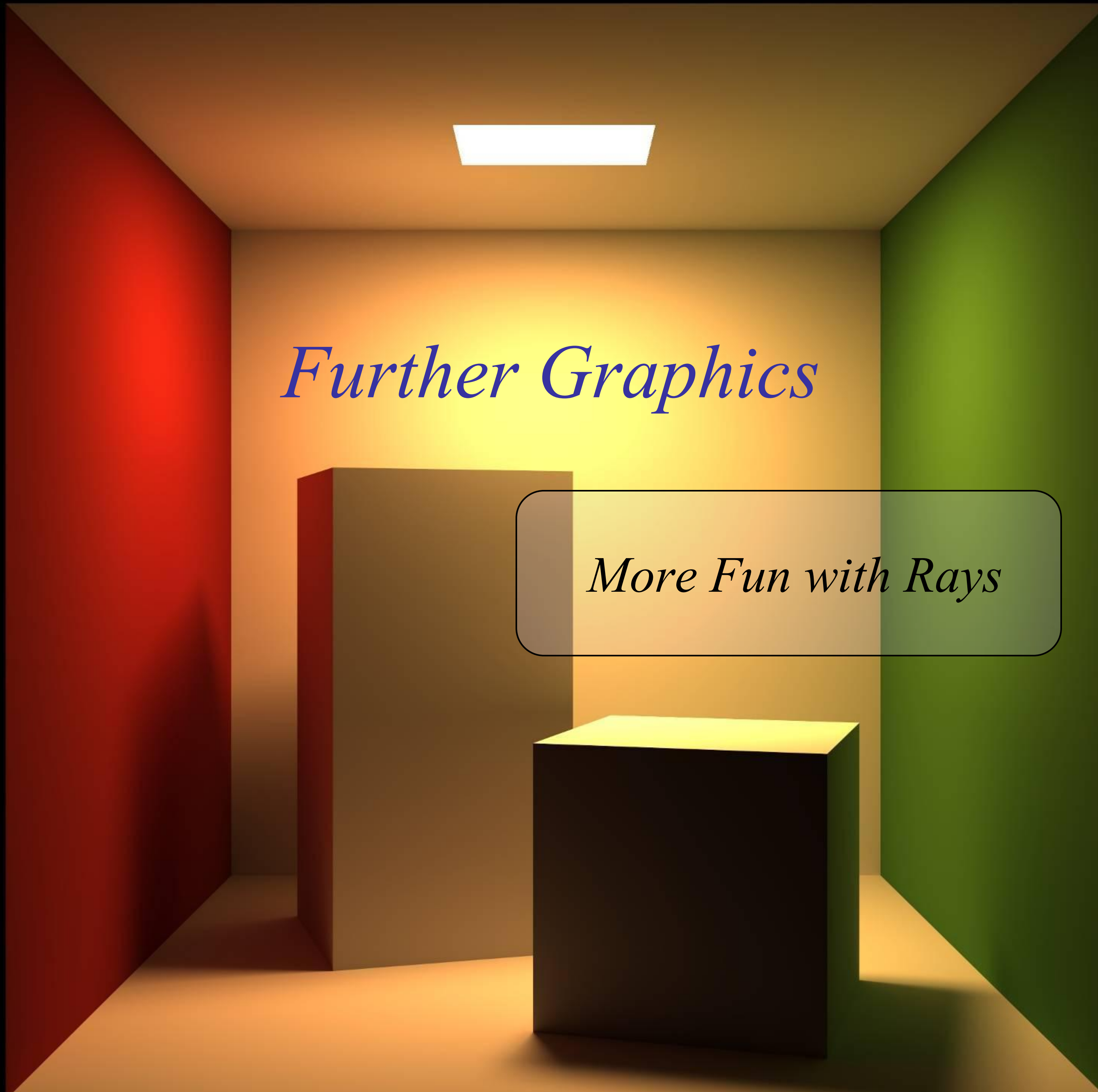
Further Graphics

More Fun with Rays

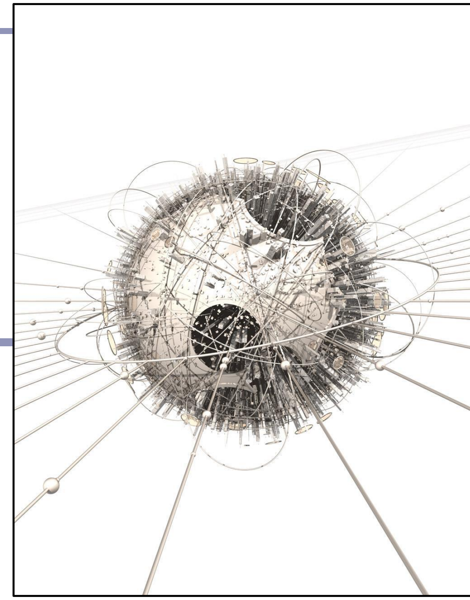


"*Cornell Box*" by Steven Parker, University of Utah.
A tera-ray monte-carlo rendering of the Cornell Box, generated in 2 CPU years on an Origin 2000. The full image contains 2048 x 2048 pixels with over 100,000 primary rays per pixel (317 x 317 jittered samples). Over one trillion rays were traced in the generation of this image.

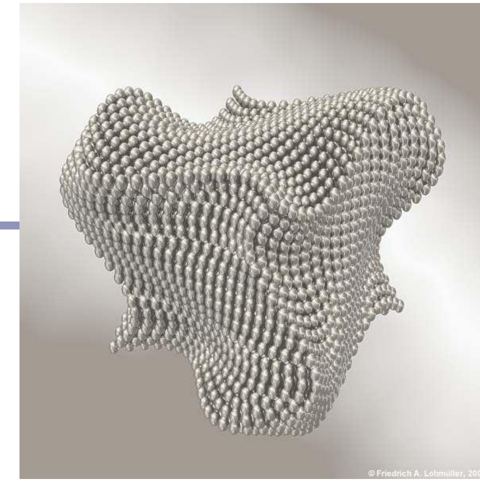
Alex Benton, University of Cambridge – alex@bentonian.com
Supported in part by Google UK, Ltd

Examples

"Scherk-Collins sculpture" by Trevor G. Quayle (2008)



"POV Planet" by Casey Uhrig (2004)



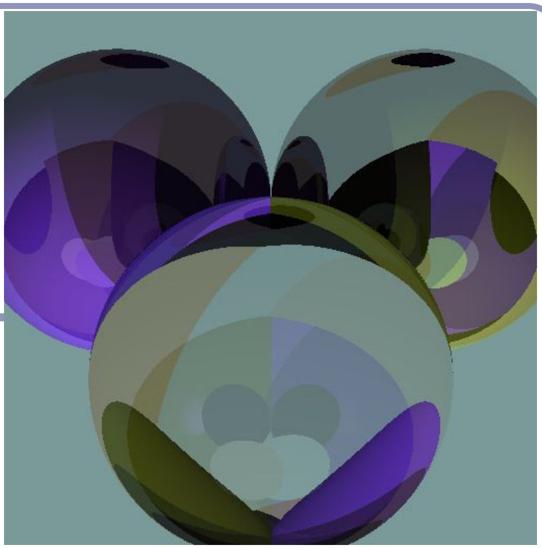
"Dancing Cube" by Friedrich A. Lohmueller (2003)

"Villarceau Circles" by Tor Olav Kristensen (2004)

"Glasses" by Gilles Tran (2006)

All images are from the POV-Ray Hall of Fame: hof.povray.org

Ray-tracing / ray-marching: It doesn’t take much code



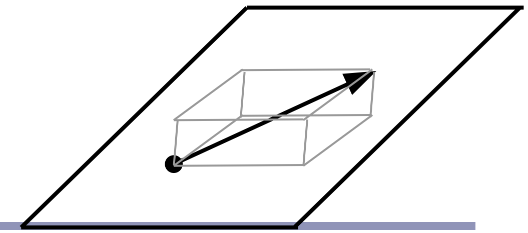
The basic algorithm is straightforward, but there's much room for subtlety

- 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,};int yx;double u,b,tmin,sqrt(),tan();double
vdot(vec A,vec B){return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(
double a,vec A,vec B){B.x+=a*A.x;B.y+=a*A.y;B.z+=a*A.z;return
B;}vec vunit(vec A){return vcomb(1./sqrt(vdot(A,A)),A,black);}
struct sphere*intersect(vec P,vec D){best=0;tmin=10000;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):10000,u=b-u>0.000001?b-u:b+u,tmin=
u>0.00001&&u<tmin?best=s,u:tmin;return best;}vec trace(int
level,vec P,vec 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(){int
d=512;printf("%d %d\n",d,d);while(yx<d*d){U.x=yx%d-d/2;U.z=d/2-
yx++/d;U.y=d/2/tan(25/114.5915590261);U=vcomb(255.,trace(3,
black,vunit(U)),black);printf("%0.f %0.f %0.f\n",U.x,U.y,U.z);}
}/*minray!*/
```

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

Hitting things with rays



A ray is defined parametrically as

$$P(t) = E + tD,\ t \geq 0 \qquad (\alpha)$$

where E is the ray's origin (our eye position) and D is the ray's direction, a unit-length vector.

