Ray Tracing in Earnest

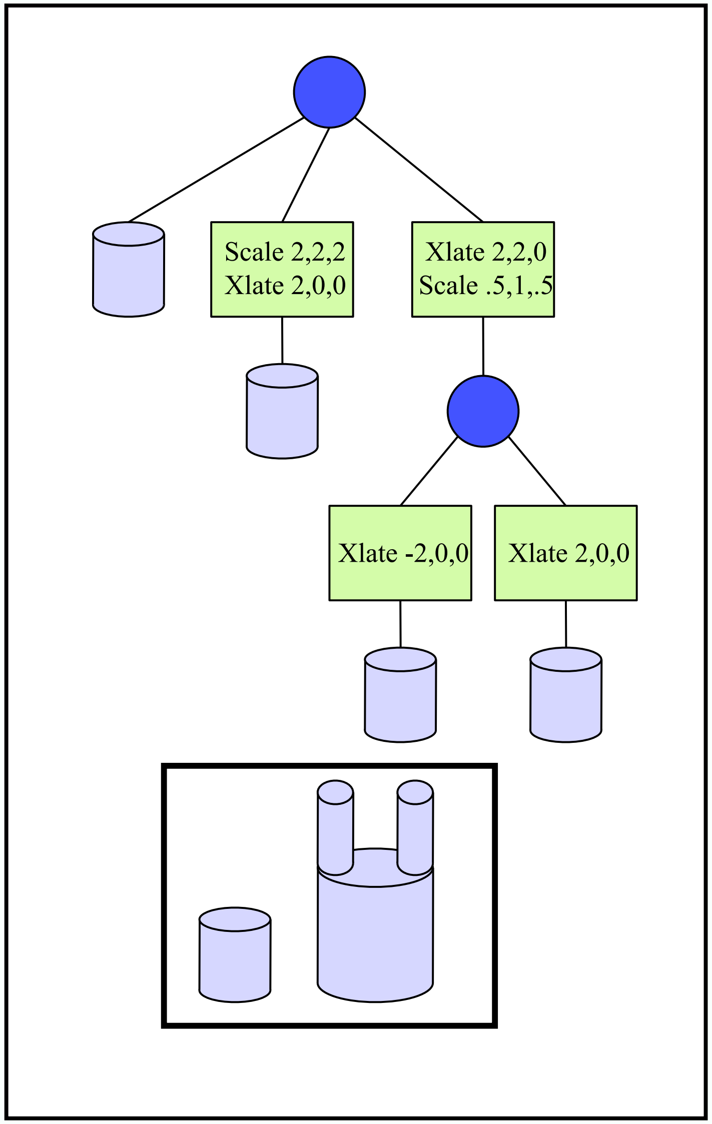
D.A. Forsyth
(using material from John Hart and others)

Issues

- Intersection with complicated models
- Accurate intersection
- Efficient intersection
- Improved rendering
  - anti aliasing (= more rays)
  - motion blur (= more rays)
  - more complex illumination phenomena (= more rays, caching)

Reminder: Scene Graphs

- Hierarchical representation of all objects in scene
  - familiar from raster graphics, etc

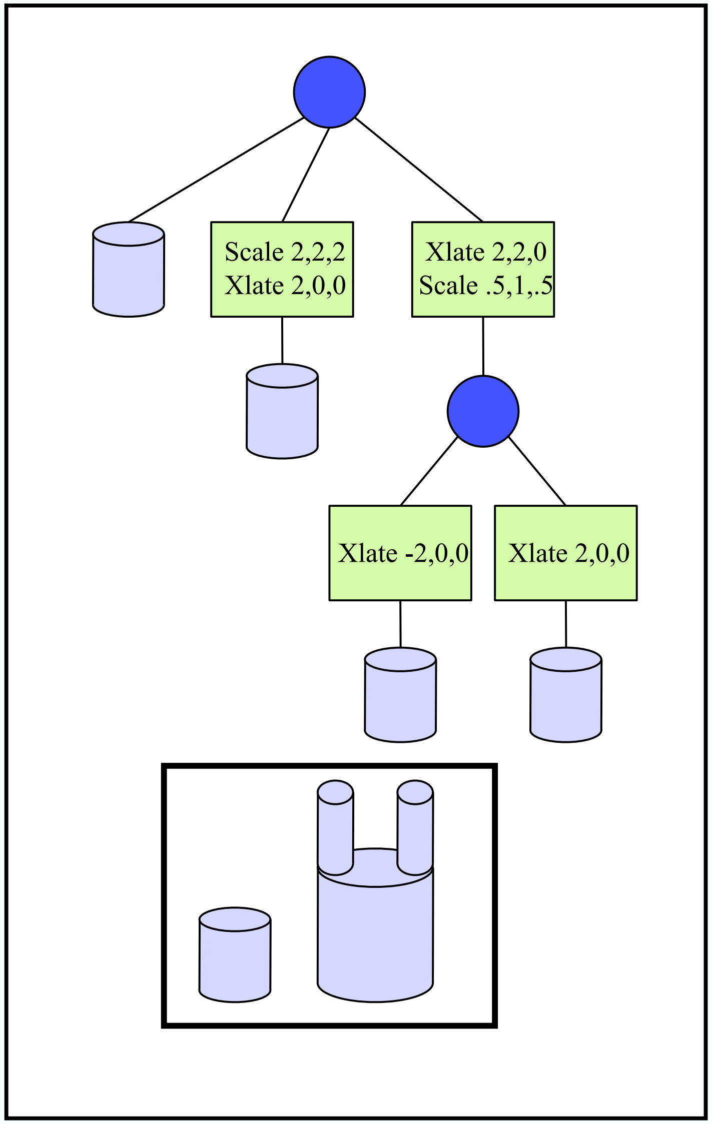


Geometric Primitives

- Primitives we can deal with
  - half-space (because we can do plane intersection)
  - sphere (because we can do sphere intersection)
  - cylinder (easy generalization of sphere)
  - convex polyhedron (easy generalization of half-space)
- Others will come as we learn more intersection techniques

