Advanced Graphics

Ray Tracing
All the maths
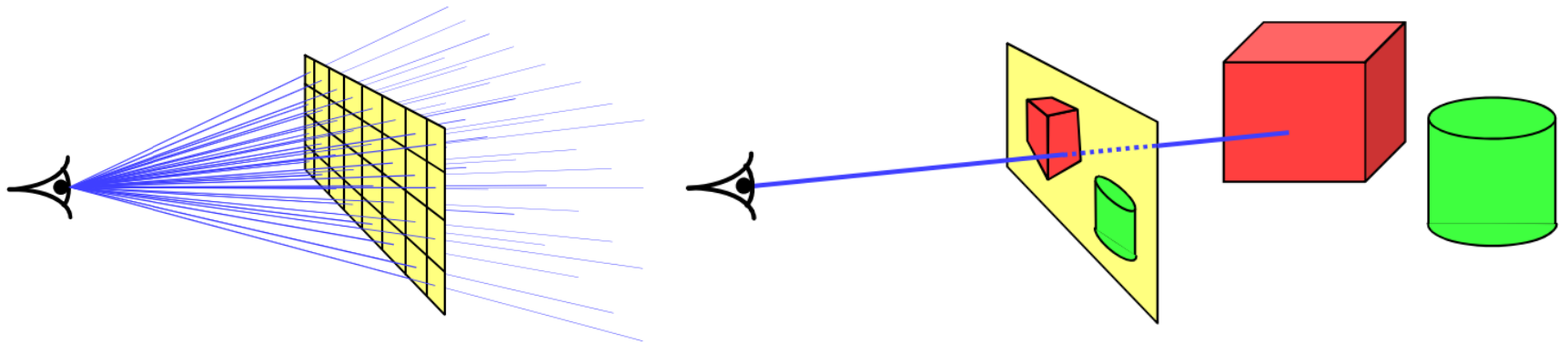
Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk

Ray tracing

- A powerful alternative to polygon scan-conversion techniques
- An elegantly simple algorithm:

Given a set of 3D objects, shoot a ray from the eye through the center of every pixel and see what it hits.

The algorithm

Select an eye point and a screen plane.

for (every pixel in the screen plane):

  Find the ray from the eye through the pixel's center.
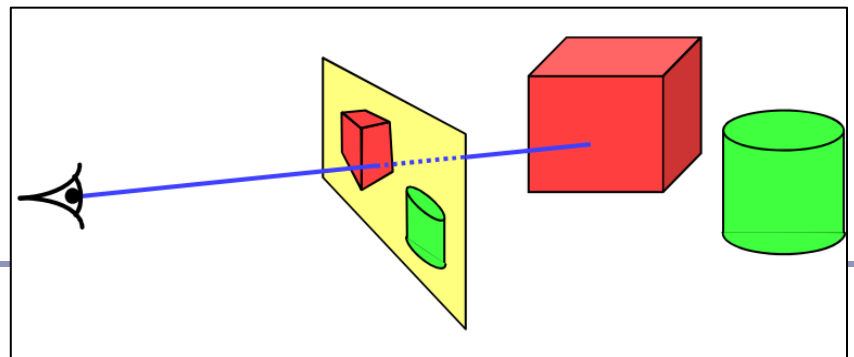
  for (each object in the scene):
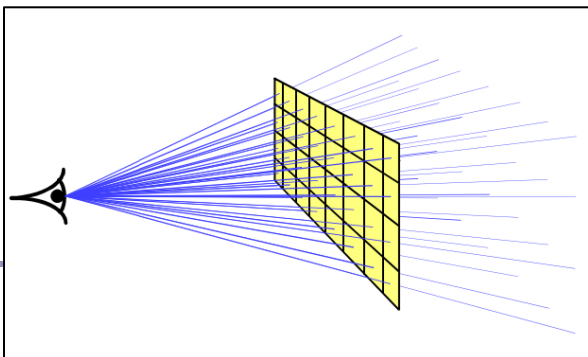
    if (the ray hits the object):

      if (the intersection is the nearest (so far) to the eye):

        Record the intersection point.

        Record the color of the object at that point.

  Set the screen plane pixel to the nearest recorded color.

