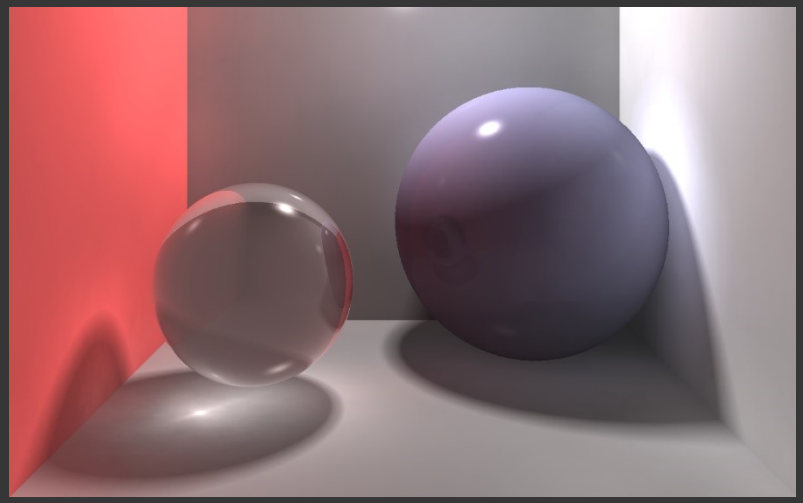
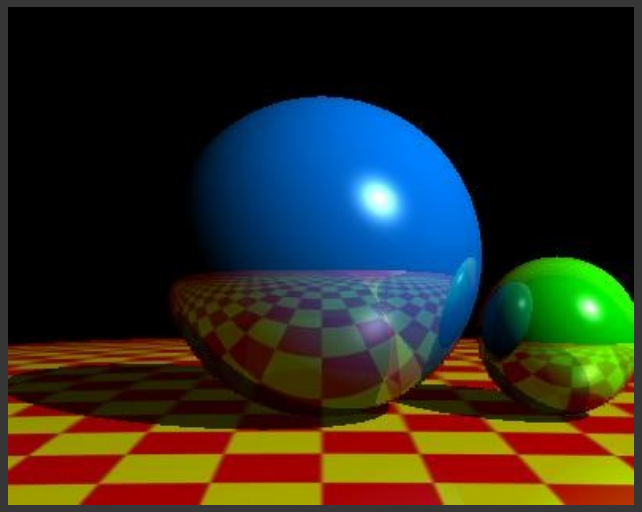
Ray-tracing

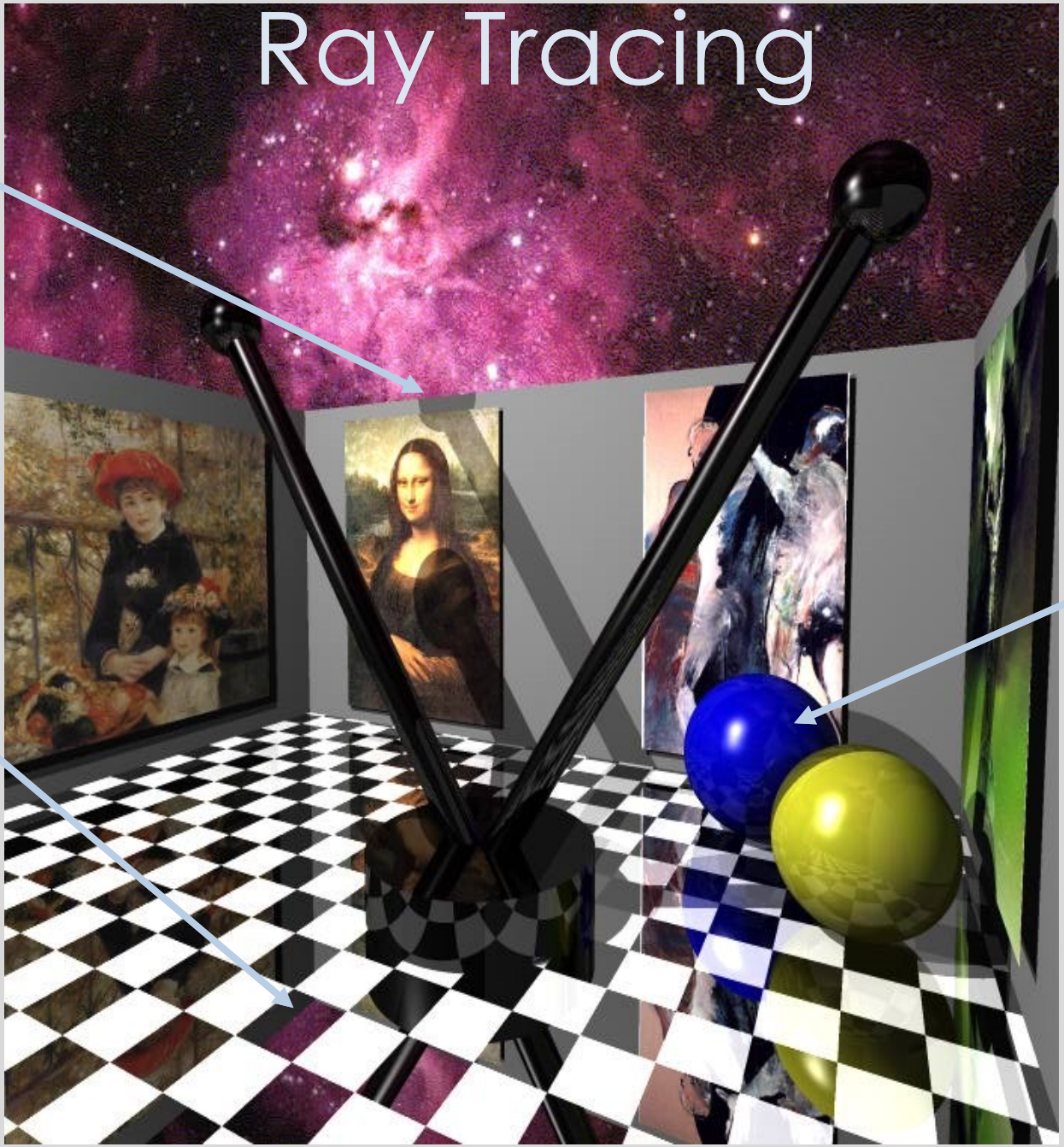
Lecturer: Rachel McDonnell
Assistant Professor in Creative Technologies
Rachel.McDonnell@cs.tcd.ie

Course www: Blackboard
Credit: some slides from Carol O'Sullivan

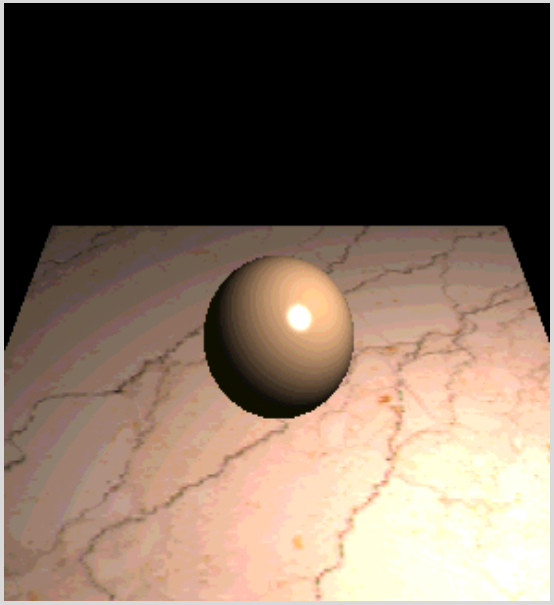




Ray Tracing

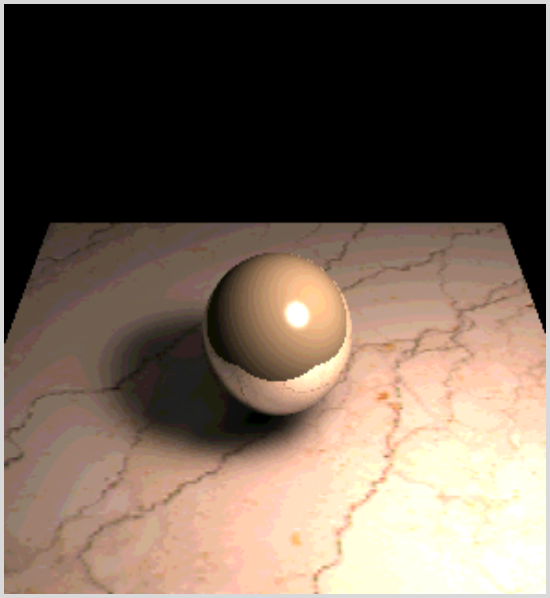

Sharp shadows
Perfectly specular reflections
Phong Illumination

Local vs. Global Illumination



Local

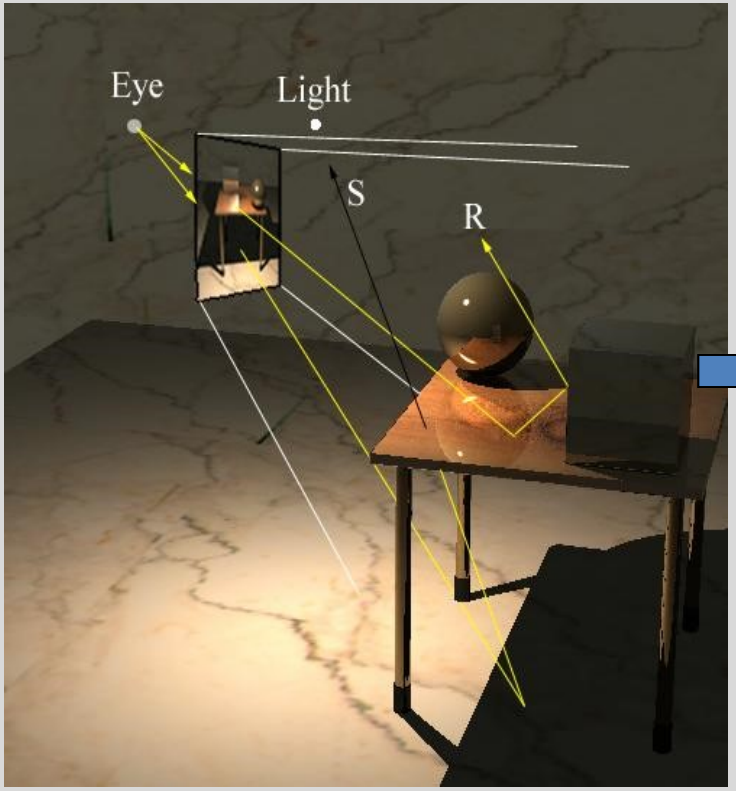
Illumination depends on local object & light sources only



Global

Illumination at a point can depend on any other point in the scene

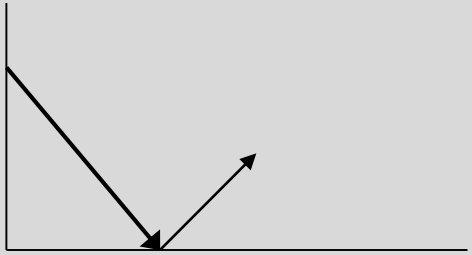
View Dependent Solution (Ray Traced)



Scene Geometry

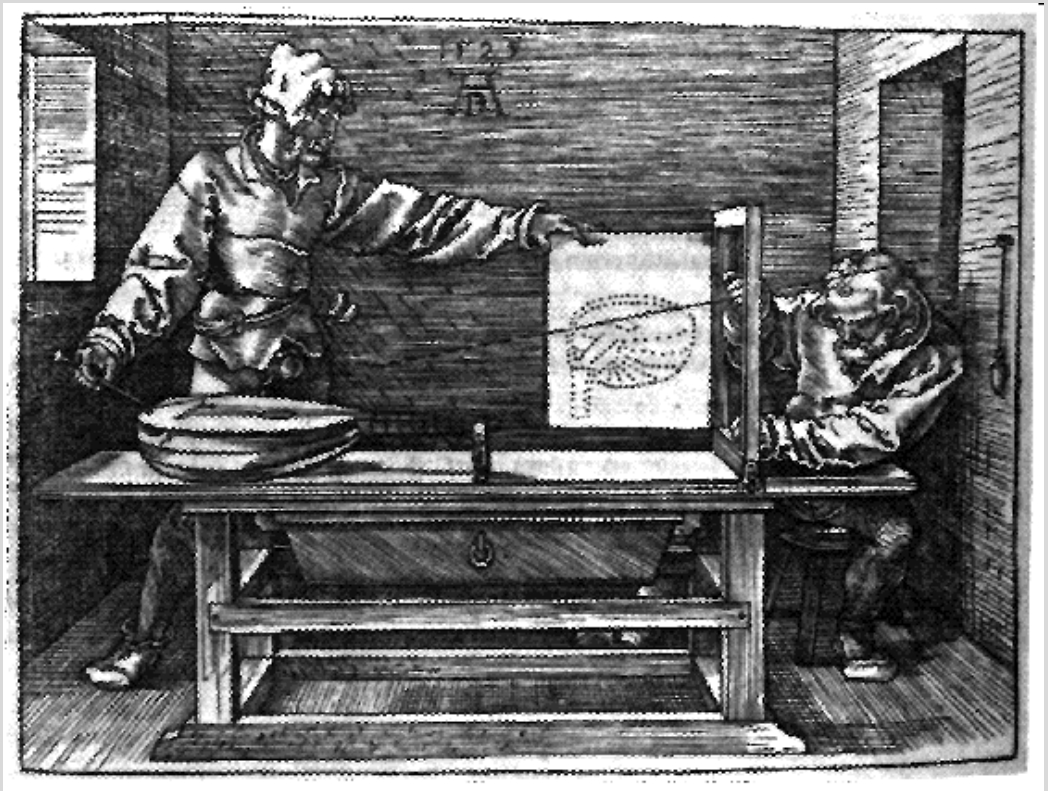
Solution determined only for directions through pixels in the viewport

Ray-tracing



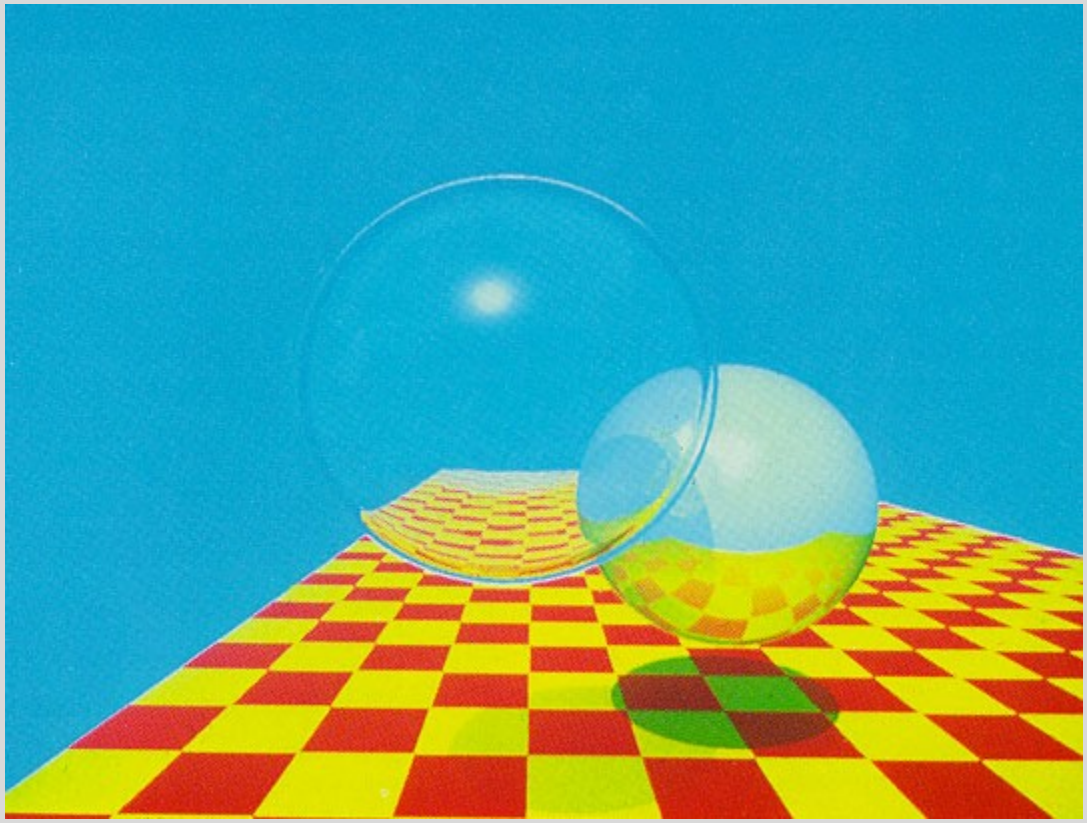
- Simplifies Light Transport
- specular to specular only, without the spread
- local illumination rays are spread (empirically), but eye rays are not
  - reflections on an object, from other items in the scene, appear as if that object was a perfect mirror
  - rays traced through objects appear as if the object was a perfect transmitter
- Typical of "quick fixes" used in CG to achieve results that look acceptable.