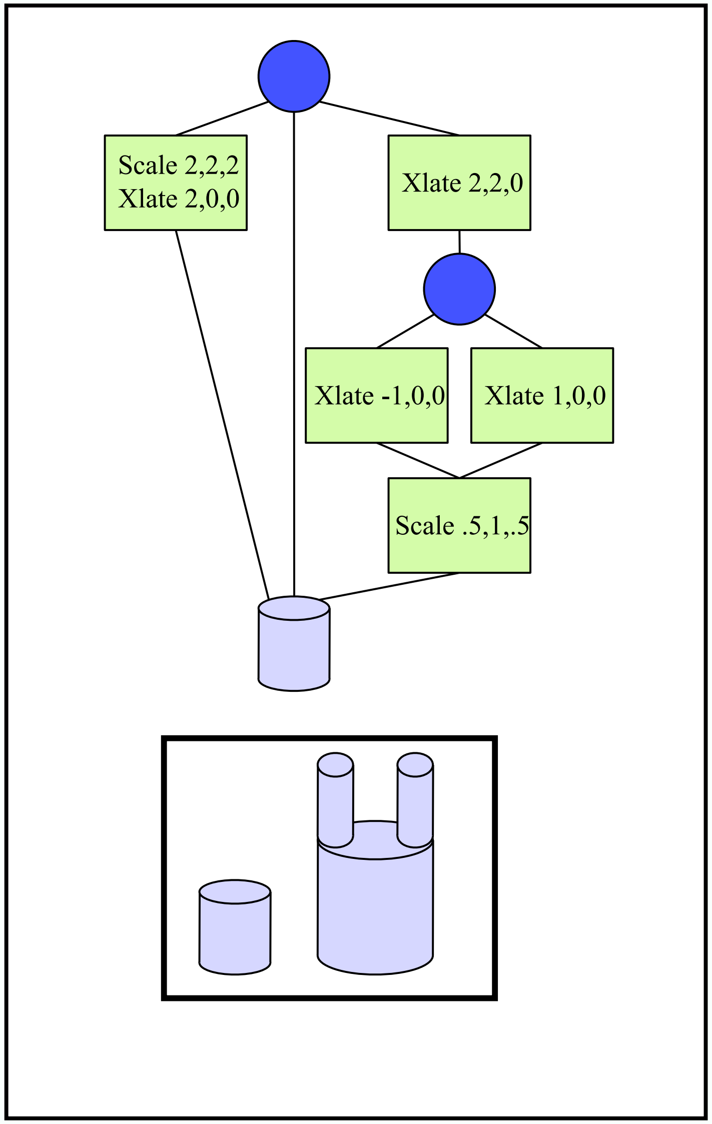
Reminder: Scene Graphs

- Hierarchical representation of all objects in scene
  - familiar from raster graphics, etc
- Transformation nodes now:
  - Intersect children with ray
    - transform ray to child's frame
    - i.e. inverted from usual
  - Returned normal must be in world frame
    - i.e. transpose(inverse(T))
  - Maintain inverse(T)