We can expand this equation to three dimensions, *x*, *y* and *z*:

$$\left.\begin{aligned} x(t) &= x_E + tx_D \\ y(t) &= y_E + ty_D \\ z(t) &= z_E + tz_D \end{aligned}\right\}\ t \geq 0 \qquad (\beta)$$

Hitting things with rays: Planes and polygons

A planar polygon P can be defined as

Polygon $P = \{v_1, \ldots, v_n\}$

which gives us the normal to P as

$$N = (v_n - v_1) \times (v_2 - v_1)$$

The equation for the plane of P is

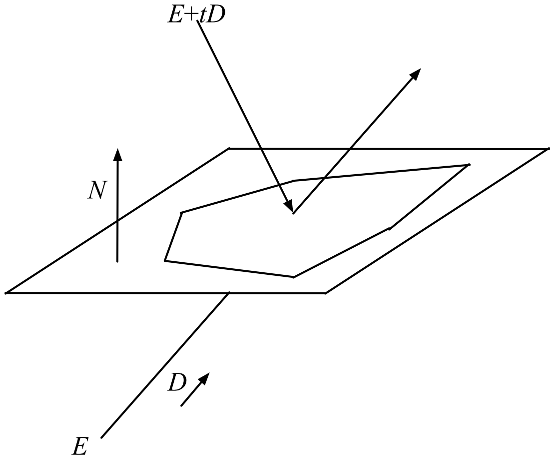
$$N \bullet (p - v_1) = 0$$

Substituting equation (α) for p yields

$$N \bullet (E + tD - v_1) = 0$$

$$x_N(x_E + tx_D - x_{v\_1}) + y_N(y_E + ty_D - y_{v\_1}) + z_N(z_E + tz_D - z_{v\_1}) = 0$$

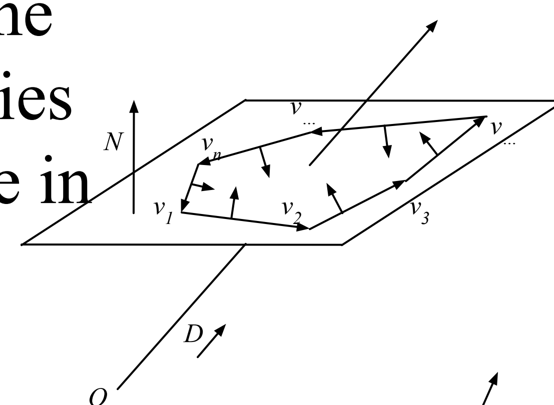
$$t = \frac{(N \bullet v1) - (N \bullet E)}{N \bullet D}$$

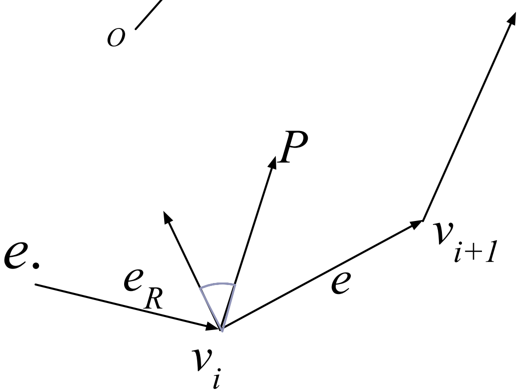


Point in convex polygon

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 e by 90° CCW around N.
    - Do this quickly by crossing N with e.
  - If $e_R \cdot (P - v_i) < 0$ then the point is outside e.
- Fastest known method.

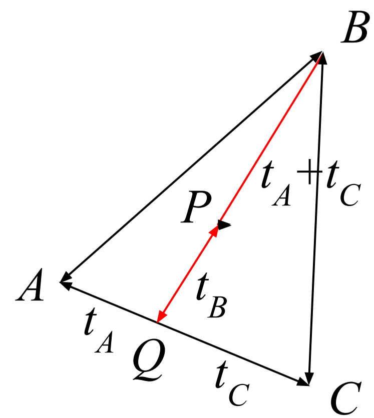


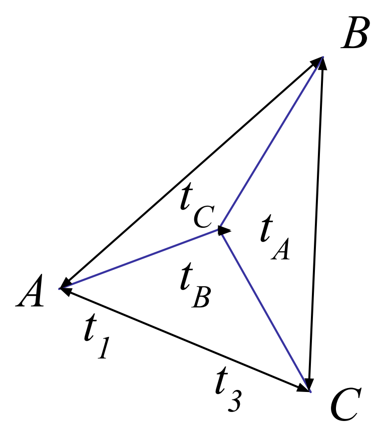


Barycentric coordinates

Barycentric coordinates (t_A, t_B, t_C) are a coordinate system for describing the location of a point P inside a triangle (A,B,C).

- You can think of (t_A, t_B, t_C) as 'masses' placed at (A,B,C) respectively so that the center of gravity of the triangle lies at P.
- (t_A, t_B, t_C) are proportional to the subtriangle areas of the three vertices.
  - The area of a triangle is ½ the length of the cross product of two of its sides.





Barycentric coordinates

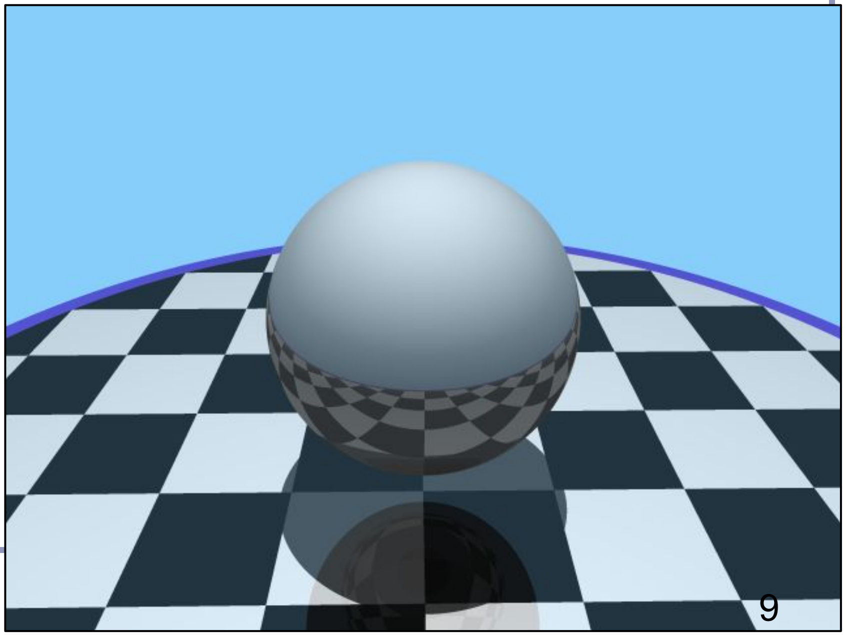
```
// Compute barycentric coordinates (u, v, w) for
// point p with respect to triangle (a, b, c)
vec3 barycentric(vec3 p, vec3 a, vec3 b, vec3 c) {
  vec3 v0 = b - a, v1 = c - a, v2 = p - a;
  float d00 = dot(v0, v0);
  float d01 = dot(v0, v1);
  float d11 = dot(v1, v1);
  float d20 = dot(v2, v0);
  float d21 = dot(v2, v1);
  float denom = d00 * d11 - d01 * d01;
  float v = (d11 * d20 - d01 * d21) / denom;
  float w = (d00 * d21 - d01 * d20) / denom;
  float u = 1.0 - v - w;
  return vec3(u, v, w);
}
```

Code credit: Christer Ericson, *Real-Time Collision Detection* (2004)
(adapted to GLSL for this lecture)

Hard shadows

To simulate shadows with rays, 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 suggest a pinpoint light source.



