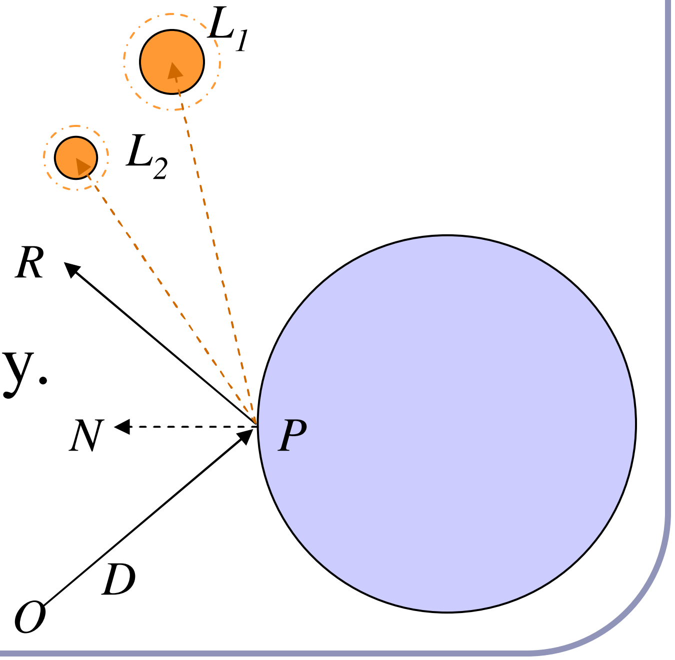
$$N=(S^{-1})^T N_L$$

  - Can ignore translations
  - For any rotation Q, $(Q^{-1})^T=Q$
  - Scaling is unaffected by transpose, and a scale of (a,b,c) becomes $(1/a,1/b,1/c)$ when inverted



Lighting revisited

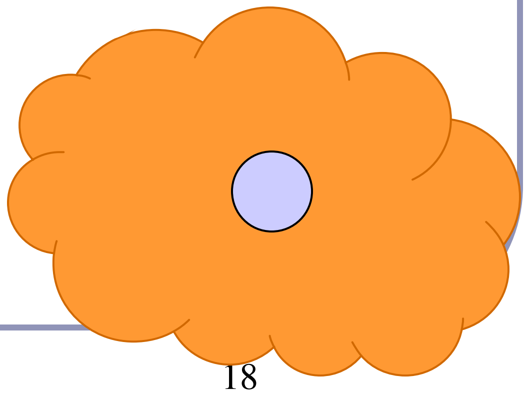
- We approximate lighting as the sum of the *ambient, diffuse,* and *specular* components of the light reflected to the eye.
  - Associate scalar parameters k_A, k_D and k_S with the surface.
  - Calculate diffuse and specular from each light source separately.



Lighting revisited—ambient lighting

- *Ambient* light is a flat scalar constant, L_A.
  - The amount of ambient light L_A is a parameter of the scene; the way it illuminates a particular surface is a parameter of the surface.
  - Some surfaces (ex: cotton wool) have high ambient coefficient k_A; others (ex: steel tabletop) have low k_A.
- Lighting intensity for ambient light alone:

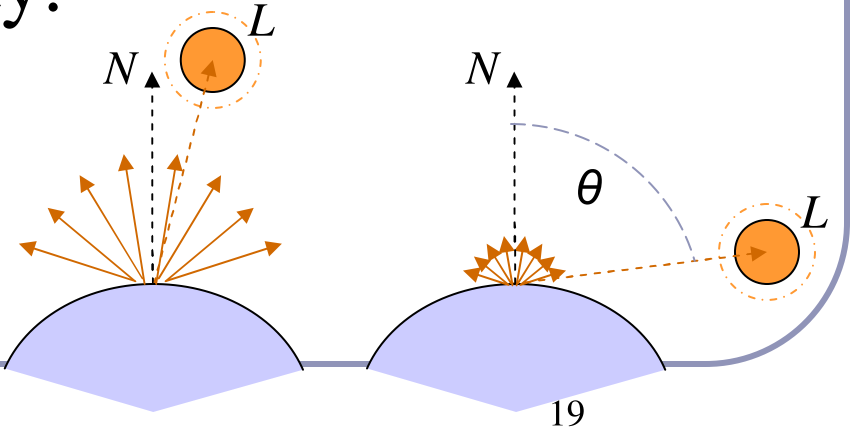
$$I_A(P) = k_A L_A$$



Lighting revisited—diffuse lighting

- The *diffuse* coefficient k_D measures how much light *scatters* off the surface.
  - Some surfaces (e.g. skin) have high k_D, scattering light from many microscopic facets and breaks. Others (e.g. ball bearings) have low k_D.
- Diffuse lighting intensity:

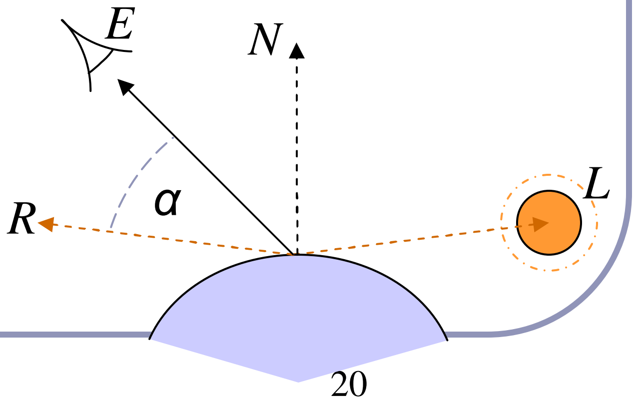
$$\begin{aligned} I_D(P) &= k_D L_D(\cos\theta) \\ &= k_D L_D(N \bullet L) \end{aligned}$$



