Ray Tracing Acceleration

Steve Marschner
CS 5630
Cornell University

Ray tracing acceleration

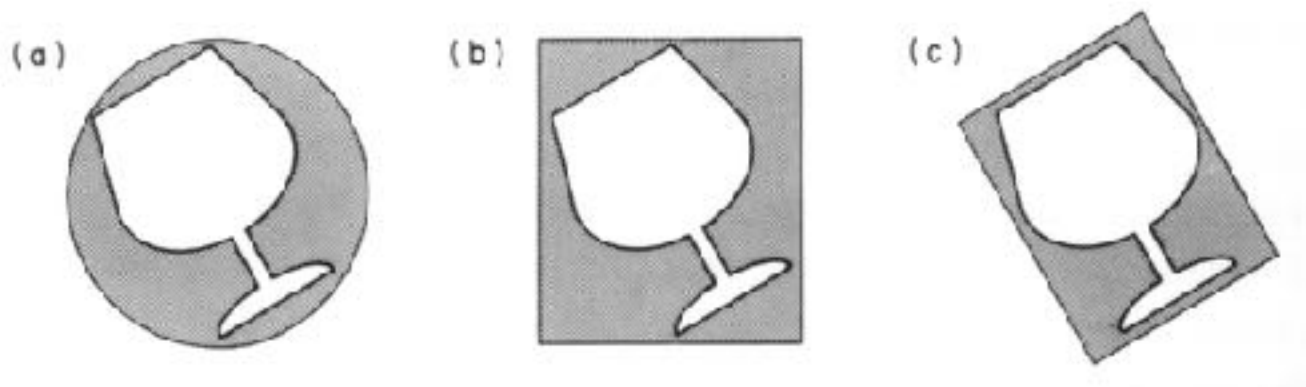
- Ray tracing is slow. This is bad!
  - Ray tracers spend most of their time in ray-surface intersection methods
- Ways to improve speed
  - Make intersection methods more efficient
    - Yes, good idea. But only gets you so far
  - Call intersection methods fewer times
    - Intersecting every ray with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

How to avoid work

- Think of a simple region of space
  - should be fast to test containment or overlap
- Strategy 1: space subdivision
  - ray (or part of ray) is inside the volume
  - object is entirely outside the volume
  - so skip the intersection test
- Strategy 2: bounding volumes
  - object is entirely inside the volume
  - ray is entirely outside
  - so skip the intersection test

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

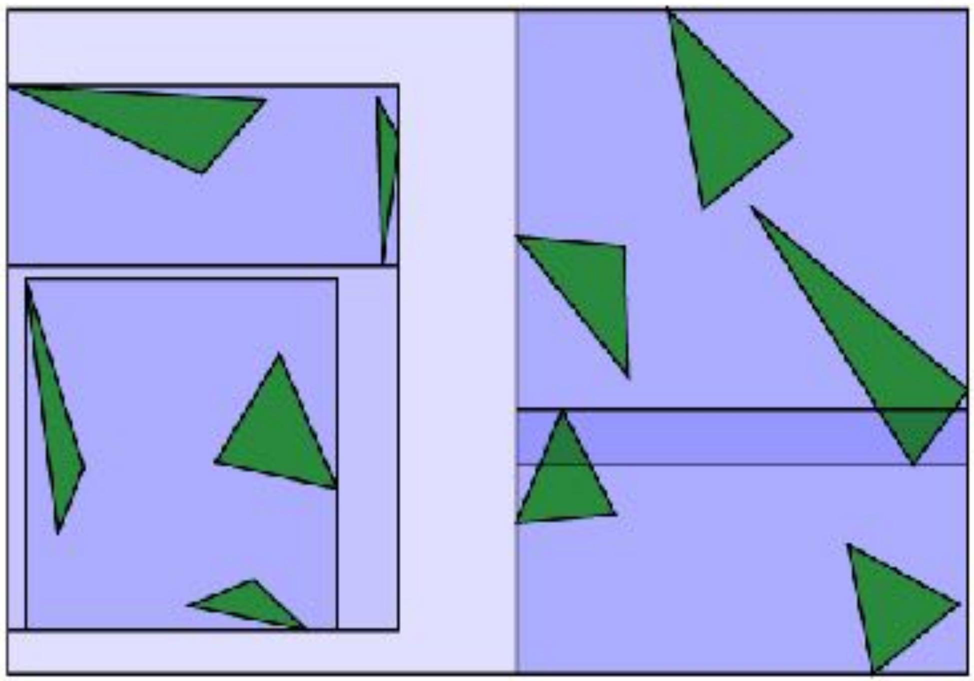
If it's worth doing, it's worth doing hierarchically!