Softer shadows

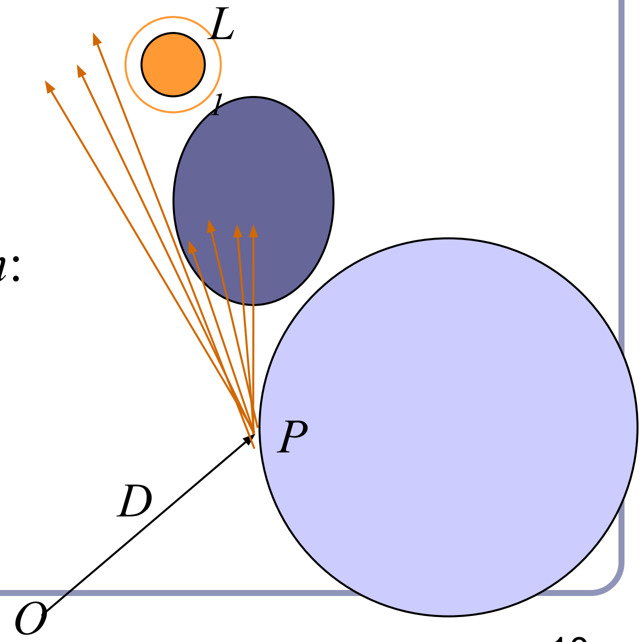
Shadows in nature are not sharp because light sources are not infinitely small.

- Also because light scatters, etc.

For lights with volume, fire many rays, covering the cross-section of your illuminated space.

Illumination is scaled by (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.



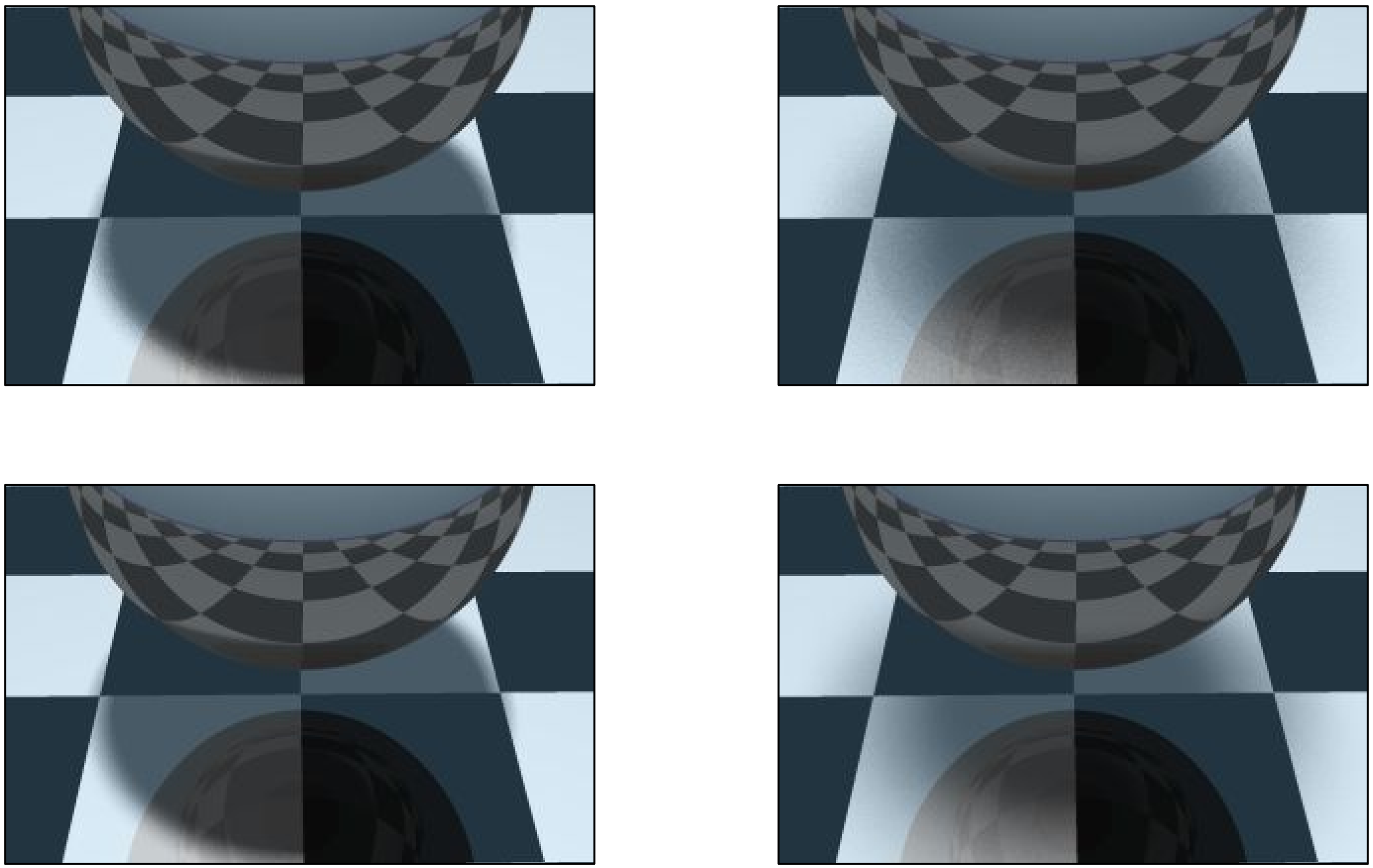
Softer shadows

Light radius: 1

Light radius: 5

Rays per shadow test: 20

Rays per shadow test: 100

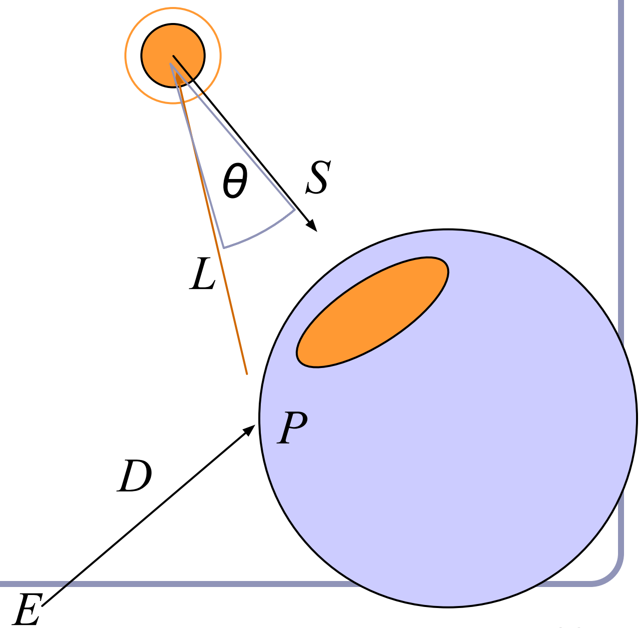


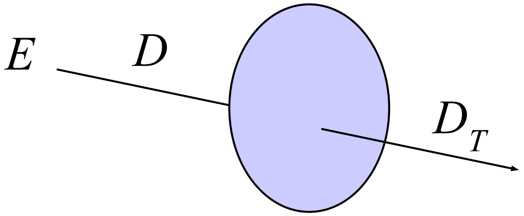
All images anti-aliased with 4x supersampling.
