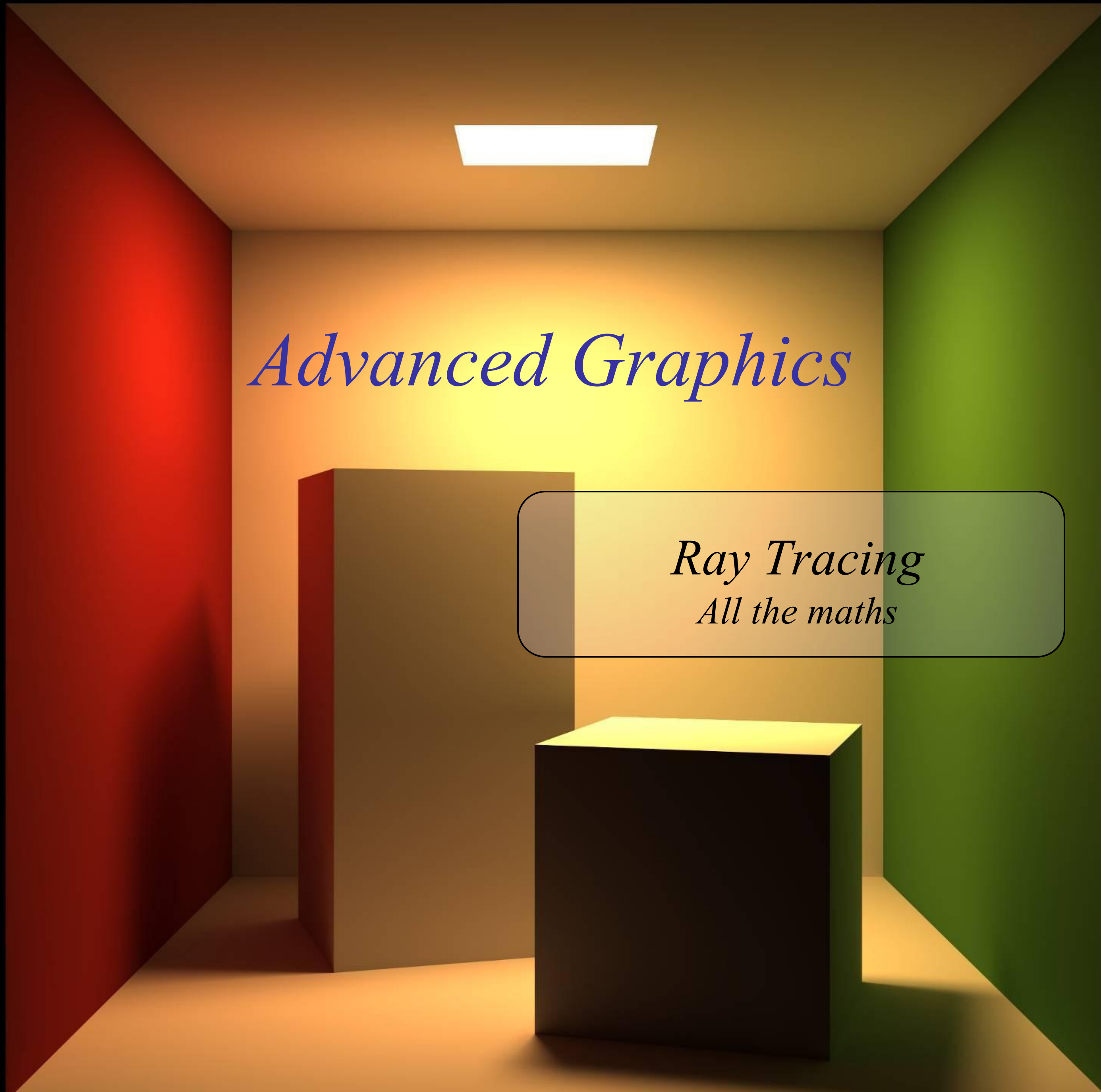
Advanced Graphics

Ray Tracing

All the maths



"*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.

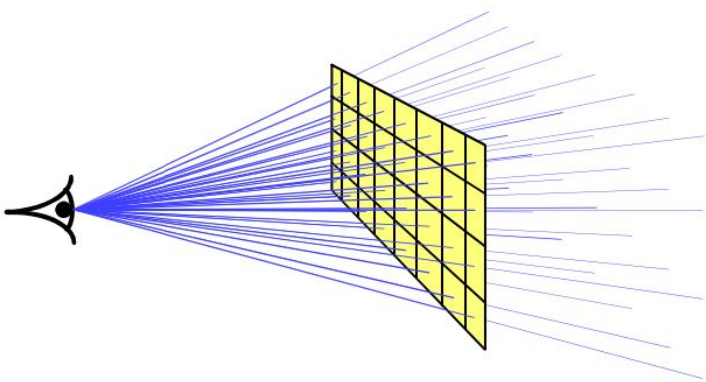
Alex Benton, University of Cambridge – alex@bentonian.com