Albrecht Dürer (1471-1528)



Ray-tracing based on ideas employed since early Renaissance artists e.g. daVinci & Dürer

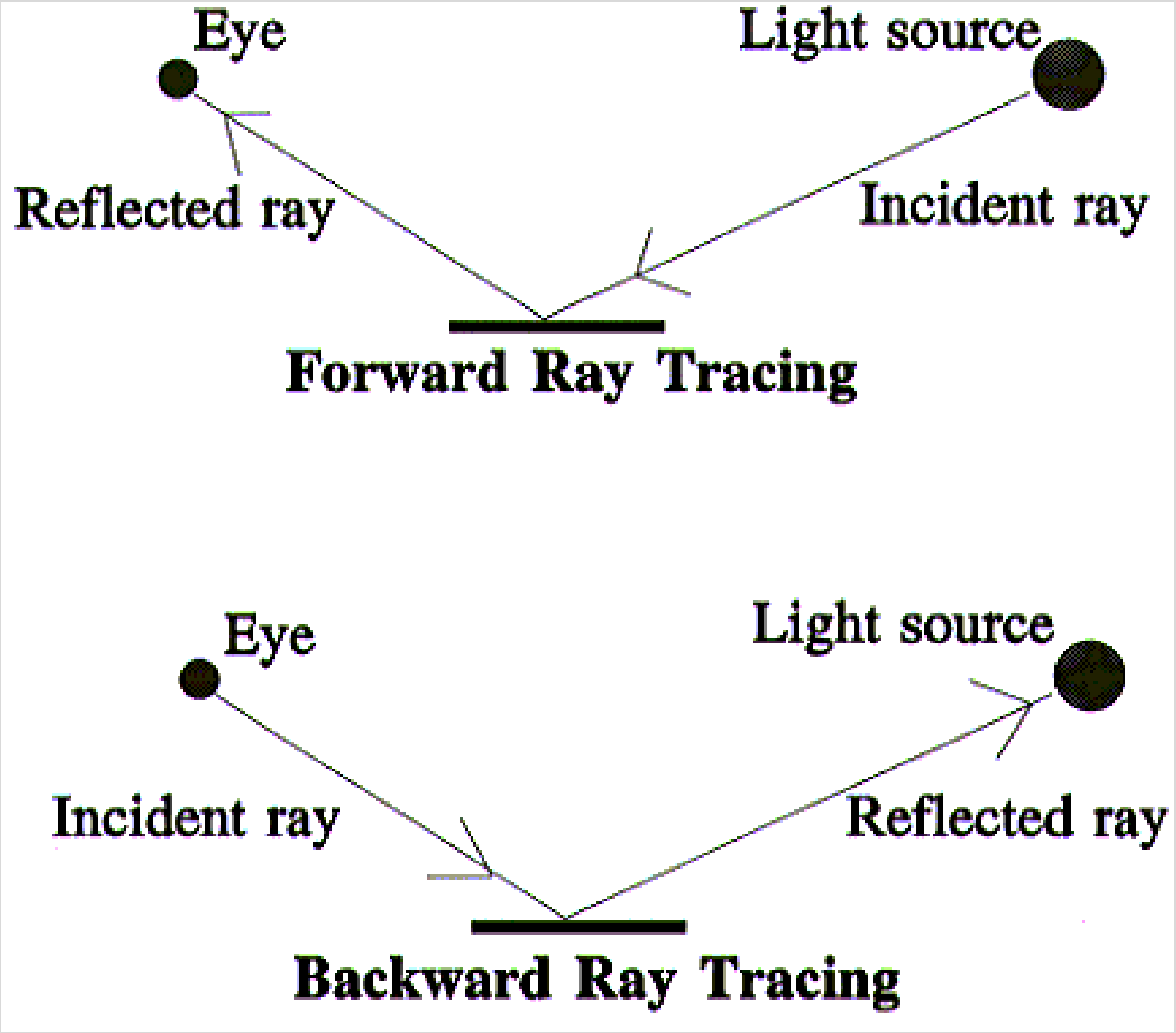
First ray-traced image



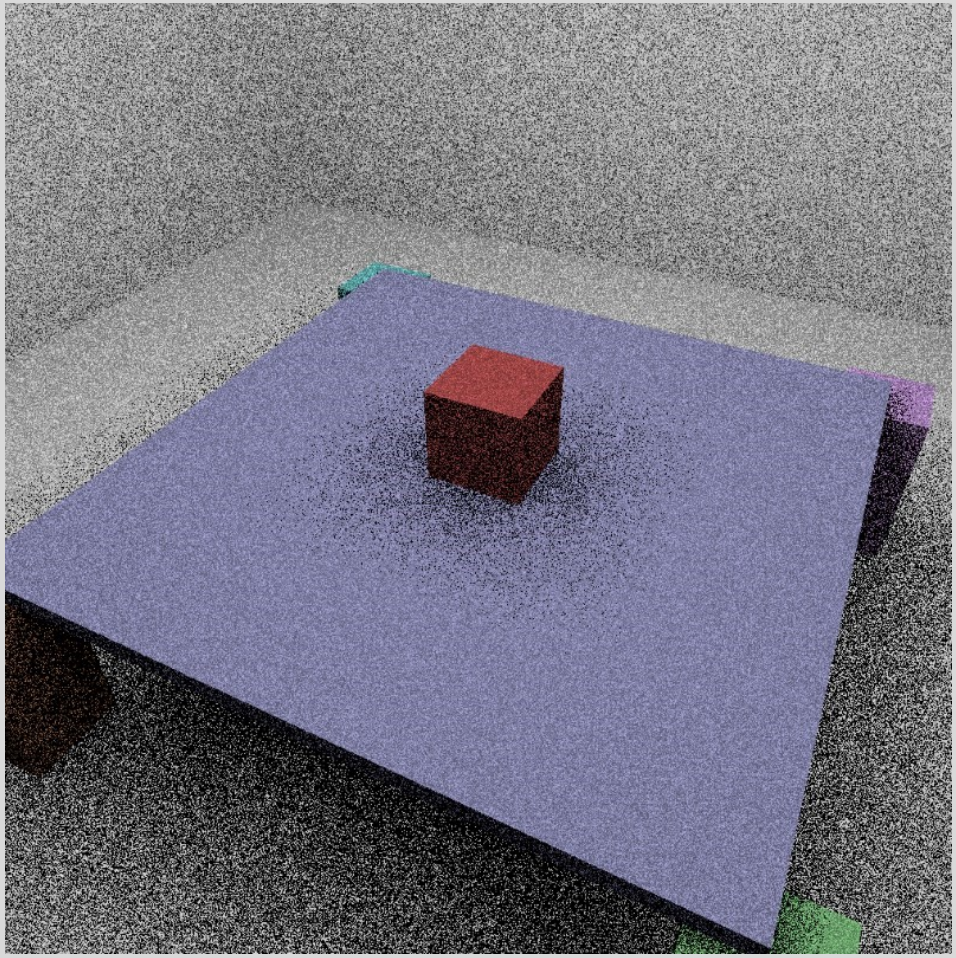
By Turner Whitted in 1979

Ray-tracing history

- First used in computer graphics in 1980
  - "super-real" images at a high cost
  - Integrated reflection, refraction, hidden surface removal, shadows in a single model
  - Rays usually considered to be infinitely thin
    - Reflection & refraction occur without any spreading
    - Perfectly smooth surfaces
    - Not real-world – like a wall of mirrors


Eye
Light source
Reflected ray
Incident ray
Forward Ray Tracing
Eye
Light source
Incident ray
Reflected ray
Backward Ray Tracing

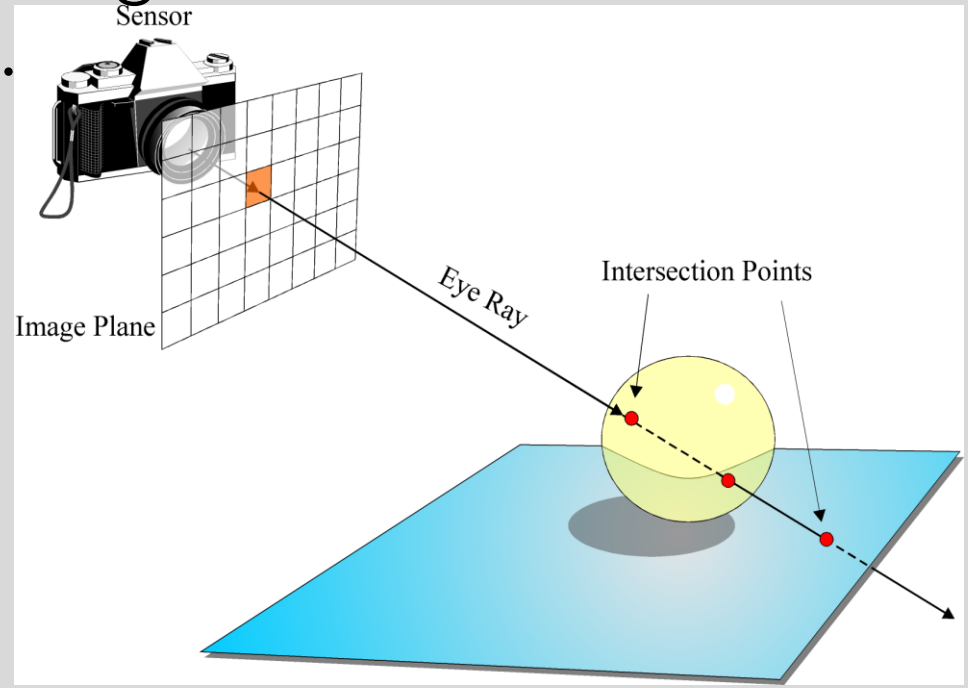
Forward Ray tracing



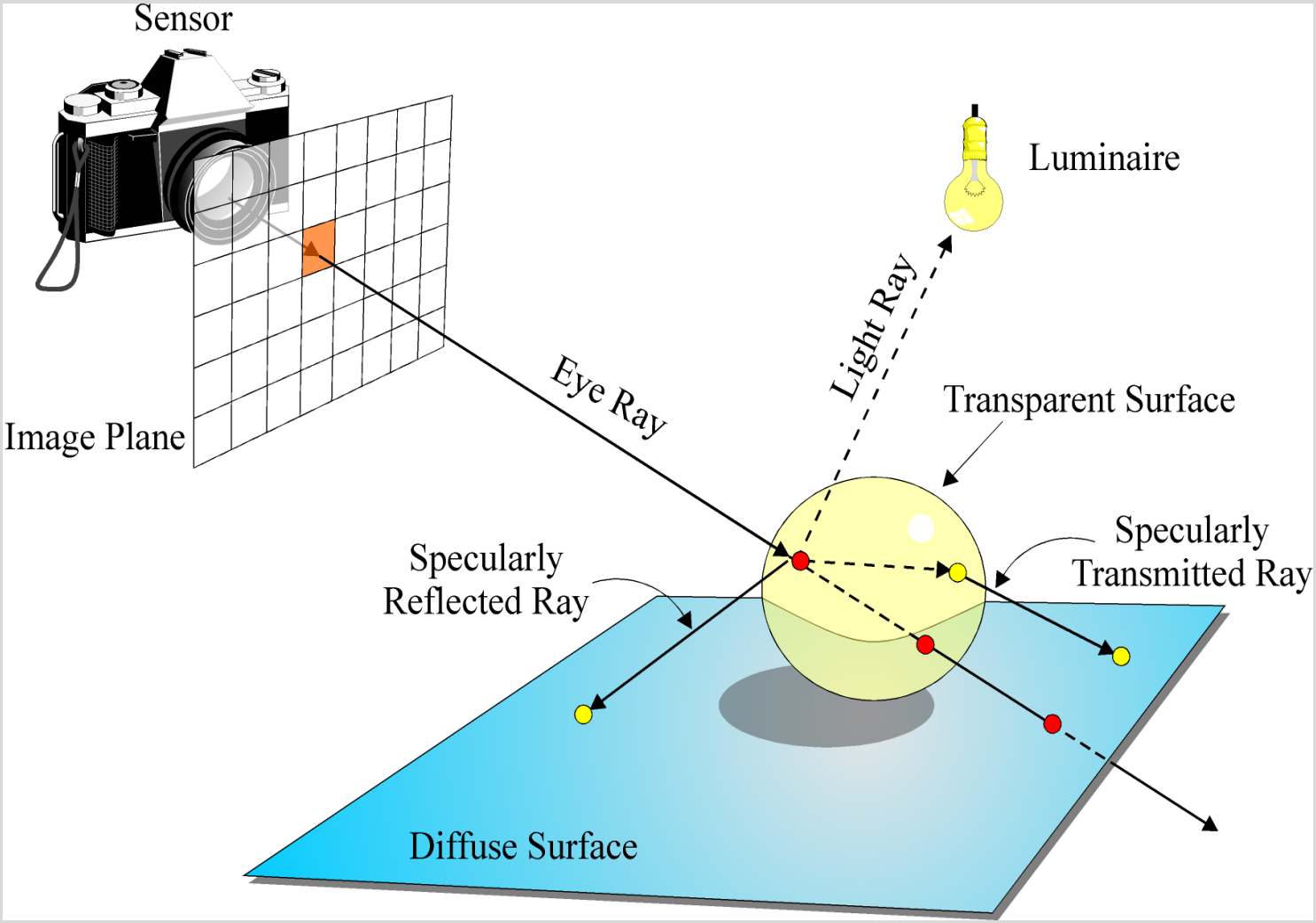
Only a fraction of rays reach the image Many, many rays are required to get a value for each pixel

Backward Ray Tracing

- For each *pixel* in the viewport, we trace a ray from the eye (called the *eye ray*) into the scene, through the pixel and determine the first object hit by the ray ⇒ *ray casting*.
- We then shade this using an extended form of the Phong Model.



Ray Tracing


Sensor
Image Plane
Eye Ray
Light Ray
Luminaire
Transparent Surface
Specularly
Reflected Ray
Specularly
Transmitted Ray
Diffuse Surface

Ray Tracing

- The eye ray will typically intersect a number of objects, some more than once ⇒ *sort intersections* to find the closest one.
- The Phong illumination model is then evaluated BUT:
  - we trace a *reflected ray* if the surface is specular
  - we trace a *refracted ray* if the surface is transparent
  - we trace *shadow rays* towards the light sources to determine which sources are visible to the point being shaded
- The reflected/refracted rays themselves will hit surfaces and we will *recursively* evaluate the illumination at these points.

⇒ a very large number of rays must be traced to illuminate a single pixel.

Ray Tracing

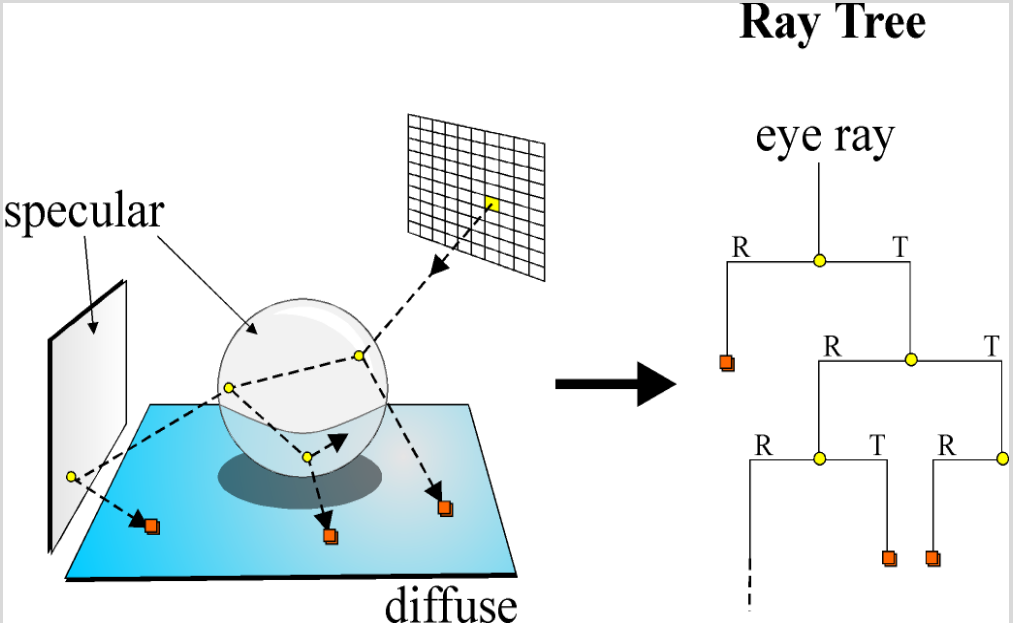
- *Whitted Illumination Model*

$$L_r(x,V) = \underbrace{L_{emitted}(x,V) + L_{Phong}(x,V)}_{\text{local contribution}} + \underbrace{L_{reflected}(x,V) + L_{refracted}(x,V)}_{\text{global contribution}}$$

- Therefore, ray tracing is a *hybrid* local/global illumination algorithm
  - we only consider global lighting effects from *ideal specular directions*
  - also, before adding each light's Phong contribution we determine if it is visible to point x, thus allowing *shadows* to be determined.