Distance to light in all images: 20 units

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. $((L \bullet S > \cos(15^\circ))\ ?\ 1 : 0)$.





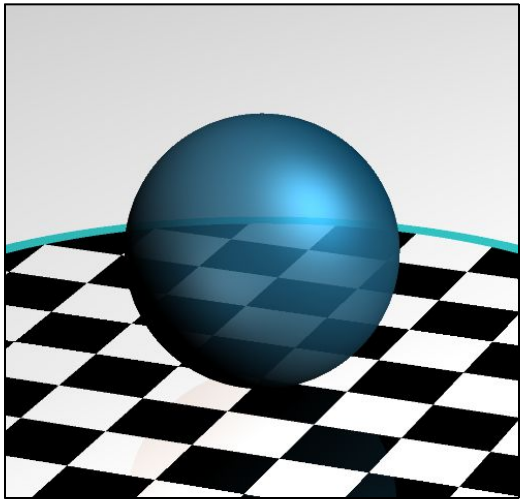
Transparency and Refraction

To add transparency, generate and trace a new *transparency ray* with $E_T = P$, $D_T = D$.

For realism, D_T should deviate (slightly) from D. 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

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."

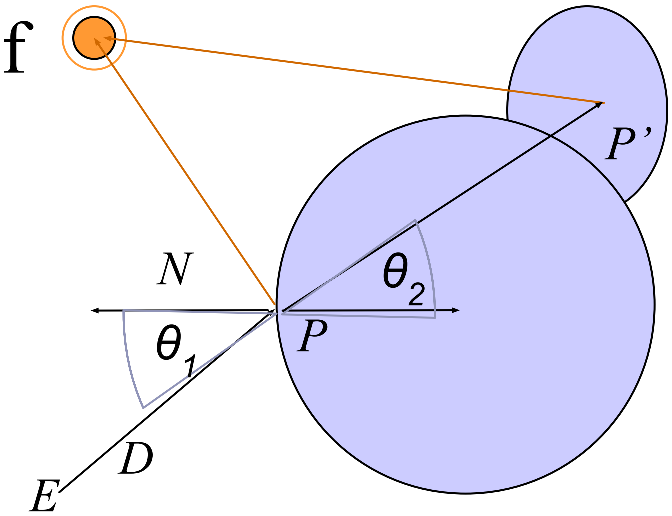
Historical note: this formula has been attributed to Willebrord Snell (1591-1626) and René Descartes (1596-1650) but first discovery goes to Ibn Sahl (940-1000) of Baghdad.

Refraction for rays

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

$$\frac{sin\,\theta_1}{sin\,\theta_2} = \frac{n_2}{n_1} \;\rightarrow\; \theta_2 = sin^{-1}(\frac{n_1}{n_2} sin\,\theta_1)$$

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.



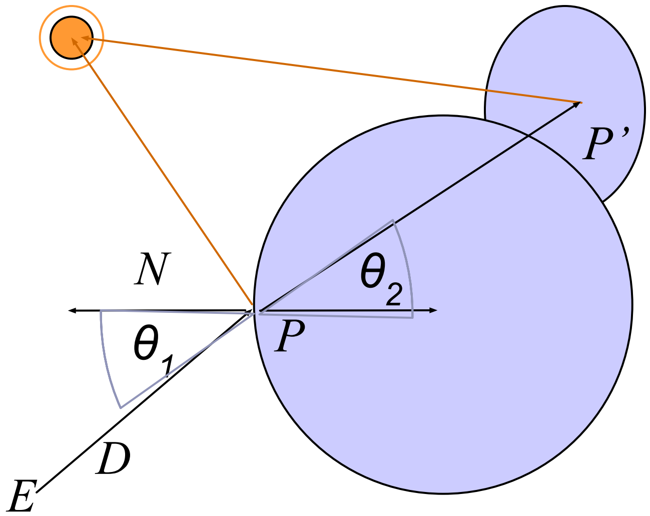
Refraction in ray tracing

What if the arcsin parameter is > 1?

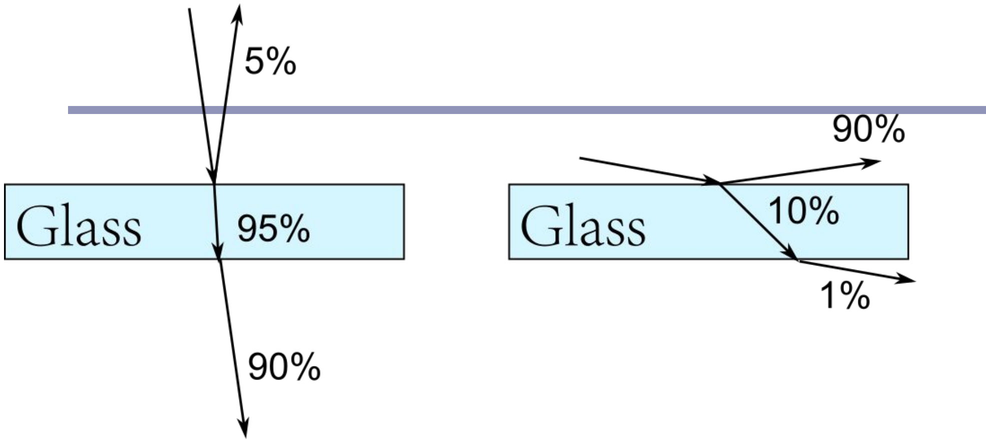
- Remember, arcsin is defined in [-1,1].
- We call this the *angle of total internal reflection*: light is trapped completely inside the surface.