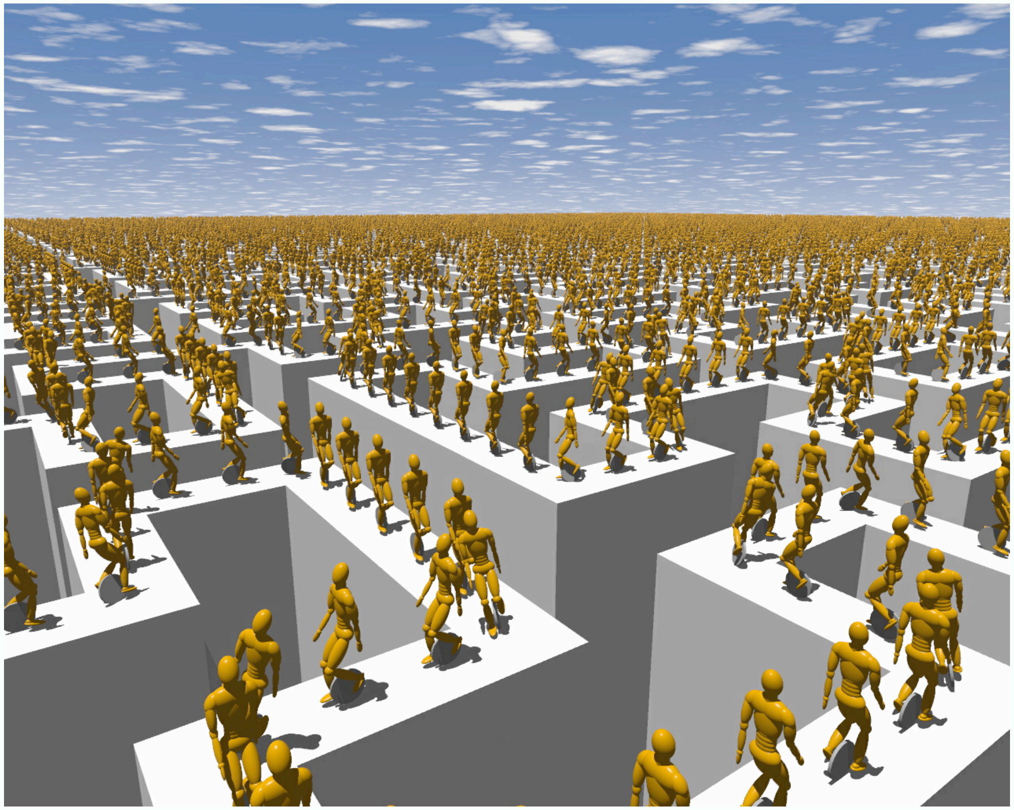


Reminder: Instancing

- Scene graph is a hierarchy
- Not necessarily a tree
- Directed acyclic graph (DAG)
- Nodes may have multiple parents
- Instance
  - Appearance of each node's geometry in scene

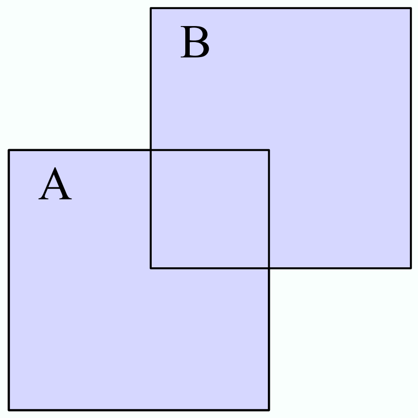


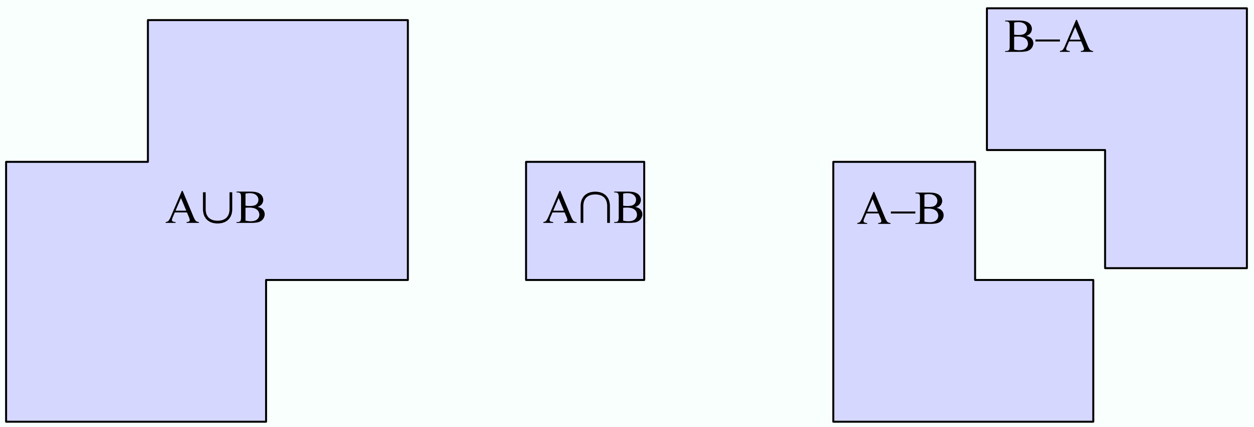
Fun with instancing



CSG

- Constructive Solid Geometry
  - objects are boolean combinations of primitive volumes
  - union, intersection, difference
    - usually regularized



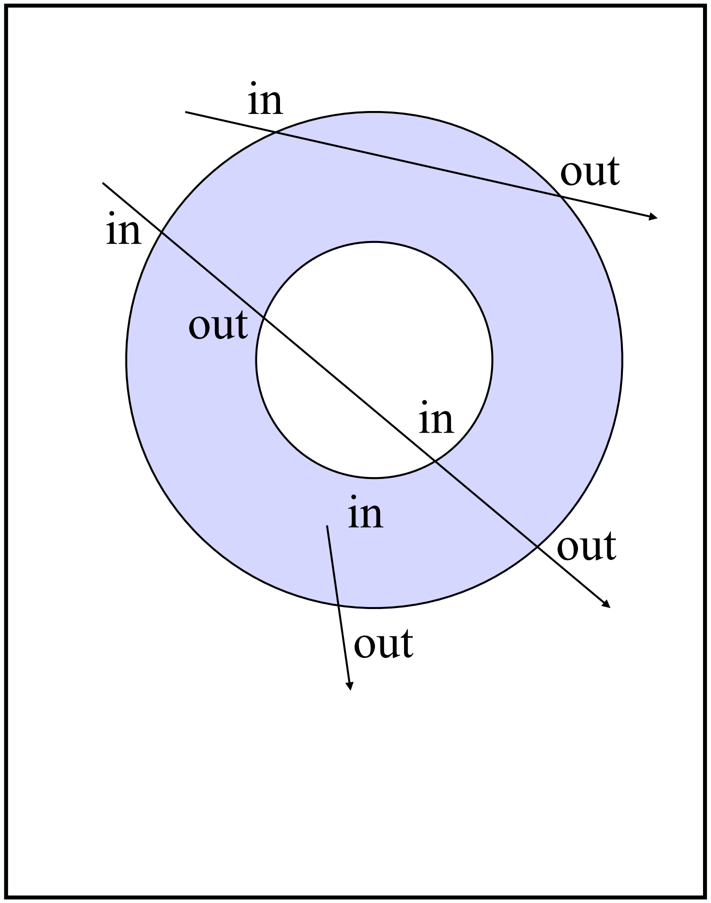


Geometric Primitives

- Primitives we can deal with
  - half-space (because we can do plane intersection)
  - sphere (because we can do sphere intersection)
  - cylinder (easy generalization of sphere)
  - convex polyhedron (easy generalization of half-space)
- Others will come as we learn more intersection techniques

