Ray Tracing Extensions

Reading

Foley *et al.*, 15.10 and 16.12

Optional:

- Glassner, An introduction to Ray Tracing, Academic Press, Chapter 1.
- T. Whitted. "An improved illumination model for shaded display". *Communications of the ACM*} 23(6), 343-349, 1980.

Goodies

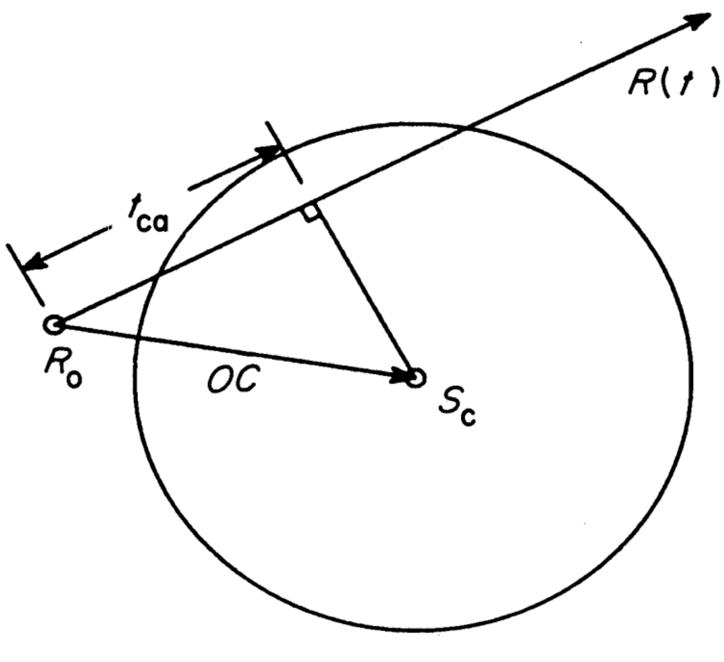
- There are some advanced ray tracing feature that self-respecting ray tracers shouldn't be caught without:
  - Acceleration techniques
  - Antialiasing
  - Distribution ray tracing
  - CSG

Acceleration Techniques

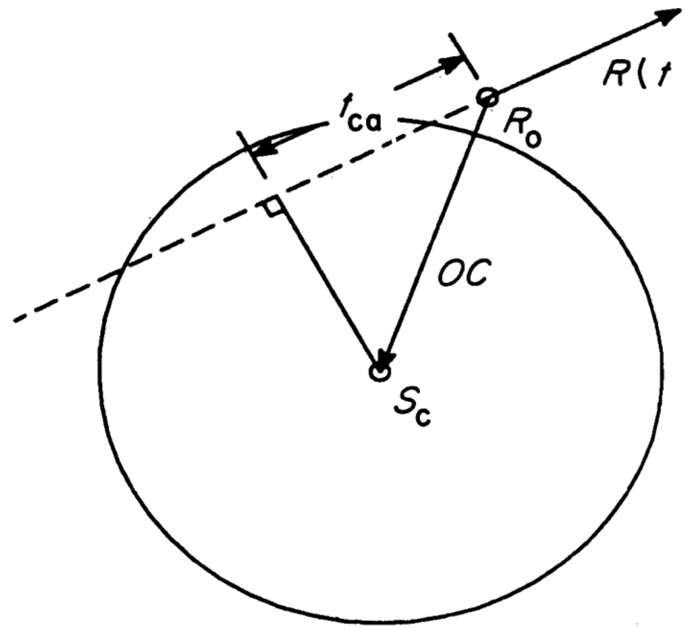
- Problem: ray-object intersection is very expensive
  - make intersection tests faster
  - do fewer tests

Fast Failure

- We can greatly speed up ray-object intersection by identifying cheap tests that guarantee failure
- Example: if origin of ray is outside sphere and ray points away from sphere, fail immediately.