Lighting revisited—specular lighting

- The *specular* coefficient k_S measures how much light *reflects* off the surface.
  - A ball bearing has high k_S; I don't.
  - 'Shininess' is approximated by a scalar power n.
- Specular lighting intensity:

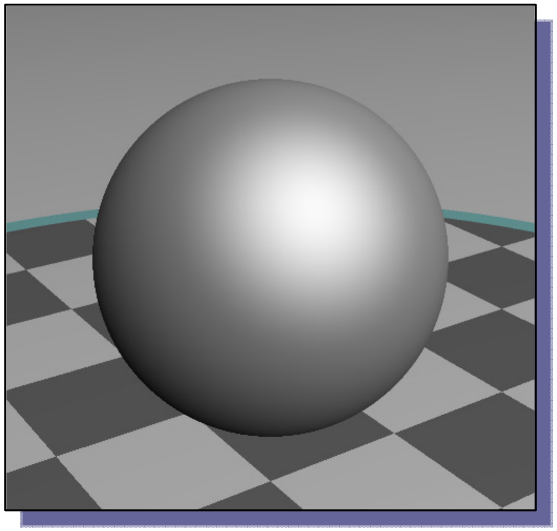
$$
\begin{aligned}
I_S(P) &= k_S L_S (\cos\alpha)^n \\
&= k_S L_S (R \bullet E)^n \\
&= k_S L_S ((2(\mathrm{L} \bullet \mathrm{N})\mathrm{N} - \mathrm{L}) \bullet E)^n
\end{aligned}
$$

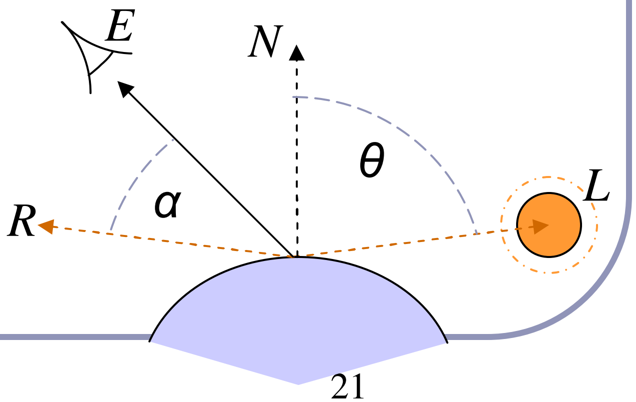


Lighting revisited—all together

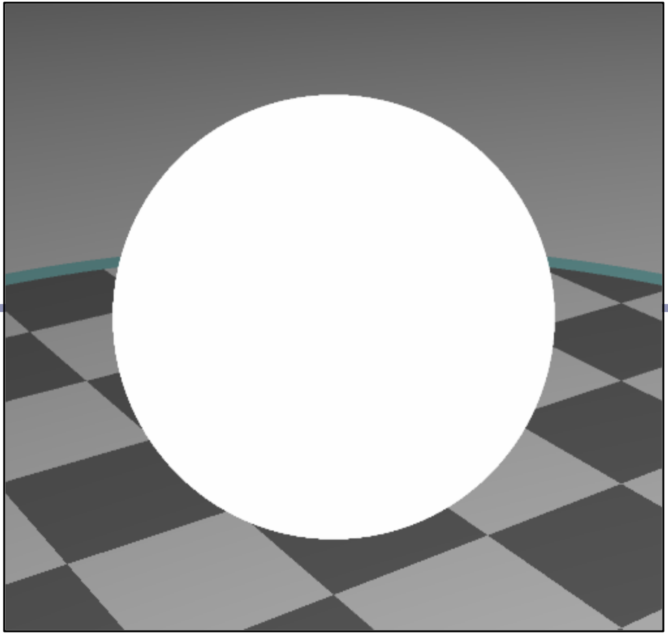
- The *total* illumination at P is therefore:

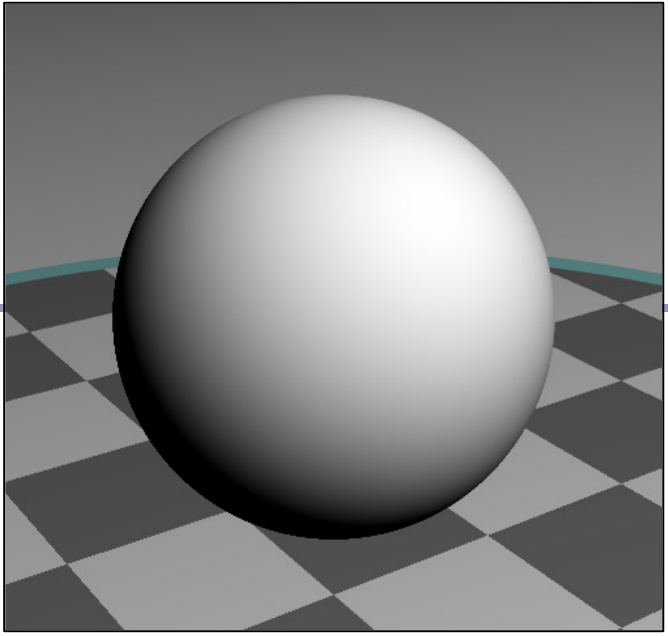
$$I(P) = k_A L_A + \sum_{Lights} k_D L_D (L \bullet N) + k_S L_S (R \bullet E)^n$$





