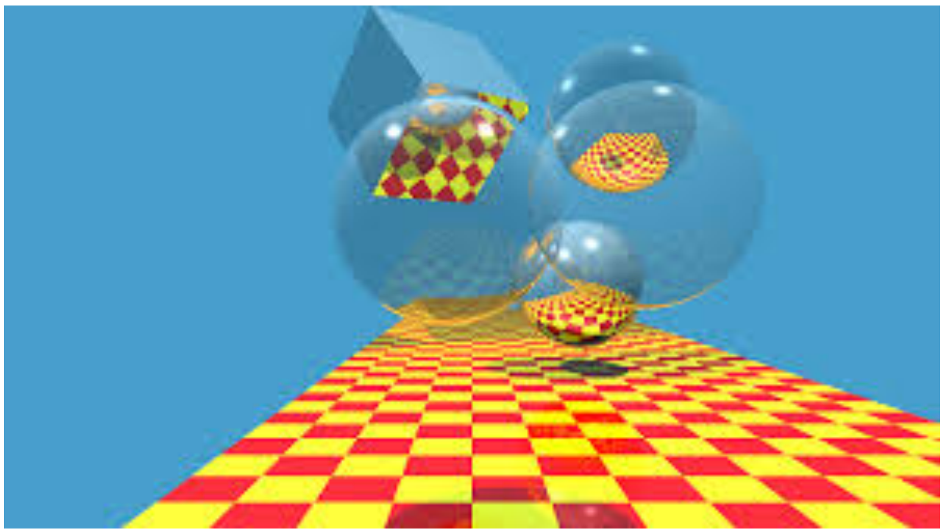
Ray Tracing

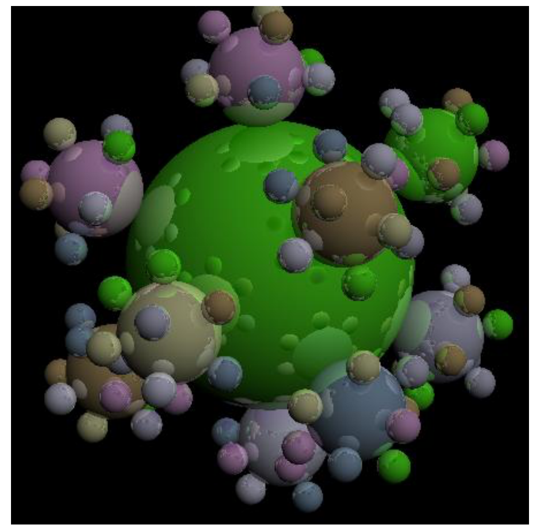
CS 4620 Lecture 34

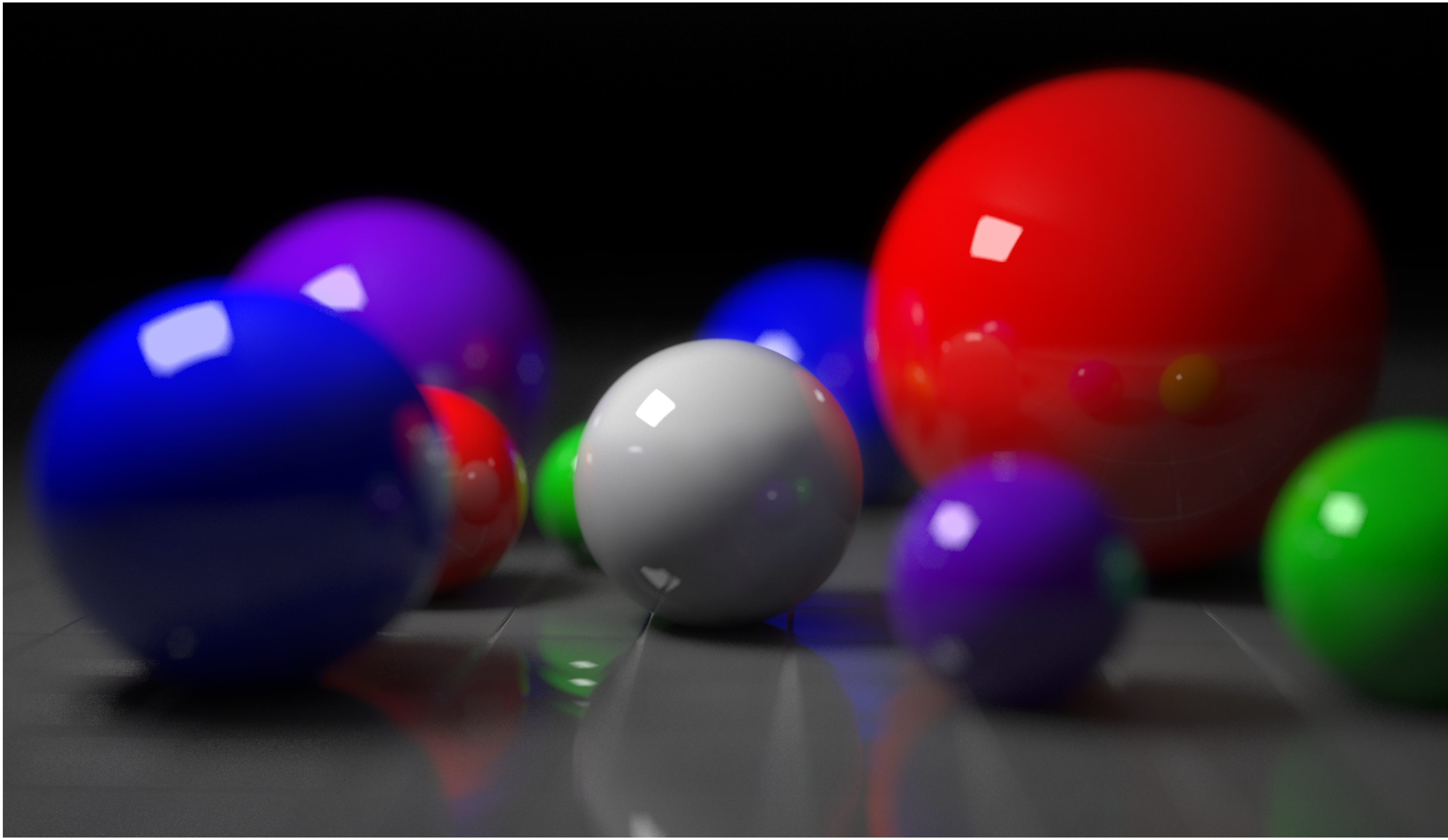
Next few weeks

- This week
  - Ray Tracing
  - 4621: Meet with TAs for feedback
  - A6 due
- Next week
  - Ray Tracing
  - TG!
- Last week of classes
  - Imaging, Research
  - A7 due

Back to ray tracing







Topics

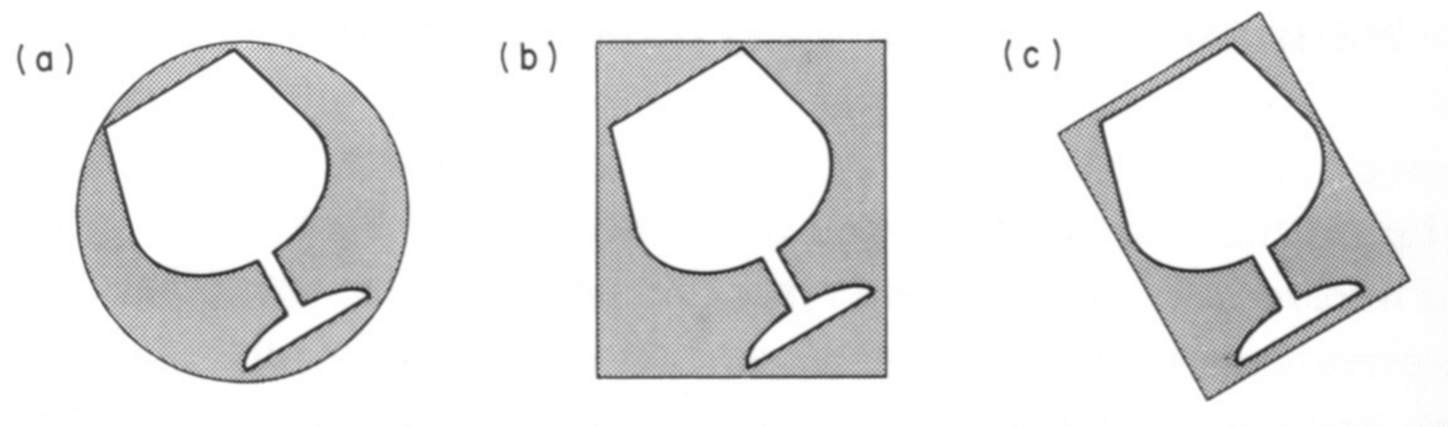
- Ray tracing acceleration structures
  - Bounding volumes
  - Bounding volume hierarchies
  - Uniform spatial subdivision
  - Adaptive spatial subdivision
- Transformations in ray tracing
  - Transforming objects
  - Transformation hierarchies

Ray tracing acceleration

- Ray tracing is slow. This is bad!
  - Ray tracers spend most of their time in ray-surface intersection methods
- Ways to improve speed
  - Make intersection methods more efficient
    - Yes, good idea. But only gets you so far
  - Call intersection methods fewer times
    - Intersecting every ray with every object is wasteful
    - Basic strategy: efficiently find big chunks of geometry that definitely do not intersect a ray

Bounding volumes

- Quick way to avoid intersections: bound object with a simple volume
  - Object is fully contained in the volume
  - If it doesn't hit the volume, it doesn't hit the object
  - So test bvol first, then test object if it hits



[Glassner 89, Fig 4.5]

Bounding volumes

- Cost: more for hits and near misses, less for far misses
- Worth doing? It depends:
  - Cost of bvol intersection test should be small
    - Therefore use simple shapes (spheres, boxes, …)
  - Cost of object intersect test should be large
    - Bvols most useful for complex objects
  - Tightness of fit should be good
    - Loose fit leads to extra object intersections
    - Tradeoff between tightness and bvol intersection cost

Implementing bounding volume

- Just add new Surface subclass, "BoundedSurface"
  - Contains a bounding volume and a reference to a surface
  - Intersection method:
    - Intersect with bvol, return false for miss
    - Return surface.intersect(ray)
  - This change is transparent to the renderer (only it might run faster)
- Note that all Surfaces will need to be able to supply bounding volumes for themselves