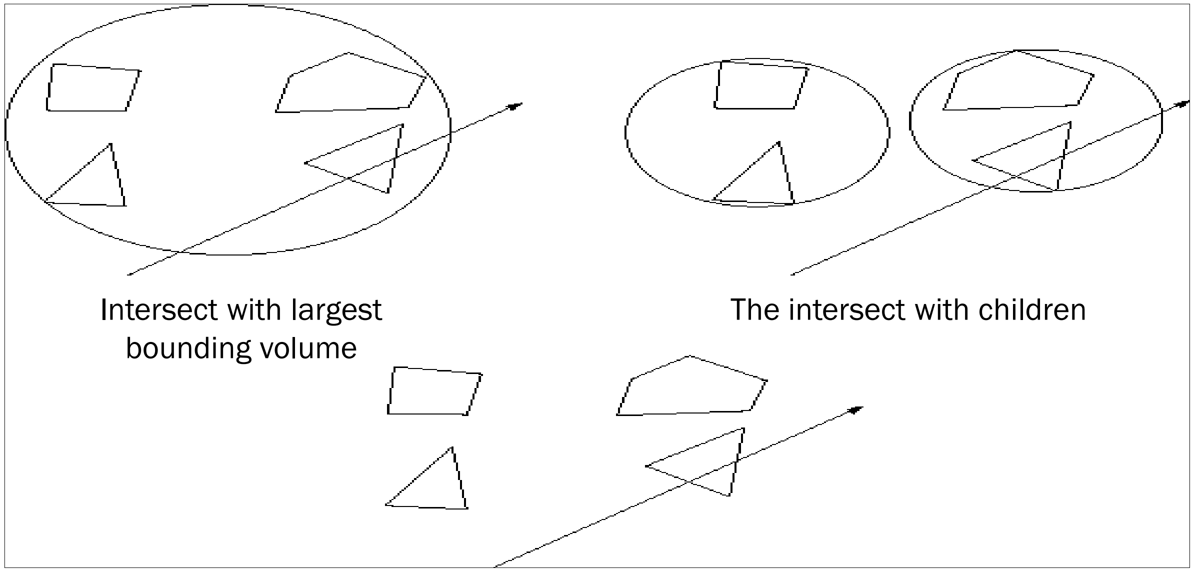


$t_{ca} > 0$, so the ray points toward the sphere



$t_{ca} < 0$, so the ray points away from the sphere

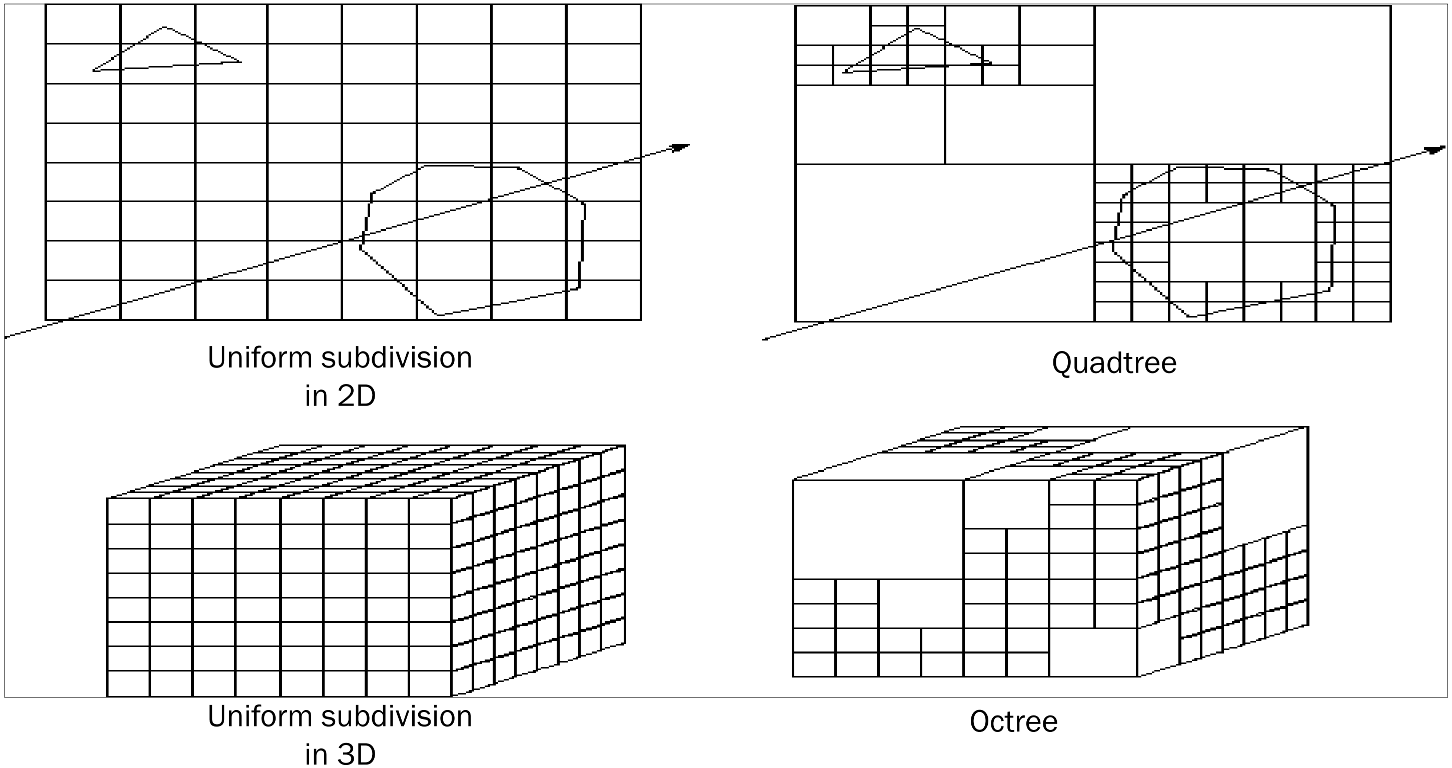
Hierarchical Bounding Volumes



Eventually, intersect with primitives

- Arrange scene into a tree
  - Interior nodes contain primitives with very simple intersection tests (e.g., spheres). Each node's volume contains all objects in subtree
  - Leaf nodes contain original geometry
- Like BSP trees, the potential benefits are big but the hierarchy is hard to build

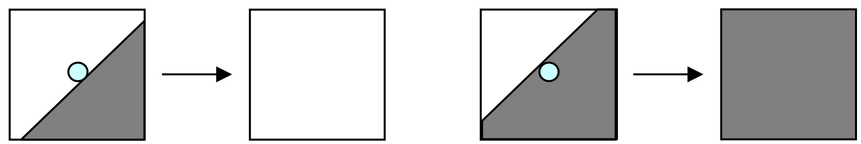
Spatial Subdivision



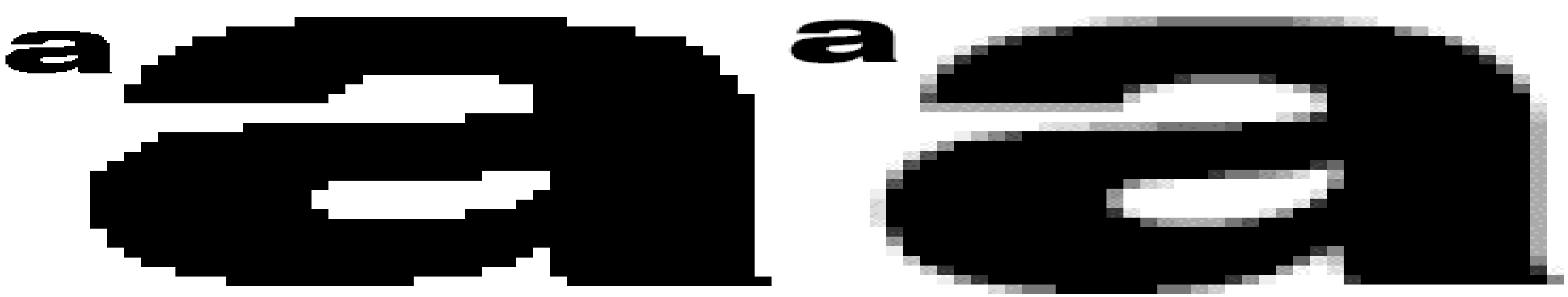
- Divide up space and record what objects are in each cell
- Trace ray through **voxel** array

Antialiasing

- So far, we have traced one ray through each pixel in the final image. Is this an adequate description of the contents of the pixel?