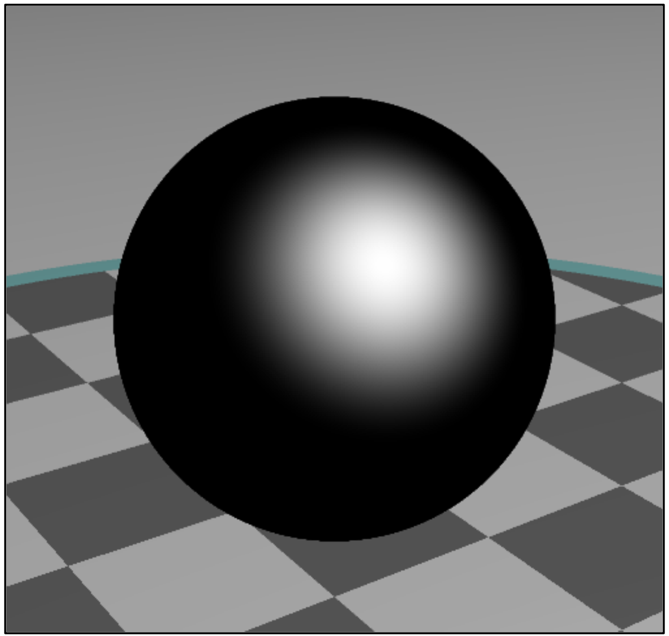
Ambient=1
Diffuse=0
Specular=0

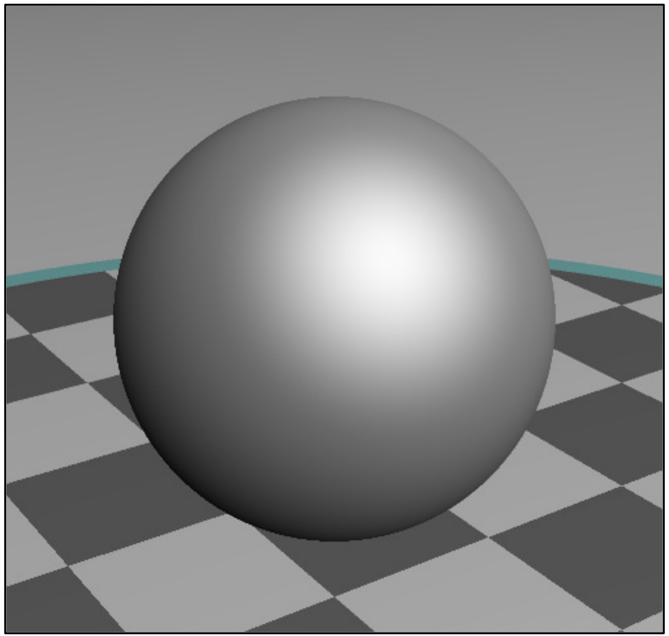




Ambient=0
Diffuse=1
Specular=0

Ambient=0
Diffuse=0
Specular=1
(n=2)

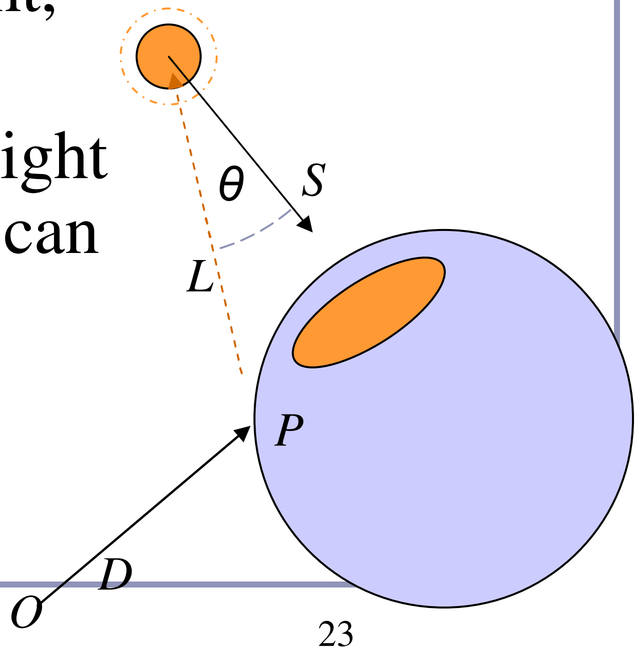




Ambient=0.2
Diffuse=0.4
Specular=0.4
(n=2)