- 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

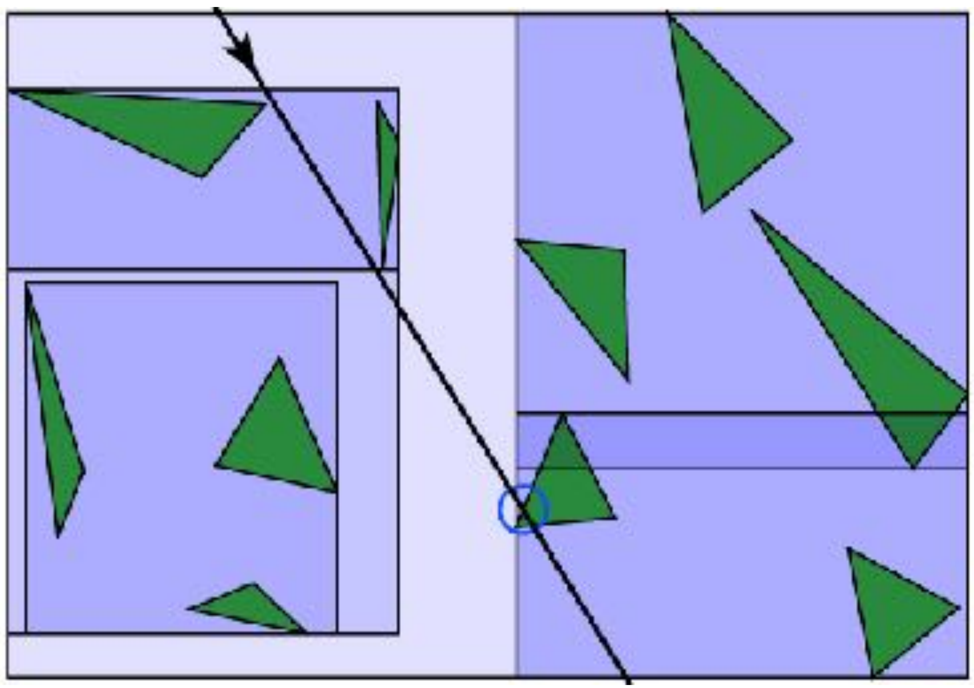
Implementing a bvol hierarchy

- A bounding volume hierarchy is a tree of boxes
  - each bounding box contains all children
  - ray misses parent implies ray misses all children
- Leaf nodes contain surfaces
  - again the bounding box contains all geometry in that node
  - if ray hits leaf node box, then we finally test the surfaces
- Replace the intersection loop over all objects in the scene with a partial tree traversal
  - test node first; test all children only ray hits parent
- Usually we use binary trees (each non-leaf box has exactly two contained boxes)