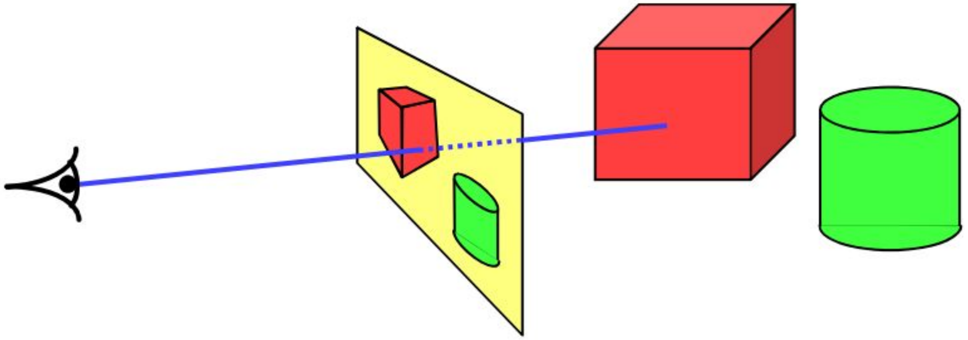
Supported in part by Google UK, Ltd

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):

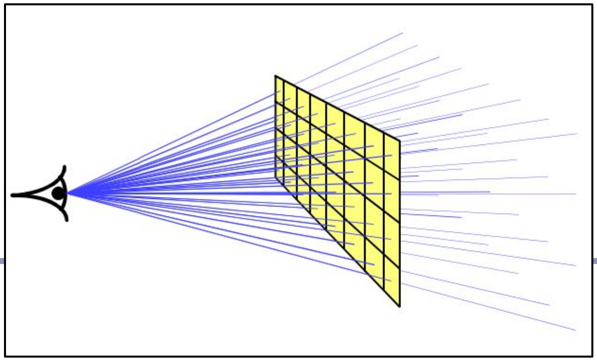
        if (the ray hits the object):

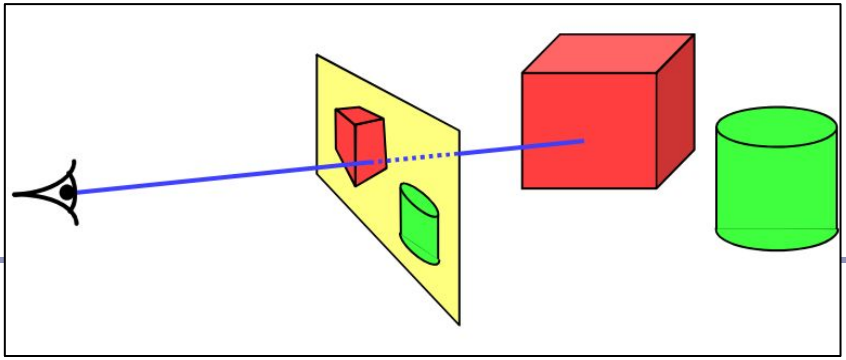
            if (the intersection is the nearest (so far) to the eye):

                Record the intersection point.

                Record the color of the object at that point.

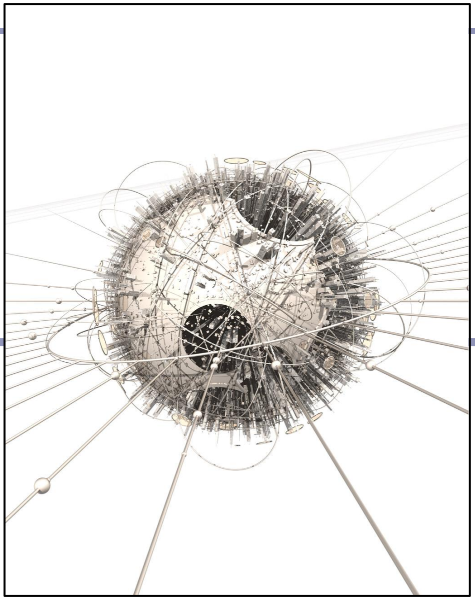
    Set the screen plane pixel to the nearest recorded color.



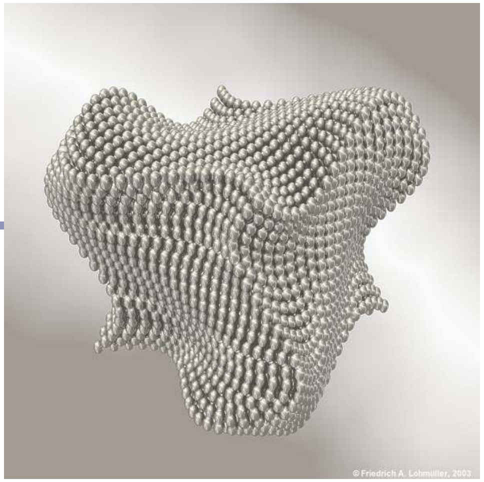


Examples

"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

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)

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.