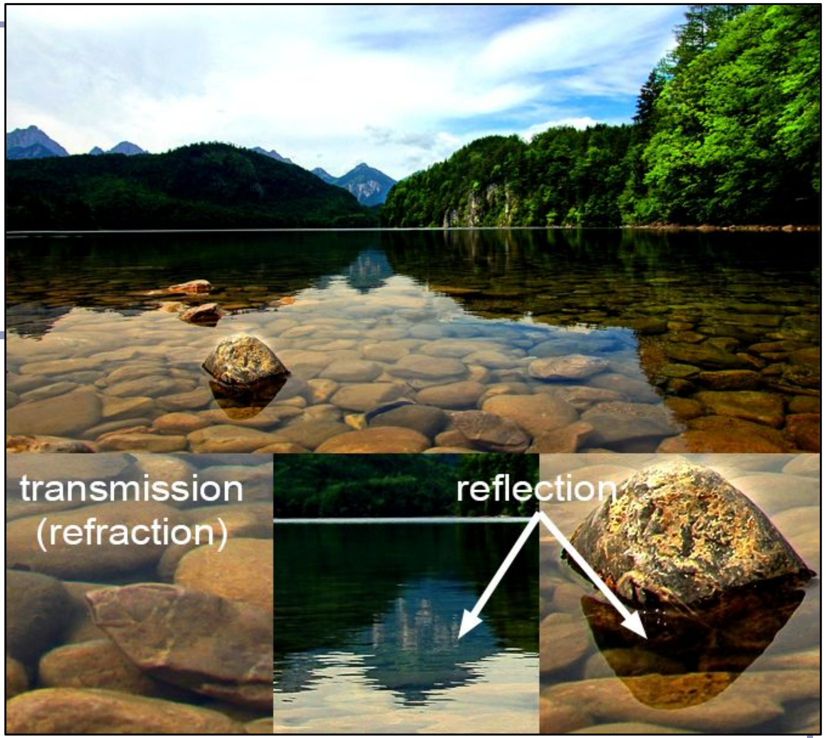
$$\theta_2 = sin^{-1}\left(\frac{n_1}{n_2} sin\,\theta_1\right)$$

Total internal reflection



Fresnel term





Example from:
https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel

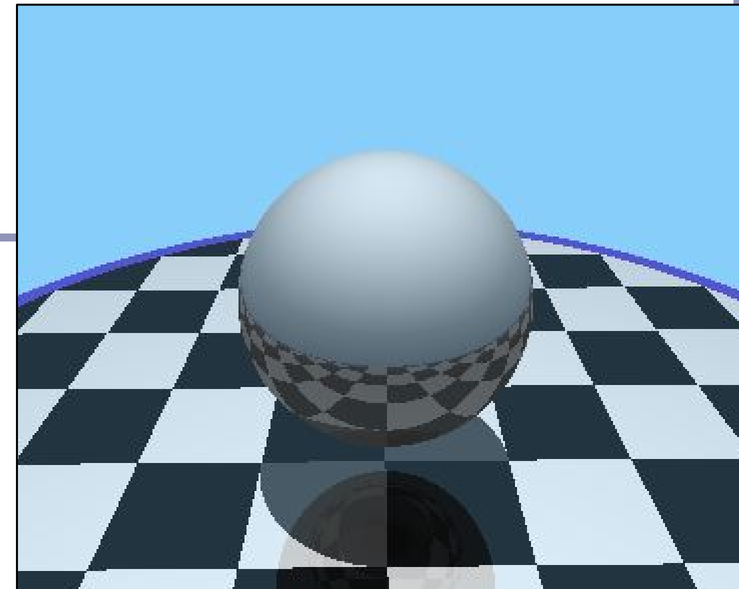
- Light is more likely to be reflected rather than transmitted near grazing angles
- This effect is modelled by *Fresnel equation*, which gives the probability that a photon is reflected rather than transmitted (or absorbed)

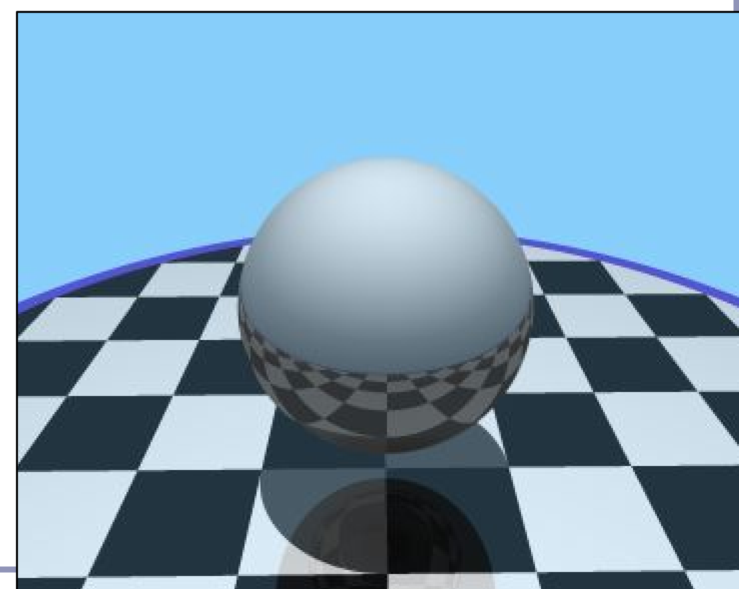
Aliasing

aliasing
/ˈeɪlɪəsɪŋ/
noun: **aliasing**
1. PHYSICS / TELECOMMUNICATIONS

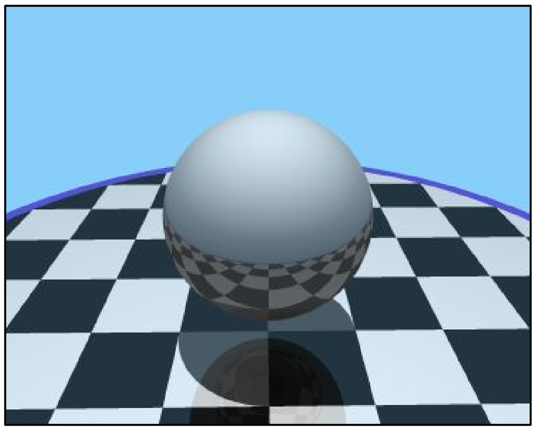
the misidentification of a signal frequency, introducing distortion or error.
"high-frequency sounds are prone to aliasing"
2. COMPUTING

the distortion of a reproduced image so that curved or inclined lines appear inappropriately jagged, caused by the mapping of a number of points to the same pixel.

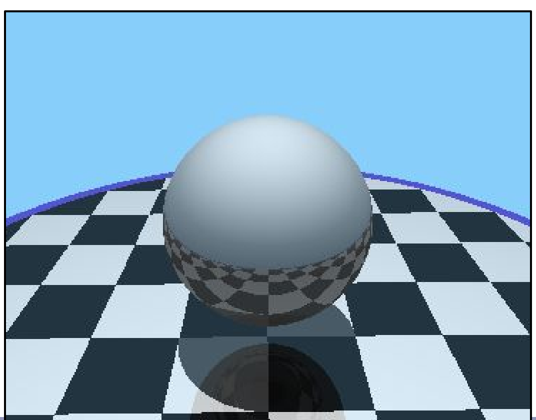




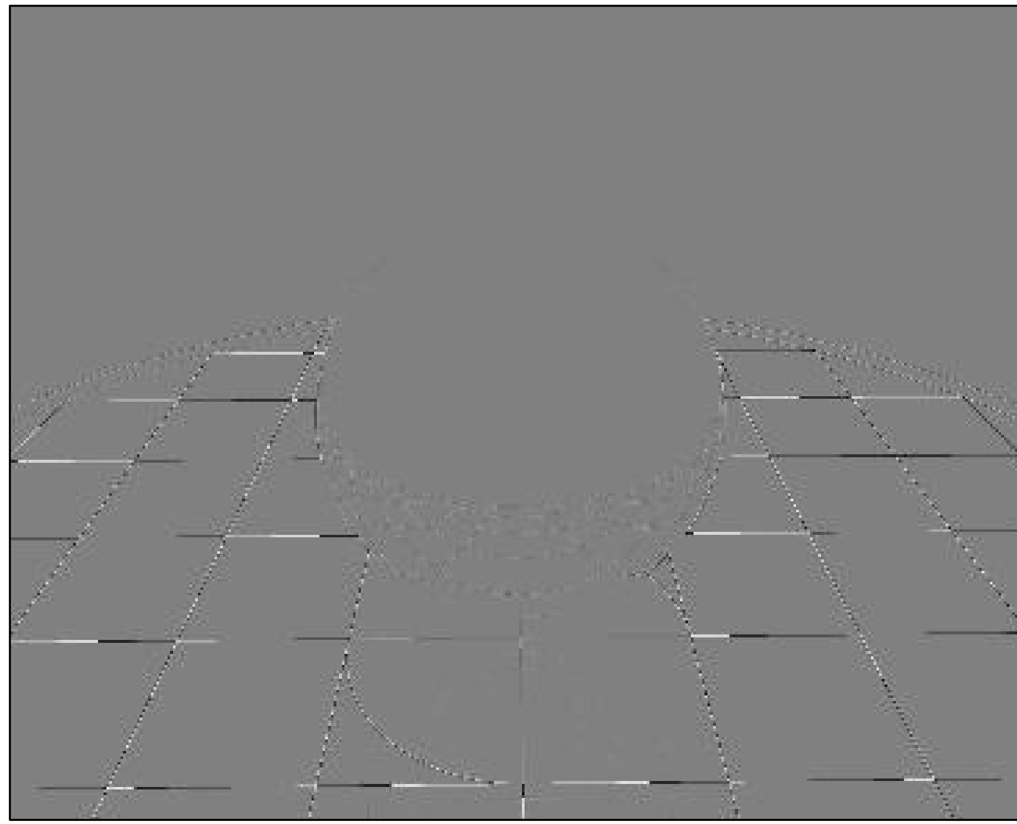
Aliasing



-



=



Anti-aliasing

Fundamentally, the problem with aliasing is that we're sampling an infinitely continuous function (the color of the scene) with a finite, discrete function (the pixels of the image).