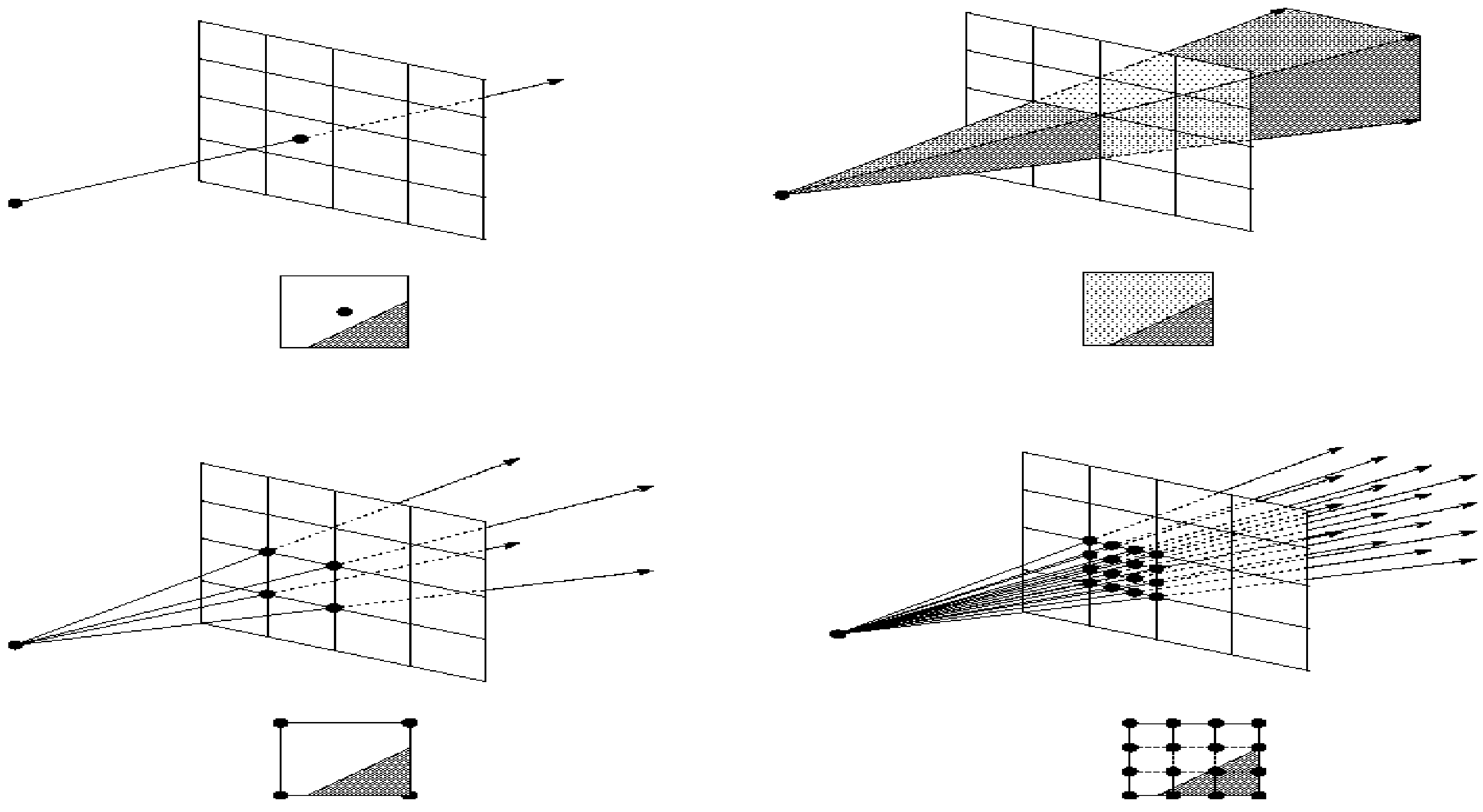


- This quantization through inadequate sampling is a form of **aliasing**. Aliasing is visible as "jaggies" in the ray-traced image.
- We really need to colour the pixel based on the *average*



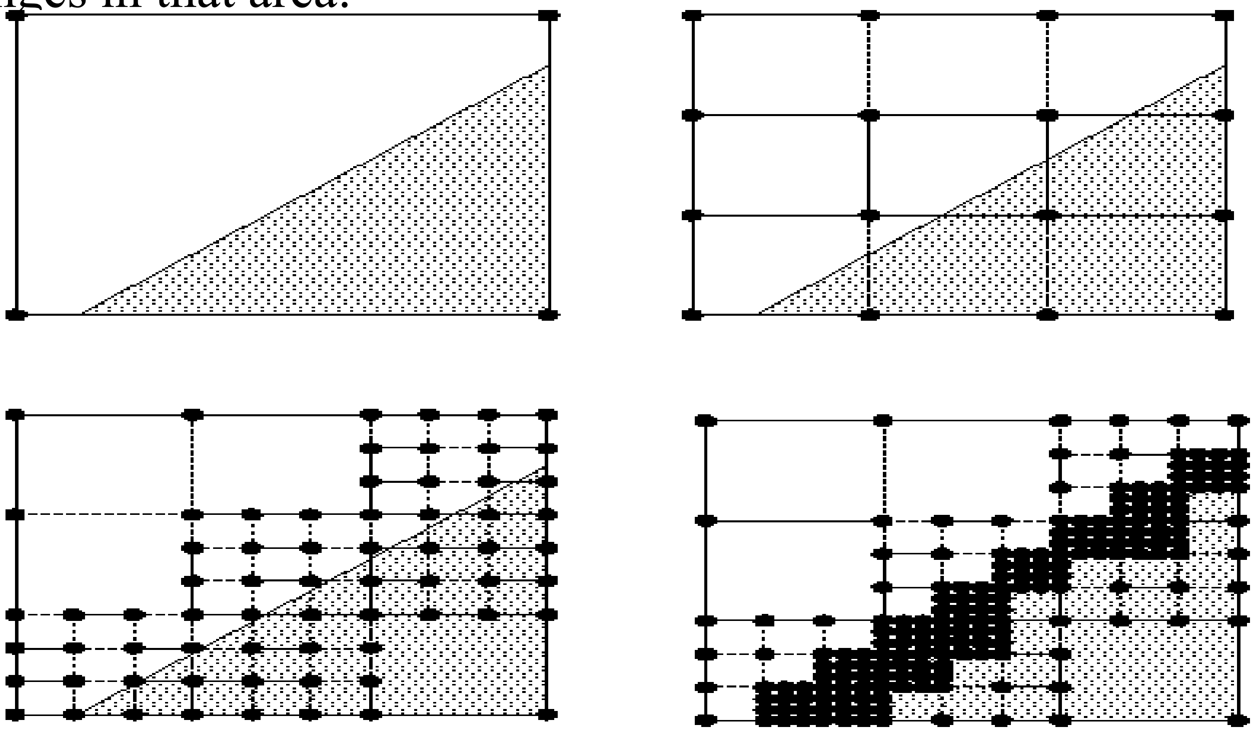
Supersampling

- We can approximate the average colour of a pixel's area by firing multiple rays and averaging the result.