Recursive Ray Tracing

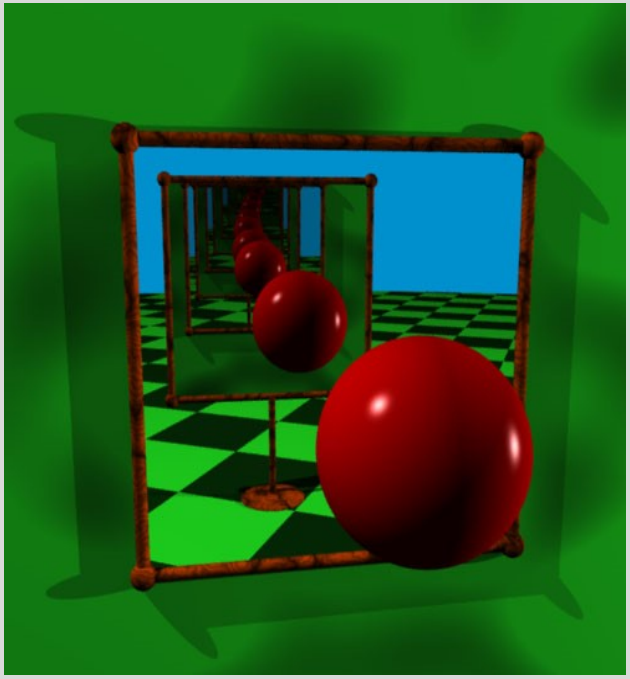
- The ray tracing algorithm is *recursive*, just as the *radiance equation is recursive*.
  - At each intersection we trace a specularly reflected and transmitted ray (if the surface is specular) or terminate the ray if diffuse.
- Thus we trace a ray *back in time* to determine its history, beginning with the *eye ray*: this leads to a *binary tree*



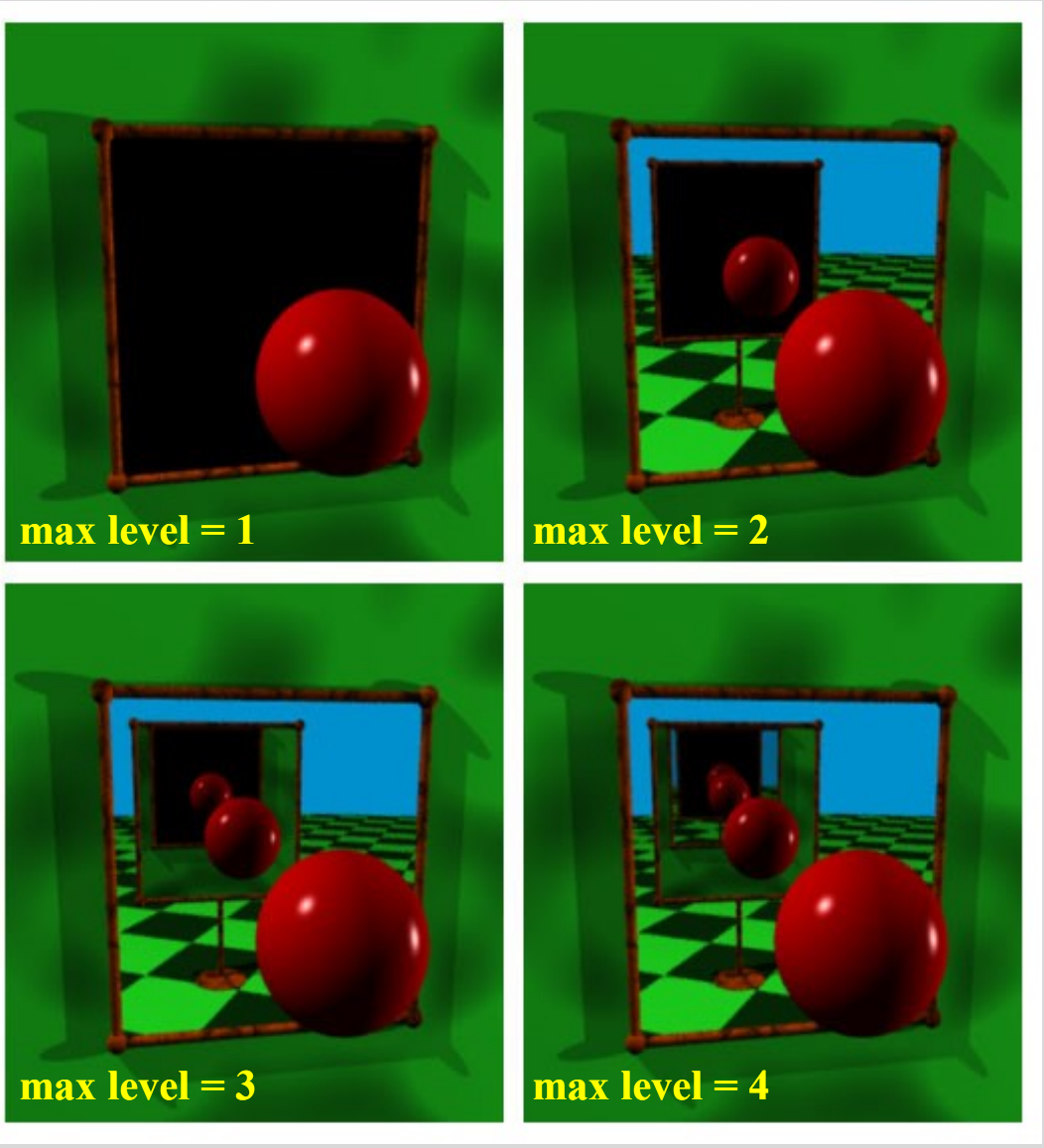
Recursive Ray Tracing

- Theoretically, this recursive process could continue indefinitely.
- In practice, at each intersection the ray loses some of its contribution to the pixel (i.e. its *importance* decreases).
  - if the eye ray hits a specularly reflecting surface with reflectivity of 50%, then only 50% of the energy hitting the surface from the reflected direction is reflected towards the pixel.
  - if the next surface hit is of the same material, the reflected ray will have its contribution reduced to 25%.
- We terminate the recursion if:
  - the current recursive depth > a *pre-determined maximum depth* or
  - if the ray's contribution to the pixel < some *pre-determined threshold* ε

Recursion Clipping



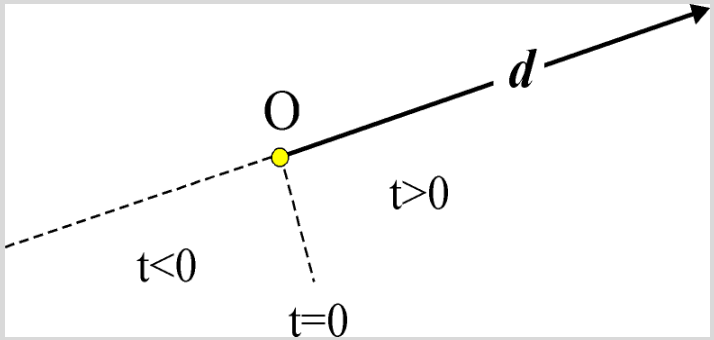
Very high maximum recursion level



The Ray

- A ray is mathematically the *affine half-space* defined by:

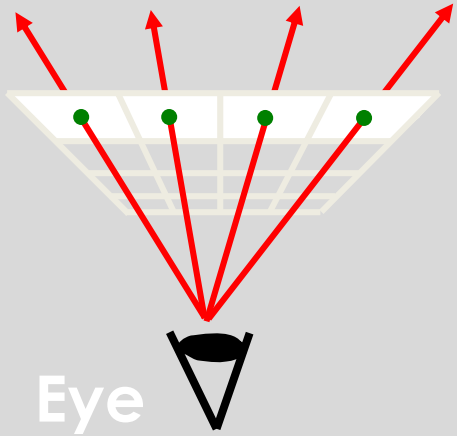
$$\mathbf{r} = O + t\vec{d} \quad t \geq 0$$



- All points on the ray correspond to some positive value of *t*, the *parametric distance* along the ray. If ***d*** is *normalised* then t is the length along the ray of the point.

The Ray Tracing Algorithm

```
for each pixel in viewport
{
    determine eye ray for pixel
    intersection = trace(ray, objects)
    colour = shade(ray, intersection)
}
```




```
trace(ray, objects)
{
    for each object in scene
        intersect(ray, object)
    sort intersections
    return closest intersection
}
```

Ray Tracing Algorithm

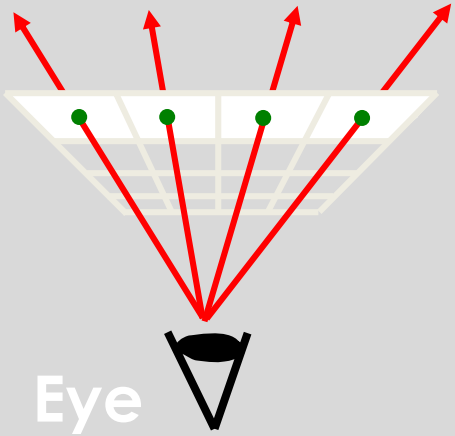
```
colour shade(ray, intersection)
{
    if no intersection
        return background colour

    for each light source
        if(visible)
            colour += Phong contribution

    if(recursion level < maxlevel and surface not diffuse)
    {
        ray = reflected ray
        intersection = trace(ray, objects)
        colour += ρ_refl*shade(ray, intersection)
      }
    return colour
}
```

Ray Casting

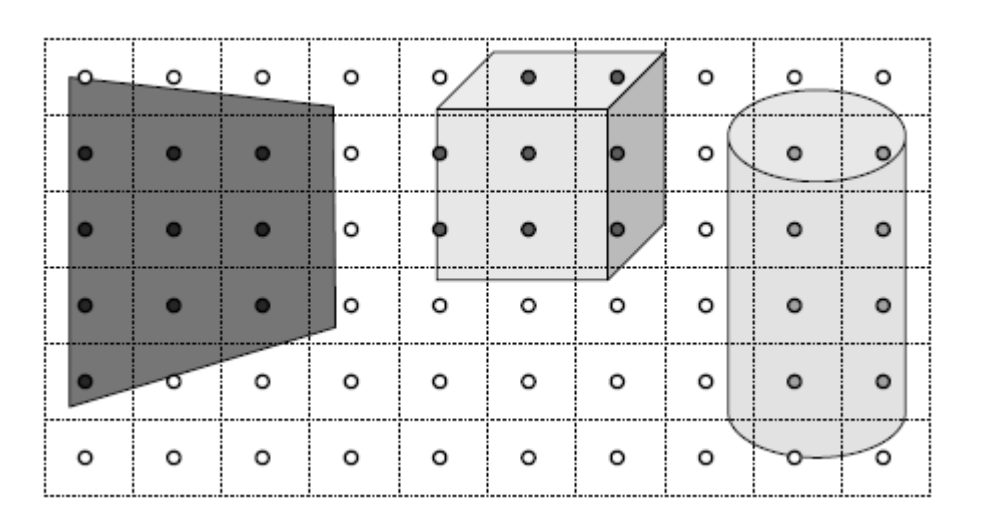
```
for each pixel in viewport
{
    determine eye ray for pixel
    intersection = trace(ray, objects)
    colour = shade(ray, intersection)
}
```




```
trace(ray, objects)
{
    for each object in scene
        intersect(ray, object)
    sort intersections
    return closest intersection
}
```

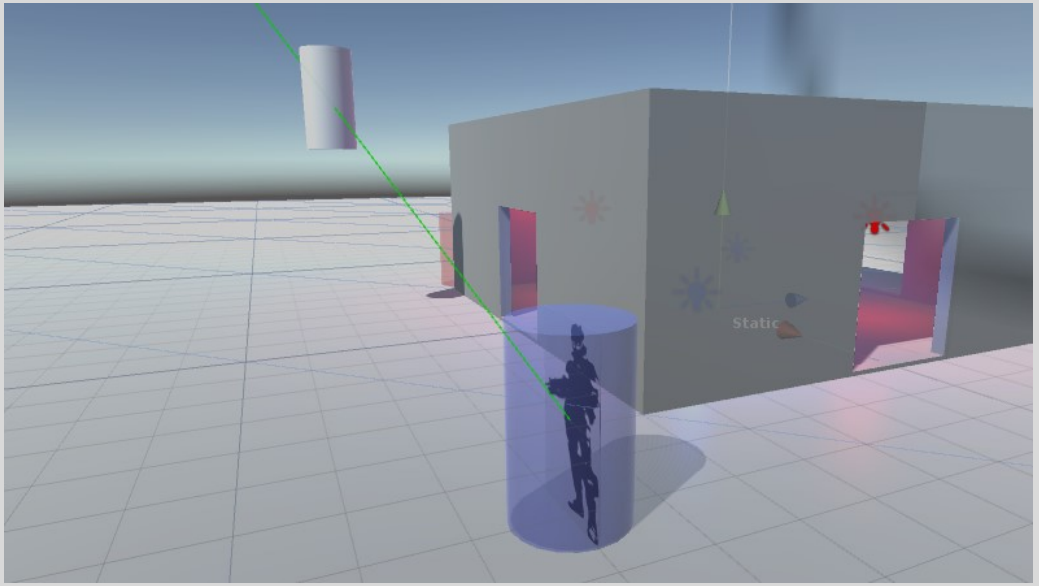
Ray Casting

- For each sample
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute colour sample based on surface radiance



Other uses of Ray Casting

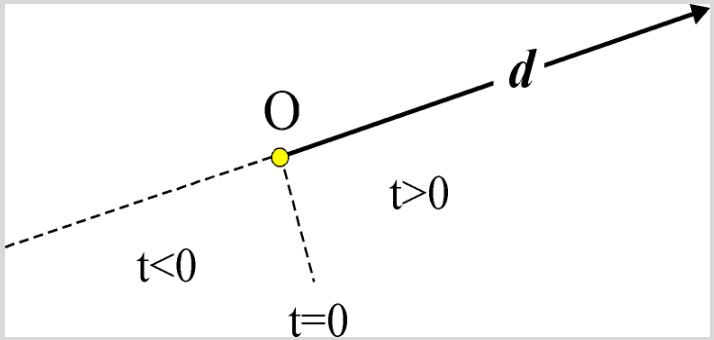
- Collision detection
- Shooting a gun
  - What does the bullet intersect?
- Line of sight
  - What's in front of me?
- Occlusion culling



The Ray

- A ray is mathematically the *affine half-space* defined by:

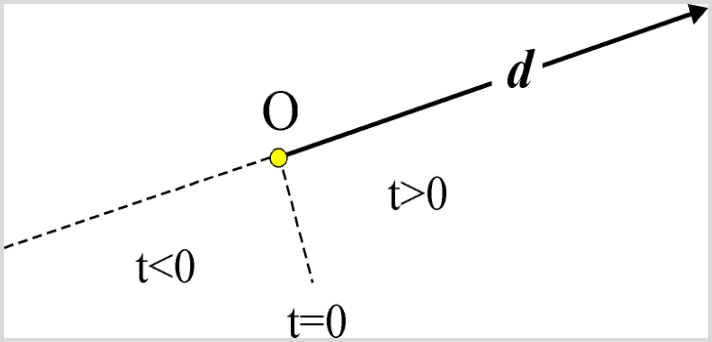
$$\mathbf{r} = O + t\vec{d} \quad t \geq 0$$



- All points on the ray correspond to some positive value of *t*, the *parametric distance* along the ray. If ***d*** is *normalised* then t is the length along the ray of the point.

Ray Object Intersection Testing

- Once we've constructed the eye rays we need to determine the *intersections* of these rays and the objects in the scene.
- Upon intersection we need the *normal* to the object at the point of intersection in order to perform shading calculations.