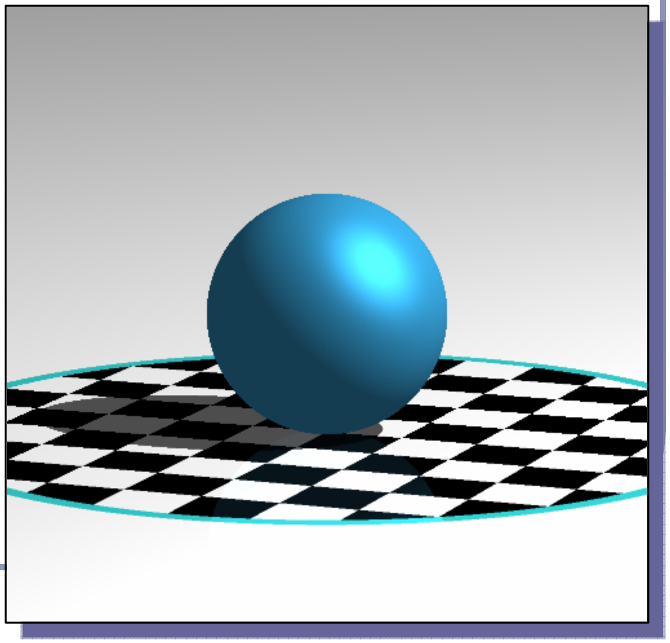
Spotlights

- To create a spotlight shining along axis S, you can multiply the (diffuse+specular) term by $(\max(L \bullet S, 0))^m$.
  - Raising m will tighten the spotlight, but leave the edges soft.
  - If you'd prefer a hard-edged spotlight of uniform internal intensity, you can use a conditional, e.g. $((L \bullet S > \cos(15^\circ))\ ?\ 1 : 0)$.



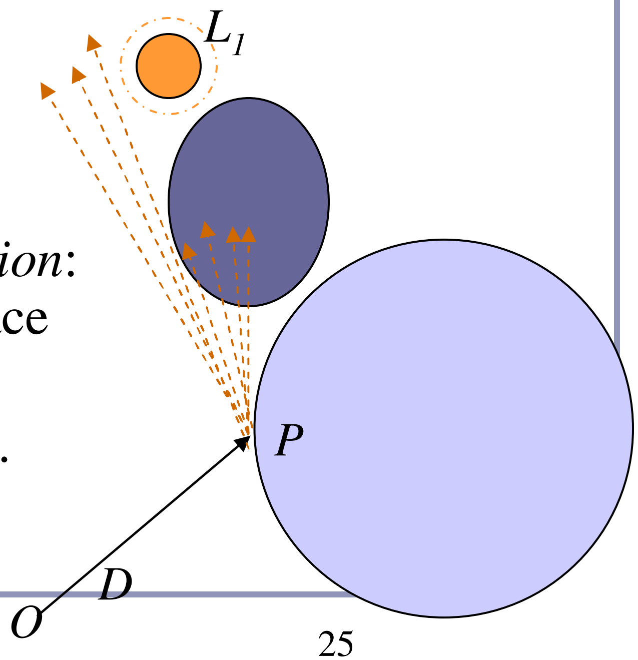
Ray tracing—Shadows

- To simulate shadow in ray tracing, fire a ray from *P* towards each light L_i. If the ray hits another object before the light, then discard L_i in the sum.
  - This is a boolean removal, so it will give *hard-edged* shadows.
    - Hard-edged shadows imply a pinpoint light source.



Softer shadows

- Shadows in nature are not sharp because light sources are not infinitely small.
  - Also because light scatters, etc.
- For lights with volume, fire many rays, covering the cross-section of your illuminated space.
- Illumination is (the total number of rays that aren't blocked) divided by (the total number of rays fired).
  - This is an example of *Monte-Carlo integration*: a coarse simulation of an integral over a space by randomly sampling it with many rays.
  - The more rays fired, the smoother the result.