Adaptive Sampling

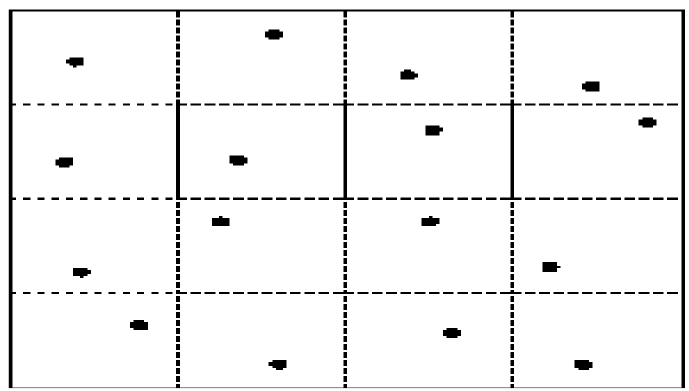
- Uniform supersampling can be wasteful if large parts of the pixel don't change much.
- So we can subdivide regions of the pixel's area only when the image changes in that area:



- How do we decide when to subdivide?

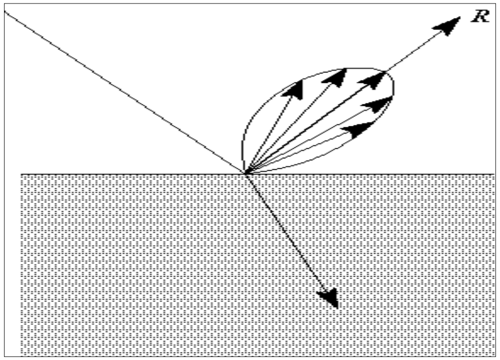
Distribution Ray Tracing

- Usually known as "distributed ray tracing", but it has nothing to do with distributed computing
- General idea: instead of firing one ray, fire multiple rays in a jittered grid

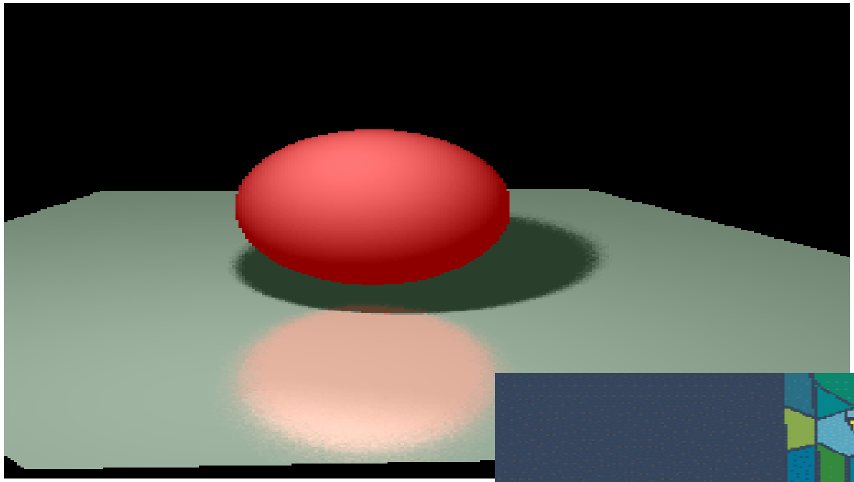


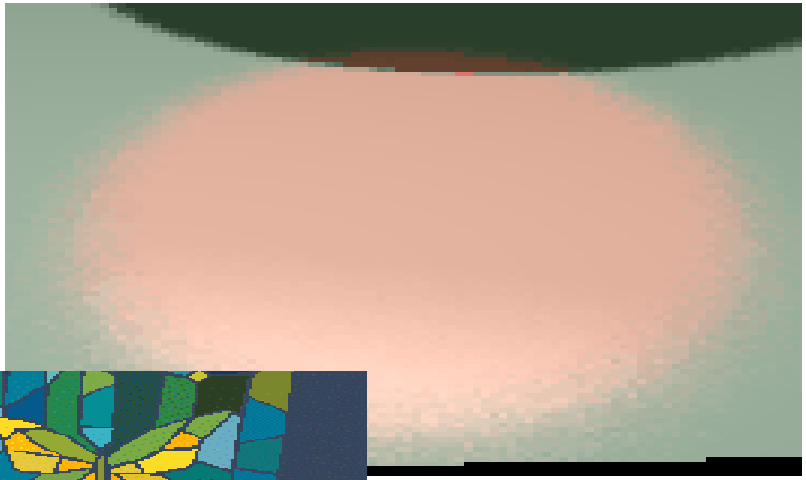
- Distributing over different dimensions gives different effects
- Example: what if we distribute rays over pixel area?

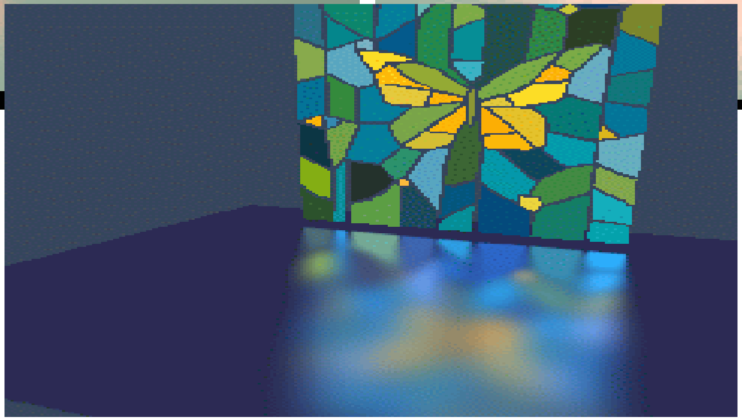
Distributing Reflections



- Distributing rays over reflection direction gives:







Disrtibuted ray tracing pseudocode

1. Partition pixel into 16 regions assigning them id 1-16
2. Partition the reflection direction into 16 angular regions and assign an id (1-16) to each
3. Select sub pixel m=1
4. Cast a ray through m, jittered within its region
5. After finding an intersection, reflect into sub-direction m, jittered within that region
6. Add result to current pixel total
7. Increment m and if m<= 16, go to step 4
8. Divide by 16, store result and move on to next pixel.