One solution to this is *super-sampling*. If we fire multiple rays through each pixel, we can average the colors computed for every ray together to a single blended color.

To avoid heavy computational load And also avoid sub-super-sampling artifacts, consider using *jittered super-sampling*.

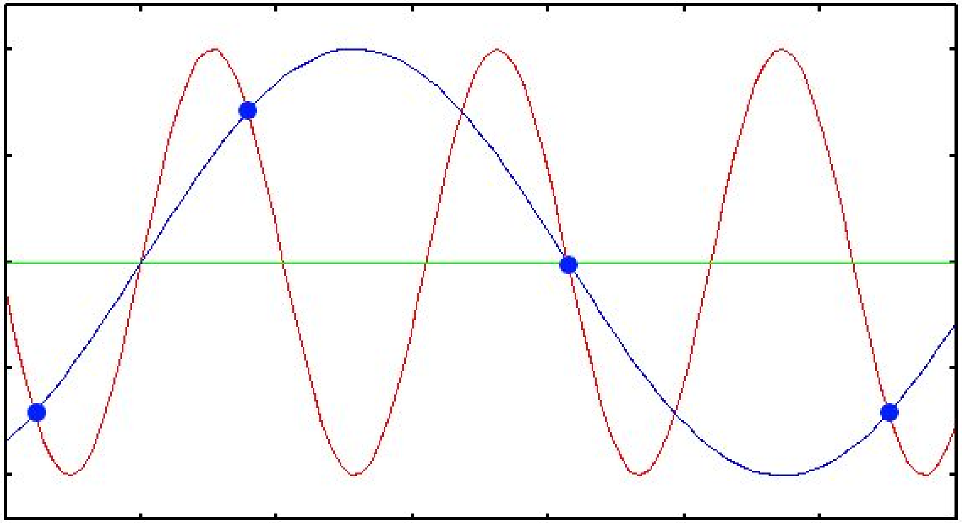
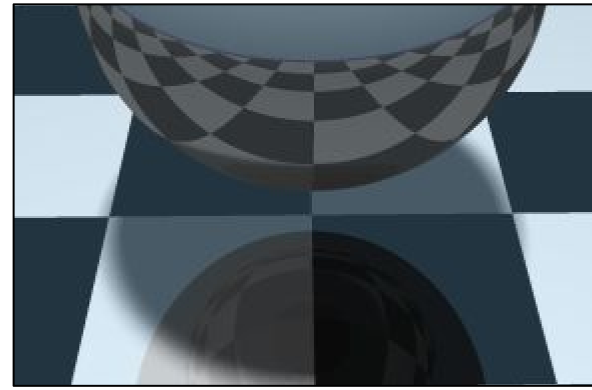
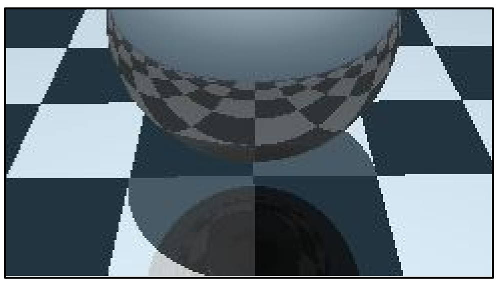


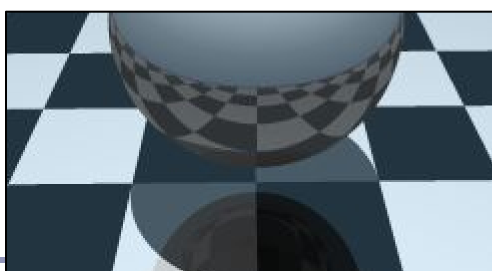
Image source: www.svi.nl

Applications of super-sampling

- Anti-aliasing
- Soft shadows
- Depth-of-field camera effects (fixed focal depth, finite aperture)







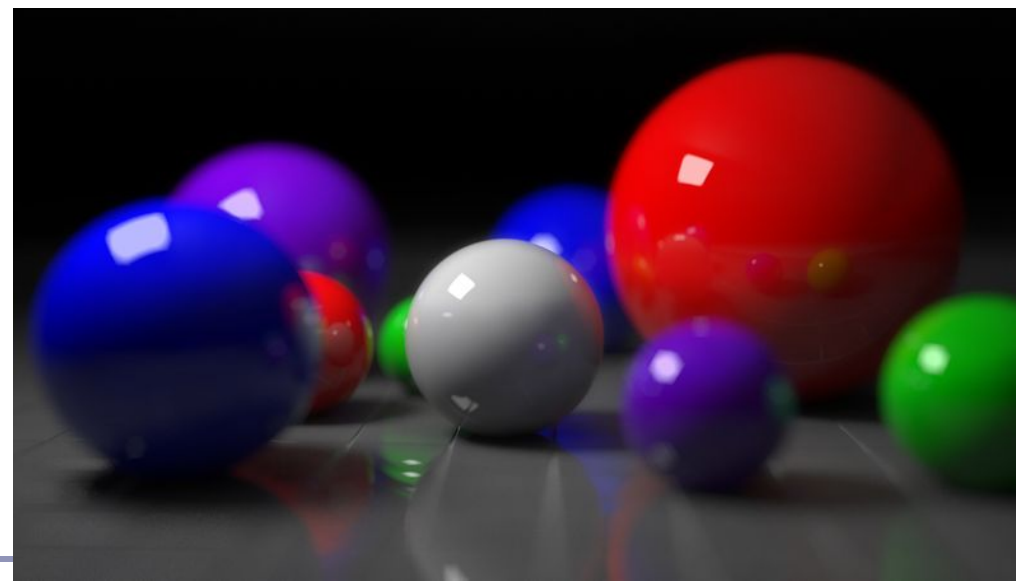


Image credit:
http://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Speed things up!
Bounding volumes

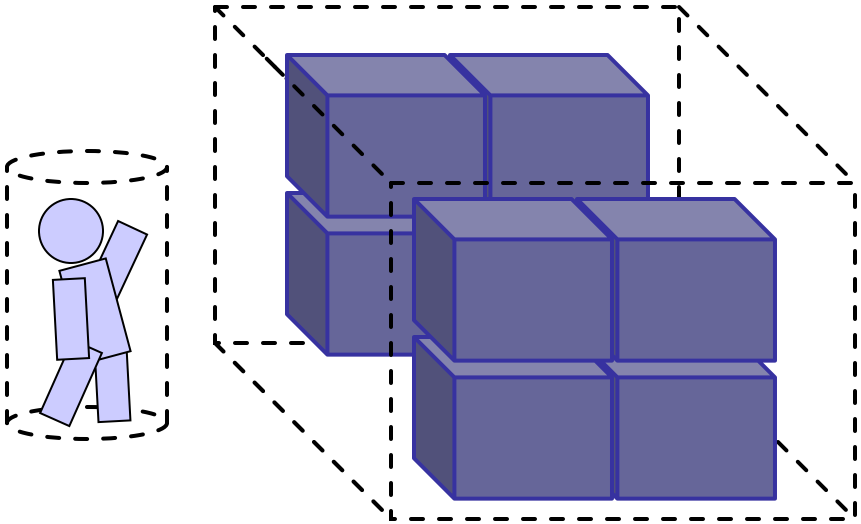
A common optimization method for ray-based rendering is the use of *bounding volumes*.

Nested bounding volumes allow the rapid culling of large portions of geometry

- Test against the bounding volume of the top of the scene graph and then work down.