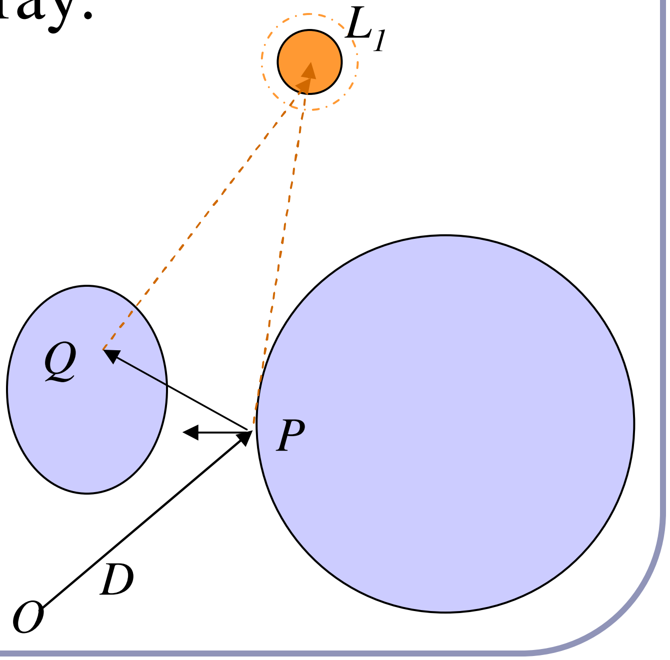
Reflection

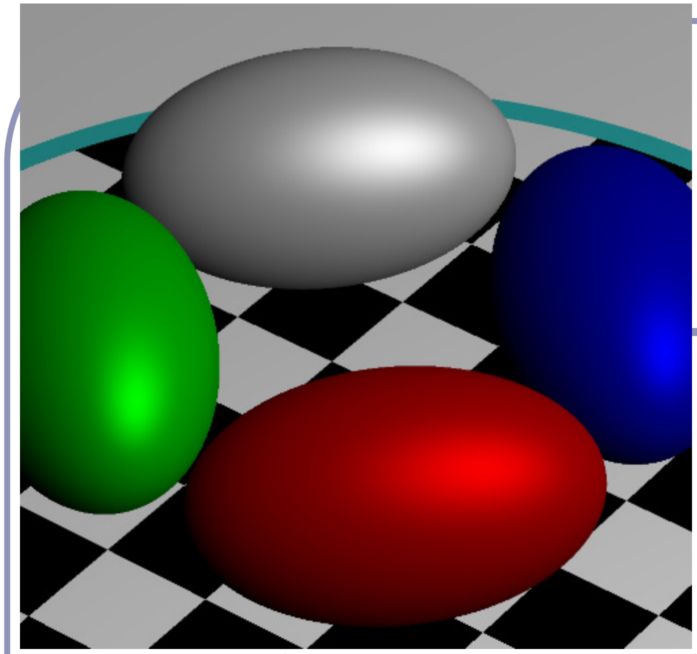
- *Reflection* rays are calculated by:

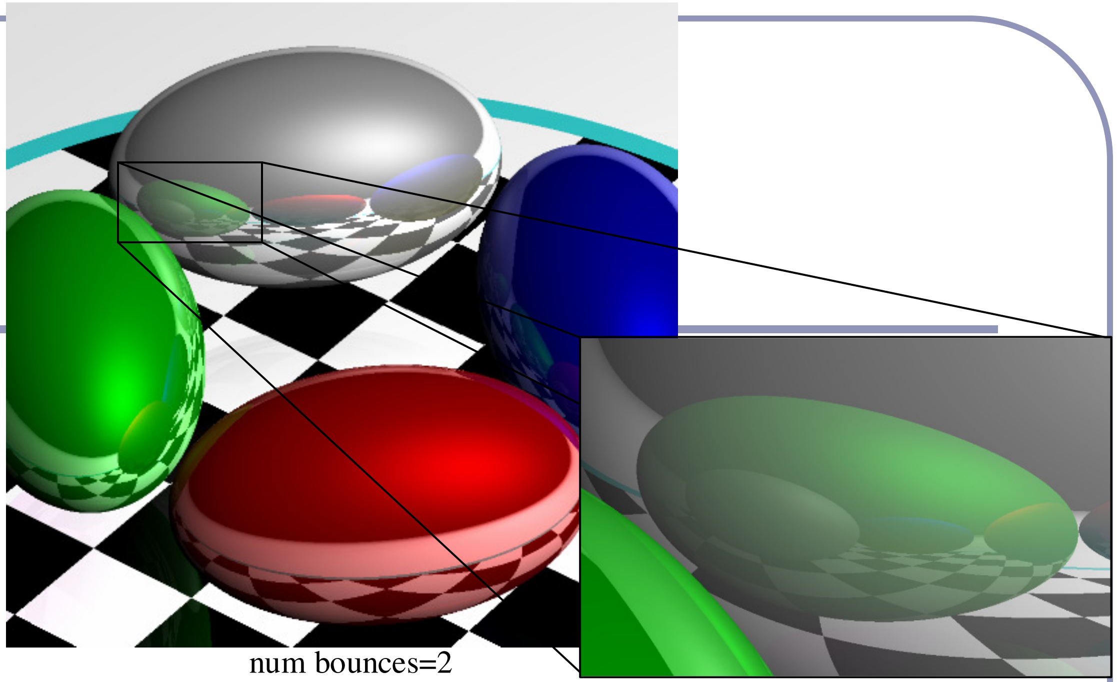
  $R = 2(-D \bullet N)N + D$

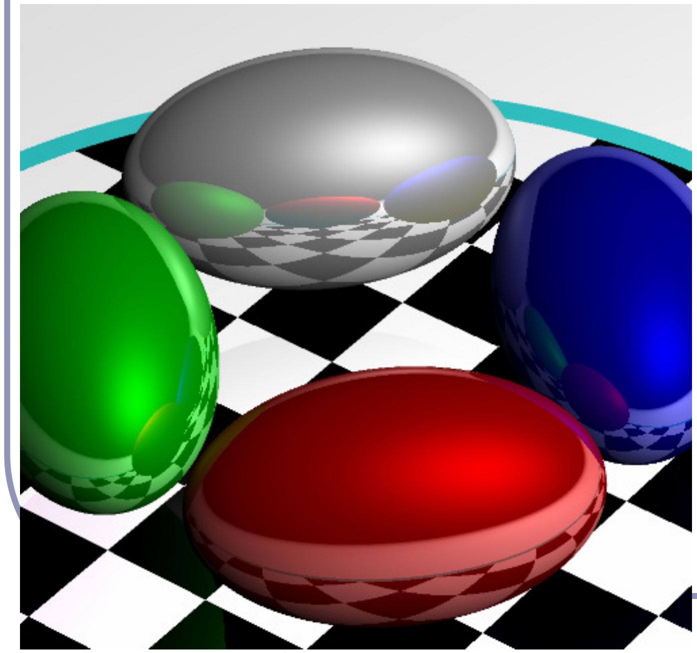
  …just like the specular reflection ray.