Ray Object Intersection Testing

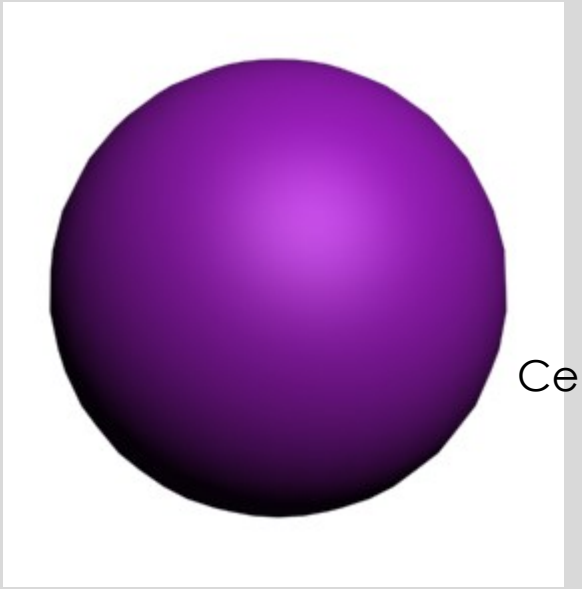
- Usually objects are defined either *implicitly* or *explicitly* (*parametrically*).
- Implicit:

$$f(\vec{v}) = f(v_0, v_1, \ldots, v_n) \quad \begin{cases} < 0 \text{ if } \vec{v} \text{ inside surface} \\ = 0 \text{ if } \vec{v} \text{ on surface} \\ > 0 \text{ if } \vec{v} \text{ outside surface} \end{cases}$$

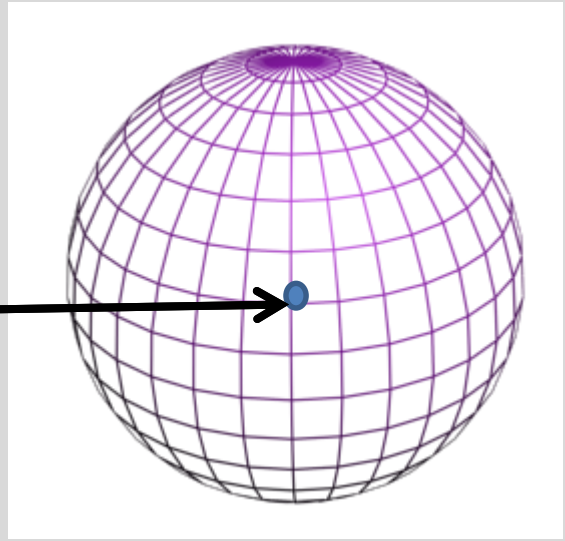
  - the set of points on the surface are the zeros of the function *f*.
- Explicit: $S^n = f(\alpha_0, \alpha_1, \ldots, \alpha_n)$
  - α_n are the generating parameters and usually have infinite ranges
  - for surfaces in 3-space, S^2, there will be 2 generating parameters.
  - we iterate over all parameters to generate the set of all points on the surface.

The Sphere

- A sphere of center (Cx, Cy, Cz) with radius r is given by: $f(x,y,z)=(x-C_x)^2+(y-C_y)^2+(z-C_z)^2-r^2=0$
- Question: is the point (5,1,0) on this sphere?



Centre = (0, 0, 0)



10cm

The Sphere

- A sphere object is defined by its center C and its radius *r*.
- *Implicit Form*: $f(\vec{v}) = |\vec{v} - C|^2 - r^2 = 0$

$$f(x, y, z) = (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 - r^2 = 0$$

- *Explicit Form*:

$$x = f_x(\theta, \phi) = C_x + r \sin\theta \cos\phi$$
$$y = f_y(\theta, \phi) = C_y + r \cos\theta$$
$$z = f_z(\theta, \phi) = C_z + r \sin\theta \sin\phi$$

- We can use either form to determine the intersection; we will choose the implicit form.

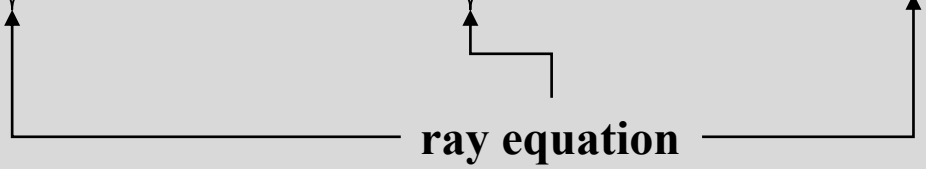
Ray Sphere Intersection

- All points on the ray are of the form: $\mathbf{r}ay = O + t\vec{d} \quad t \geq 0$
- All points on the sphere satisfy:

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 - r^2 = 0$$

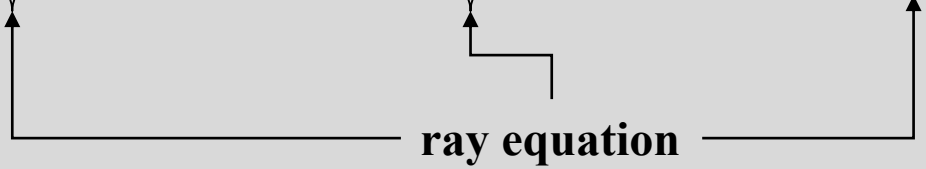
- any intersection points (= points shared by both) must satisfy both, so substitute the ray equation into the sphere equation and solve for *t*:

$$([O_x + td_x] - C_x)^2 + ([O_y + td_y] - C_y)^2 + ([O_z + td_z] - C_z)^2 - r^2 = 0$$



Problem

$$(([O_x + td_x] - C_x)^2 + ([O_y + td_y] - C_y)^2 + ([O_z + td_z] - C_z)^2 - r^2 = 0$$



- Expand the first term
  - remember $(a-b)^2 = a^2 - 2ab + b^2$
- Rearrange into: $At^2 + Bt + C = 0$

Ray Sphere Intersection

- Rearrange and solving for *t* leads to a *quadratic form* (which is to be expected as the sphere is a quadratic surface):

$$At^2 + Bt + C = 0$$

$$A = \left(d_x^2 + d_y^2 + d_z^2\right) = 1$$

$$B = 2d_x\left(O_x - C_x\right) + 2d_y\left(O_y - C_y\right) + 2d_z\left(O_z - C_z\right)$$

$$C = \left(O_x - C_x\right)^2 + \left(O_y - C_y\right)^2 + \left(O_z - C_z\right)^2 - r^2$$

- We employ the classic quadratic formula to determine the 2 possible values of *t*:

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} = \frac{-B \pm \sqrt{B^2 - 4C}}{2}$$

Intersection Classification

- Depending on the number of *real roots* we have a number of outcomes which have nice geometric interpretations:
  - we use the *discrimin*

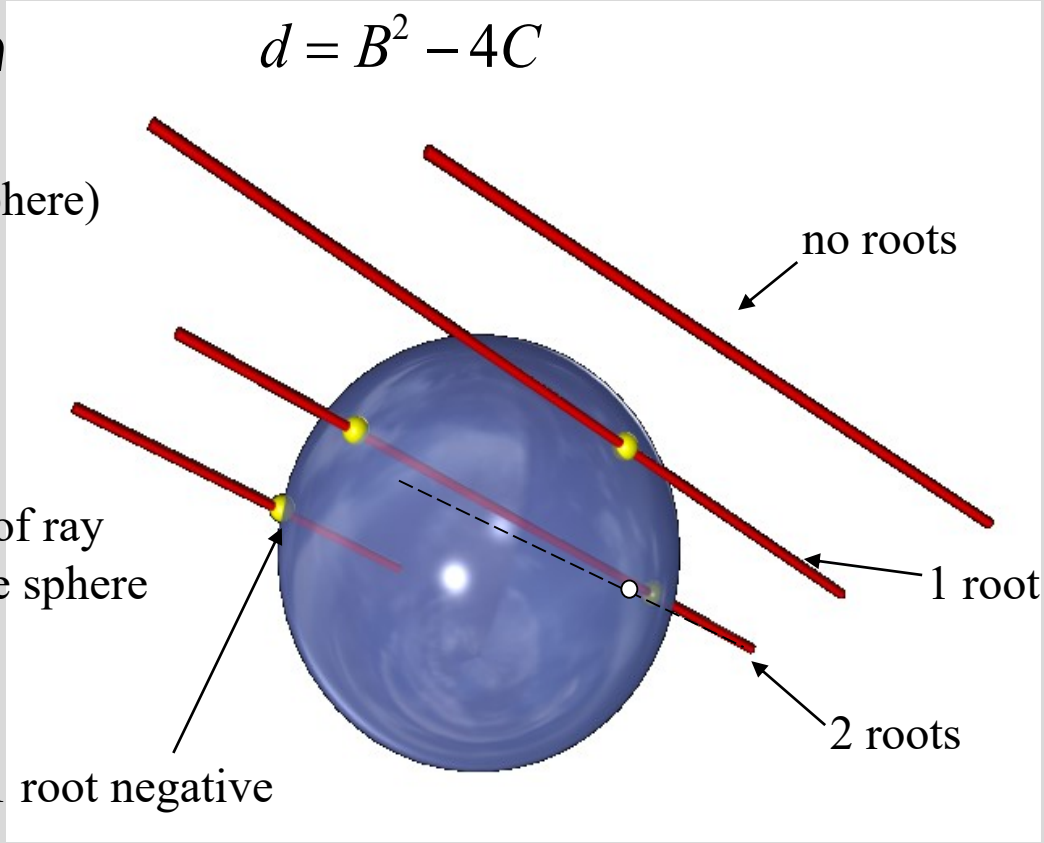
$$d = B^2 - 4C$$

$d = 0 \Rightarrow$ 1 root (ray is tangent to sphere)

$d < 0 \Rightarrow$ no real roots (ray misses)

$d > 0 \Rightarrow$ 2 real roots:

Both positive $\Rightarrow$ sphere in front of ray
One negative $\Rightarrow$ ray origin inside sphere



Ray Sphere Intersection Test

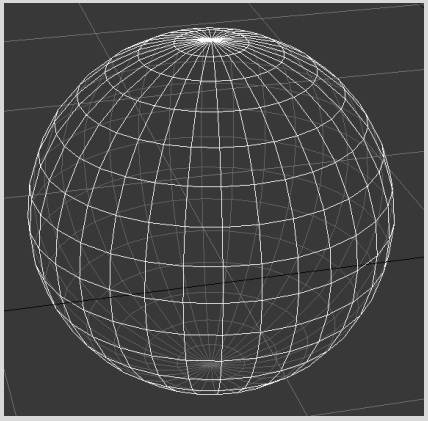
- If we have 2 positive values of *t* we use the smallest (i.e. the nearest to the origin of the ray).
- *t* is substituted back into the ray equation yielding the point of intersection:

$$P_{int} = O_{ray} + t_{int} d_{ray}$$

- We then evaluate the *Phong model* at this point. To do so we need the normal to the surface of the sphere at the point of intersection.
- The normal and the original ray direction are then used to determine the directions of *reflected* and *refracted* rays.

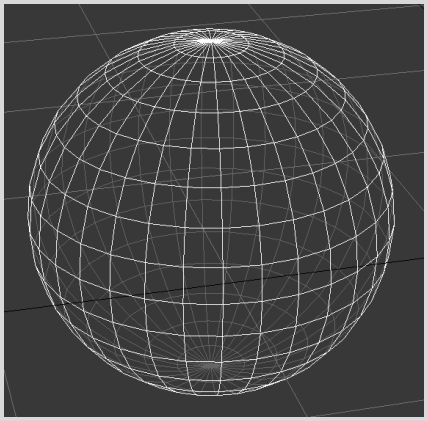
Normal to Sphere

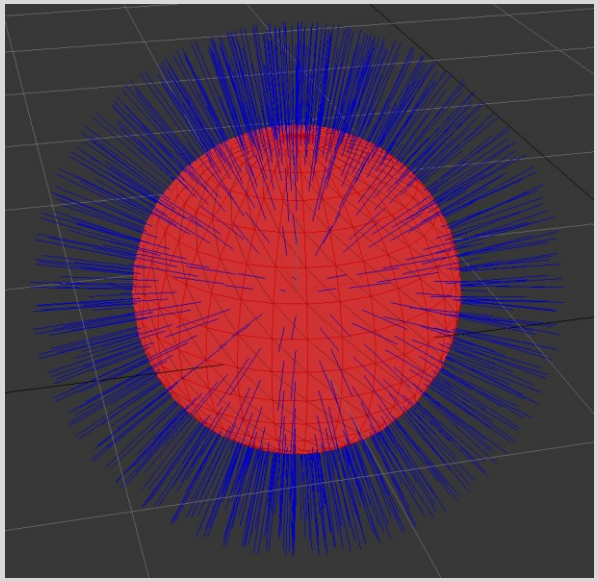
- We can compute the normal to a sphere at a point x:



Normal to Sphere

- We can compute the normal to a sphere at a point x:

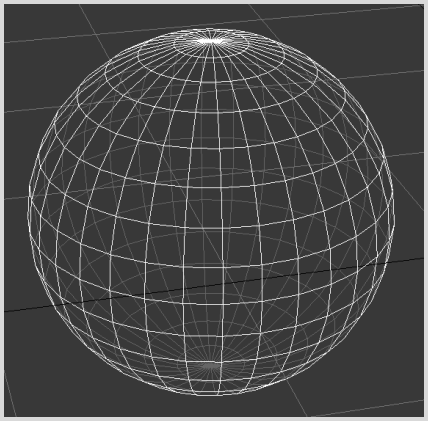


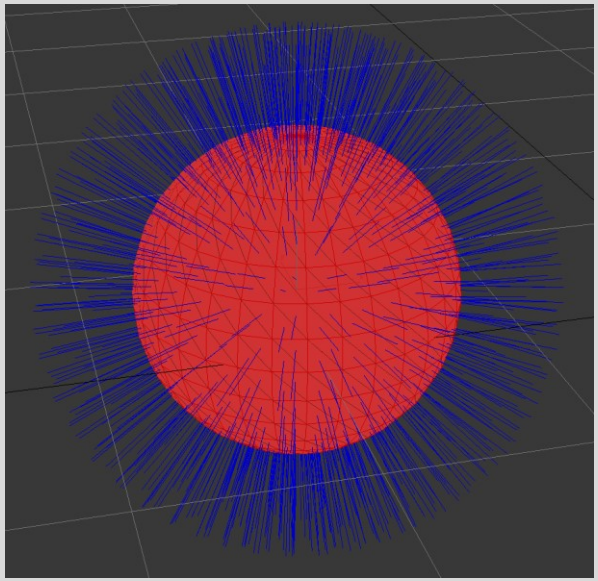


Normal to Sphere

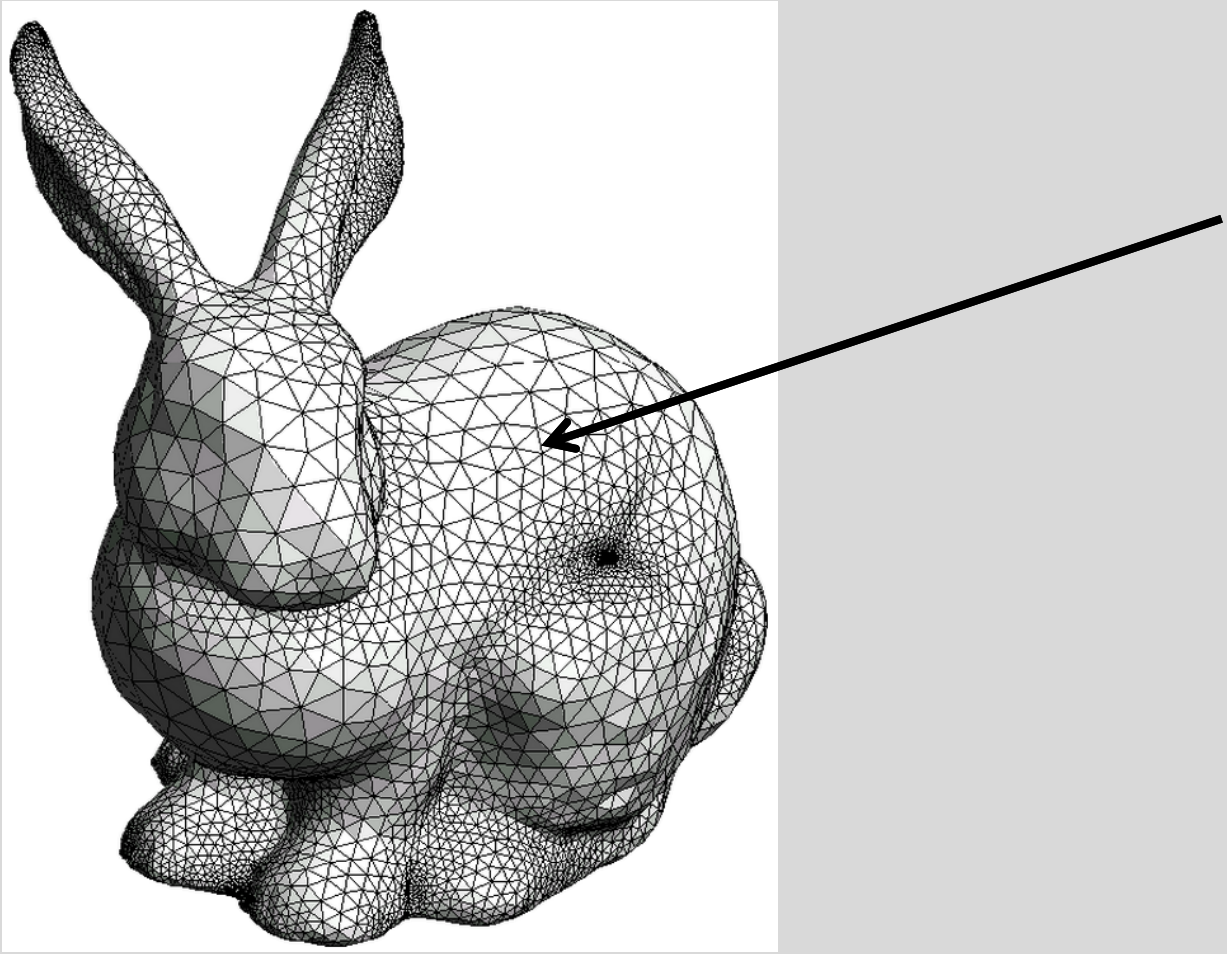
- We can compute the normal to a sphere at a point x: $N = \frac{\vec{x} - C}{|\vec{x} - C|}$

i.e. the normal to the sphere is the normalised vector associated with the point. For a sphere with center C we compute the normal as:





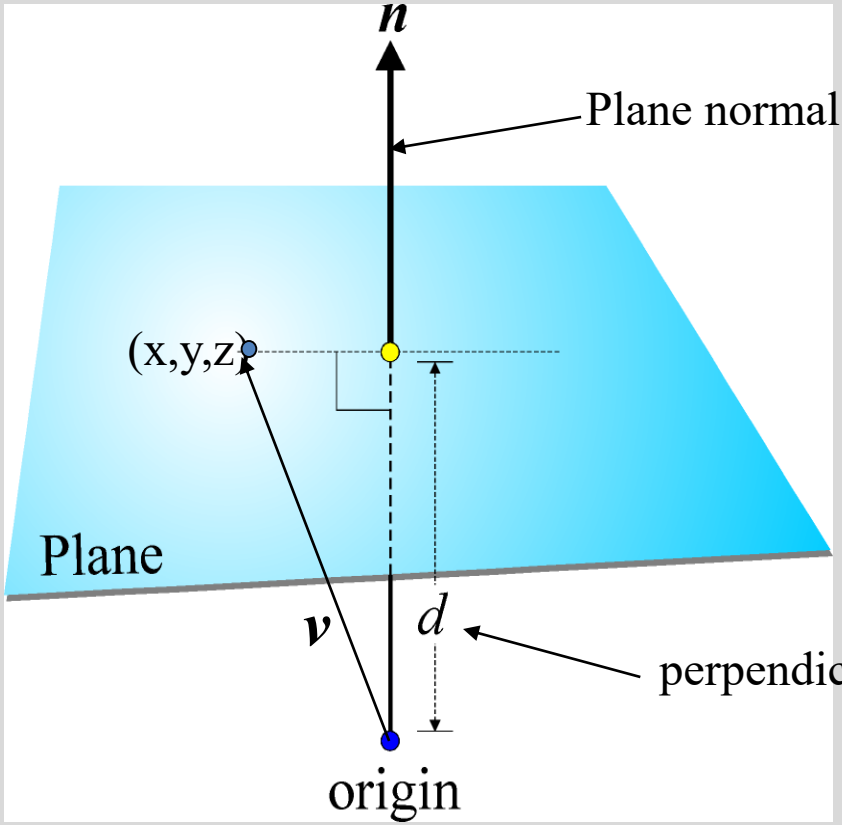
Ray Polygon Intersection



Ray Plane Intersection

- A *plane* may be defined by its normal and the perpendicular distance of the plane to the origin

$$\vec{v} \cdot \vec{n}_P = d_p \quad \text{or} \quad n_x x + n_y y + n_z z - d_P = 0$$



perpendicular distance to the origin (d_p)

Ray Plane Intersection

- Does ray intersect plane?

Ray Plane Intersection

- Does ray intersect plane?
- dot product between the normalised ray and the triangle's normal $\vec{n}_P.\vec{d}_r$
  - If the result == 0 means that ray & normal are perpendicular
    - No intersection!
  - If the result > 0 means that the ray and the triangle's normal are in the same direction
    - -> back face intersection
  - if the result < 0 then you know they intersect (plane facing direction of ray)
- Note there can only be *one intersection*. The normal to the plane at each point is the same, i.e. the plane normal $\boldsymbol{n_P}$

Ray Plane Intersection

- To intersect a ray with a plane we use the same approach as with the sphere i.e. substitute in the ray equation and solve for t
-
$$\left(O_r + td_r\right)\cdot \vec{n}_P = d_P$$
- *O* is origin of the ray and d_r is the direction of the ray
- solve for *t* and that will give you the point on the plane

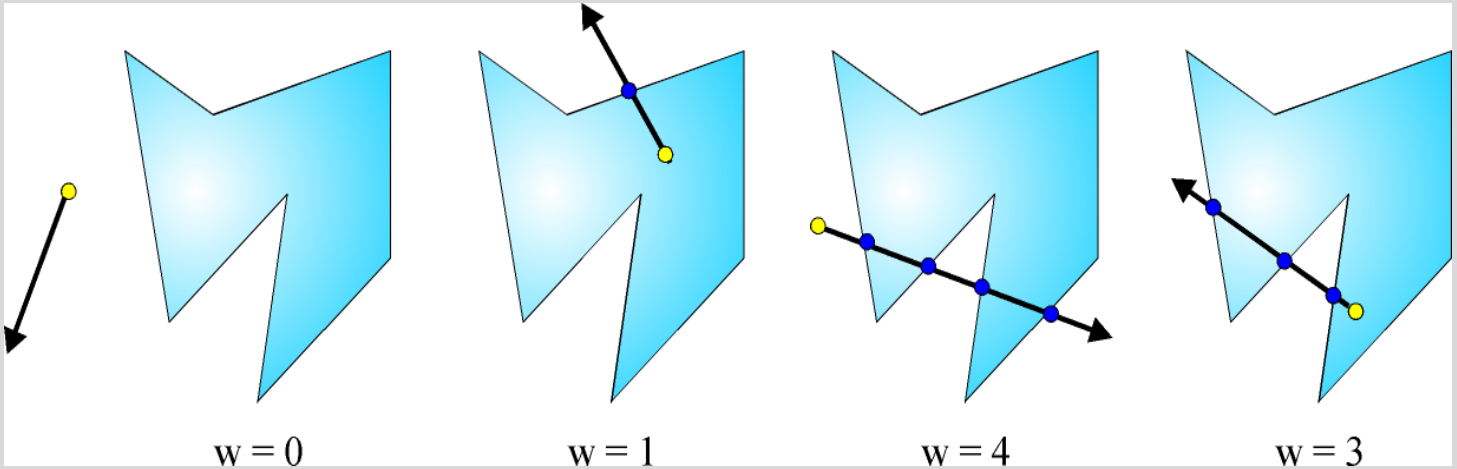
$$\Rightarrow t = \frac{d_P - \vec{n}_P \cdot O_r}{\vec{n}_P \cdot \vec{d}_r}$$

- *t* is substituted back into the ray equation yielding the point of intersection:

$$P_{int} = O_{ray} + t_{int} d_{ray}$$

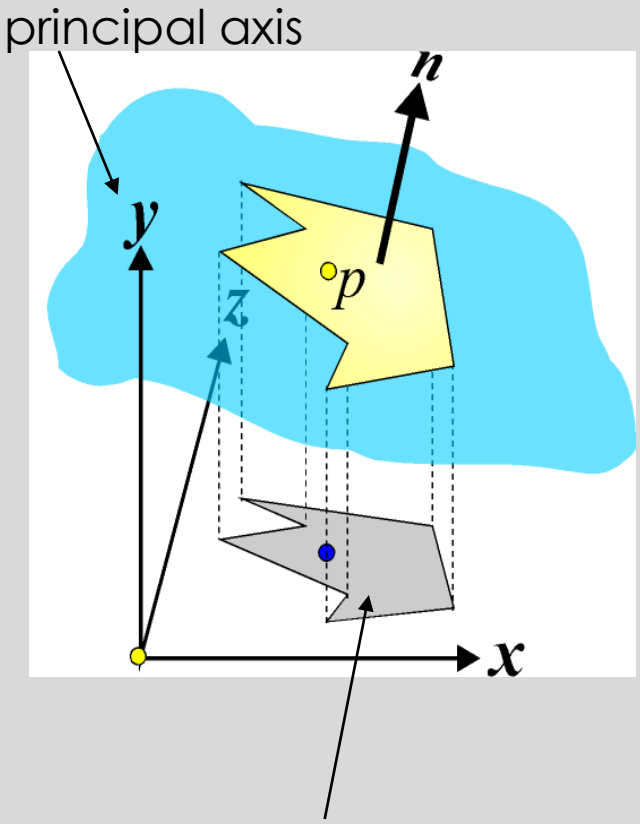
Ray Polygon Intersections

- First intersect the ray with the polygon plane: given this intersection point (assuming ray is not parallel to the plane) determine if it is within the polygon interior using the *Jordan Curve Theorem*:
  - construct any ray with the intersection point as an origin
  - count the number of polygon edges this ray crosses (= *winding number*)
  - if *w* is odd then the point is in the interior



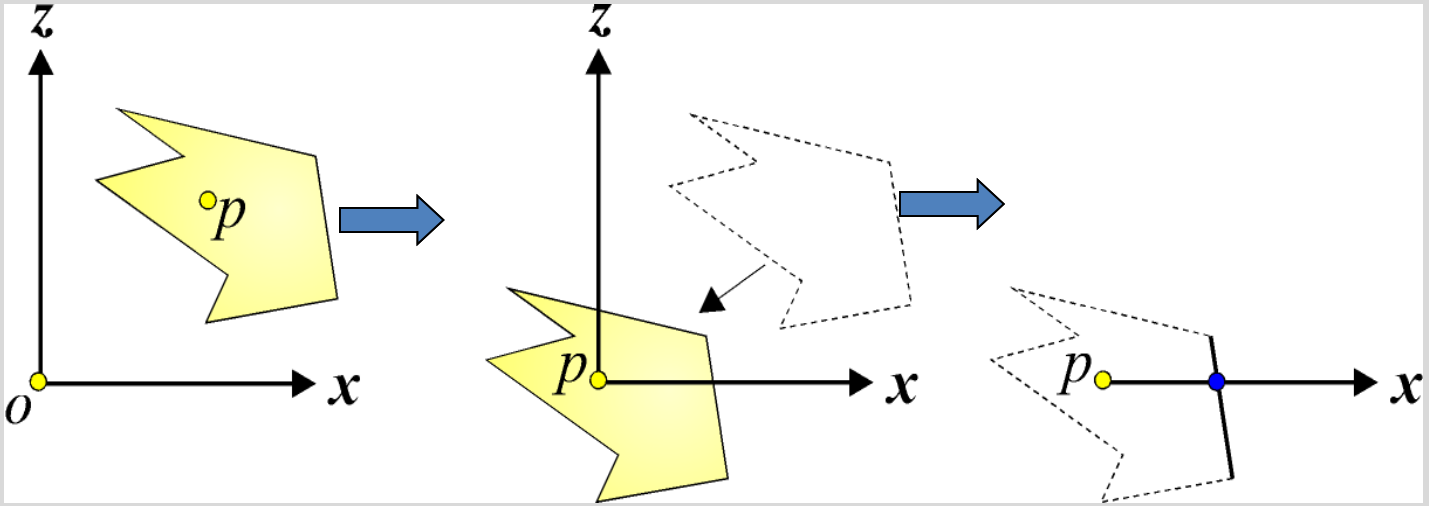
Ray Polygon Intersections

- Note that this is essentially a 2 dimensional problem:
  - after intersecting the ray and the plane, reduce the problem to 2D by projecting parallel to the polygon's *principal axis* onto the plane formed by the other 2 axes
  - the principal axis is given by the *largest ordinate* of the polygon normal
  - e.g. if ***n*** = [*0.2 0.96 0.3*] what is the principal axis?
  - What plane do we project onto?
  - This projection preserves the *topology* (assuming the projection direction is not parallel to the polygon)



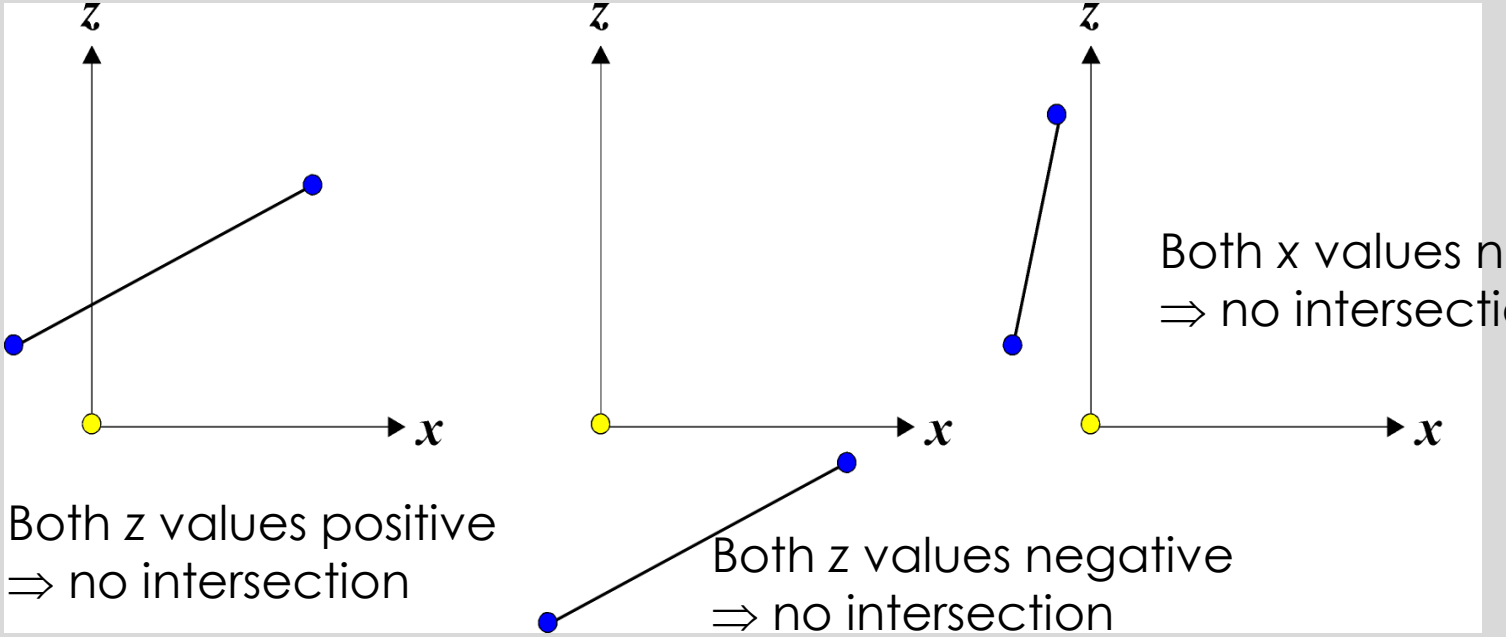
Ray Polygon Intersections

- Note that this projection is *constant for each polygon*, therefore we can pre-compute the polygon projection and principal axis.
- Once projected, the *winding number* is determined.
  - translate the intersection point to the origin (apply same translation to all projected polygon vertices).
  - use an axis (not the principal axis) as the direction to try
  - determine the number of *edge crossings* along the *positive axis*.