DRT pseudocode

TraceImage() looks basically the same, except now each pixel records the average color of jittered sub-pixel rays.

function *traceImage* (scene):
for each pixel (i, j) in image **do**
    I(i, j) ← 0
    for each sub-pixel id in (i,j) **do**
        s ← *pixelToWorld*(jitter(i, j, id))
        p ← **COP**
        d ←(**s** - **p**).normalize()
        I(i, j) ← I(i, j) + *traceRay*(scene, **p**, **d,** id)
    end for
    I(i, j) ← I(i, j)/numSubPixels
end for
end function

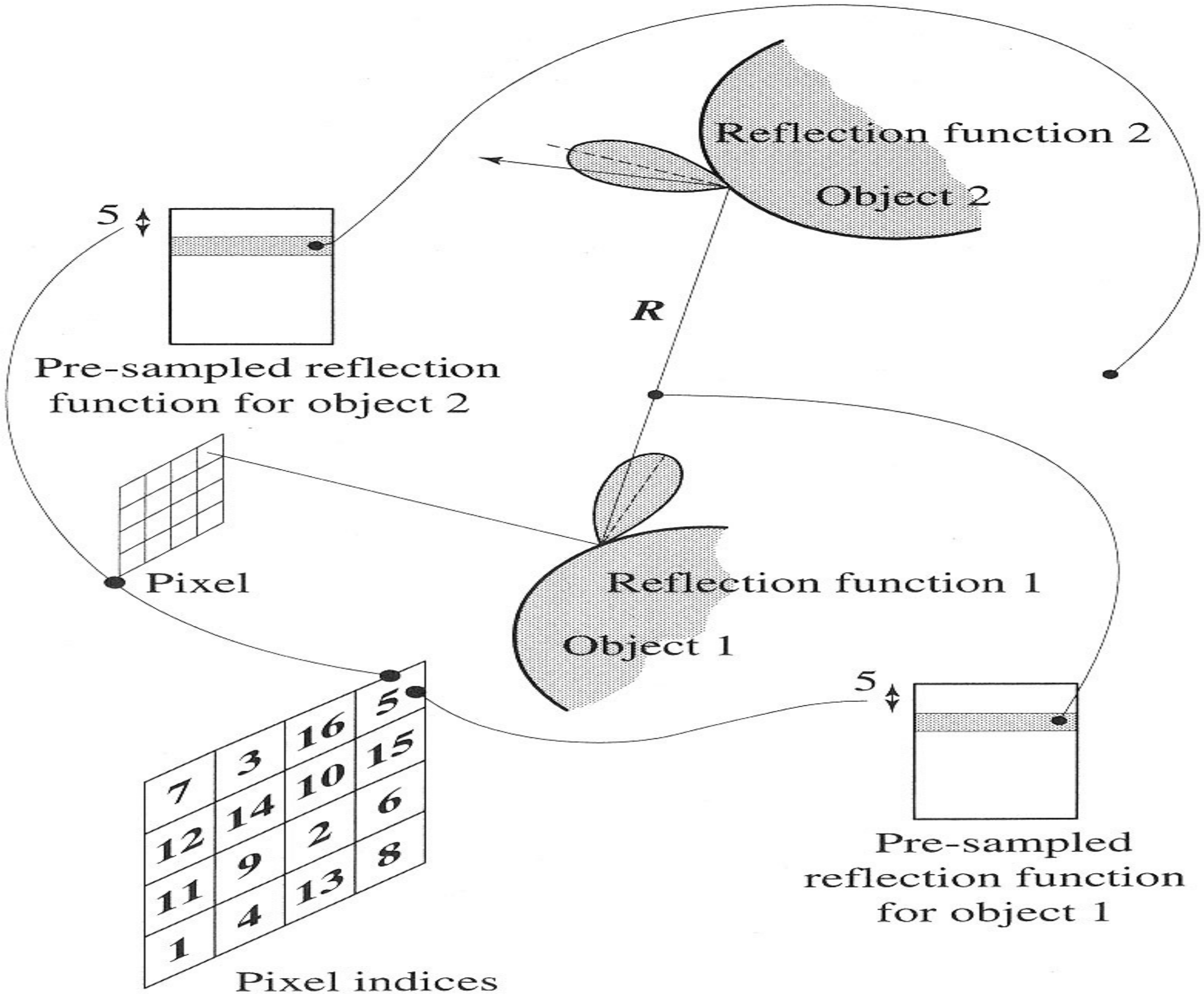
A typical choice is numSubPixels = 4*4.

DRT pseudocode (cont'd)

Now consider *traceRay*(), modified to handle (only) opaque glossy surfaces:

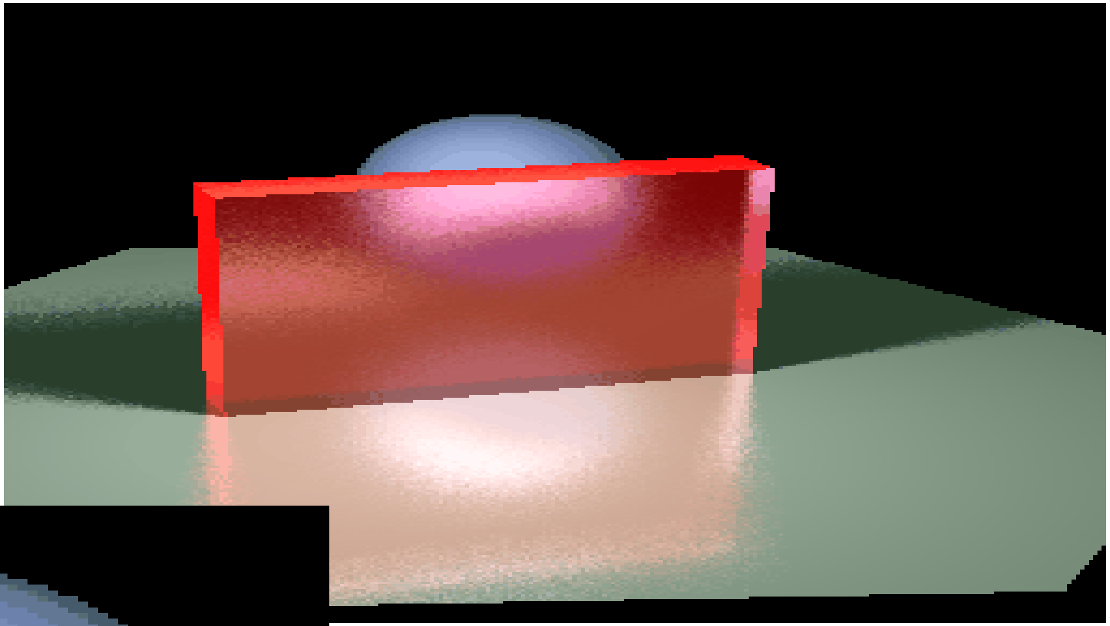
function *traceRay*(scene, **p**, **d, id**):
(**q**, **N**, material) $\leftarrow$ *intersect* (scene, **p**, **d**)
I $\leftarrow$ *shade*(…)
R $\leftarrow$ *jitteredReflectDirection*(**N**, **-d**, **id**)
I $\leftarrow$ I + material.k_r * *traceRay*(scene, **q**, **R, id**)
return I
end function

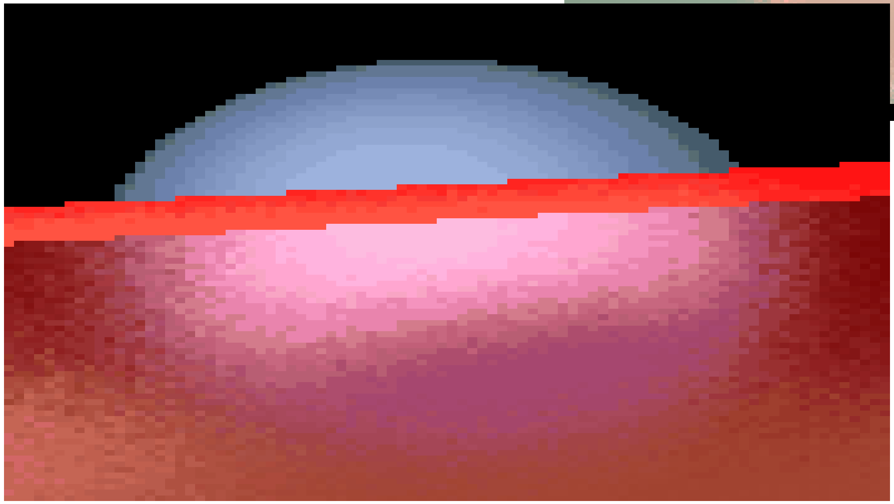
Pre-sampling glossy reflections



Distributing Refractions

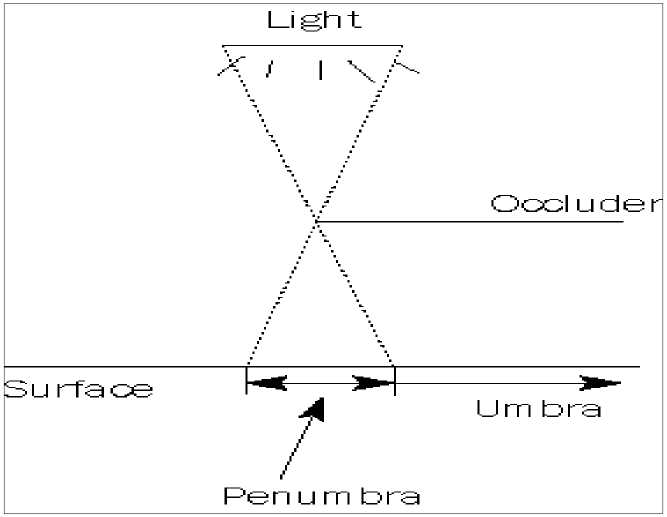
- Distributing rays over transmission direction gives:

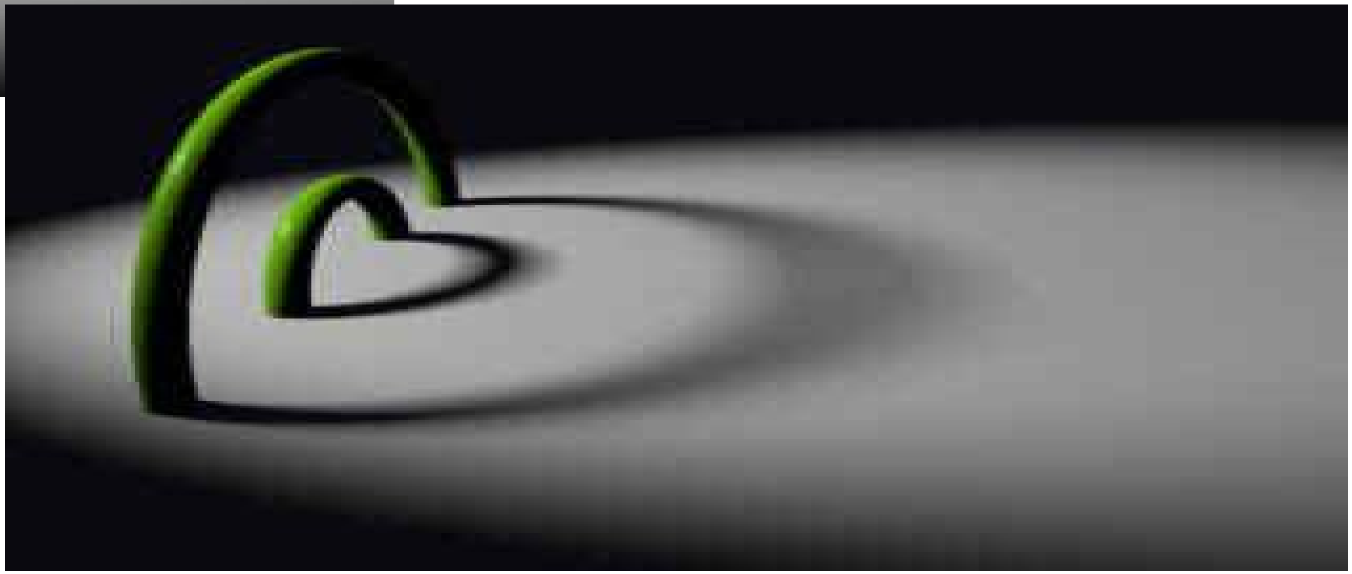




Distributing Over Light Area

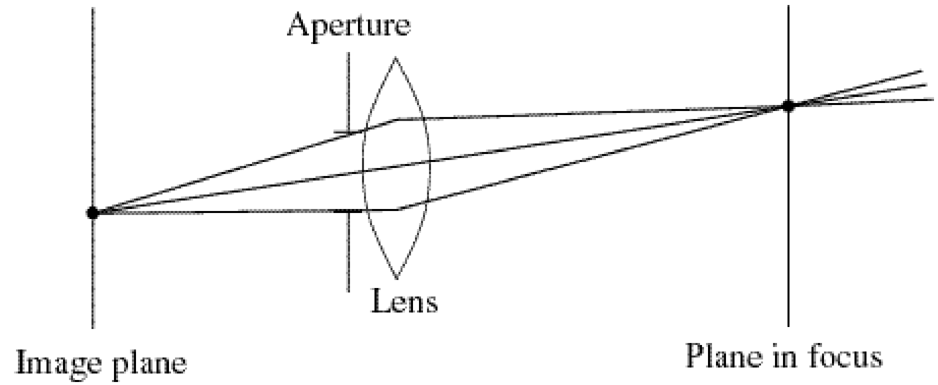
- Distributing over light area gives:





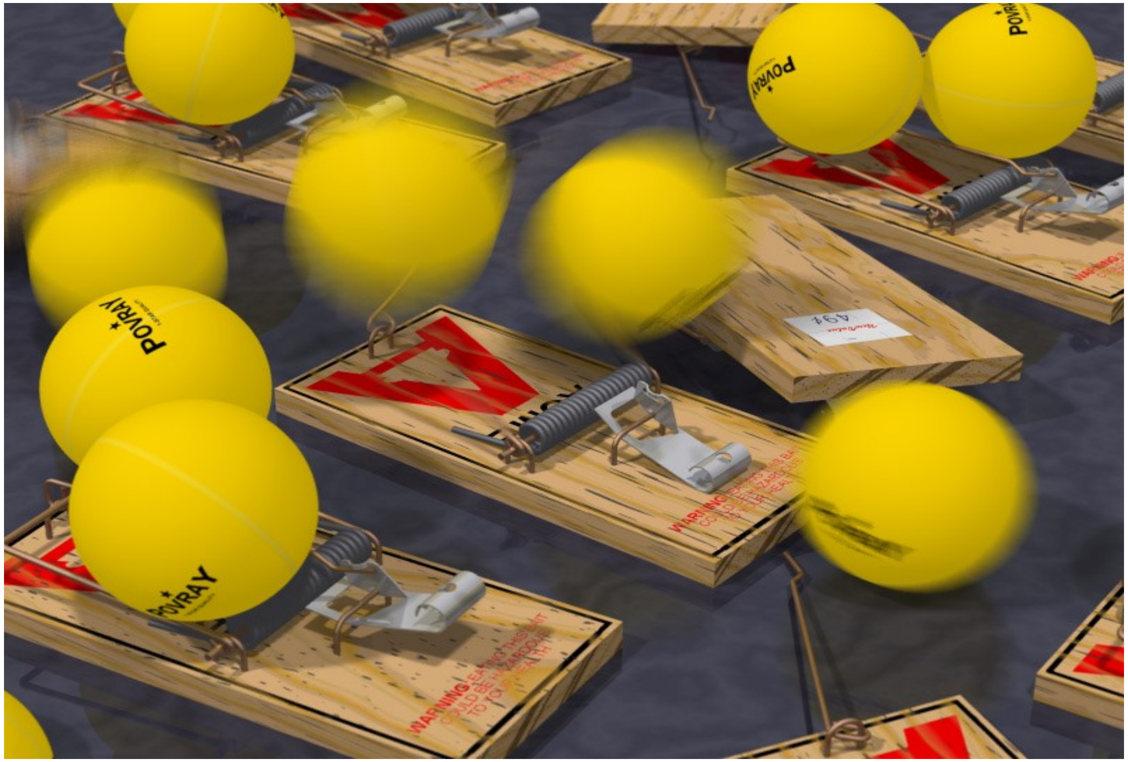
Distributing Over Aperature

Choose a point on a finite aperature and trace through the "in-focus point".