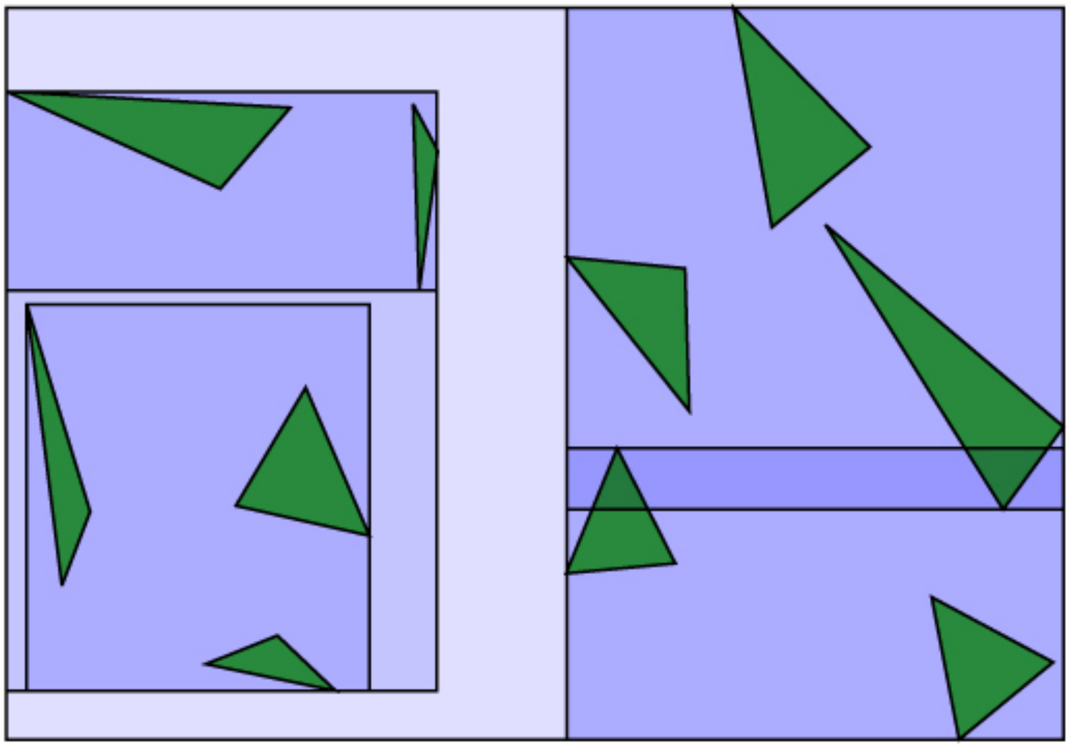
If it's worth doing, it's worth doing hierarchically!

- Bvols around objects may help
- Bvols around groups of objects will help
- Bvols around parts of complex objects will help
- Leads to the idea of using bounding volumes all the way from the whole scene down to groups of a few objects

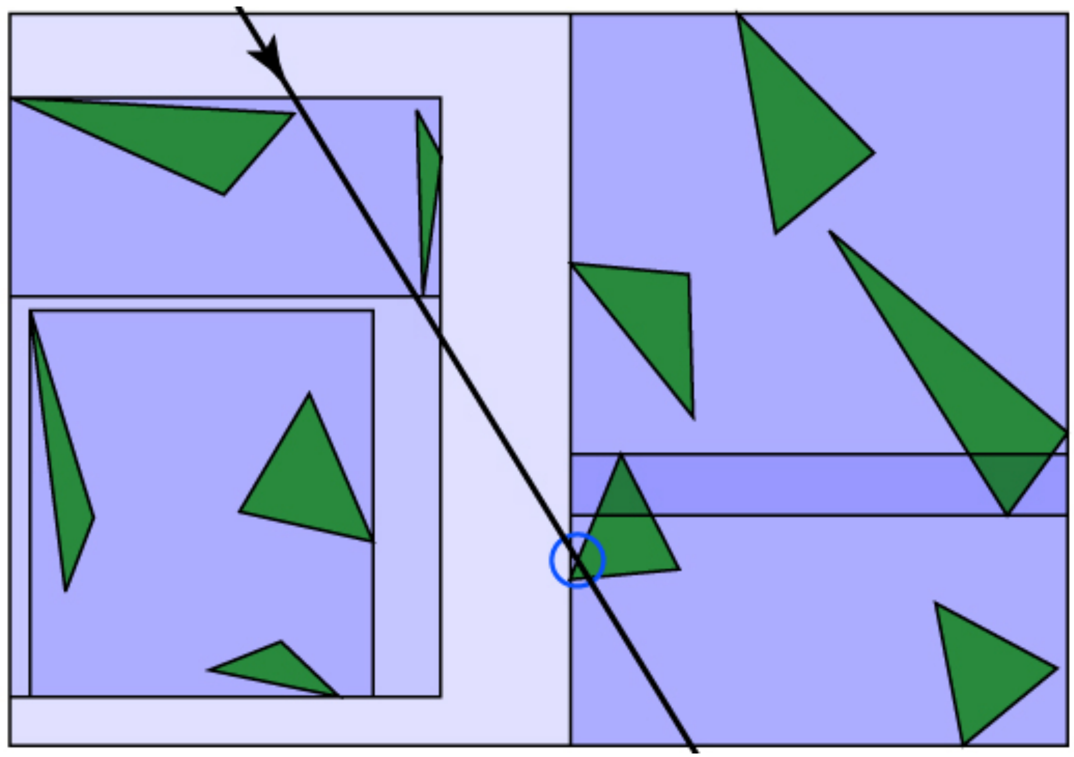
Implementing a bvol hierarchy

- A BoundedSurface can contain a list of Surfaces
- Some of those Surfaces might be more BoundedSurfaces
- Voilà! A bounding volume hierarchy
  – And it's all still transparent to the renderer

BVH construction example



BVH ray-tracing example



BVH Intersection

- Trace ray with root node

- If intersection, trace rays with ALL children
  - If no intersection, eliminate tests with all children