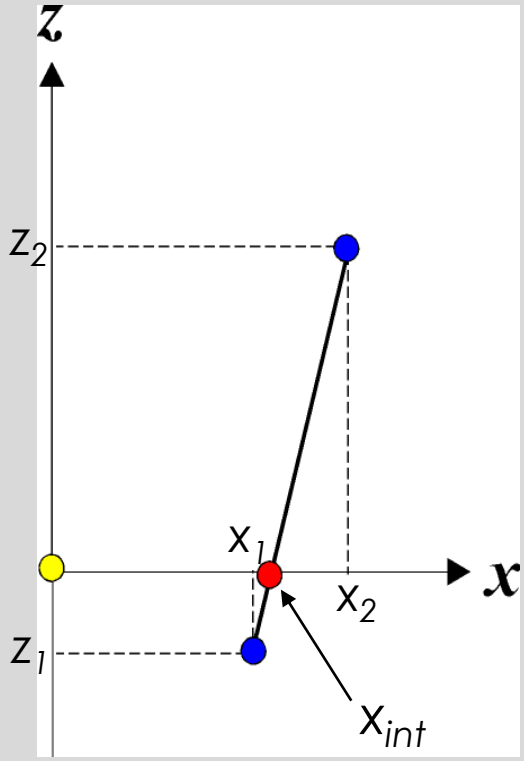
Ray Polygon Intersections

- We can speed up the process of determining edge intersection with an axis. There are 3 redundant cases of edge axis topology:



Ray Polygon Intersections

- If none of these cases are satisfied then we must compute the intersection with the axis and determine if it lies on the positive half of the axis:



Remember the general equation of a line:

$$y = mx + c$$

c: intersection with the *y* axis

$$x_{int} = x_1 - z_1 \frac{(x_2 - x_1)}{(z_2 - z_1)}$$

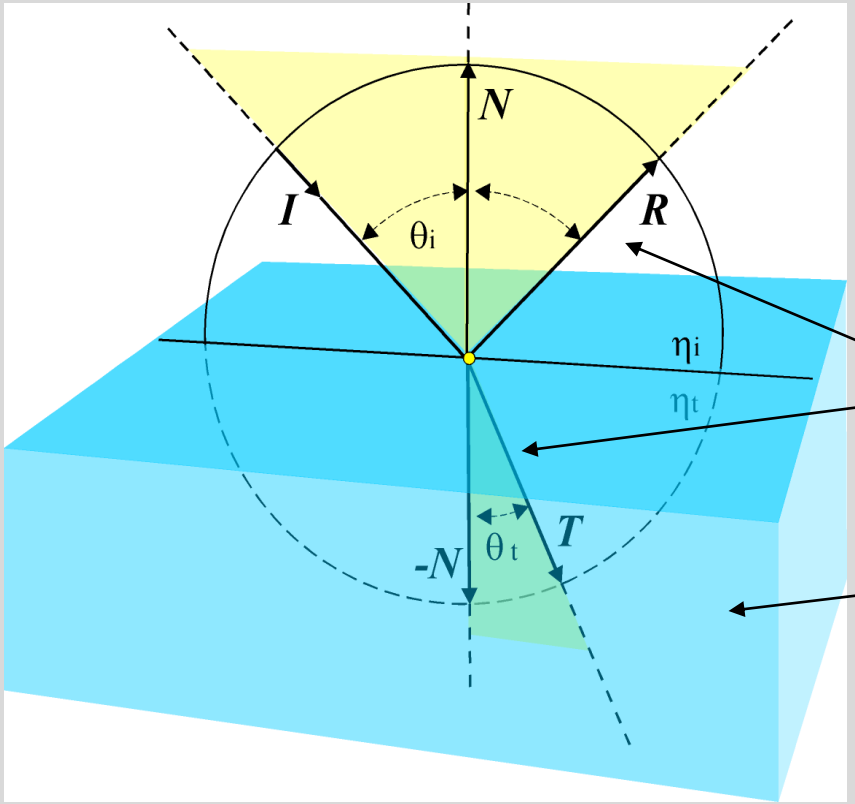
If $x_{int} > 0$ then the edge crosses the axis and the winding number is incremented.

Summary

1. Assume planar polygons
2. Check if ray is parallel or behind plane
3. If not, find intersection with plane
   - Substitute the ray equation into the plane equation
4. Now that you have intersection point on the plane, find out if it is contained in the polygon

Constructing Secondary Rays

- Having determined the closest intersection along a ray path we then evaluate the *Whitted illumination model* at this point.
- This requires the construction of a *reflected* and a *refracted* ray:



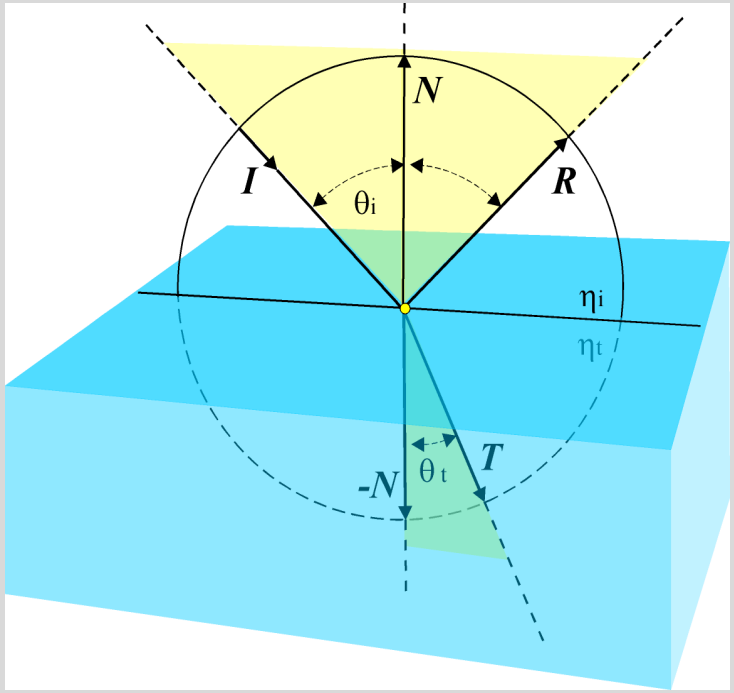
N, ***I***, ***R*** and ***T*** all lie in the same plane

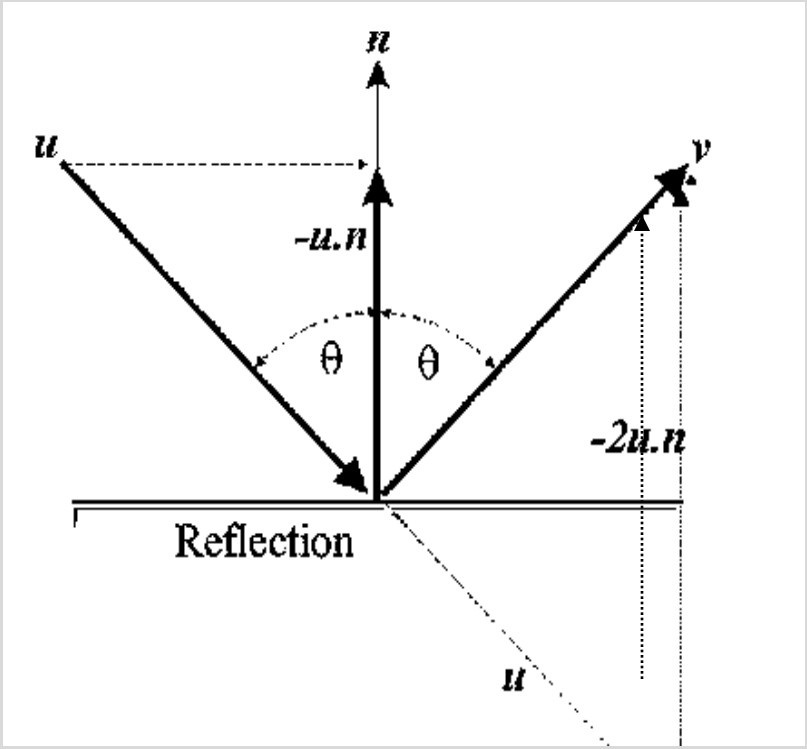
secondary rays

reflective and transparent medium (e.g. glass)

Secondary Rays

- As discussed previously, the rays will have the intersection point, **x**, as their *origin*.
  - The *reflected ray direction* ***R*** is given by: $R = I - 2N(I \cdot N)$

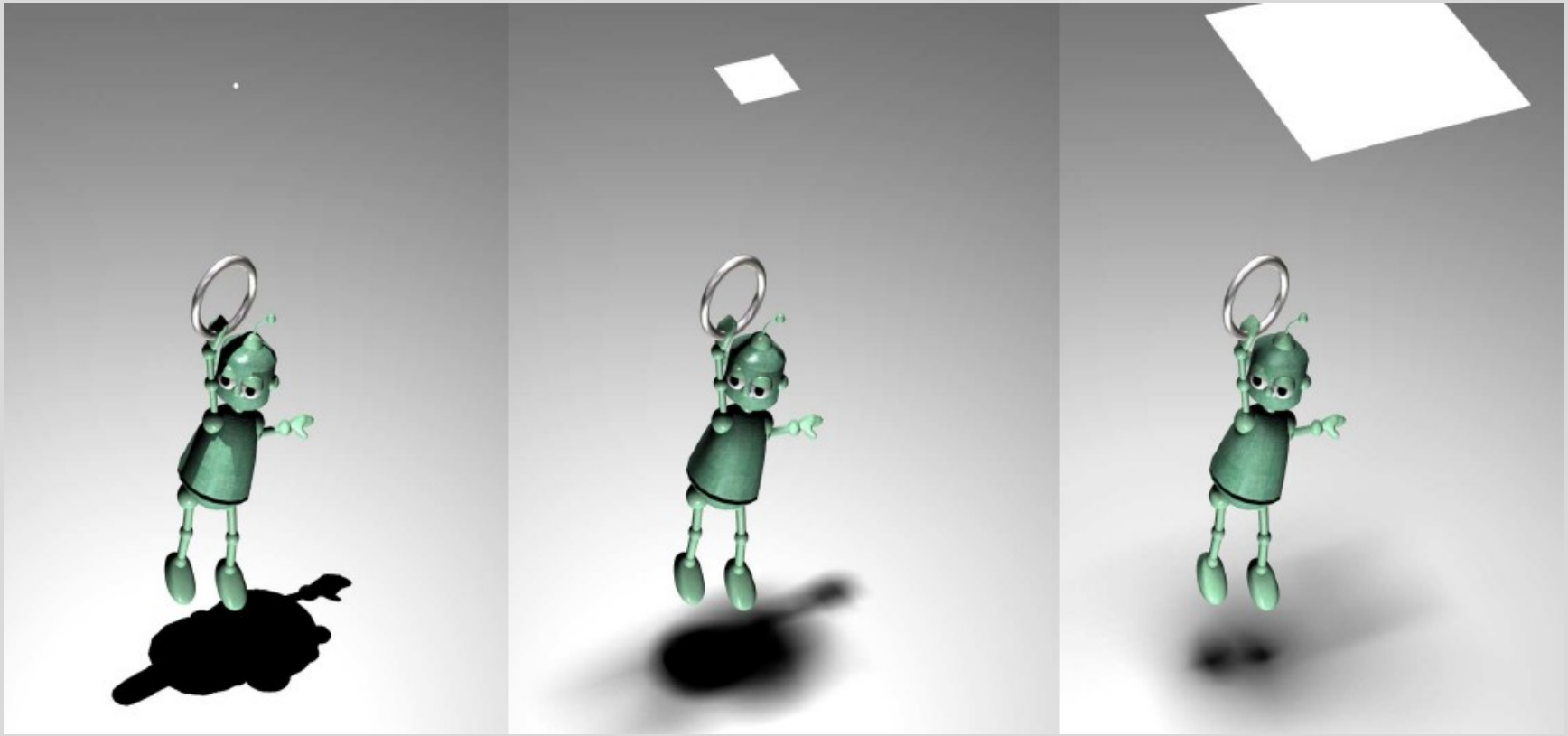




Shadows

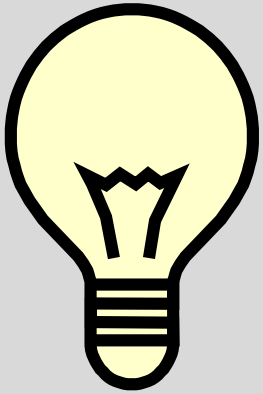
- Whether a point is in shadow or not is determined by casting a ray from each intersection point to the light source
  - Shadow feeler
- If it intersects any object then the point of interest is deemed to be in shadow
  - Easier than ray/object intersections, as we only need to know if intersection has occurred (do not need to find the nearest object)
- Shadow calculations impose a computational overhead in ray tracing that increases rapidly as the number of light sources increases

Soft Shadows



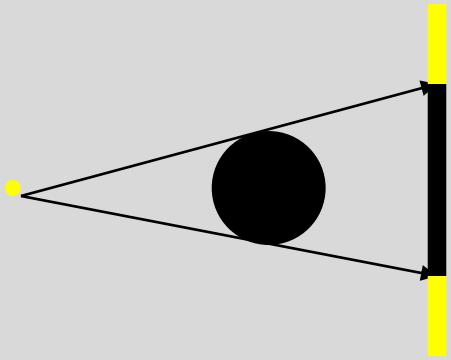
Point Light Sources

- A “point” light source is usually not truly a point light source. Considering a light source as a point is just a convenient model.
- Consider a light bulb, for example. It is not an infinitesimally small point. It has volume.
- Implication: Real light sources produce soft shadows.

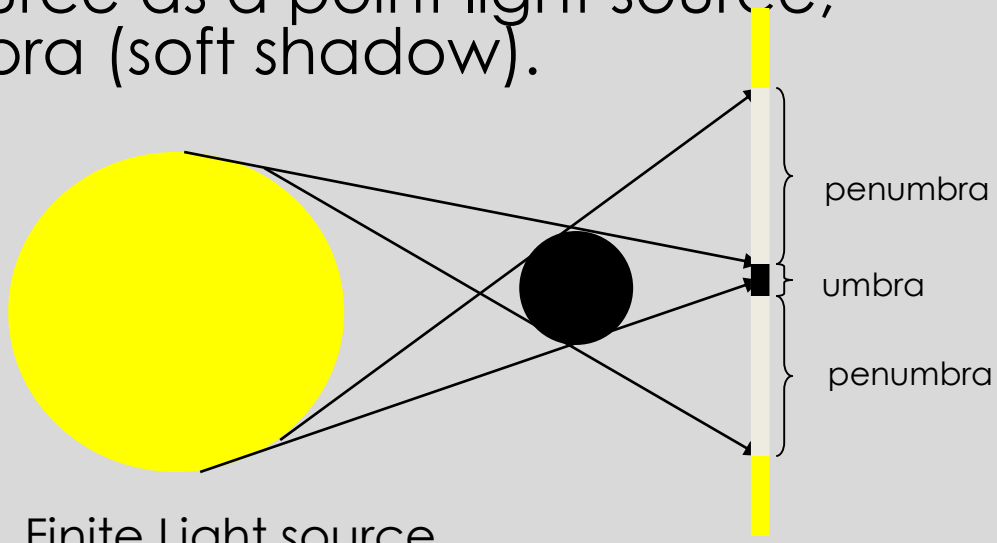


Soft Shadows

- Shadows are not uniformly dark.
- The shadow is divided into two parts: the umbra and penumbra.
- No light at all from the light source reaches the umbra. It is completely dark.
- Some light from the light source reaches the penumbra. It is partially dark.
- If we model the light source as a point light source, there will be no penumbra (soft shadow).



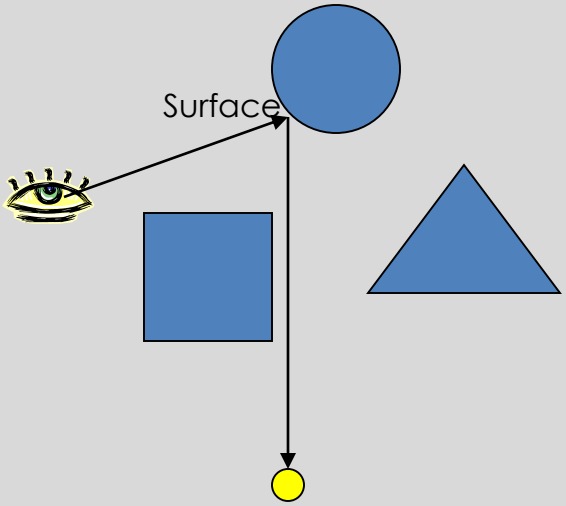
Point Light source



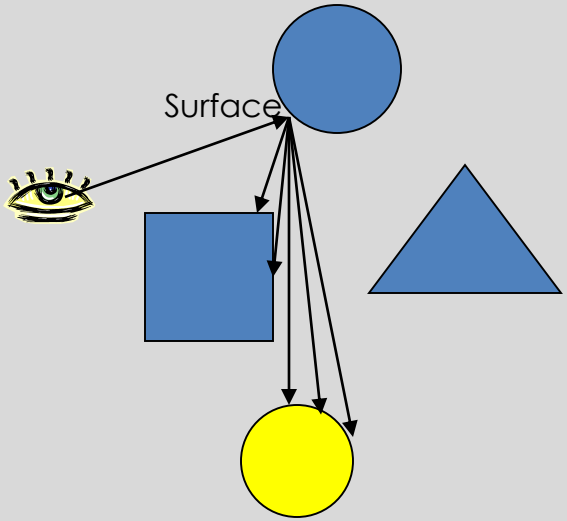
Finite Light source

Soft Shadows in Ray Tracing

- For more realistic shadows, model light sources as volumes rather than points.
- Use a sphere to model a light source, rather than a point.
- This is often quite realistic because many light sources in real life are spherical, e.g. light bulbs and lanterns.
- When calculating lighting, shoot several rays to the light source, instead of just one ray.
- Calculate lighting for each ray, and take the average.