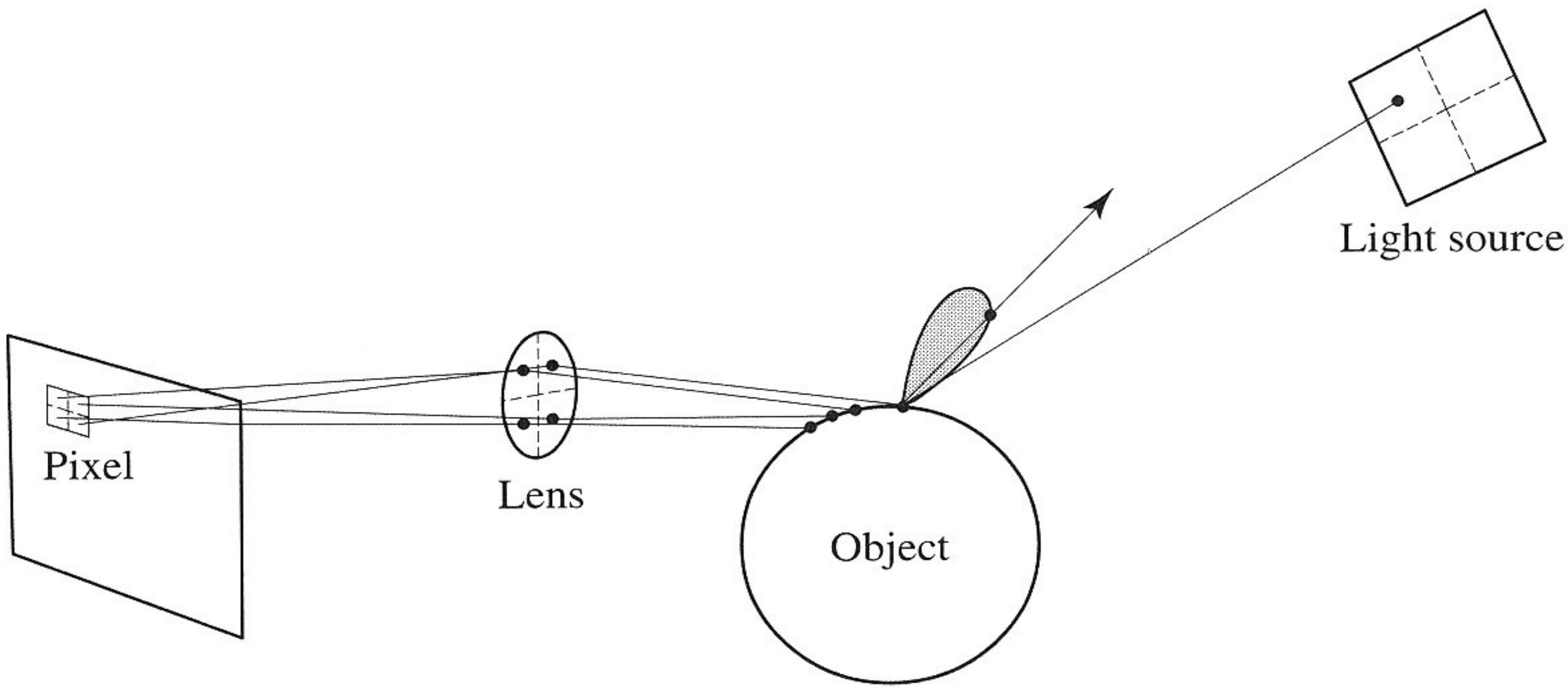
Distributing Over Time

- We can endow models with velocity vectors and distribute rays over *time*. this gives:



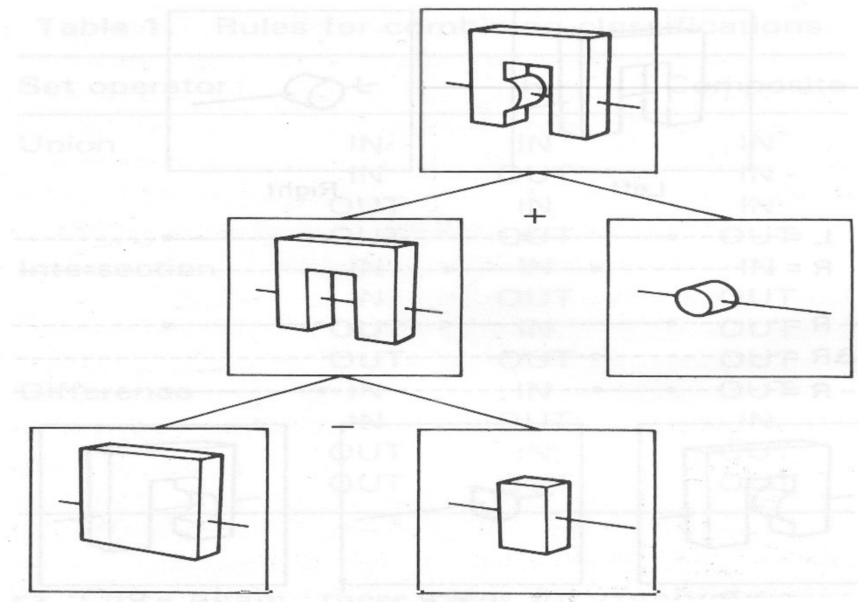
Chaining the ray id's

In general, you can trace rays through a scene and keep track of their id's to handle *all* of these effects:



CSG

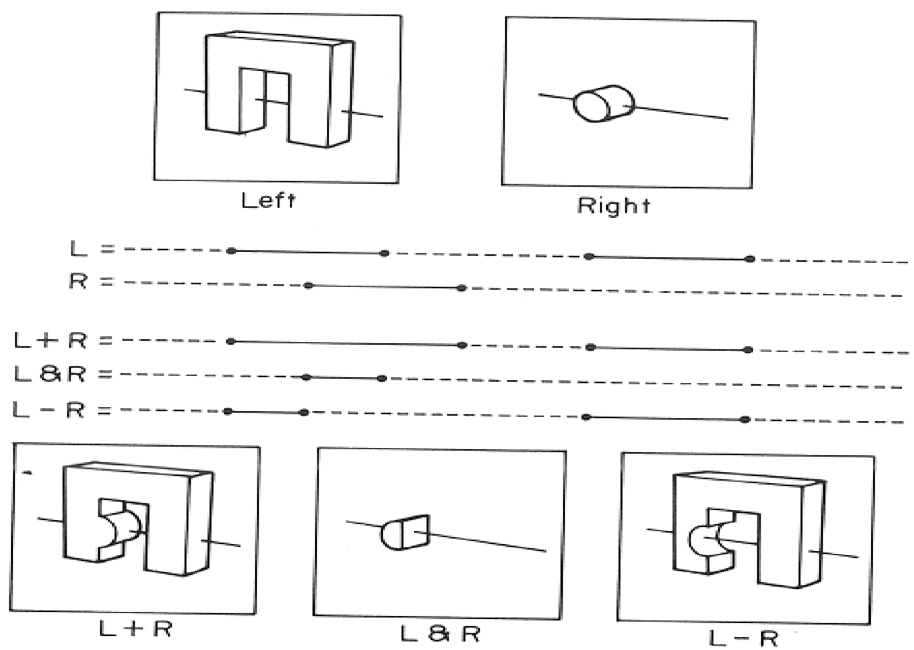
- CSG (constructive solid geometry) is an incredibly powerful way to create complex scenes from simple primitives.



- CSG is a modeling technique; basically, we only need to modify ray-object intersection.

CSG Implementation

- CSG intersections can be analyzed using “Roth diagrams”.
  - Maintain description of *all intersections* of ray with primitive
  - Functions to combine Roth diagrams under CSG operations



- An elegant and extremely slow system