Examples
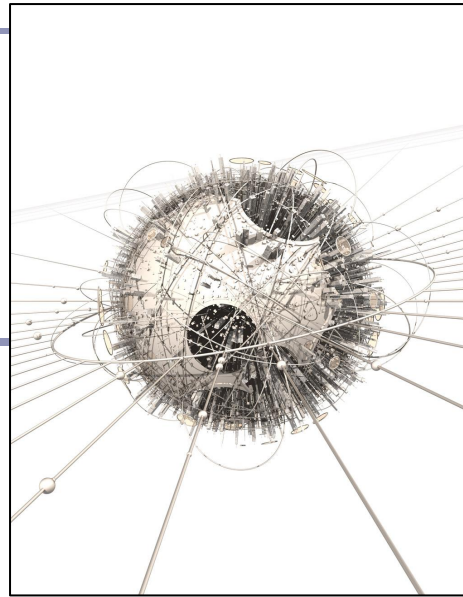
"POV Planet" by Casey Uhrig (2004)
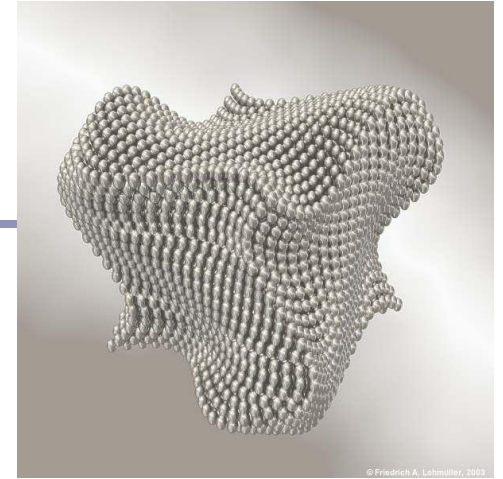
"Dancing Cube" by Friedrich A. Lohmueller (2003)

© 2004 Tor Olav Kristensen
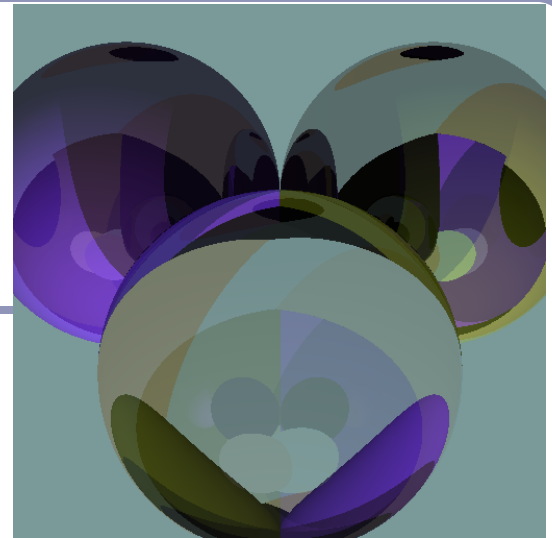
"Villarceau Circles" by Tor Olav Kristensen (2004)

"Glasses" by Gilles Tran (2006)