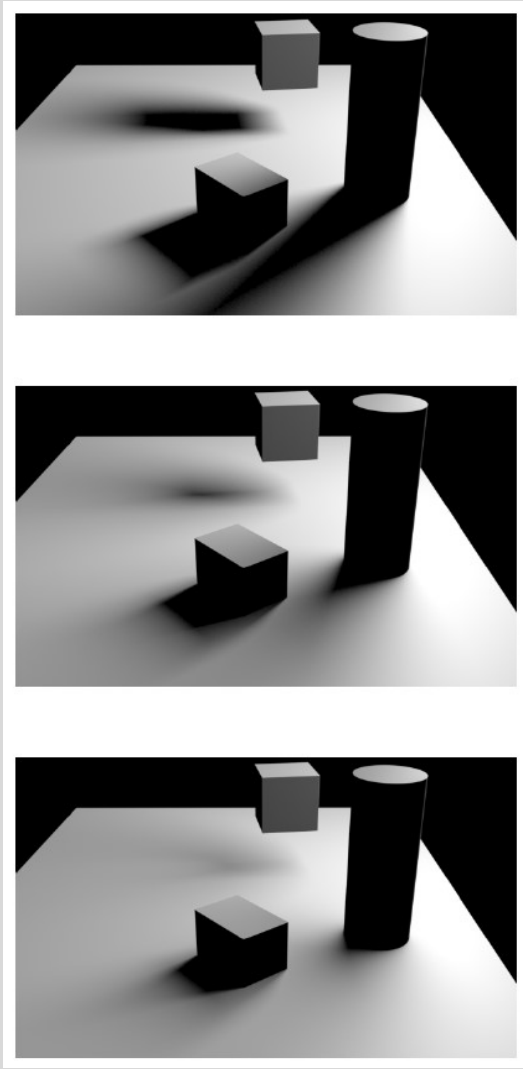


Point light source: The surface is completely lit by the light source.



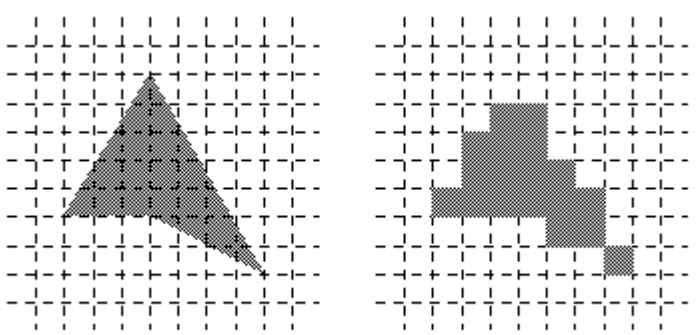
Finite light source: 3/5 of the rays reach the light source. The surface is partially lighted.

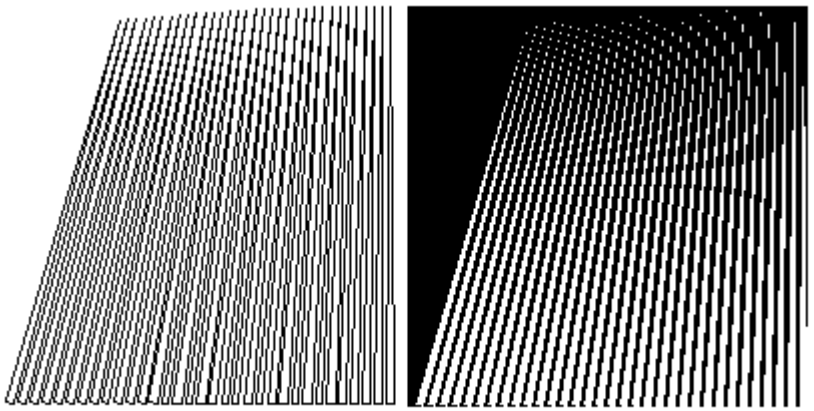
Soft Shadow examples



Aliasing

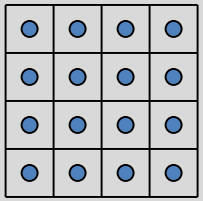
- Ray tracing gives a colour for every possible point in the image
- But a square pixel contains an *infinite* number of points
  - These points may not all have the same colour
  - Sampling: choose the colour of one point (centre of pixel)
  - Regular sampling leads to *aliasing*
    - Jaggies, Moiré patterns (object with regular pattern, ray interference)





Anti-aliasing

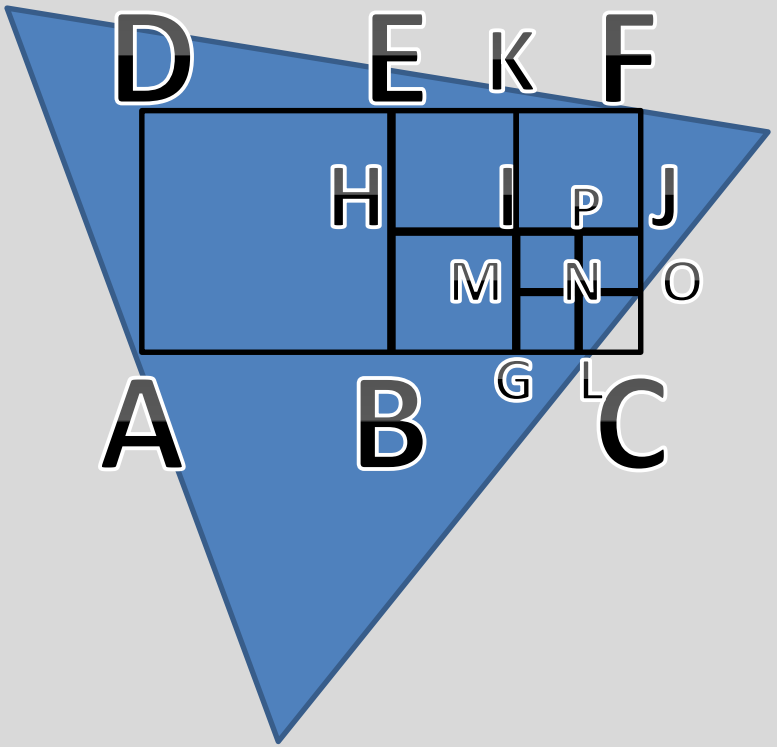
- Supersampling
  - Fire more than one ray for each pixel
  - (e.g., a 4x4 grid of rays)
  - Average the results (perhaps using a filter)



Anti-aliasing: Supersampling

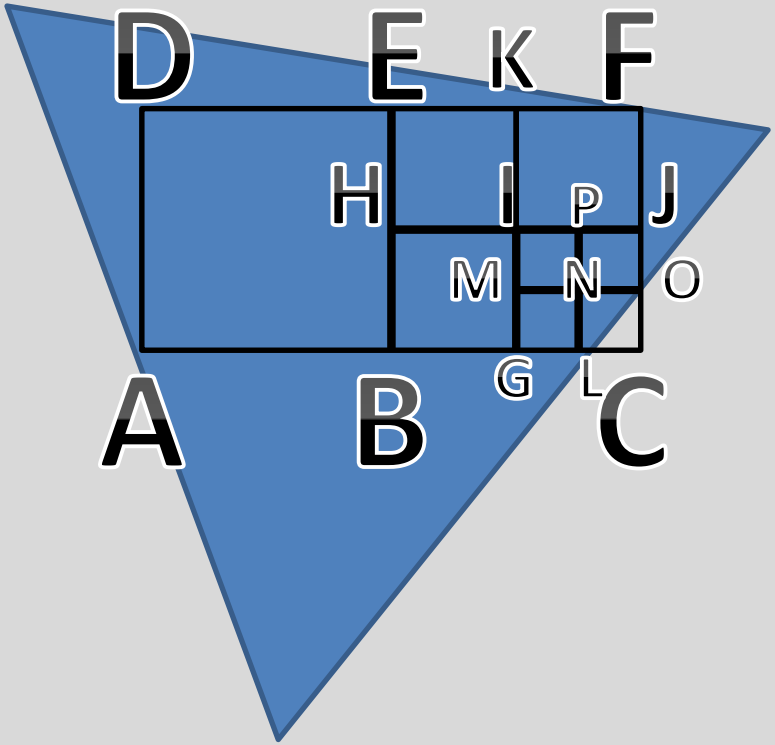
- Can be done *adaptively*
  - divide pixel into 2x2 grid, trace 5 rays (4 at corners, 1n at centre)
  - if the colours are similar then just use their average
  - otherwise recursively subdivide each cell of grid
  - keep going until each 2x2 grid is close to uniform or limit is reached
  - filter the result

Example



- **First pixel colour**
- **(D+E+A+B)/4**
- **What colour is the second pixel (EFBC)??**

Solution



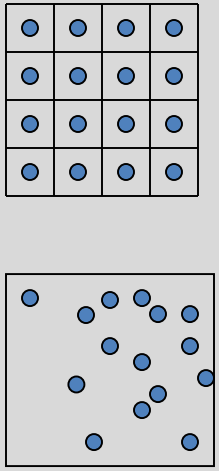
- **¼(E+K+H+I) + ¼(K+F+I+J) + ¼(H+I+B+G) + ¼ (¼(I+P+N+M)+ ¼ (P+J+O+N) + ¼ (M+N+G+L) + ¼ (N+O+L+C))**

Adaptive Supersampling

- **Is adaptive supersampling the answer?**
  - Areas with fairly constant appearance are sparsely sampled (good)
  - Areas with lots of variability are heavily sampled (good)
- But...
  - even with massive supersampling visible aliasing is possible when the sampling grid interacts with regular structures
  - problem is, objects tend to be almost aligned with sampling grid
  - noticeable beating, moire patterns are possible
- So use *stochastic sampling*
  - instead of a regular grid, subsample randomly
  - then adaptively subsample

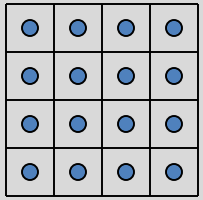
Anti-Aliasing

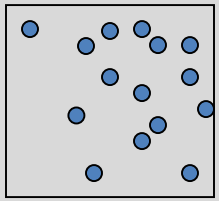
- Splitting a pixel into a regular sub-pixel grid may still have aliasing if the underlying pattern matches the sub-pixel division.
- Alternative: We can randomly sample from the pixel
  - Problem: Randomly picked positions may be very uneven

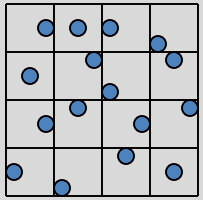


Anti-Aliasing

- Splitting a pixel into a regular sub-pixel grid may still have aliasing if the underlying pattern matches the sub-pixel division.
- Alternative: We can randomly sample from the pixel
  - Problem: Randomly picked positions may be very uneven
- Alternative: Pick from random positions within regular sub-pixel grid







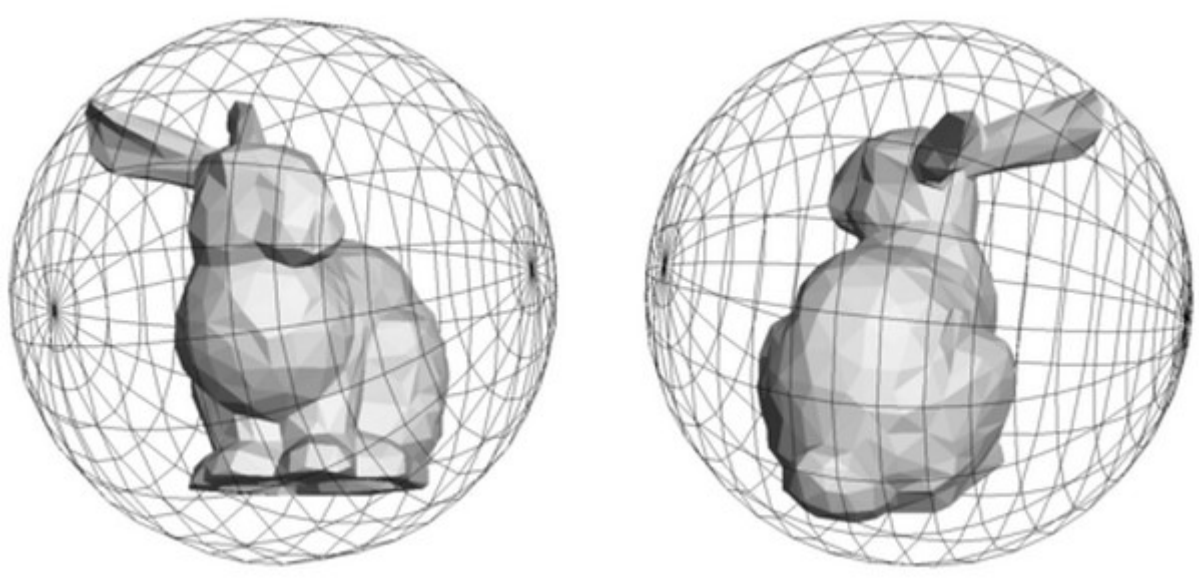
Speeding Up Ray-Tracing

- Now, we need to test each object in the scene for intersection with each ray
- This is too time-consuming.
- Let's try to keep the objects in some **data structure** so that we can quickly determine if a set of objects will definitely not intersect a ray.

Efficiency Considerations

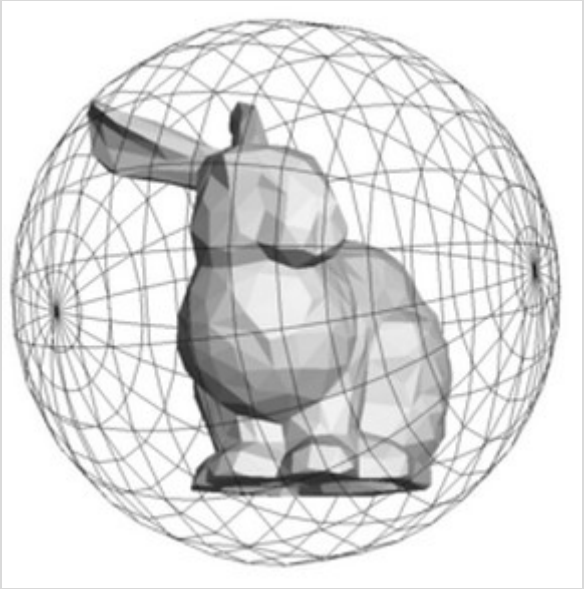
- 1024 * 1024 image, with 100 objects in the scene
- How many intersection tests?
  - Every eye-ray through each pixel tests against 100 objects
  - 1024*1024*100 = >100M intersection calculations!
  - Whitted found that 75 – 95% of his system's time was spent in the intersection routine
- Need to speed up individual intersection calculations or avoid them entirely

Bounding Volumes



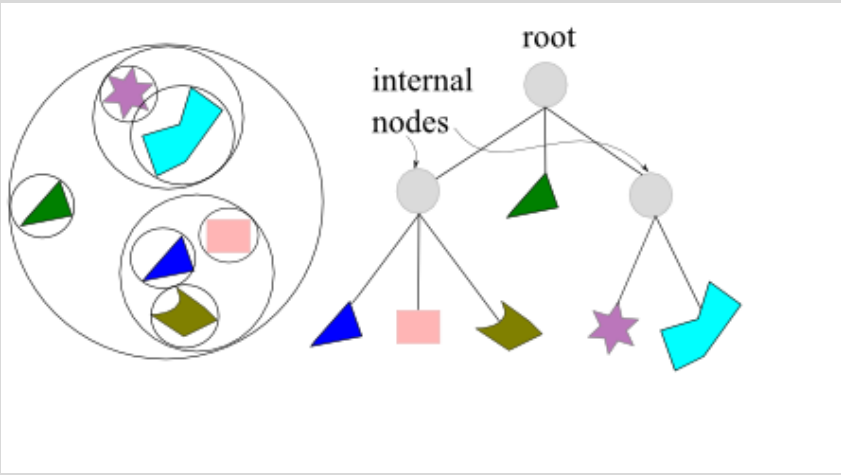
Bounding Volumes

- Decrease the amount of time spent on intersection calculations
- An object that is relatively expensive to test for intersections may be enclosed in a bounding volume whose intersection test is less expensive (sphere, ellipsoid)
- The object can be excluded from testing if the ray fails to hit the bounding volume