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):

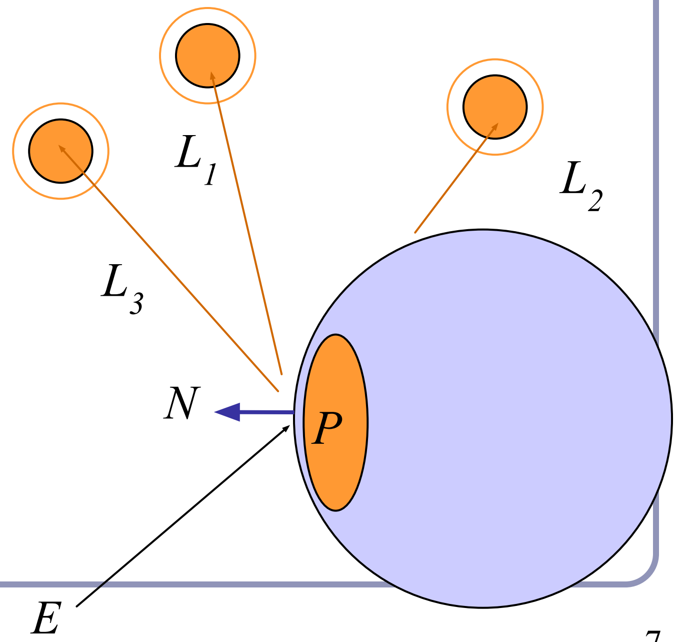
  if $(N \bullet L) > 0$:

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

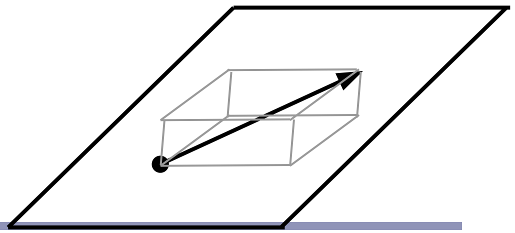
    $diffuse \mathrel{+}= k_D(N \bullet L)$

    $specular \mathrel{+}= k_S(R \bullet E)^n$

intensity at P = ambient + diffuse + specular



Hitting things with rays



A ray is defined parametrically as

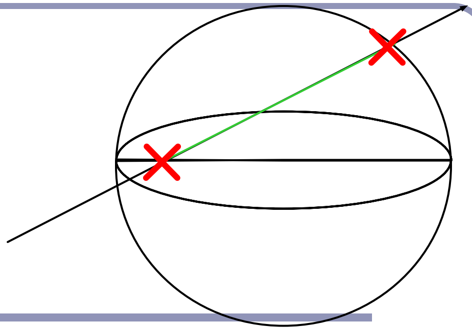
$$P(t) = E + tD, \; t \geq 0 \qquad (\alpha)$$

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

We 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: Sphere



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

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

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

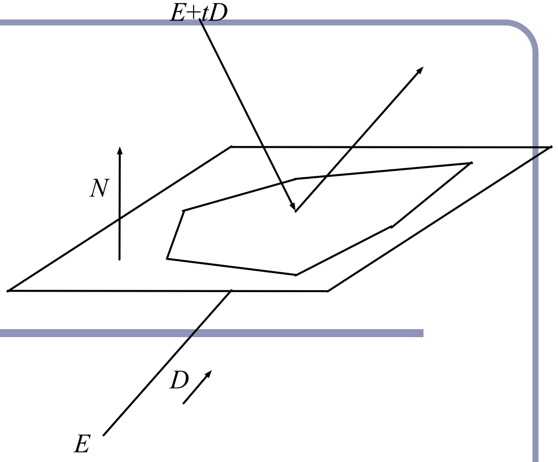
$$at^2+bt+c=0$$

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: 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

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

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

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

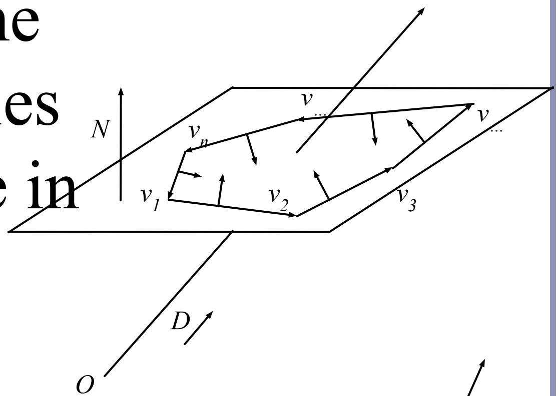
$x_N(x_E + tx_D - x_{v1}) + y_N(y_E + ty_D - y_{v1}) + z_N(z_E + tz_D - z_{v1}) = 0$

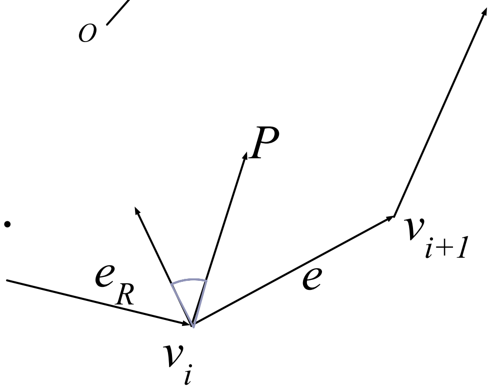
$$t = \frac{(N \bullet v1) - (N \bullet E)}{N \bullet D}$$

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 $e_R \bullet (P - v_i) < 0$ then the point is outside e.
- Fastest known method.