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

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
- …

```c
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)

Running time

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
(num transparent surfaces) x
(num shadow rays) x
(ray reflection depth) x …

Image by nVidia

Contrast this to polygon rasterization: time is a function of the number of elements in the scene times the number of lights.

Coloring pixels

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.
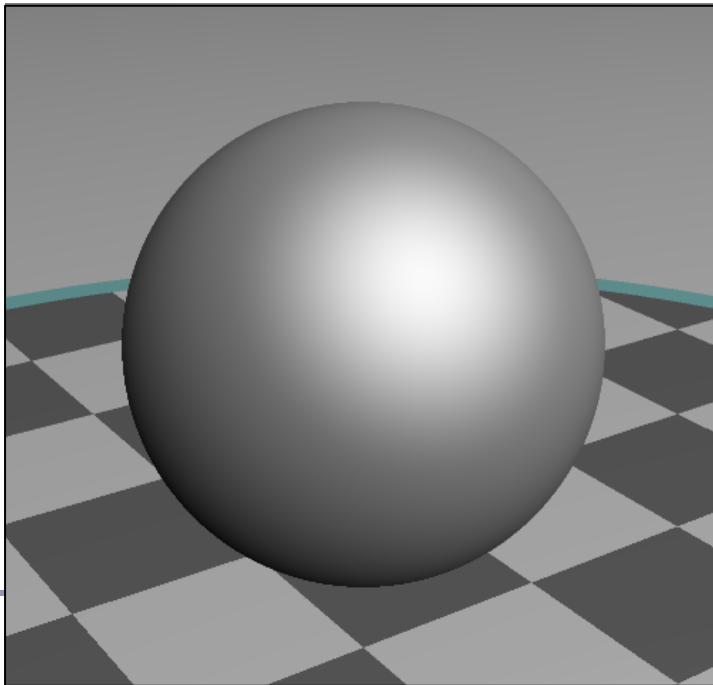- Calculate diffuse and specular from each light source separately.

L_1

L_2

R

N

P

D

E
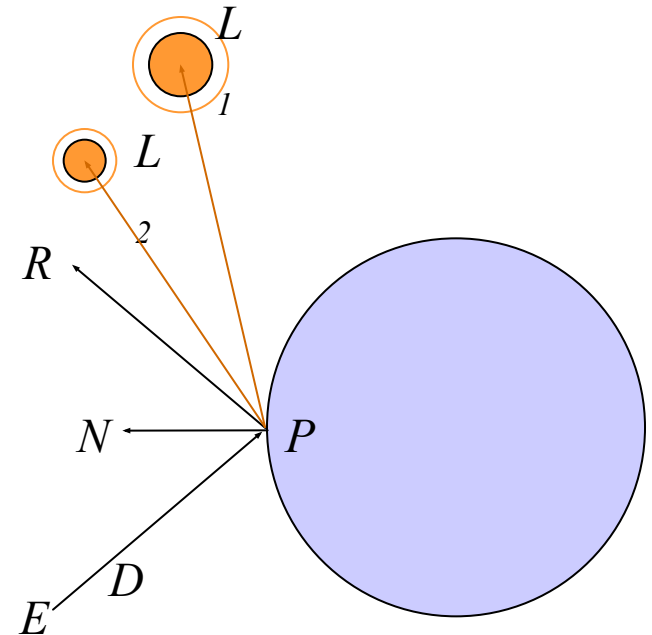
Lighting revisited

The *total illumination at P* is:

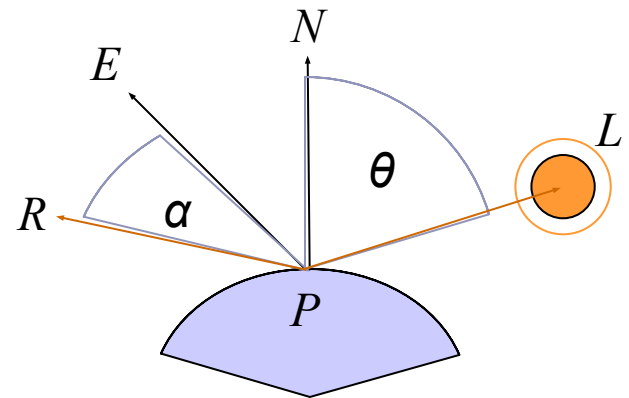$$I(P) = k_A + k_D(N \bullet L) + k_S(R \bullet E)^n$$

summed over all lights *L*.

Ambient light: k_A

Diffuse light: $k_D(N \bullet L)$

Specular light: $k_S(R \bullet E)^n$

You remember this from Computer Graphics and Image Processing last year, right?

Ray-traced illumination

Once you have the point P, the intersection of the ray with the nearest object, you'll compute how much each of the lights in the scene illuminates P.

diffuse = 0

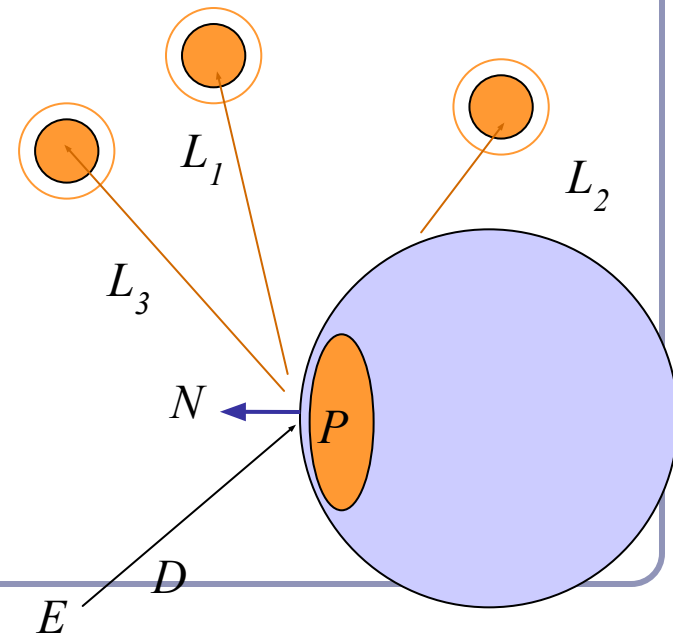*specular = 0*

for (each light L_i in the scene):

    if $(N \cdot L) > 0$:

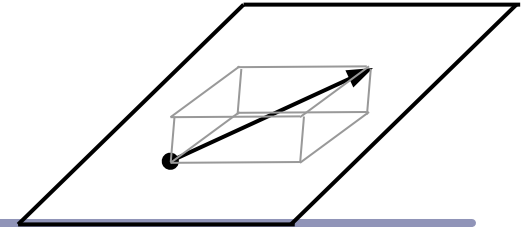   [Optionally: if (a ray from P to L_i can reach L_i):]

        diffuse += $k_D(N \cdot L)$

        specular += $k_S(R \cdot E)^n$