Great for…

- Collision detection between scene elements
- Culling before rendering
- Accelerating ray-tracing, -marching



Types of bounding volumes

The goal is to accelerate volumetric tests, such as “does the ray hit the cow?” → *speed* trumps *precision*

- choose fast hit testing over accuracy
- ‘bboxes’ don’t have to be tight

Axis-aligned bounding boxes

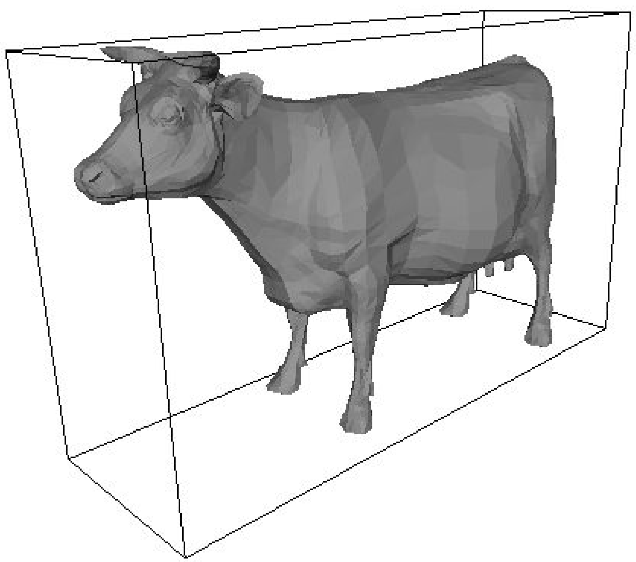
- max and min of x/y/z.

Bounding spheres

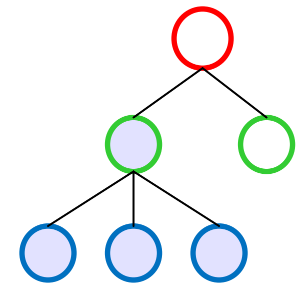
- max of radius from some rough center

Bounding cylinders

- common in early FPS games

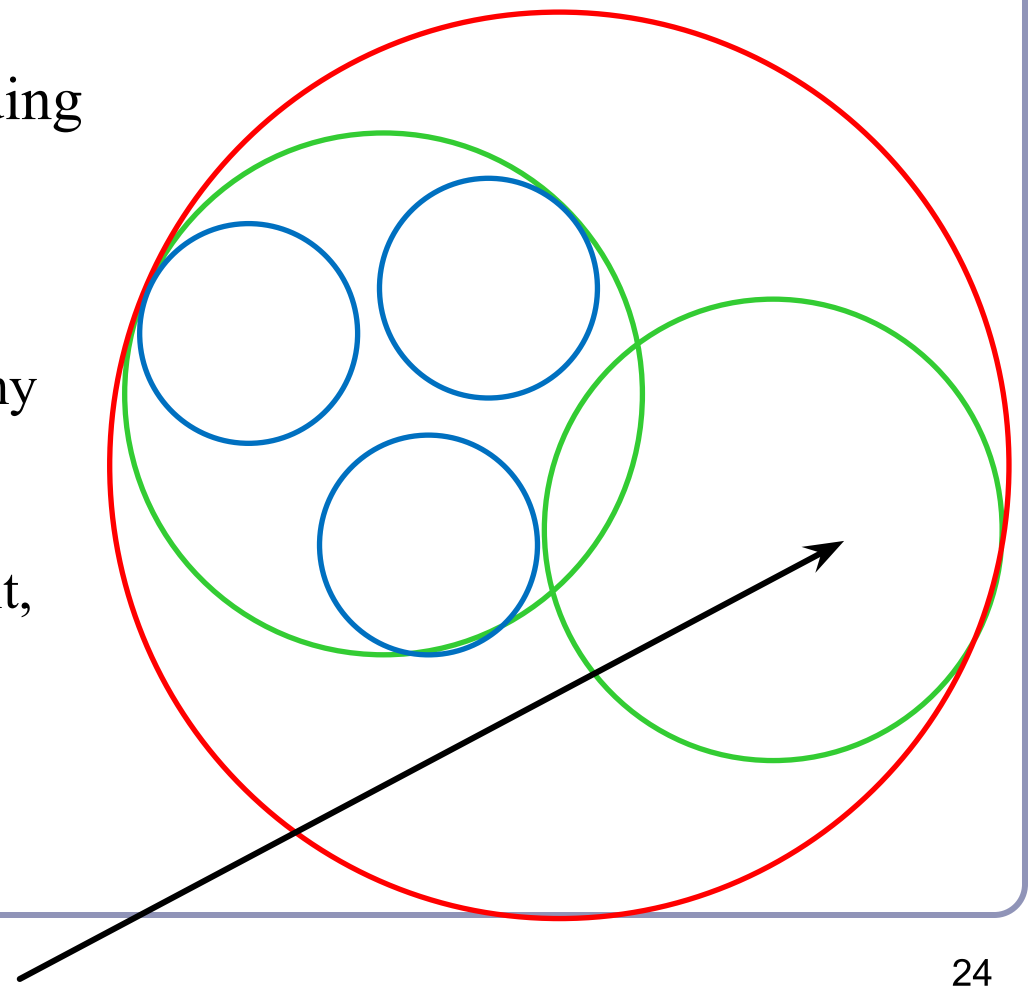


Bounding volumes in hierarchy



Hierarchies of bounding volumes allow early discarding of rays that won't hit large parts of the scene.

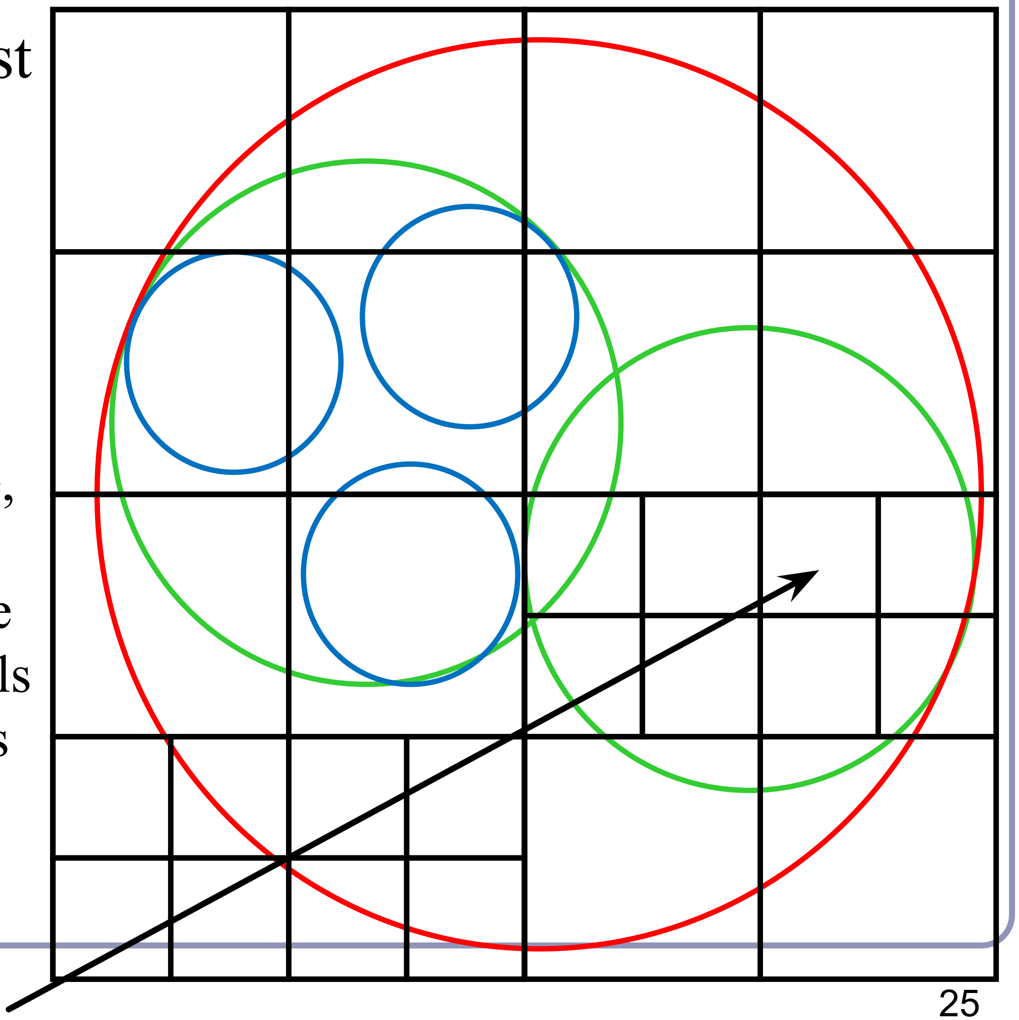
- Pro: Rays can skip subsections of the hierarchy
- Con: Without spatial coherence ordering the objects in a volume you hit, you'll still have to hit-test every object



Subdivision of space

Split space into cells and list in each cell every object in the scene that overlaps that cell.