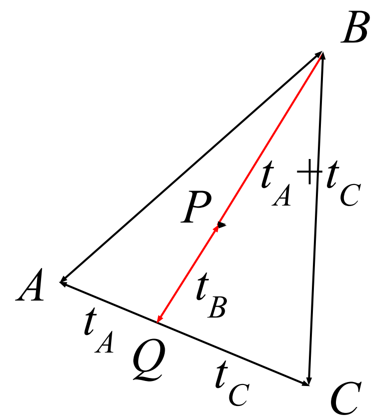


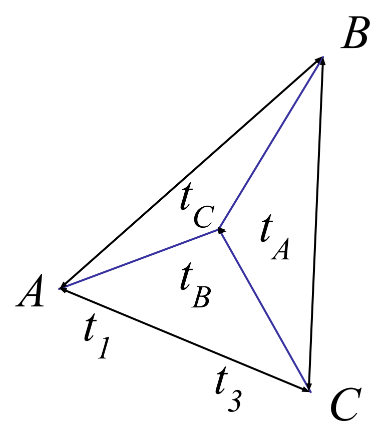


Barycentric coordinates

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.



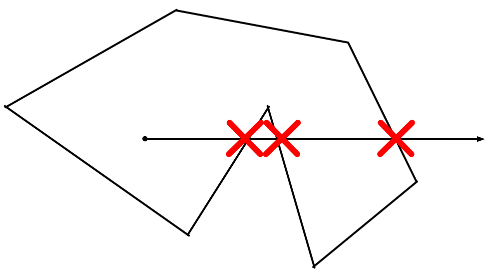


Barycentric coordinates

```
// 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)

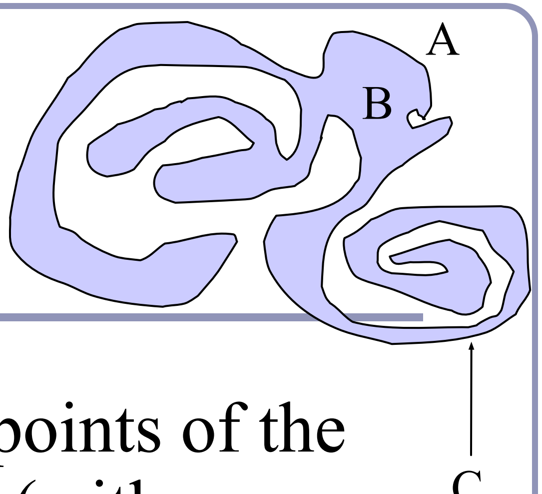
Point in nonconvex polygon



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*.

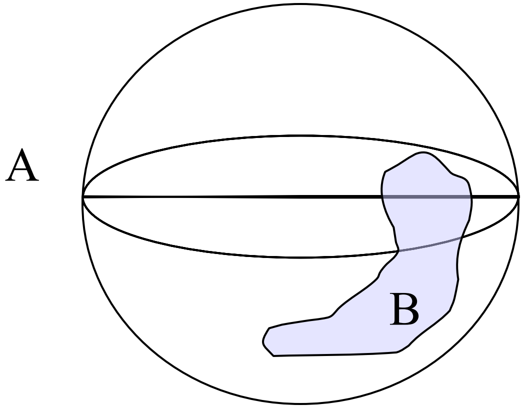
Sketch of proof : (For full proof see Courant & Robbins, 1941.)

- 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

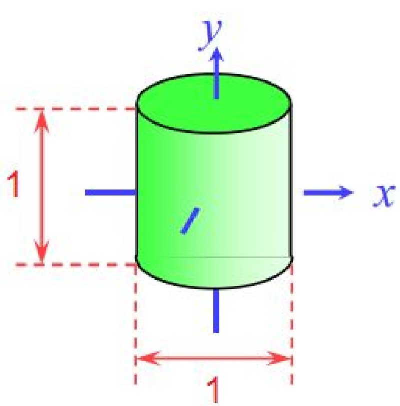
Note that the Jordan curve theorem can be extended to a curve on a sphere, or anything which is topologically equivalent to a sphere.

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



Local coordinates, world coordinates

A very common technique in graphics is to associate a *local-to-world transform,* T, with a primitive.



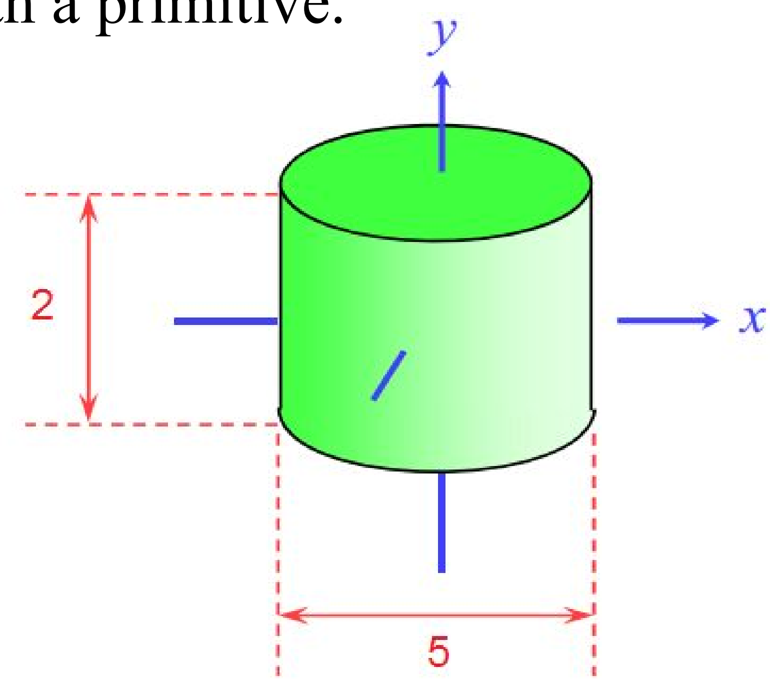