Choice of bounding volumes

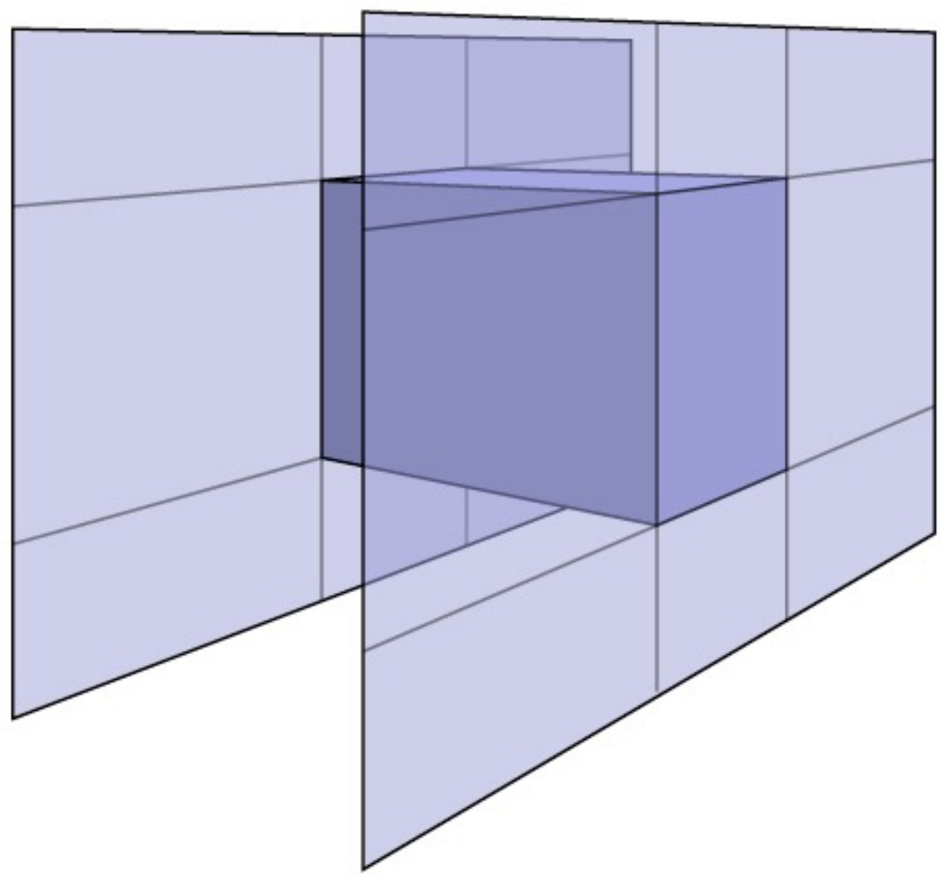
- Spheres -- easy to intersect, not always so tight
- Axis-aligned bounding boxes (AABBs) -- easy to intersect, often tighter (esp. for axis-aligned models)
- Oriented bounding boxes (OBBs) -- easy to intersect (but cost of transformation), tighter for arbitrary objects
- Computing the bvols
  – For primitives -- generally pretty easy
  – For groups -- not so easy for OBBs (to do well)
  – For transformed surfaces -- not so easy for spheres

Axis aligned bounding boxes

- Probably easiest to implement
- Computing for primitives
  - Cube: duh!
  - Sphere, cylinder, etc.: pretty obvious
  - Groups or meshes: min/max of component parts
- AABBs for transformed surface
  - Easy to do conservatively: bbox of the 8 corners of the bbox of the untransformed surface
- How to intersect them
  - Treat them as an intersection of slabs (see Shirley)

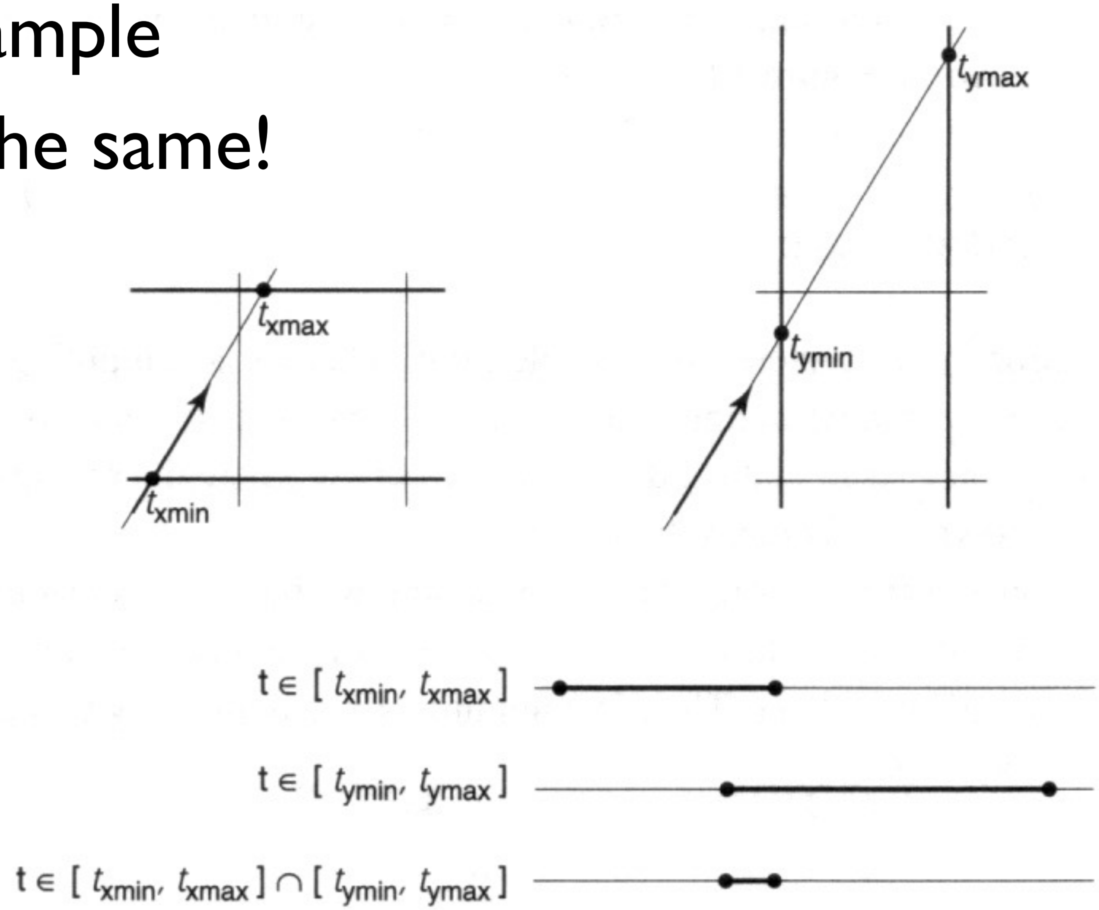
Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs