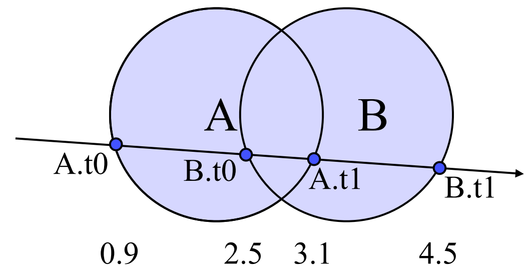
Raytracing CSG

- Represent all intersections in a hit record
  - list
- If we know where focal point is (in/out), parity classifies all others

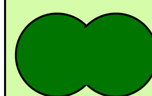
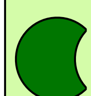


Raytracing CSG

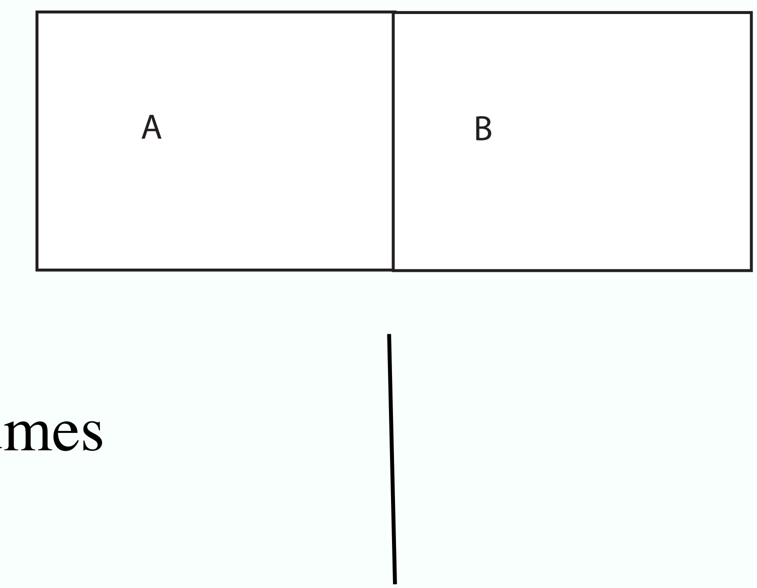
- List of t-values for A, B w/in-out classification
  - A.t_list = {0.9, 3.1} = {0.9in, 3.1out}
  - B.t_list = {2.5, 4.5} = {2.5in, 4.5out}
    - Use dot(r.d,n) to determine in,out
- Merge both lists into a single t-ordered list
  - { 0.9 Ain Bout,
    2.5 Ain Bin,
    3.1 Aout Bin,
    4.5 Aout Bout }
    - Keep track of A and B in/out classification
- Use Roth table to classify t-values



Roth Table

Op	A	B	Res
+ 	in	in	in
	in	out	in
	out	in	in
	out	out	out
* 	in	in	in
	in	out	out
	out	in	out
	out	out	out
– 	in	in	out
	in	out	in
	out	in	out
	out	out	out

Regularizing CSG



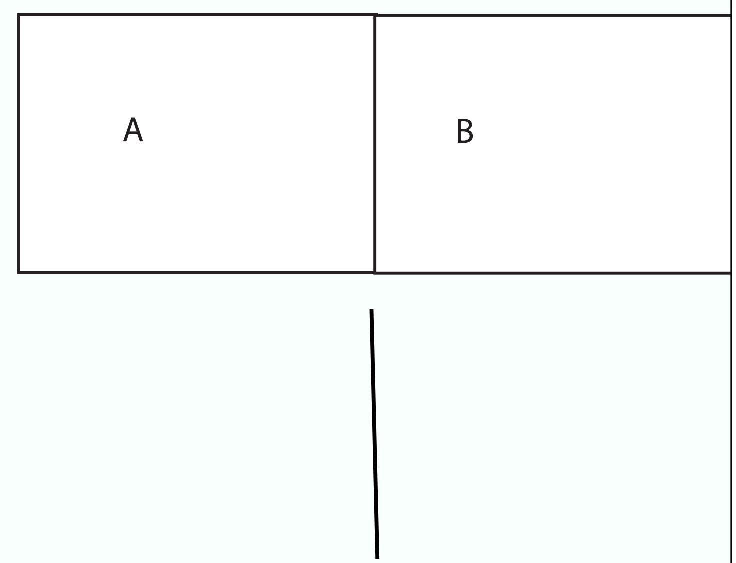
- Primitives can produce non-volumes
  - e.g. A intersect B in pic gives line

$$A \cap^* B = \text{closure}\left(\text{interior}(A) \cap \text{interior}(B)\right)$$

There’s a general phenomenon here

- Points that lie on top of one another
  - but we may not be able to tell
- Our t-values aren’t precisely correct
  - numerical representations aren’t precise
    - could be for polynomial surfaces, but this is not worth the effort
- This means
  - intersections aren’t precisely where we think they are
  - eg shadow ray eczema
- Tolerable solution
  - regard points that are “very close” as the same point
  - cures shadow ray eczema by ignoring surface as blocker
  - can be used to cure previous problem

Regularizing CSG



- Primitives can produce non-volumes
  - e.g. A intersect B in pic gives line
- Regularize
  - eg

$$A \cap^* B = \text{closure}\left(\text{interior}(A) \cap \text{interior}(B)\right)$$

  - equivalently
    - require Bin to occur some small distance before Aout to get hit

This makes the line go away. (ex: how do you regularize union, difference?)