- Pro: The ray can skip empty cells
- Con: Depending on cell size, objects may overlap many filled cells or you may waste memory on many empty cells
- Popular for voxelized games (ex: *Minecraft*)



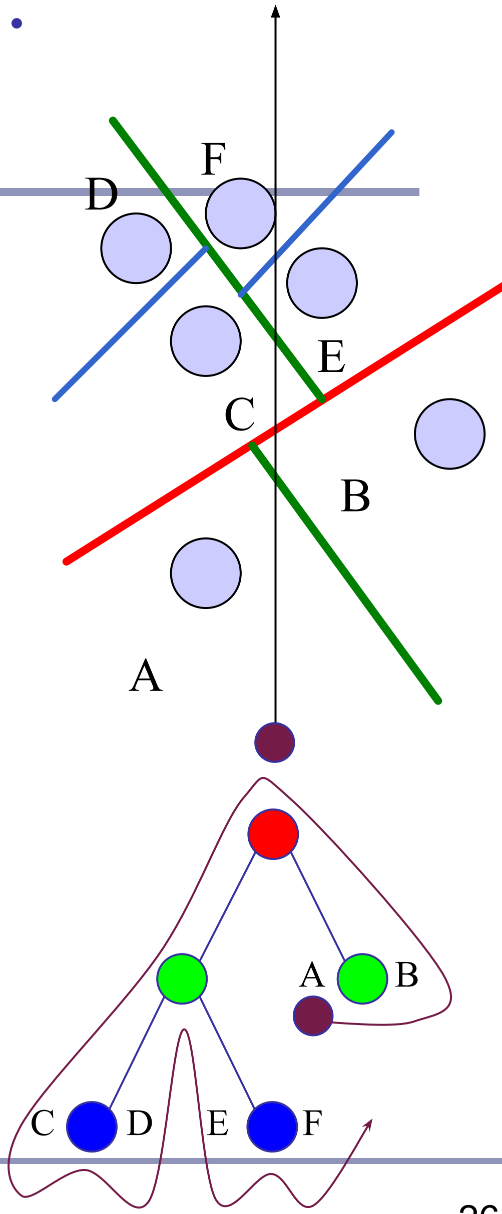
Popular acceleration structures: BSP Trees

The *BSP tree* **pre-**partitions the scene into objects in front of, on, and behind a tree of planes.

- This gives an ordering in which to test scene objects against your ray
- When you fire a ray into the scene, you test all near-side objects before testing far-side objects.

Challenges:

- requires slow pre-processing step
- strongly favors static scenes
- choice of planes is hard to optimize



Popular acceleration structures: *kd-trees*

The *kd-tree* is a simplification of the BSP Tree data structure

- Space is recursively subdivided by axis-aligned planes and points on either side of each plane are separated in the tree.
- The *k*d-tree has O(n log n) insertion time (but this is very optimizable by domain knowledge) and O($n^{2/3}$) search time.
- *k*d-trees don't suffer from the mathematical slowdowns of BSPs because their planes are always axis-aligned.

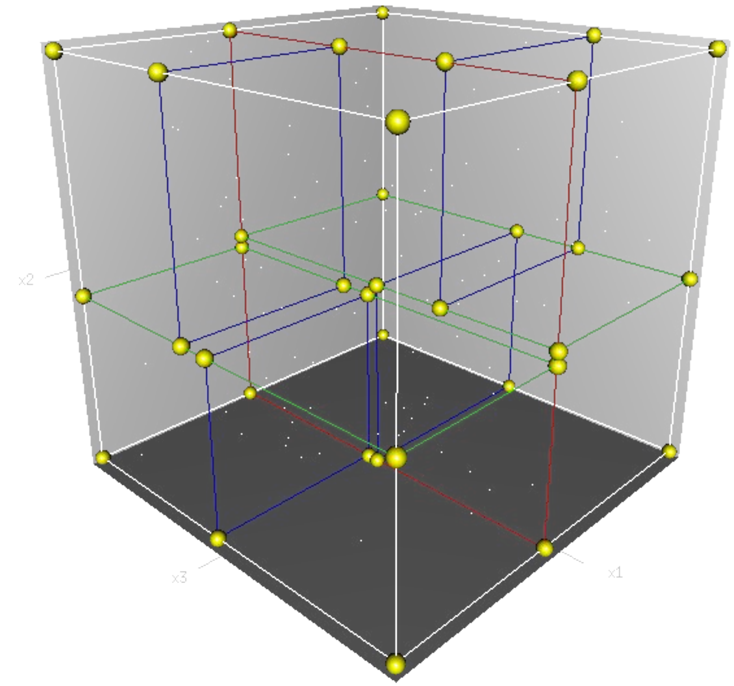


Image from Wikipedia, bless their hearts.

Popular acceleration structures: *Bounding Interval Hierarchies*

The *Bounding Interval Hierarchy* subdivides space around the volumes of objects and shrinks each volume to remove unused space.

- Think of this as a "best-fit" *k*d-tree
- Can be built dynamically as each ray is fired into the scene

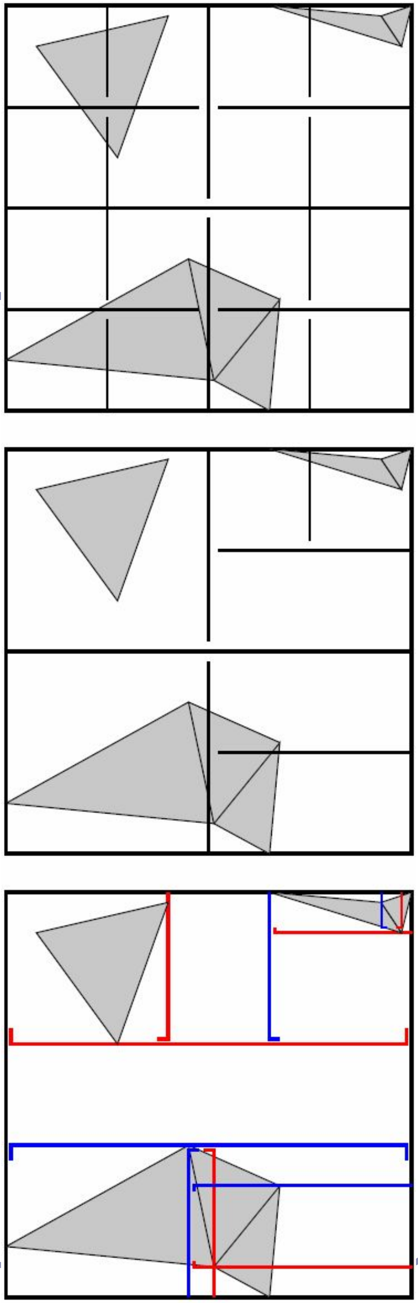


Image from Wächter and Keller's paper, *Instant Ray Tracing: The Bounding Interval Hierarchy*, Eurographics (2006)

References

Intersection testing
http://www.realtimerendering.com/intersections.html
http://tog.acm.org/editors/erich/ptinpoly
http://mathworld.wolfram.com/BarycentricCoordinates.html

Ray tracing
Peter Shirley, Steve Marschner. *Fundamentals of Computer Graphics*. Taylor & Francis, 21 Jul 2009
Hughes, Van Dam et al. *Computer Graphics: Principles and Practice*. Addison Wesley, 3rd edition (10 July 2013)