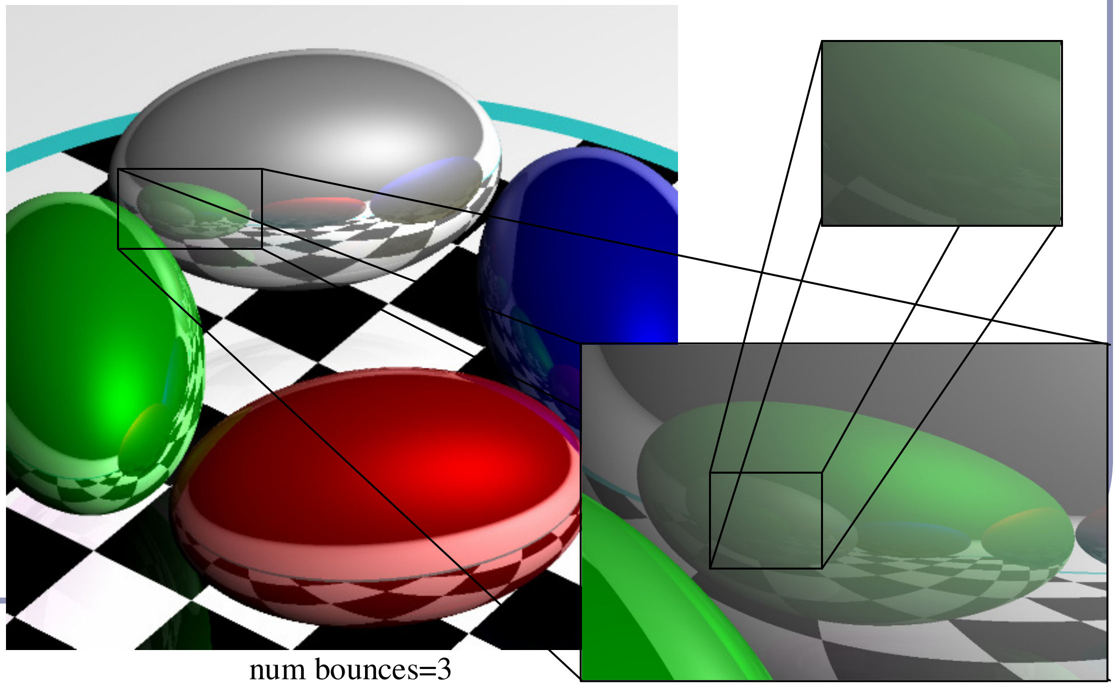
  - Finding the reflected color is a recursive raycast.
  - Reflection has *scene-dependant* performance impact.




num bounces=0

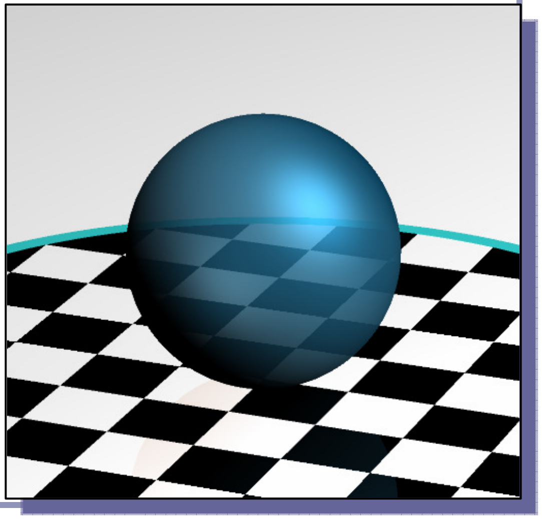

num bounces=2


num bounces=1


num bounces=3

Transparency

- To add transparency, generate and trace a new *transparency ray* with $O_T=P$, $D_T=D$.
- To support this in software, make color a 1x<u>4</u> vector where the fourth component, ‘alpha’, determines the weight of the recursed transparency ray.



Refraction

- *Snell's Law*:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1} = \frac{v_1}{v_2}$$

"The ratio of the sines of the *angles of incidence* of a ray of light at the interface between two materials is equal to the inverse ratio of the *refractive indices* of the materials is equal to the ratio of the speeds of light in the materials."

Historical note: this formula has been attributed to Willebrord Snell (1591-1626) and Rene' Descartes (1596-1650) but first discovery goes to Ibn Sahl (940-1000) of Baghdad.

Refraction

- The *angle of incidence* of a ray of light where it strikes a surface is the acute angle between the ray and the surface normal.
- The *refractive index* of a material is a measure of how much the speed of light[1] is reduced inside the material.
  - The refractive index of air is about 1.003.
  - The refractive index of water is about 1.33.

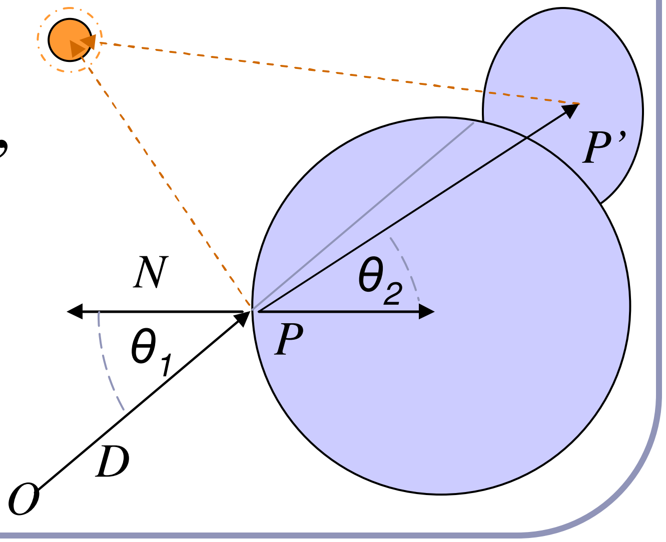
[1] Or sound waves or other waves

Refraction in ray tracing

$$\theta_1 = \cos^{-1}(N \bullet D)$$

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1} \quad \rightarrow \quad \theta_2 = \sin^{-1}\left(\frac{n_1}{n_2}\sin\theta_1\right)$$

- Using Snell's Law and the angle of incidence of the incoming ray, we can calculate the angle from the negative normal to the outbound ray.