Implicit Surfaces

- Surface is:
- points in vector form:
- ray is:
- intersections are:
  - and are obtained by root finding

$$f(x, y, z) = 0$$

$$f(\mathbf{x}) = 0$$

$$\mathbf{a} + t\mathbf{v}$$

$$f(\mathbf{a} + t\mathbf{v}) = 0$$

Accurate Intersection: Computing roots

- Options: numerical root finding
  - Interval halving
  - Newton's method with deflation
  - Bracketing with Sturm sequence

Interval halving

- Assume we have two points on ray
  - perhaps generated by some form of spatial subdivision scheme
  - one on positive side, one on negative side of intersection
- Split the interval in half
  - One half has the root (+-)
  - Other doesn’t (++, --)
- Keep the one that does, and go again if it is too big

Newton's method

- Estimate is:
- Observe that:
- so update is:
-

$$t_n$$

$$f(t_n + \Delta t) = f(t_n) + \Delta t \frac{df}{dt} = 0$$

$$\Delta t = -\frac{f(t_n)}{\left(\frac{df}{dt}\right)}$$

Practicalities

- Deflation: if you have found a root, divide the polynomial by (t-root) to reduce degree
- Newton's method can behave badly
  - start in a good place
  - e.g. root from previous ray with this object
- Newton's method not efficient for shadow rays
- Newton's method doesn't guarantee closest root

Sturm sequences

- Build a sequence of polynomials

$$p_0(t) = f(t)$$
$$p_1(t) = \frac{df}{dt}$$
$$\ldots$$
$$p_k(t) = -\mathrm{rem}(p_{k-2}, p_{k-1})$$
$$\ldots$$
$$p_m$$
$$0$$

  - (where rem stands for remainder; f should not have repeated roots)

Sturm sequences

- write $\sigma(\xi)$ for the number of sign changes in

$$(p_0(\xi), p_1(\xi), p_2(\xi), ..., p_m(\xi))$$

- then for a<b, number of real roots in (a, b] is

$$\sigma(a) - \sigma(b)$$

Can bracket root using interval halving, use for shadow rays

Sturm sequences: example

$$p_0 = t^3 + 3t^2 - 1$$
$$p_1 = 3t^2 + 6t \qquad \text{so } p_0 = (t/3)p_1 + (1/3)p_1 - 2t - 1$$
$$p_2 = 2t + 1$$
$$p_3 = \text{constant}$$

-9/4

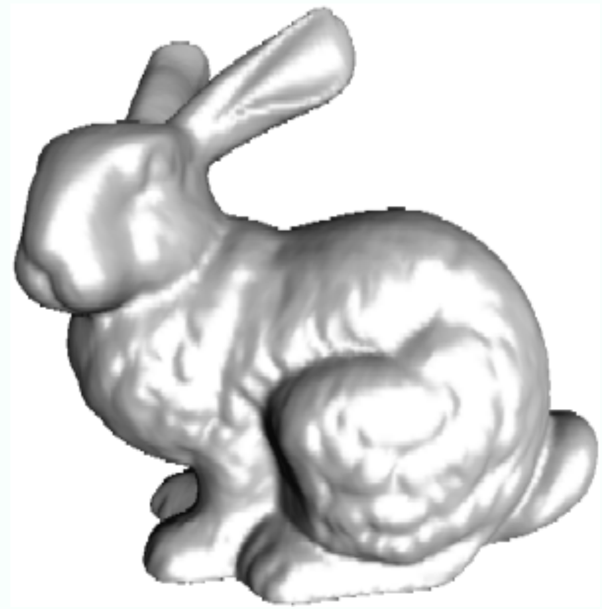
Ex: how many roots in 0-1 interval?

how many roots in 0 - infinity interval?

find root in 0-1 interval

Making Ray Tracing Faster

- Coherence
  - Image coherence: rays through nearby pixels go through nearby things
  - Spatial coherence: similar rays go through similar things
  - Temporal coherence: the same ray at the next time goes through similar things