intensity at P = ambient + diffuse + specular

L_1

L_2

L_3

N

P

E

D

Hitting things with rays
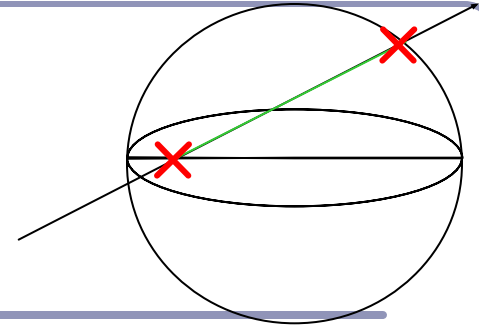
A ray is defined parametrically as

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

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

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

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

Hitting things with rays: Sphere

The unit sphere, centered at the origin, has the implicit equation

$$x^2 + y^2 + z^2 = 1 \qquad\qquad (\gamma)$$

Substituting equation (β) into (γ) gives

$$(x_E + tx_D)^2 + (y_E + ty_D)^2 + (z_E + tz_D)^2 = 1$$

which expands to

$$t^2(x_D{}^2 + y_D{}^2 + z_D{}^2) + t(2x_Ex_D + 2y_Ey_D + 2z_Ez_D) + (x_E{}^2 + y_E{}^2 + z_E{}^2 - 1) = 0$$
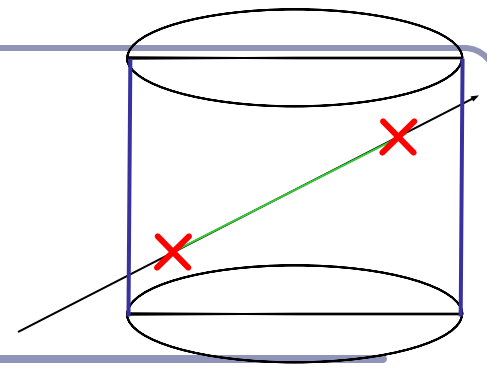
which is of the form

$$at^2 + bt + c = 0$$

which can be solved for t:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

giving us two points of intersection.

Hitting things with rays: Cylinder

The infinite unit cylinder, centered at the origin, has the implicit equation

$$x^2 + y^2 = 1 \qquad\qquad\qquad (\delta)$$

Substituting equation (β) into (δ) gives

$$(x_E + tx_D)^2 + (y_E + ty_D)^2 = 1$$

which expands to

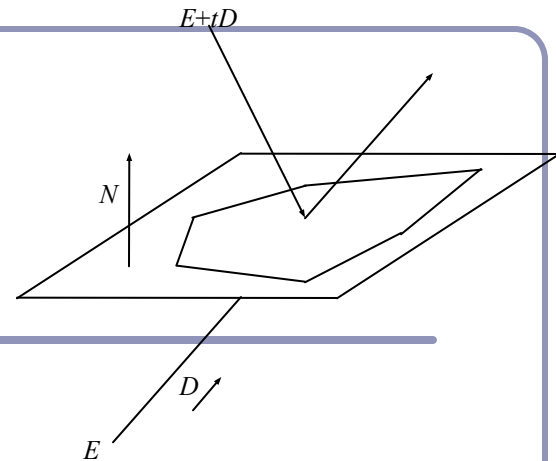$$t^2(x_D{}^2 + y_D{}^2) + t(2x_E x_D + 2y_E y_D) + (x_E{}^2 + y_E{}^2 - 1) = 0$$

which is of the form

$$at^2 + bt + c = 0$$

which can be solved for t as before, giving us two points of intersection.

The cylinder is infinite; there is no z term.