Intersecting boxes: 2D

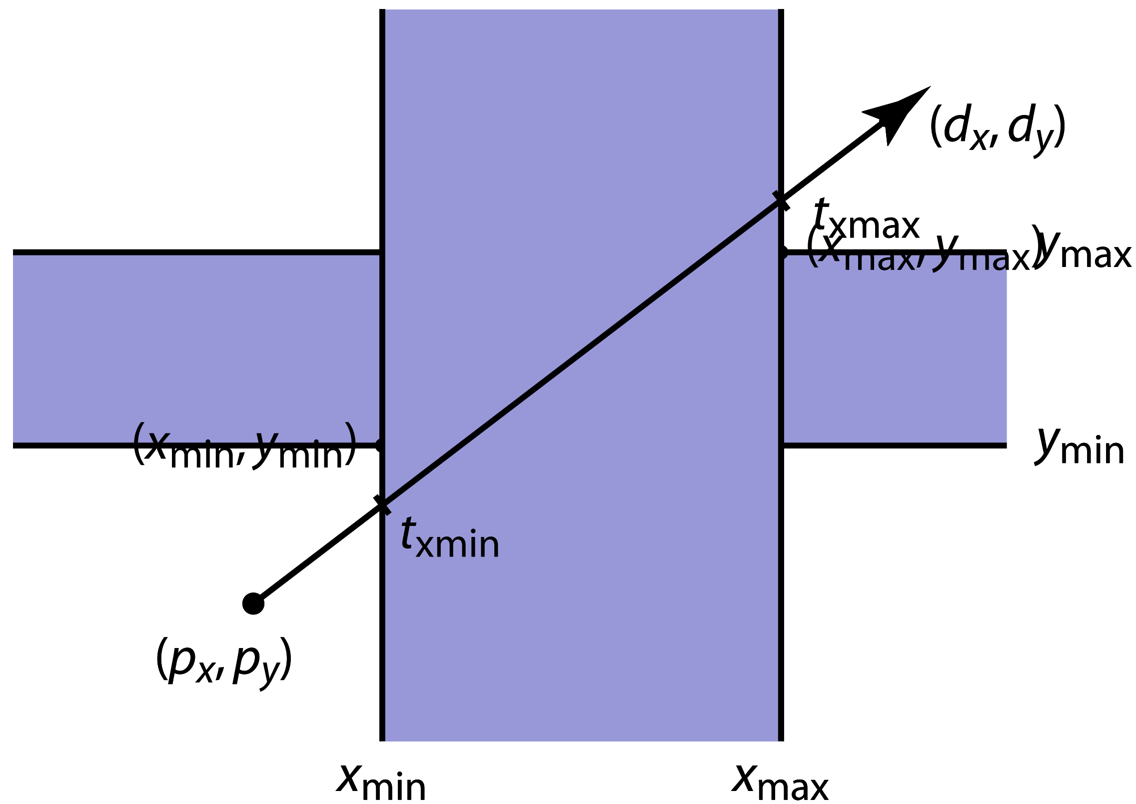
- 2D example
- 3D is the same!



Ray-slab intersection

$$p_x + t_{x\min}\, d_x = x_{\min}$$
$$t_{x\min} = (x_{\min} - p_x)/d_x$$

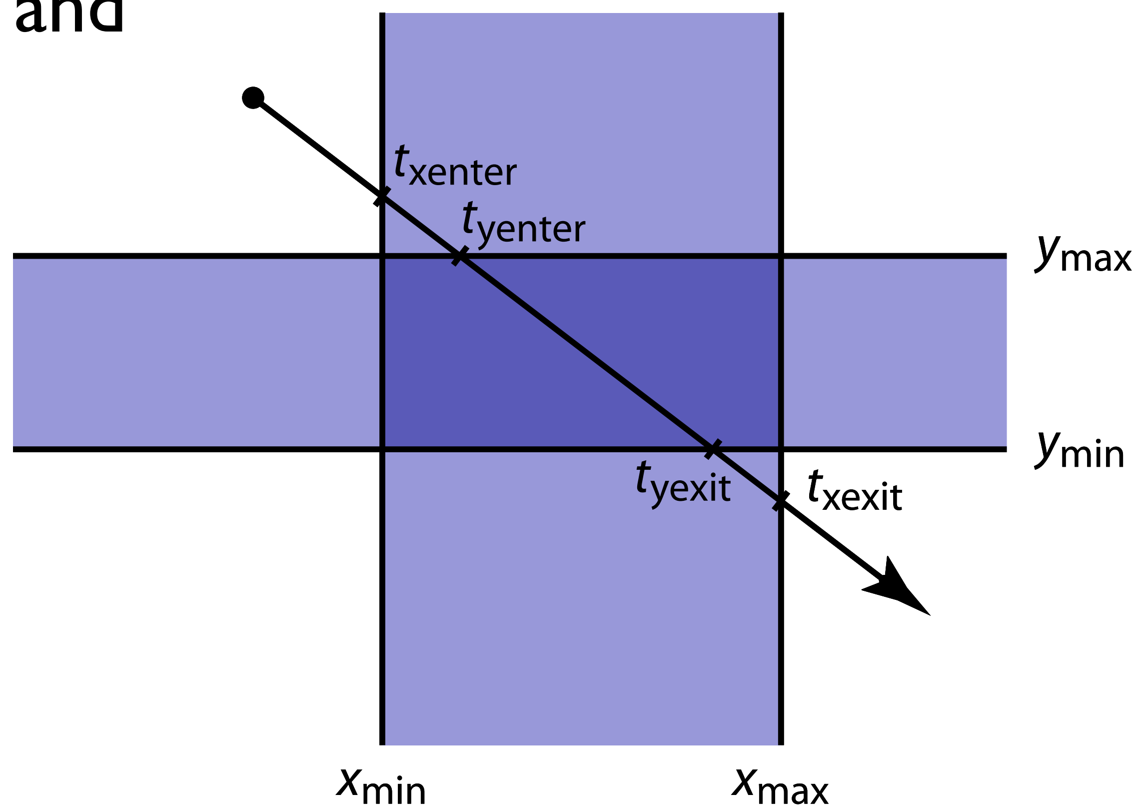
$$p_y + t_{y\min}\, d_y = y_{\min}$$
$$t_{y\min} = (y_{\min} - p_y)/d_y$$