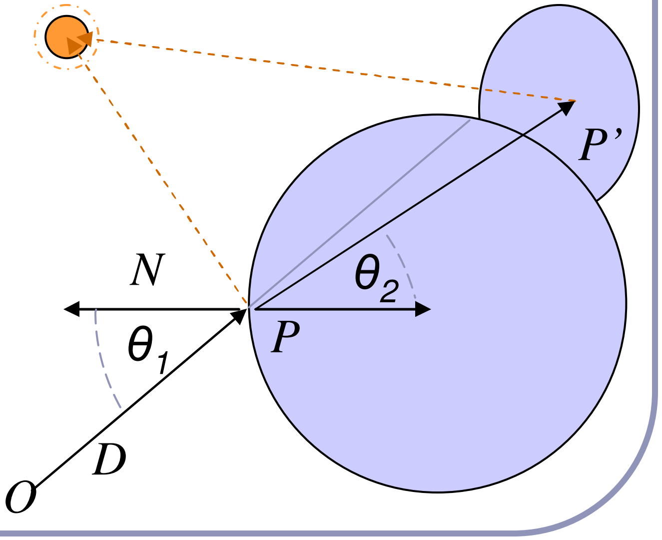


Refraction in ray tracing

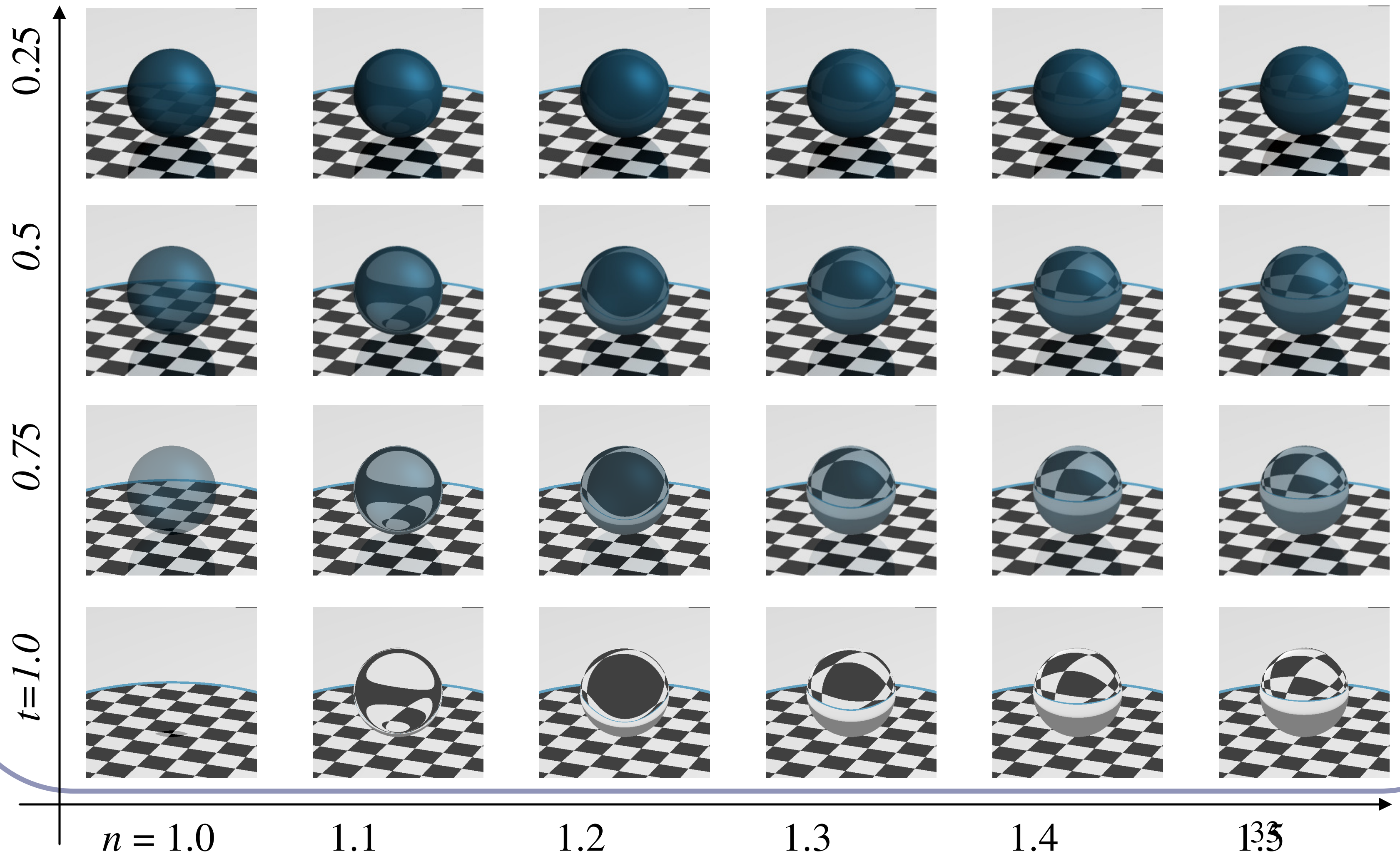
- What if the arcsin parameter is > 1?
  - Remember, arcsin is defined in [-1,1].
- We call this the *angle of total internal reflection*, where the light becomes trapped completely inside the surface.