The cylinder "as it sees itself", in local coordinates

*

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.

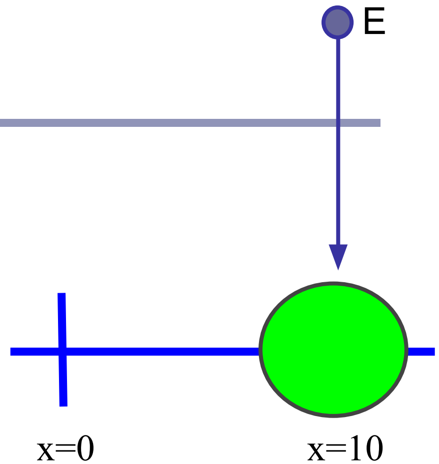
=



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

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*, $\mathtt{T}^{-1}$.



World coordinates

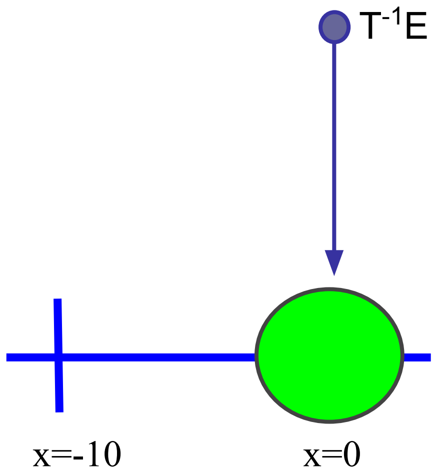
If the ray is defined by

$$\mathtt{P(t) = E + tD}$$

then the ray in local coordinates is defined by

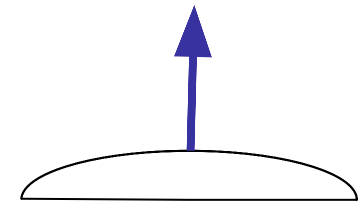
$$\mathtt{T^{-1}(P(t)) = T^{-1}(E) + t(T^{-1}_{3x3}D)}$$

where $\mathtt{T}^{-1}_{\mathtt{3x3}}$ is the top left 3x3 submatrix of $\mathtt{T}^{-1}$.



Local coordinates

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…

<u>Primitive type</u>	<u>Equation for N</u>
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*	$N = n$

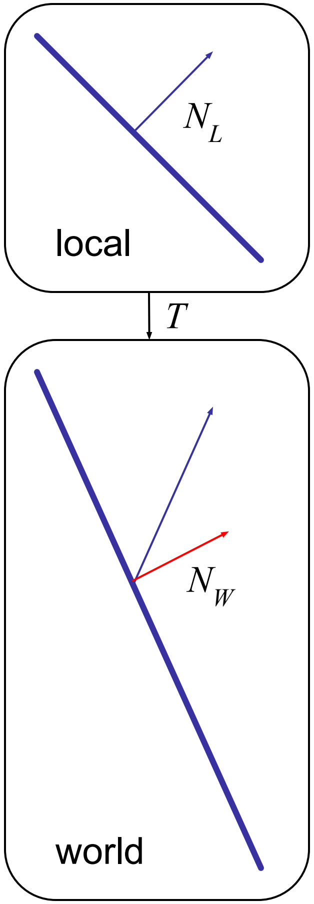
We use the normal for color, reflection, refraction, shadow rays...

Converting the normal from local 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:

$$\mathrm{N} = ((\mathrm{T}_{3\mathrm{x}3})^{-1})^{\mathrm{T}}\ \mathrm{N}_{\mathrm{L}}$$

- We want the top left 3x3 to discard 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



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
   $P'(t) = T^{-1}(P(t)) = T^{-1}(E) + t(T^{-1}_{3x3}(D))$
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 $((T^{3x3})^{-1})^{T}$.

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

Your scene graph and you

Many 2D GUIs today favor an event model in which events ‘bubble up’ from child windows to parents. This is sometimes mirrored in a scene graph.

- Ex: a child changes size, changing the size of the parent’s bounding box
- Ex: the user drags a movable control in the scene, triggering an update event

If you do choose this approach, consider using the *Model View Controller* or *Model View Presenter* design pattern. 3D geometry objects are good for displaying data but they are not the proper place for control logic.

- For example, the class that stores the geometry of the rocket should not be the same class that stores the logic that moves the rocket.
- <u>Always separate logic from representation</u>.

Your scene graph and you

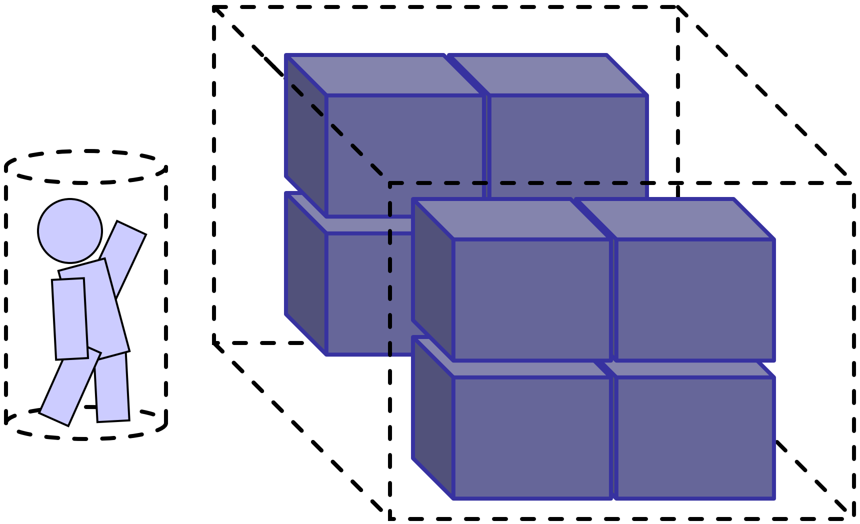
A common optimization derived from the scene graph is the propagation of *bounding volumes*.

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



Speed up ray-tracing with *bounding volumes*

Bounding volumes help to quickly accelerate volumetric tests, such as “does the ray hit the cow?”

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

Axis-aligned bounding boxes

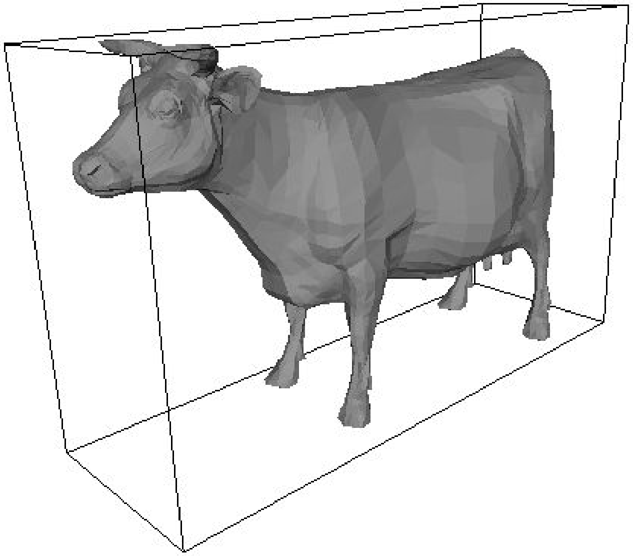
- max and min of x/y/z.

Bounding spheres

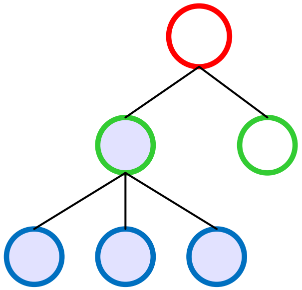
- max of radius from some rough center

Bounding cylinders

- common in early FPS games

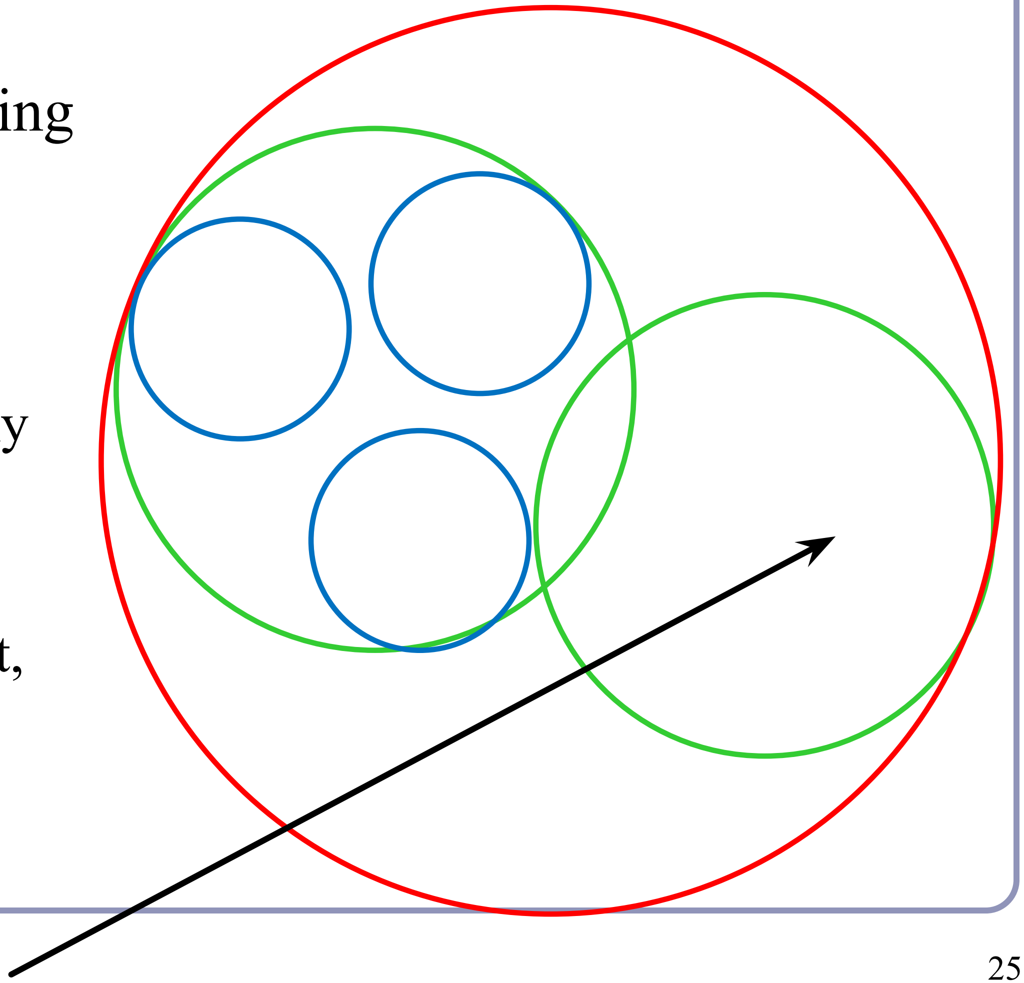


Bounding volumes in hierarchy



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

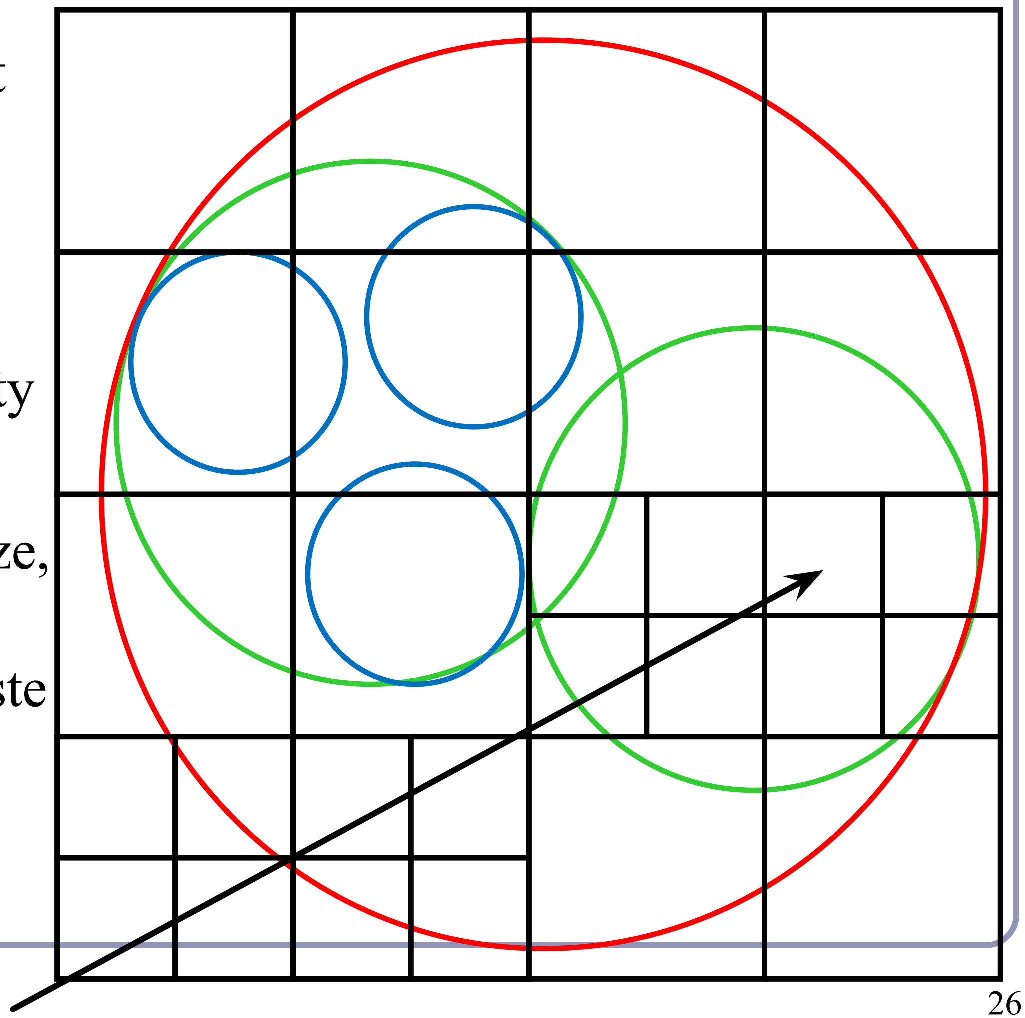