Intersecting intersections

- Each intersection is an interval
- Want last entry point and first exit point

$$t_{x\text{enter}} = \min(t_{x\min}, t_{x\max})$$
$$t_{x\text{exit}} = \max(t_{x\min}, t_{x\max})$$
$$t_{y\text{enter}} = \min(t_{y\min}, t_{y\max})$$
$$t_{y\text{exit}} = \max(t_{y\min}, t_{y\max})$$
$$t_{\text{enter}} = \max(t_{x\text{enter}}, t_{y\text{enter}})$$
$$t_{\text{exit}} = \min(t_{x\text{exit}}, t_{y\text{exit}})$$



Building a hierarchy

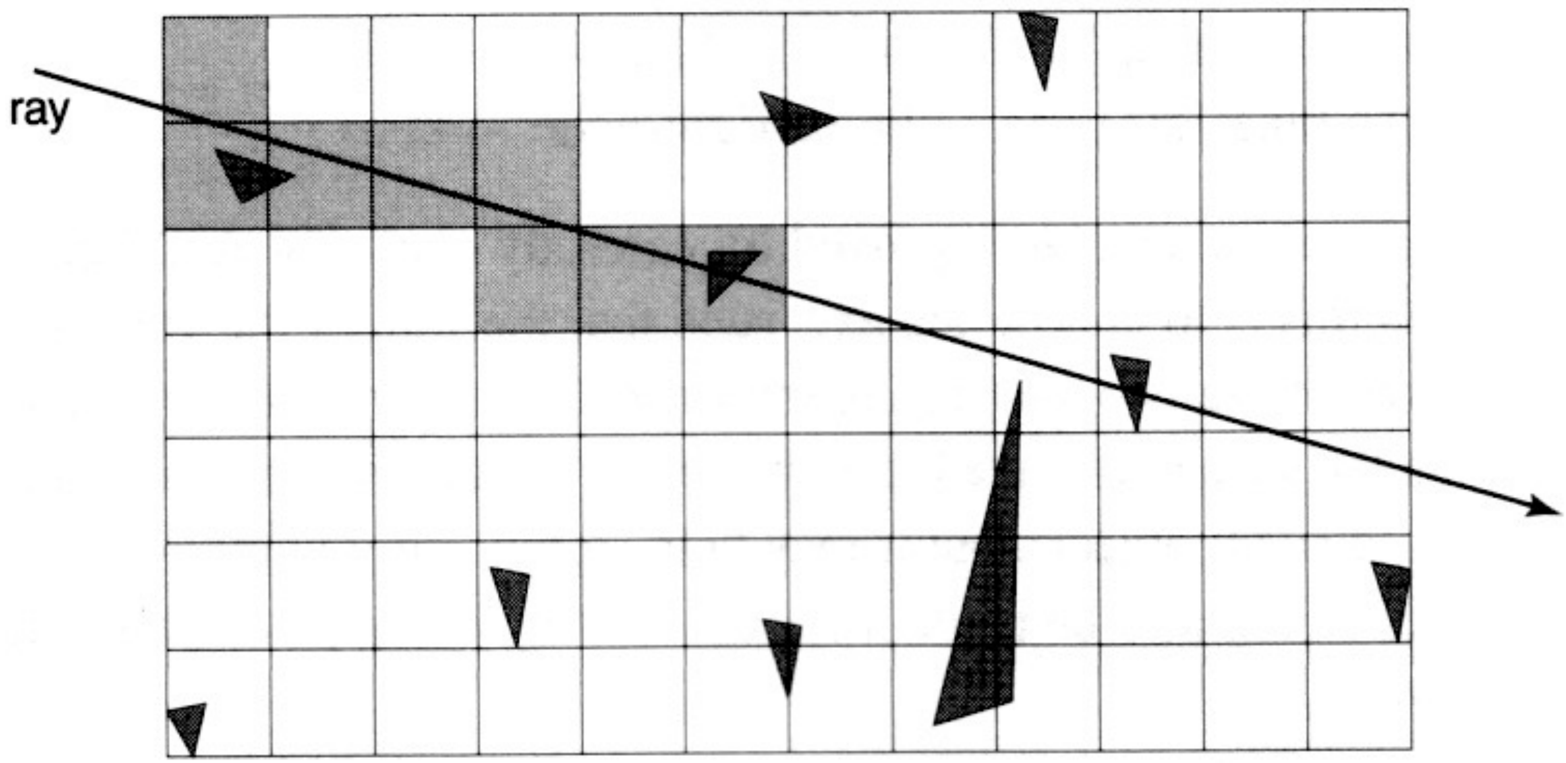
- Top Down vs Bottom Up
- Top down
  - Make bbox for whole scene, then split into (maybe 2) parts
    - Recurse on parts
    - Stop when there are just a few objects in your box
- Bottom Up
  - Ideal: partitions
  - Expensive, but optimal
  - Good for static (maybe)