Hitting things with rays: Planes and polygons

A planar polygon P can be defined as

Polygon $P = \{v^1, ..., v^n\}$

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

$N \bullet (p - v^1) = 0$ (ζ)

Substituting equation (α) into (ζ) for p yields

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

$x_N(x_E + tx_D - x_v^1) + y_N(y_E + ty_D - y_v^1) + z_N(z_E + tz_D - z_v^1) = 0$

$$t = \frac{(N \bullet v^1) - (N \bullet E)}{N \bullet D}$$

Hit test: 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
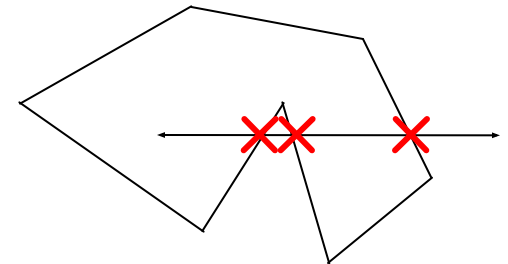
Point in nonconvex polygon

Winding number

- 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}.
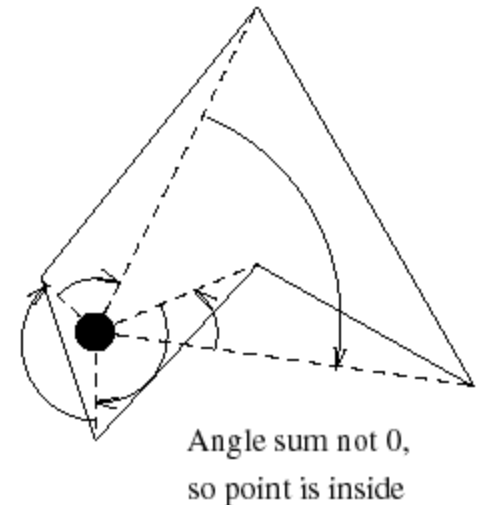 - Caveat: This method is elegant but slow.


Angle sum not 0,
so point is inside

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 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.
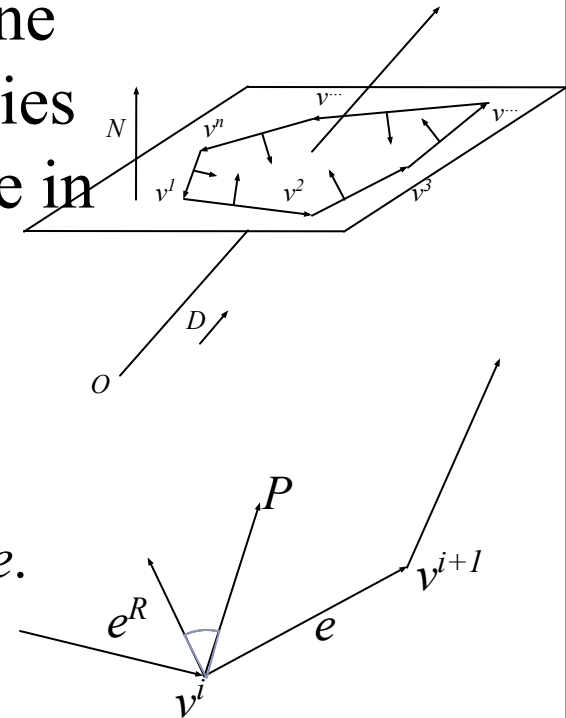
Barycentric coordinates

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 also proportional to the subtriangle areas.
  - The area of a triangle is ½ the length of the cross product of two of its sides.
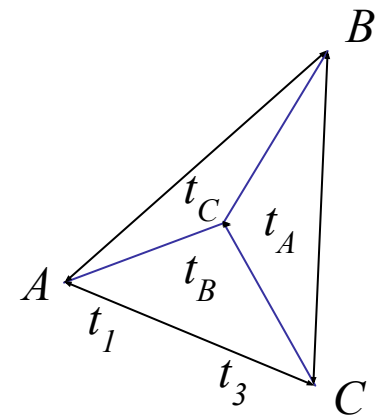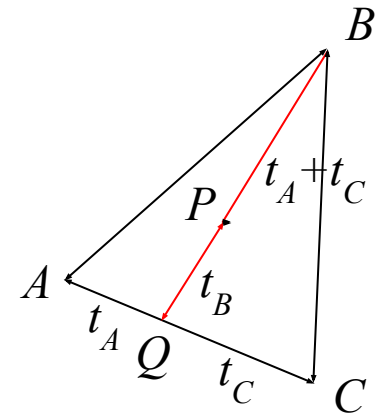
The *Jordan curve theorem*
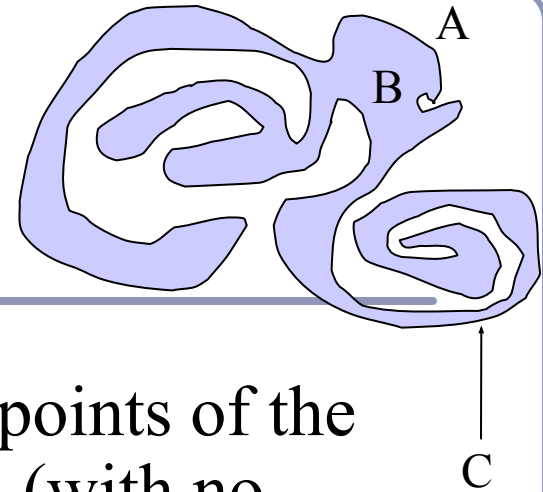
"Any simple closed curve C divides the points of the plane not on C into two distinct domains (with no points in common) of which C is the common boundary."

- 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.
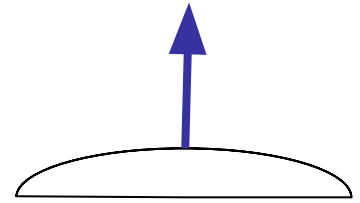