BVH construction example



BVH ray-tracing example



Choice of bounding volumes

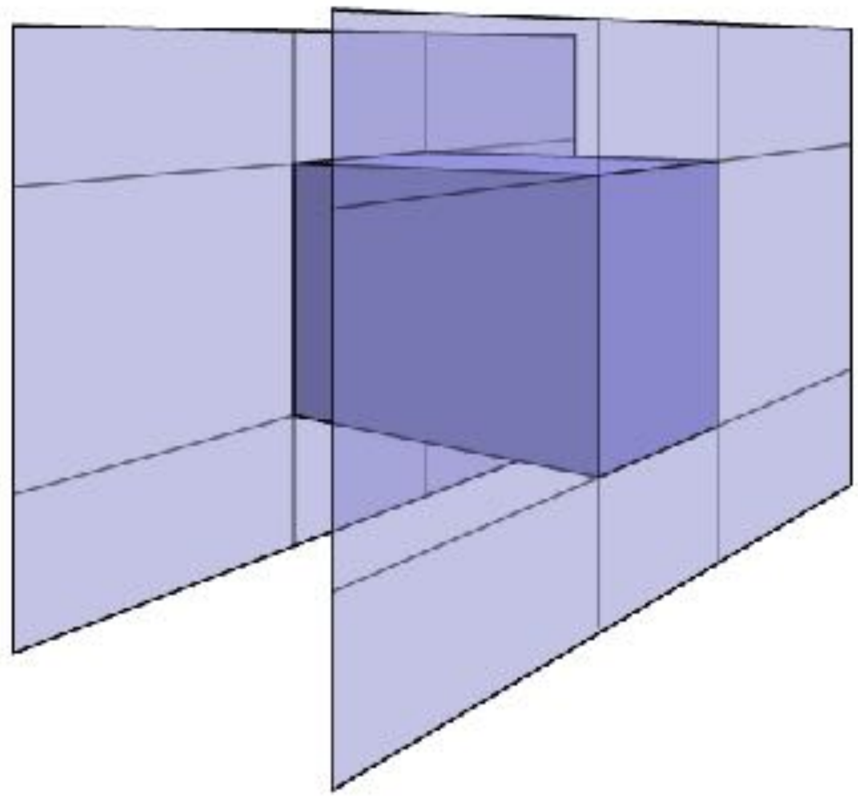
- Spheres -- easy to intersect, not always so tight
- Axis-aligned bounding boxes (AABBs) -- easy to intersect, often tighter (esp. for axis-aligned models)
- Oriented bounding boxes (OBBs) -- easy to intersect (but cost of transformation), tighter for arbitrary objects
- Computing the bvols
  - For 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 (axis-aligned) primitives
  - Cube: duh!
  - Sphere, cylinder, etc.: pretty obvious
  - Triangles: compute min/max of vertex coordinates
  - Groups or meshes: min/max of component parts
- How to intersect them
  - Treat them as an intersection of slabs (see also textbook)