Building a hierarchy

- How to partition?
  - Ideal: clusters
  - Practical: partition along axis
    - Center partition
      - Less expensive, simpler
      - Unbalanced tree
    - Median partition
      - More expensive
      - More balanced tree
    - Surface area heuristic
      - Model expected cost of ray intersection
      - Generally produces best-performing trees

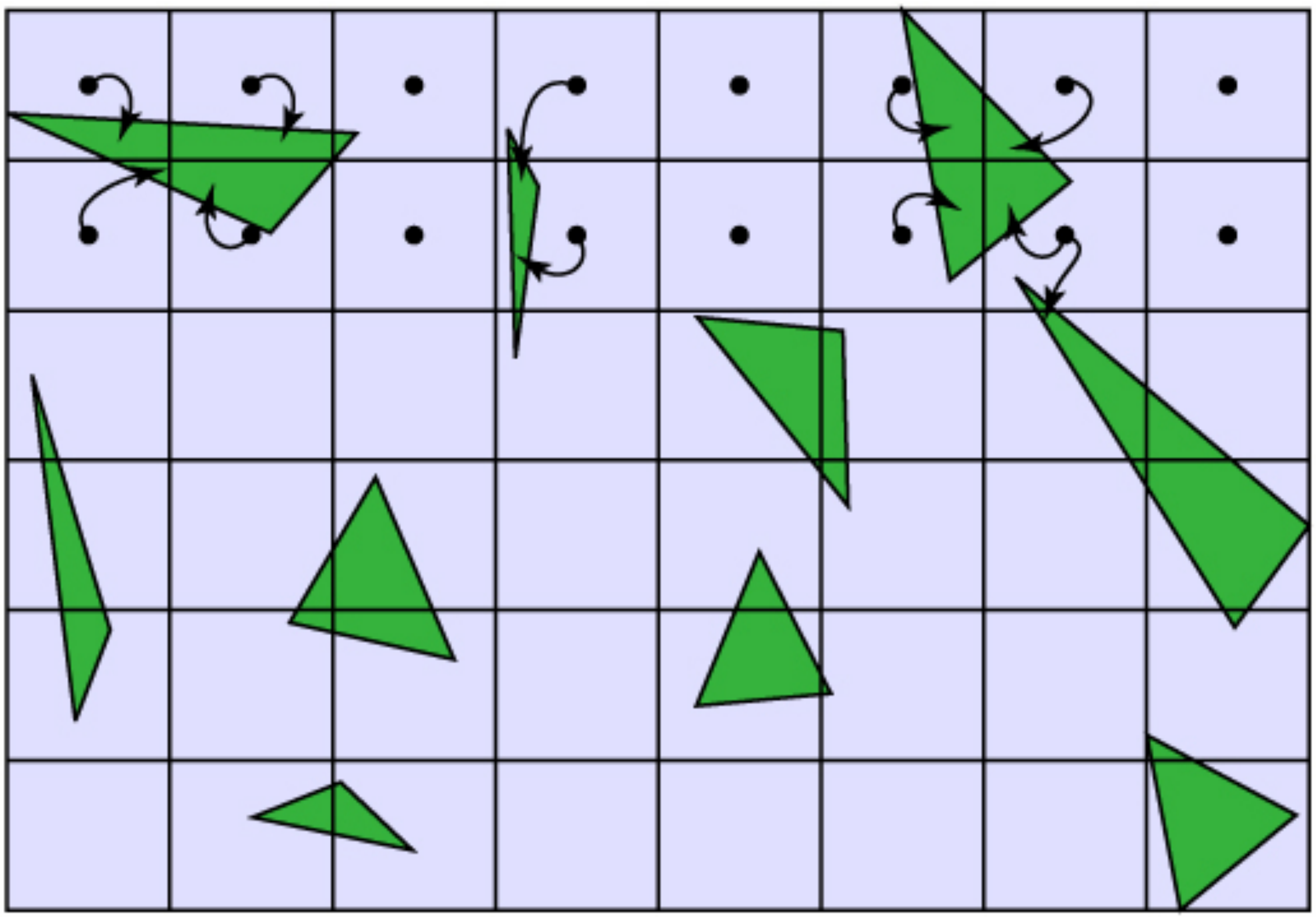
Regular space subdivision