Stanford Bunny
~70K triangles

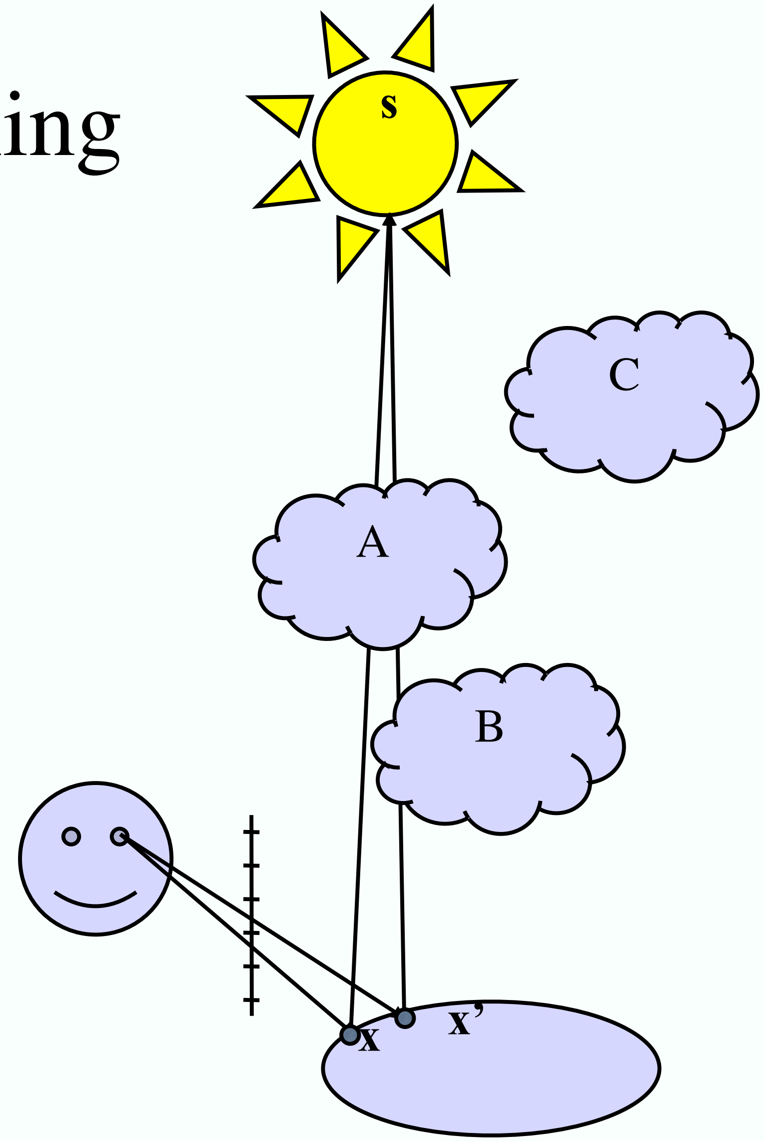
Do we need 70K ray-triangle
intersections for each ray?

Item buffer

- Use conventional z-buffer renderer to render surfaces
  - shade with pointer, not illumination
  - this gives pointer to closest surface
  - not much used now (ex: why?)

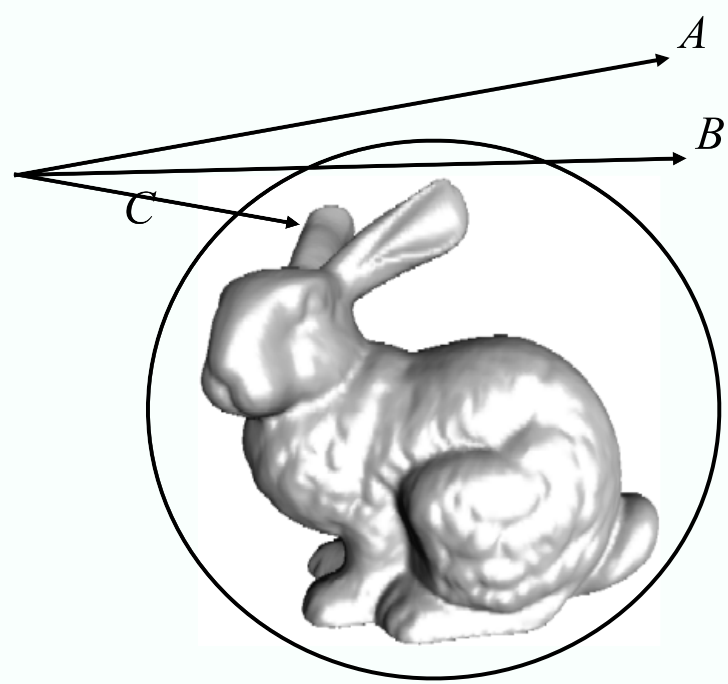
Shadow Caching

- Any interloper between surface point **x** and the light source **s** will cast a shadow
  - Doesn't matter how many
  - Doesn't matter which is closest
  - Stop ray intersections once *any* intersection found
- Neighboring shadowed surface points **x** and **x'** probably shadowed by the same object
  - Start shadow ray intersection search with object intersected in last shadow search