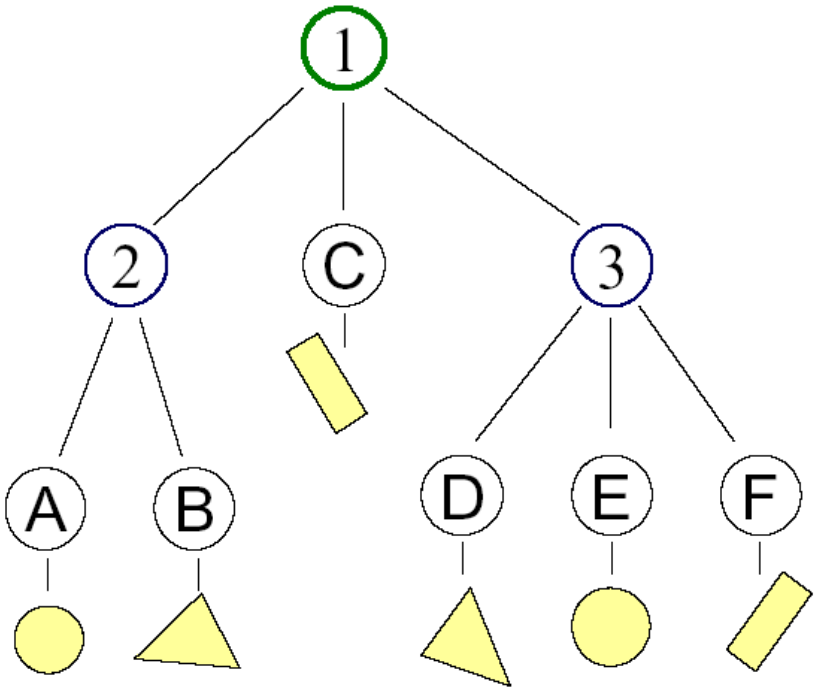
Hierarchy

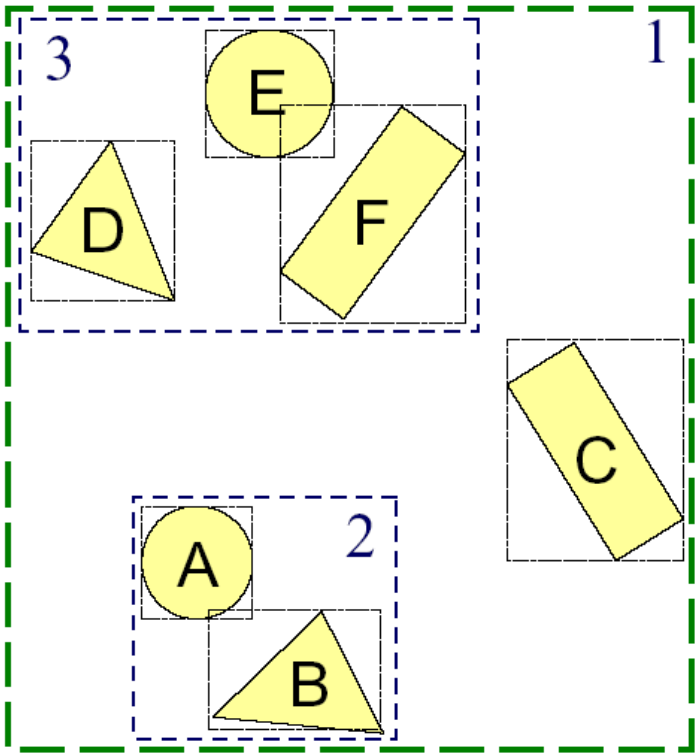
- Bounding volumes by themselves do not determine the order or frequency of intersection tests
  - May be organised in nested hierarchies with objects at the leaves and internal nodes that bound their children
  - **Key concept**: a child volume is guaranteed not to intersect with a ray if its parent does not
  - Thus, if intersection tests begin with the root, many branches of the hierarchy may be trivially rejected



Bounding Volume Hierarchies 1

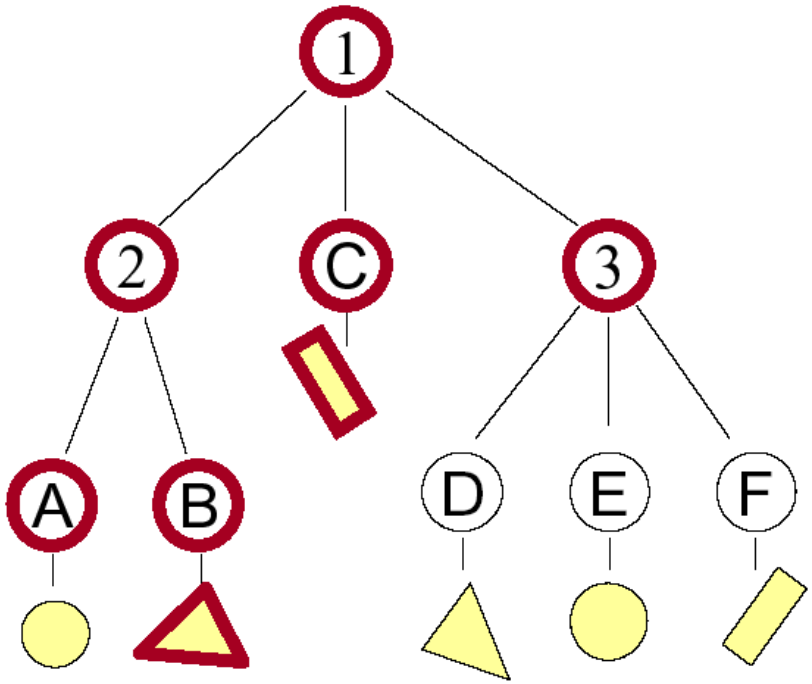
- Build hierarchy of bounding volumes
  - Bounding volume of interior node contains all children

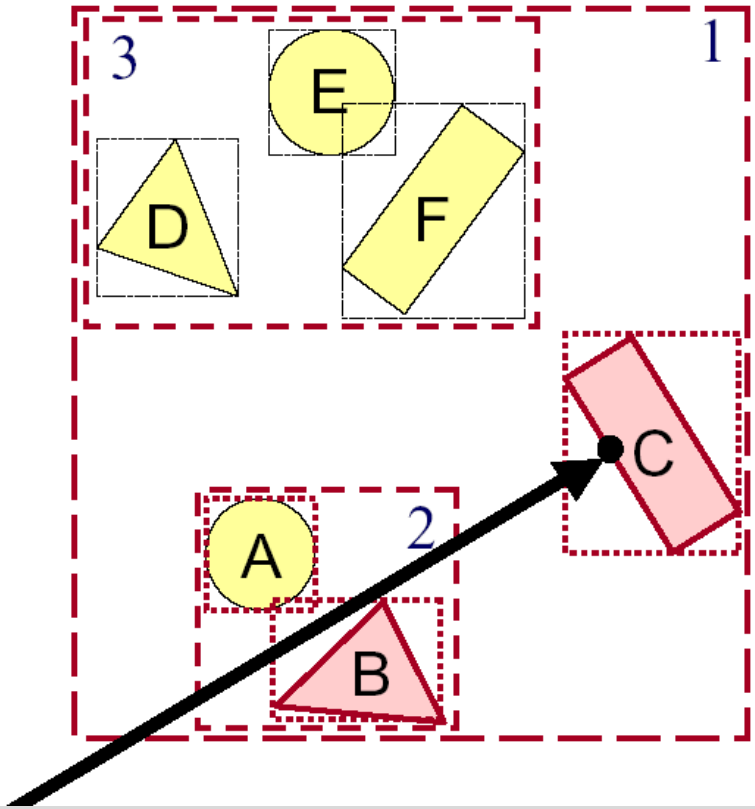




Bounding Volume Hierarchies 2

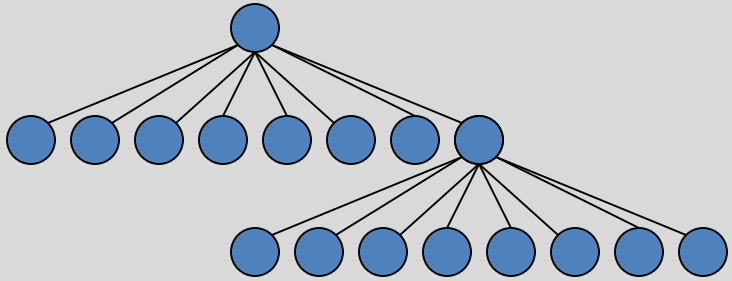
- Use hierarchy to accelerate ray intersections
  - Intersect node contents only if hit bounding volume

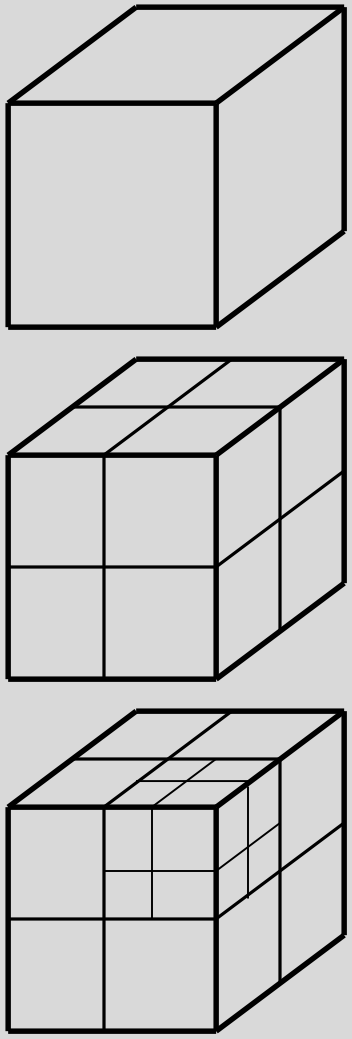




Octree

- Represent the 3D volume with a cube.
- Subdivide cube by half in each direction: We get 8 sub-cubes.
- Repeatedly subdivide each sub-cube the same way.





Octree

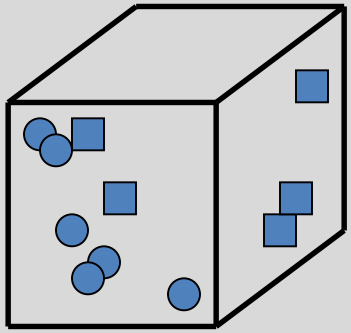
- A data structure that describes how the objects in a scene are distributed throughout the 3d space occupied by the scene
- Objects that are **close to each other** in space are represented by nodes that are close to eachother in an octree
- When tracing a ray, instead of doing intersection calculations between the ray and every object in the scene, we can now trace the ray from sub-region to sub-region in the subdivision of occupied space.
- For each sub-region that the ray passes through, there will only be a small number of objects with which it could intersect
  - Lower number of intersection tests

Octree Construction Example

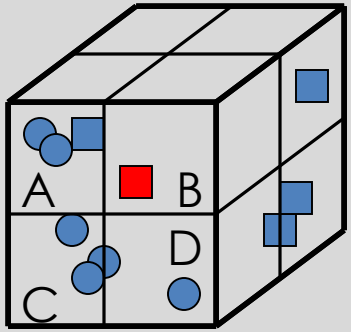
Scene

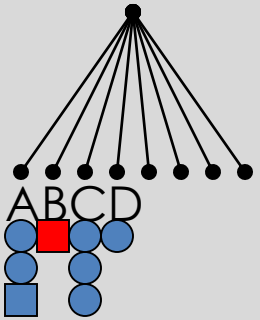
Octree

1. Put a bounding cube around all objects.
This is the root of the octree.

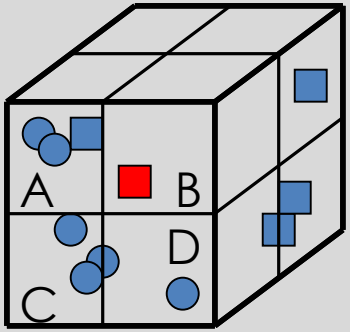


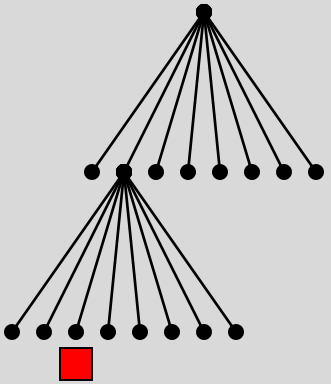
2. Subdivide each node into 8 equal cubes. These will be the children of the node. Put each object in the respective node.
*The nodes A, B, C and D are shown.





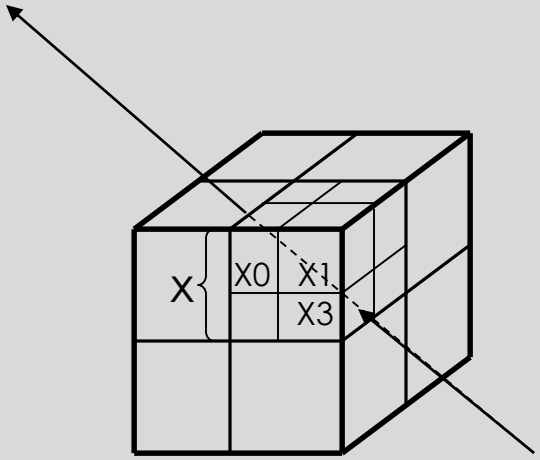
3. For each node, subdivide again recursively, until MAX_LEVEL has been reached or there is no object left in the node.
*The node B is shown.





Octree Traversal Example

- Suppose now, we have a ray, and we want to know which objects it intersects.
- We don't have to test all the objects in the scene.
- We only need to test the objects in the nodes intersected by the ray.



Example:

1. Test intersection of ray with root node.

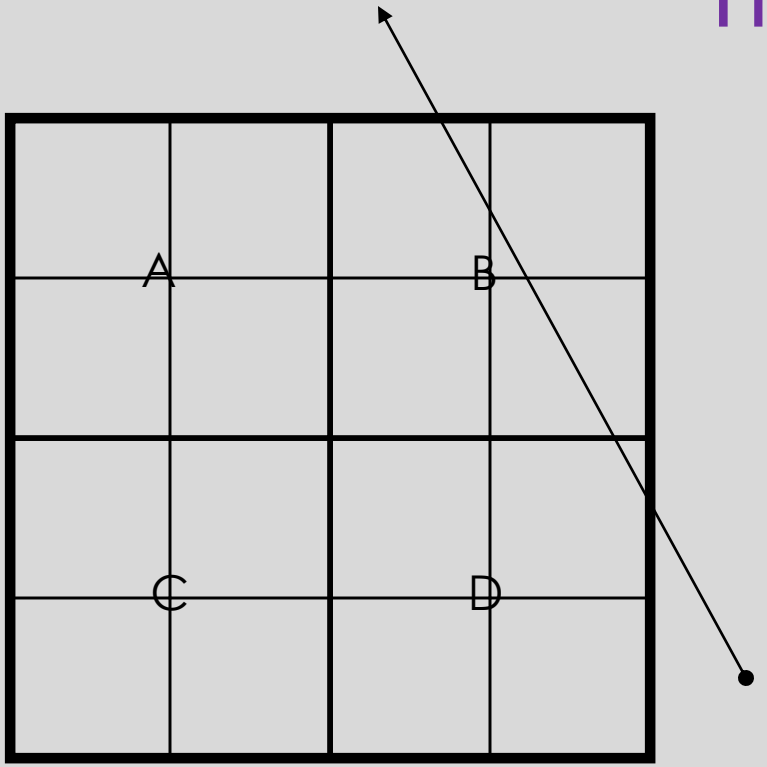
2. Intersect root, therefore check all children of root node.

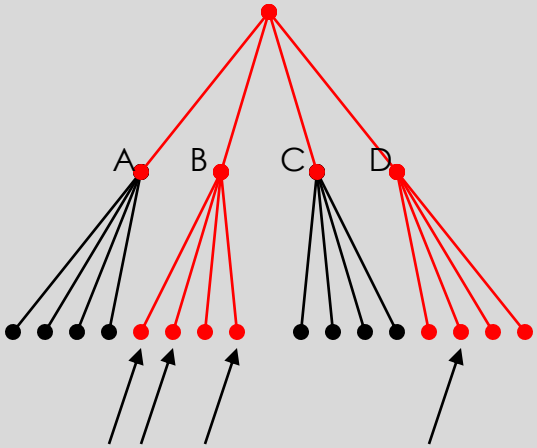
3. Find that ray only intersects node X. Therefore, only traverse the octree down node X.

4. Find that ray only intersects node X0, X1 and X3, so traverse only those nodes.

5. Nodes X0, X1 and X3 are leaf nodes, so now just test ray-object intersections for all objects contained in these leaf nodes.

Octree to speed up Ray-Tracing





This example shows a quad-tree, a 2D version of an octree, for simplicity.
The camera and ray are shown.
Only the red nodes need to be tested for intersections.
All the objects in the marked leaf nodes need to be tested for intersections.

Problems

- Tradeoff
  - Limit the depth of any branch
    - Increases the number of intersection candidates for the small volume
  - Subdivide until a voxel contains a single object
    - The maximum length of any branch may become long
- Disadvantage
  - Possible to have large volumes that contain only a single small object
  - Many rays may enter this region that do not intersect the object
  - Similar to having bounding volume that contains a large void area because shape is less than optimum for the shape of the object it encloses

Additional Reading

- Advanced Animation and Rendering Techniques: theory and practice, Alan Watt and Mark Watt
- Fully worked examples of intersection tests: http://www.vis.uky.edu/~ryang/teaching/cs535-2012spr/Lectures/13-RayTracing-II.pdf