$$\theta_2 = \sin^{-1}\left(\frac{n_1}{n_2}\sin\theta_1\right)$$

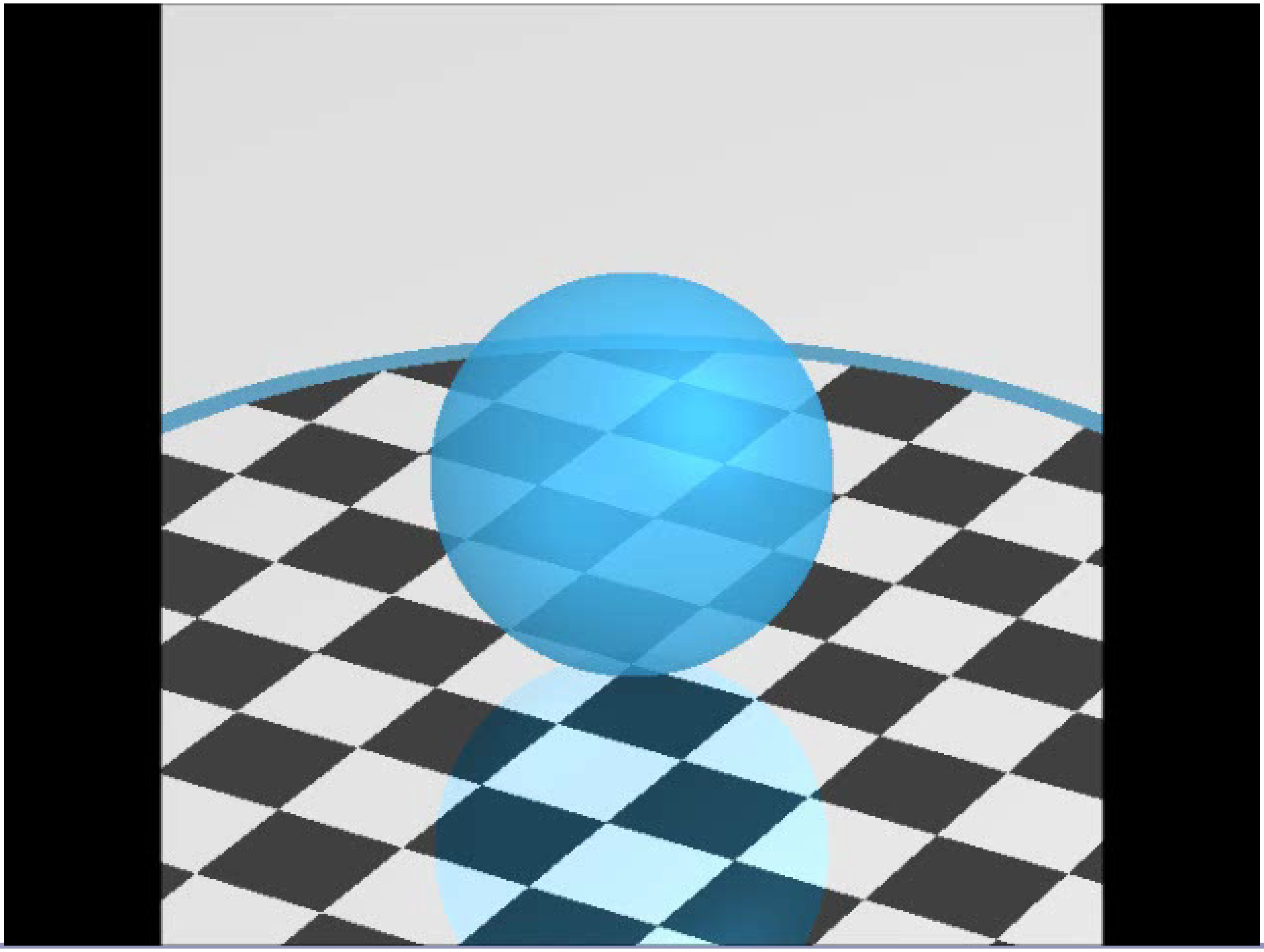
Total internal reflection



Refractive index vs transparency



Refraction in action



References

Jordan curves

- R. Courant, H. Robbins, *What is Mathematics?*, Oxford University Press, 1941
- http://cgm.cs.mcgill.ca/~godfried/teaching/cg-projects/97/Octavian/compgeom.html

Point-in-polygon

- http://tog.acm.org/editors/erich/ptinpoly/
- http://mathworld.wolfram.com/BarycentricCoordinates.html

Ray tracing

- Foley & van Dam, *Computer Graphics* (1995)
- Jon Genetti and Dan Gordon, *Ray Tracing With Adaptive Supersampling in Object Space*, http://www.cs.uaf.edu/~genetti/Research/Papers/GI93/GI.html (1993)
- Zack Waters, “Realistic Raytracing”, http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/realistic_raytracing.html