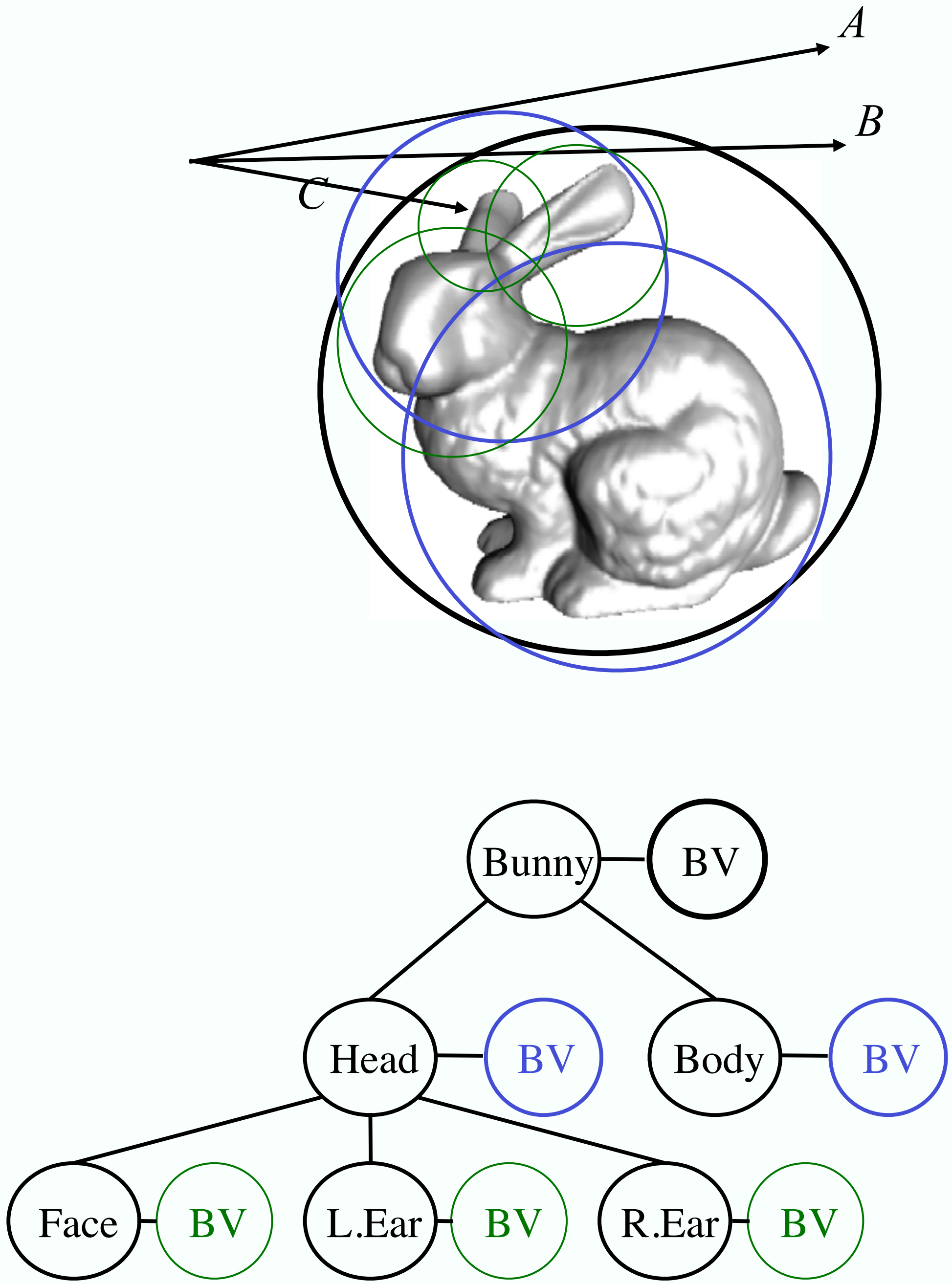
Bounding Volume

- Ray-bunny intersection takes 70K ray-triangle intersections even if ray misses the bunny
- Place a sphere around bunny
  - Ray *A* misses sphere so ray *A* misses bunny without checking 70K ray-triangle intersections
  - Ray *B* intersects sphere but still misses bunny after checking 70K intersections
  - Ray *C* intersects sphere and intersects bunny
- Can also use axis-aligned bounding box
  - Easier to create for triangle mesh

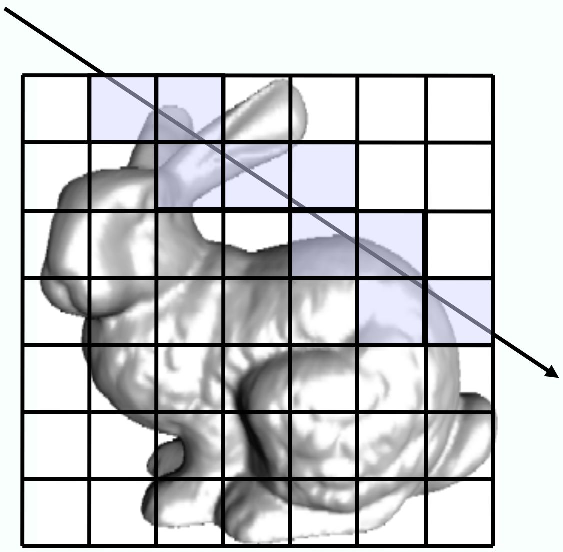


Bounding Volume Hierarchy

- Associate bounding volume with each node of scene graph
- If ray misses a node's bounding volume, then no need to check any node beneath it
- If ray hits a node's BV, then replace it with its children's BV's (or geometry)
- Breadth first search of tree
  - Maintain heap ordered by ray-BV intersection *t*-values
  - Explore children of node w/least pos. ray-BV *t*-value