- An entirely different approach: uniform grid of cells

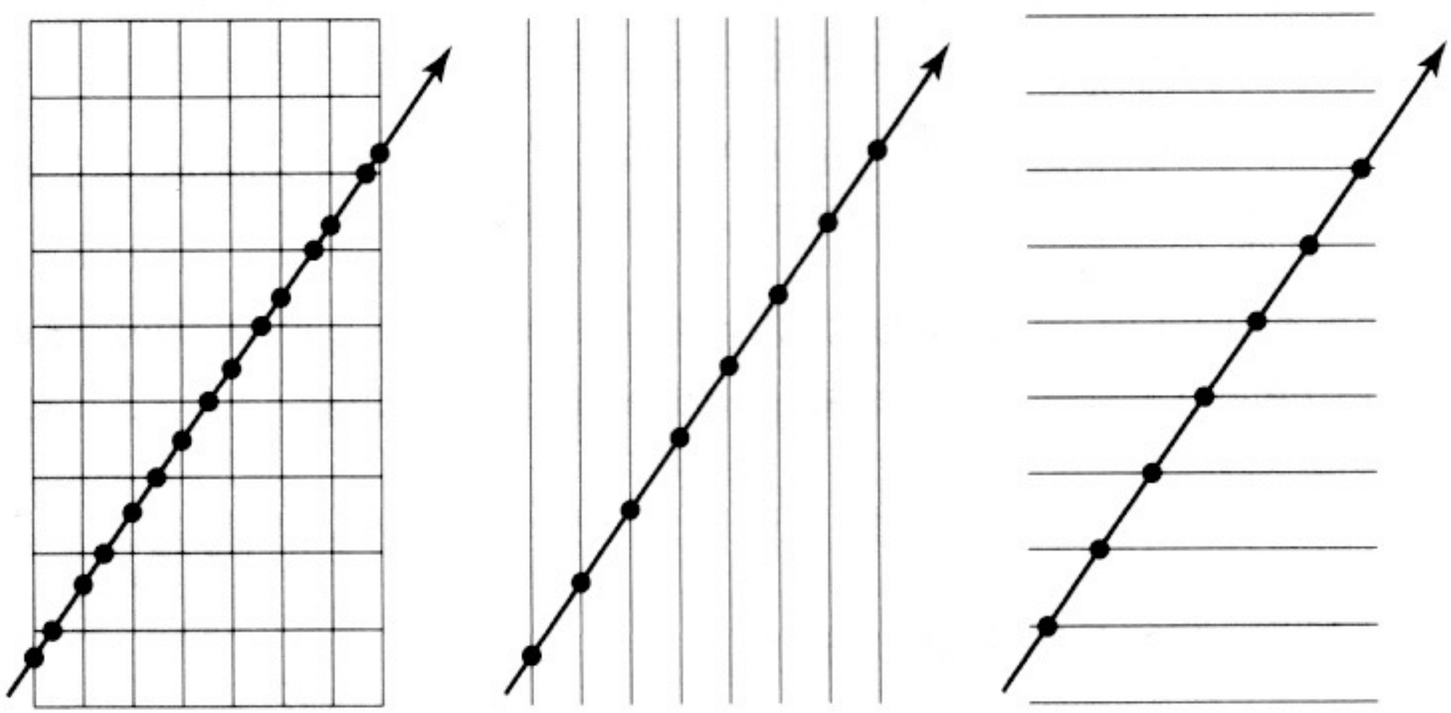


Regular grid example

- Grid divides space, not objects

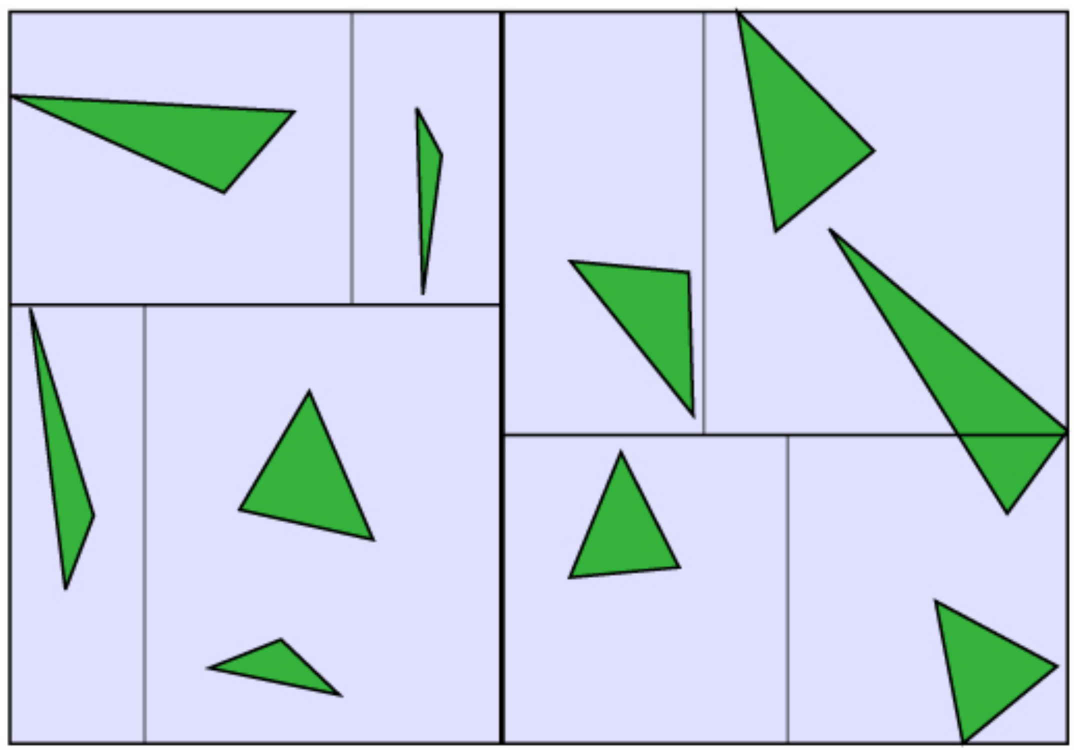


Traversing a regular grid



Non-regular space subdivision

- *k*-d Tree
  - subdivides space, like grid
  - adaptive, like BVH



Implementing acceleration structures

- Conceptually simple to build acceleration structure into scene structure
- Better engineering decision to separate them