The Jordan curve theorem on a sphere

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."
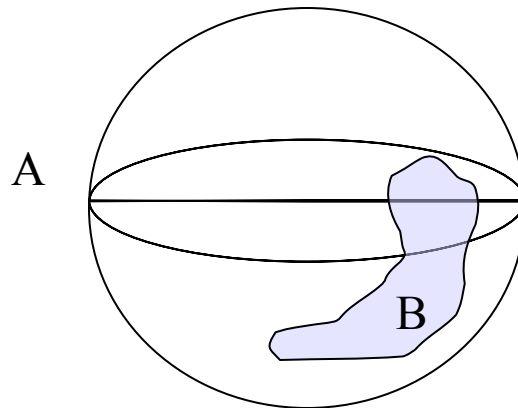
Finding the normal

We often need to know *N*, the *normal to the surface* at the point where a ray hits a primitive.

If the ray R hits the primitive P at point X then N is…

Primitive type	Equation for N
Unit Sphere centered at the origin	$N = X$
Infinite Unit Cylinder centered at the origin	$N = [\, x_X,\, y_X,\, 0\,]$
Infinite Double Cone centered at the origin	$N = X \times (X \times [\, 0,\, 0,\, z_X\,])$
Plane with normal *n*	$N = n$

We'll use the normal for *color*, *reflection*, *refraction* and *shadow* rays.

Local coordinates, world coordinates

A very common technique in graphics is to associate a *matrix transform*, typically a 4x4 matrix, with a primitive.

5	0	0	0
0	2	0	0
0	0	5	0
0	0	0	1

A 4x4 *scale matrix*, which multiplies x and z by 5, y by 2.
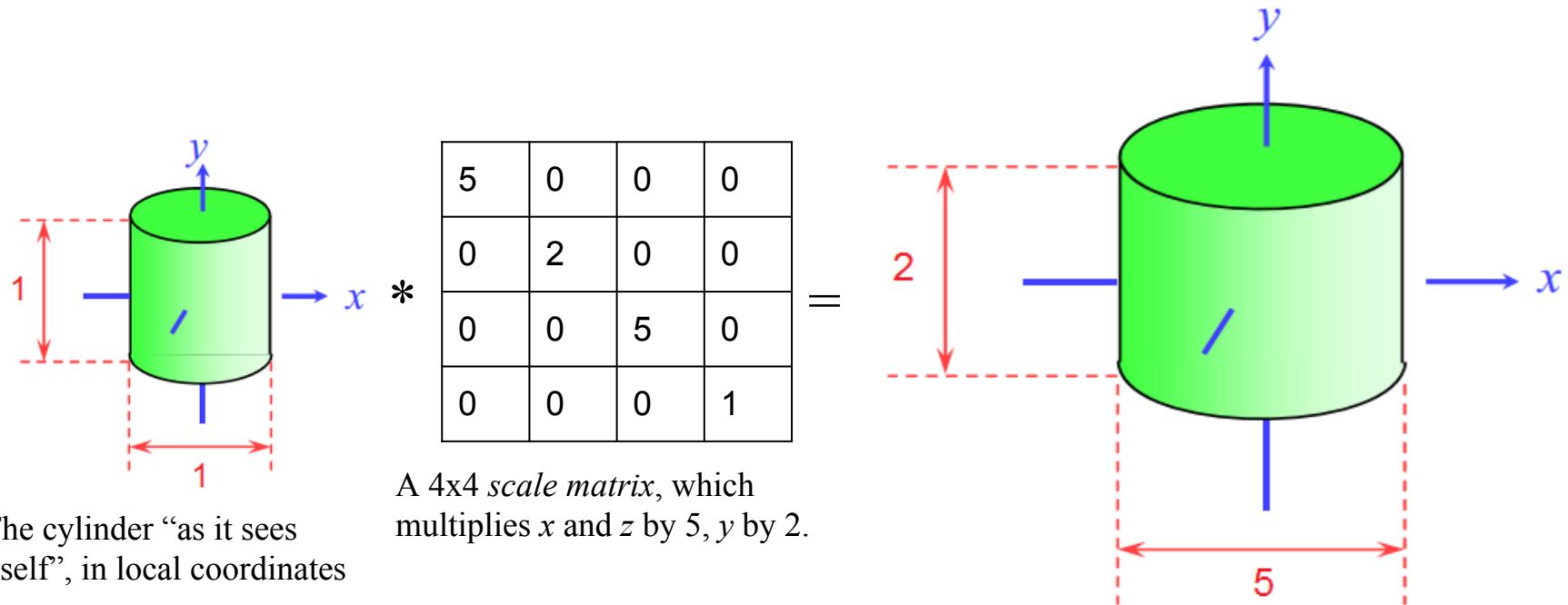
The cylinder "as it sees itself", in local coordinates

The cylinder "as the world sees it", in world coordinates

Local coordinates, world coordinates

 We can use the *local to world transform* to describe an object's position, orientation, and size in the world, while still coding the object as centered around its local origin.
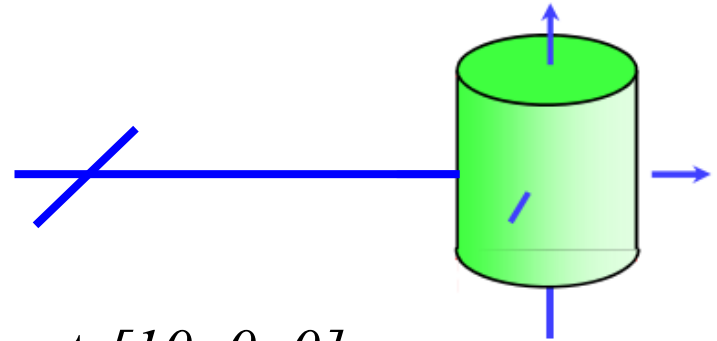

 The trick is to hit an object described in its local coordinates with a ray described in world coordinates.
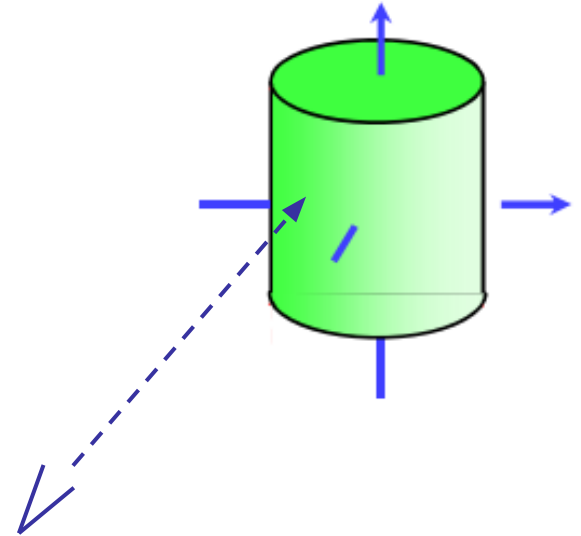
Local coordinates, world coordinates

Consider the following transform T:

```
        [ 1 0 0 10 ]
 T =    [ 0 1 0 0  ]
        [ 0 0 1 0  ]
        [ 0 0 0 1  ]
```

If we multiply *T* times *[0, 0, 0]*, we get *[10, 0, 0]*.

In other words, *T* describes an object which has been translated 10 units up the positive X axis. *T* is a *local to world* transformation.
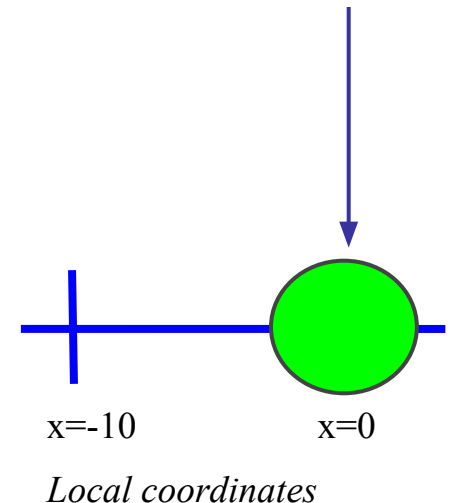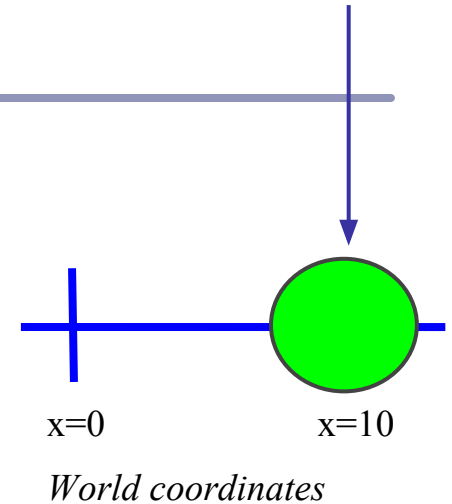
Local coordinates, world coordinates: Transforming the ray

 In order to test whether a ray hits a transformed object, we need to describe the ray in the object's *local coordinates*. We transform the ray by the *inverse of the local to world matrix*:

```
        [ 1 0 0 -10 ]
 T⁻¹ = [ 0 1 0 0   ]
        [ 0 0 1 0   ]
        [ 0 0 0 1   ]
 P'(t) = T⁻¹(P(t))
```

 This will take a ray passing through *x=10* and transform it to a ray through *x=0*.

x=0   x=10

World coordinates

x=-10   x=0

Local coordinates

Local coordinates, world coordinates: Transforming the ray's direction

It would be convenient to expand `P'(t)`. However, we can't write

Wrong! →

$$P'(t)=T^{-1}(P(t))=T^{-1}(E)+t(T^{-1}(D))$$

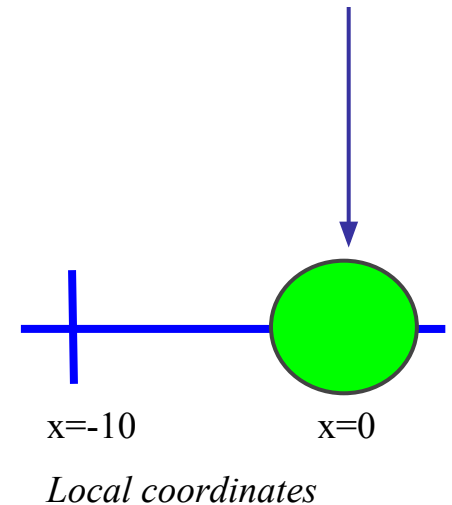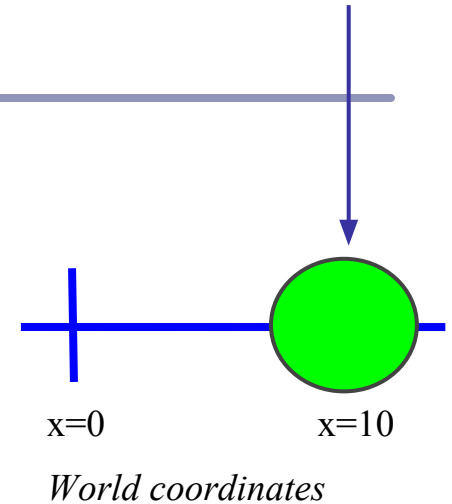because T^{-1} includes a translation and D is a unit direction. Instead write:

Right! →

$$L=(((T^{-1})^{3x3})^{-1})^{T}$$

$$P'(t)=T^{-1}(E)+t(L(D))$$

where $(((T^{-1})^{3x3})^{-1})^{T}$ is the top left 3x3 matrix of T^{-1}, inverted and transposed.

This eliminates the translation and addresses inverted scale issues.

x=0   x=10

World coordinates

x=-10   x=0

Local coordinates

Finding the normal

Much like the direction D, we have to take care when converting the normal N of the hit surface from local coordinates back to world coordinates.

To find the world-coordinates normal N from the local-coordinates N_L, multiply N_L by the transpose of the inverse of the top left-hand 3x3 submatrix of T:

$$\texttt{N=((T}^{\texttt{3x3}}\texttt{)}^{\texttt{-1}}\texttt{)}^{\texttt{T}}\ \texttt{N}_{\texttt{L}}$$

- 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

N_L

local

T

N_W
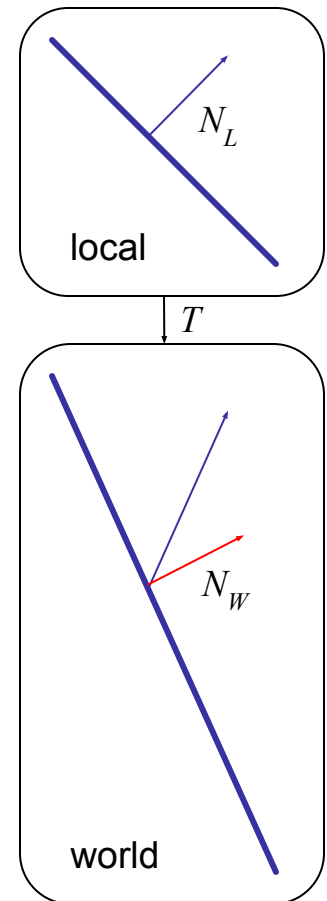
world

Local coordinates, world coordinates Summary

To compute the intersection of a ray R=E+tD with an object transformed by local-to-world transform T:

1. Compute R', the ray R in local coordinates, as
   ```
   P'(t) = T(P(t)) = T(E) + t((((T⁻¹)³ˣ³)⁻¹)ᵀ(D))
   ```
2. Perform your hit test in local coordinates.
3. Convert all hit points from local coordinates back to world coordinates by multiplying them by T.
4. Convert all hit normals from local coordinates back to world coordinates by multiplying them by `((T³ˣ³)⁻¹)ᵀ`.

This will allow you to efficiently and quickly fire rays at arbitrarily-transformed primitive objects.

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

Intersection testing
http://www.realtimerendering.com/intersections.html
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