A
B
C
Bunny
BV
Head
BV
Body
BV
Face
BV
L.Ear
BV
R.Ear
BV

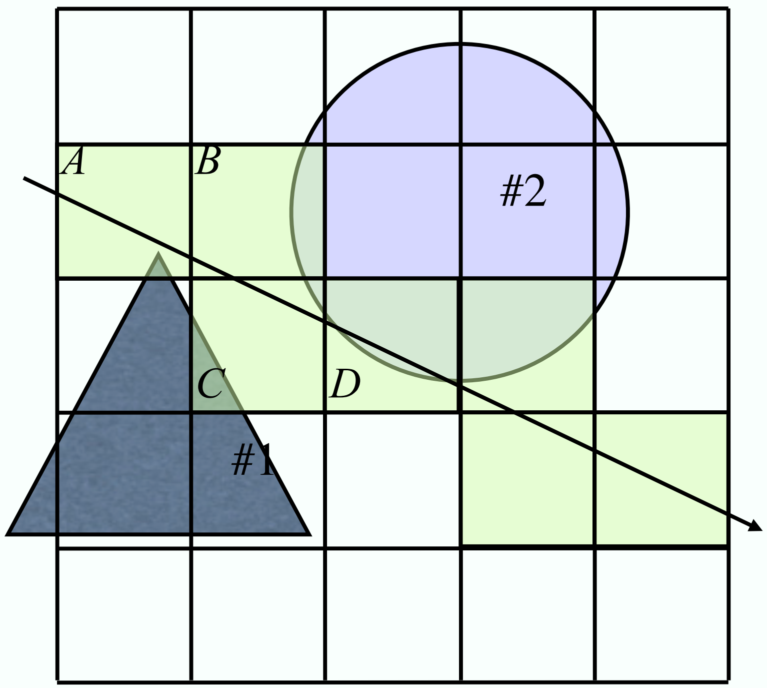
Grids



- Encase object in a 3-D grid of cubes
  - each has list of all triangles it intersects
- Rasterize ray to find which cells it intersects
  - 3D Bresenham algorithm
  - All cells that contain any part of ray
- Working from first ray-cell to last…
  - Find least positive intersect of ray with triangles in cell's list
  - If no intersection, move on to next cell

Tagging

- Ray-object intersection test valid for ray with entire object
  - not just portion of object inside current cell
  - Need only intersect object once for each ray
- Tags
  - does not intersect
  - intersection at ...

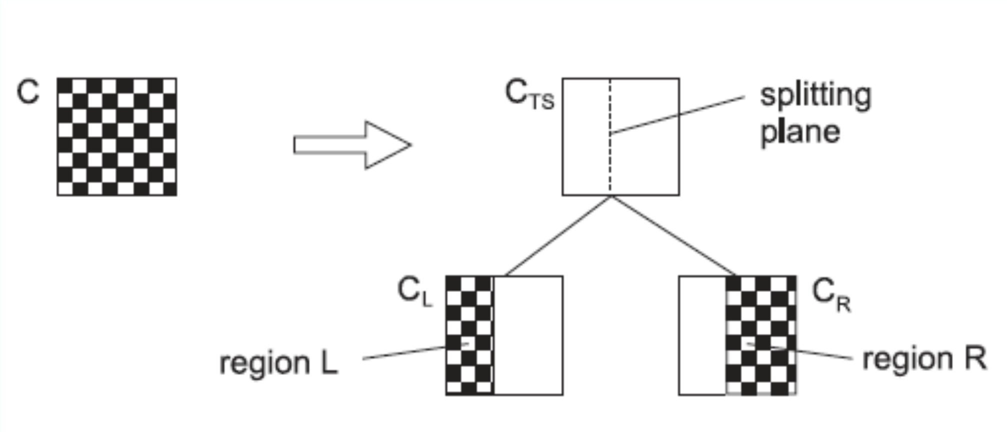


K-D trees

- Put bounding box around all objects
  - split with coordinate plane (x, y, or z) into two boxes
  - distribute objects into boxes
    - split each child box recursively until stop
- Questions:
  - how do we compute intersections?
    - easy
      - pass ray into children it intersects
      - intersect with objects in leaf nodes
  - what is a good split?
  - how should we stop splitting?

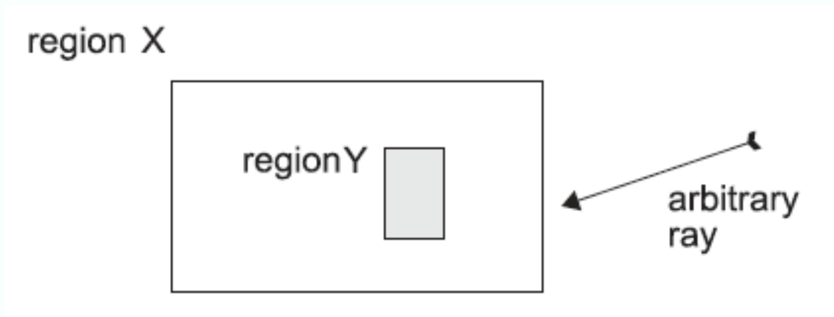
K-D trees - what is a good split?

- Keep track of intersection costs
  - cheap to intersect with nearly empty boxes
  - expensive to intersect with a box with lots of stuff
  - expensive to look at many small boxes
- Cost of split=
  - Cost of traversal+Cost Left Intersect +Cost Right Intersect
- Need a model for intersect costs



K-D trees - what is a good split?

- Intersect cost model:
  - Each box contains voxels on some fine grid
  - Filled voxels might be convex
  - If they were, probability of intersection would be ratio of surface areas



$$\text{Expected cost of ray entering box} = \frac{S_y}{S_x}\text{Base cost of intersection}$$

K-D trees - what is a good split?

- Expected cost of split =
  - expected cost of LHS box+
  - expected cost of RHS box+
  - cost of traversal
- Notice expression does not depend on probability ray visits parent

K-D trees

- Splits occur only on planes that bound filled voxels
- Search all splits for lowest cost, using model
- Stopping
    - fixed depth
    - threshold number of objects per voxel
    - both
    - adaptive (i.e. make cost estimate for each leaf, split of each leaf)

http://www.flipcode.com/archives/Raytracing_Topics_Techniques-Part_7_Kd-Trees_and_More_Speed.shtml