- Pro: Rays can skip subsections of the hierarchy
- Con: Without spatial coherence ordering the objects in a volume you hit, you'll still have to hit-test every object



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



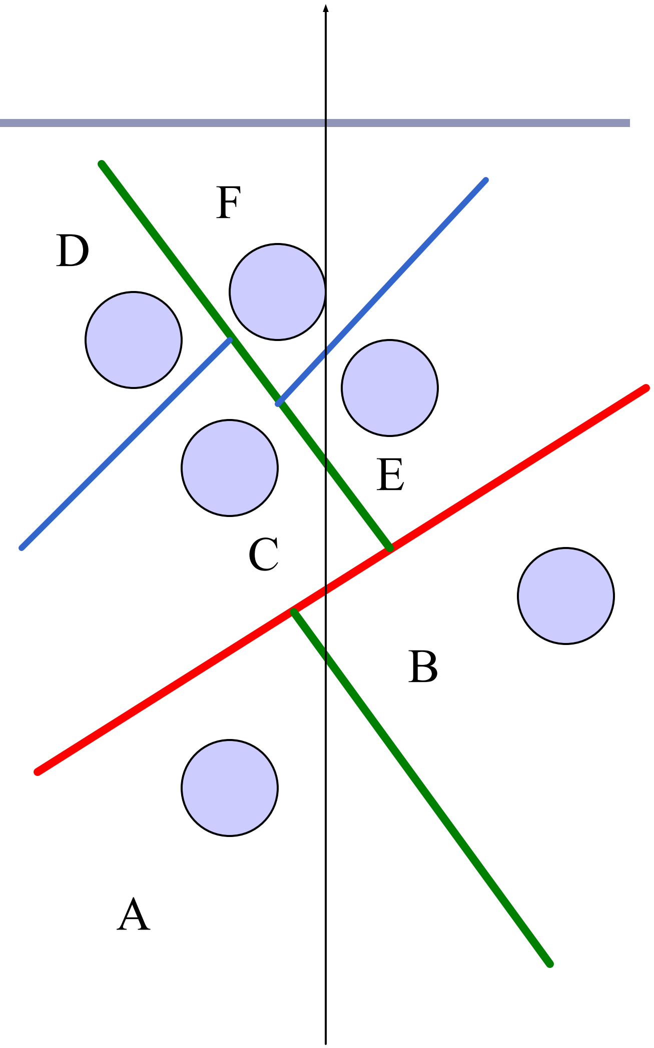
Popular acceleration structures: BSP Trees

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

- When you fire a ray into the scene, you test all near-side objects before testing far-side objects.

Problems:

- choice of planes is not obvious
- computation is slow
- plane intersection tests are heavy on floating-point math.



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.

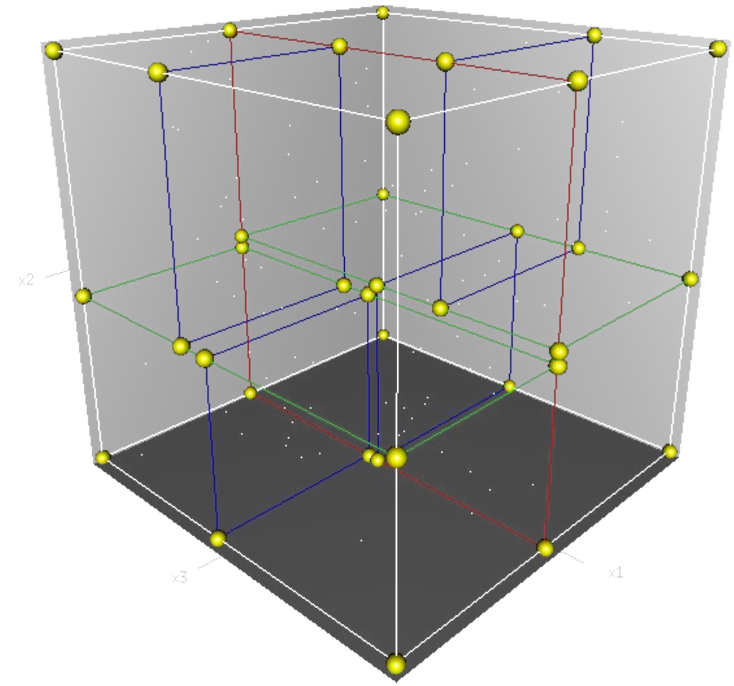


Image from Wikipedia, bless their hearts.

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

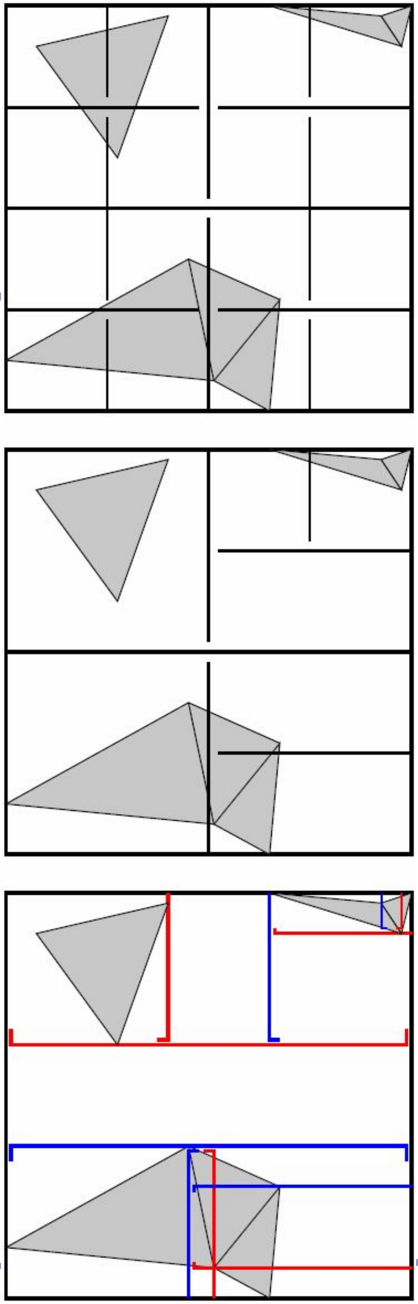


Image from Wächter and Keller’s paper, *Instant Ray Tracing: The Bounding Interval Hierarchy*, Eurographics (2006)

References

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