Ray-box intersection

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

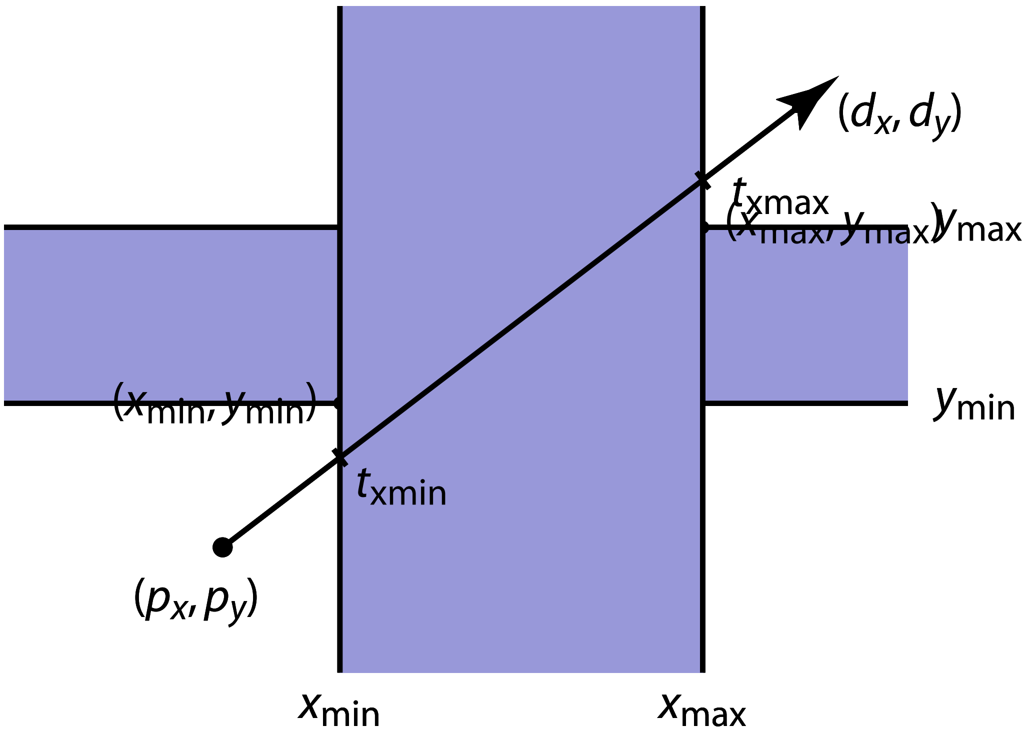


Ray-slab intersection

- 2D example
- 3D is the same!

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

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



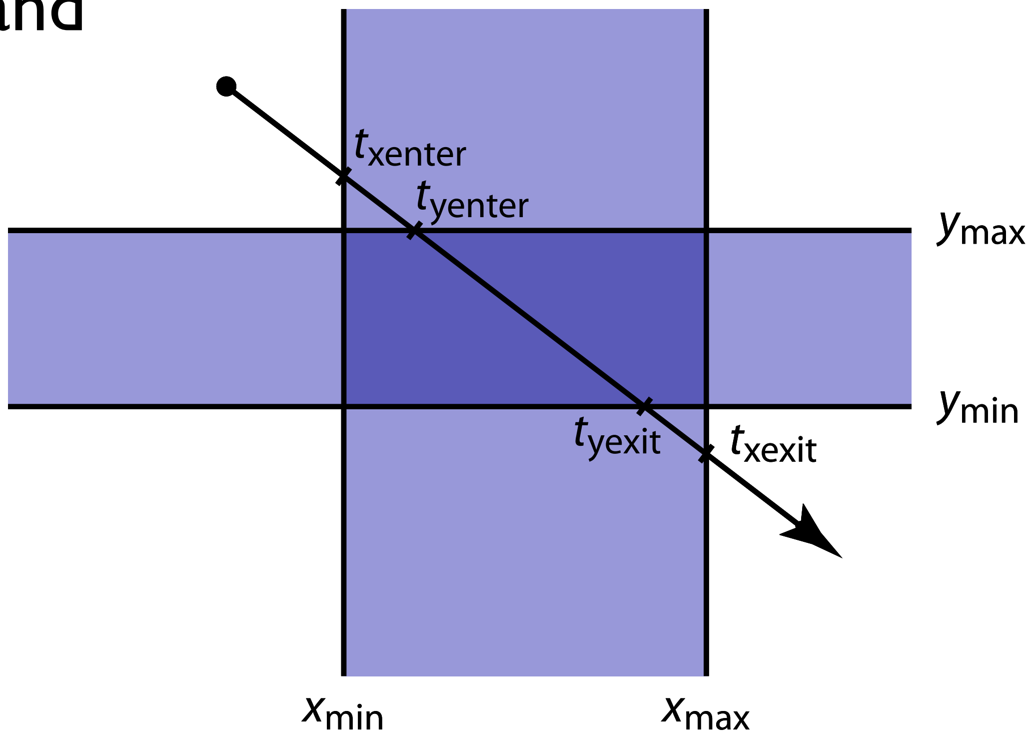
Intersecting intersections

- 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

- Input: list of triangles
- Output: tree
- Top-down strategy:
  - make bbox for the whole list
  - if list is short enough:
    - return a leaf node with all the triangles in it
  - if list is is too long:
    - split list into 2 parts
    - recursively build subtree for each part
    - return an internal node with those 2 children

Building the hierarchy

- How to partition?
  - Ideal: clusters
  - Practical: partition along the longest axis
    - Center partition
      - less expensive, simpler
      - unbalanced tree (but may sometimes be better)
    - Median partition
      - more expensive
      - more balanced tree
    - Surface area heuristic
      - models expected cost of ray intersection
      - generally produces best-performing trees

Surface area heuristic

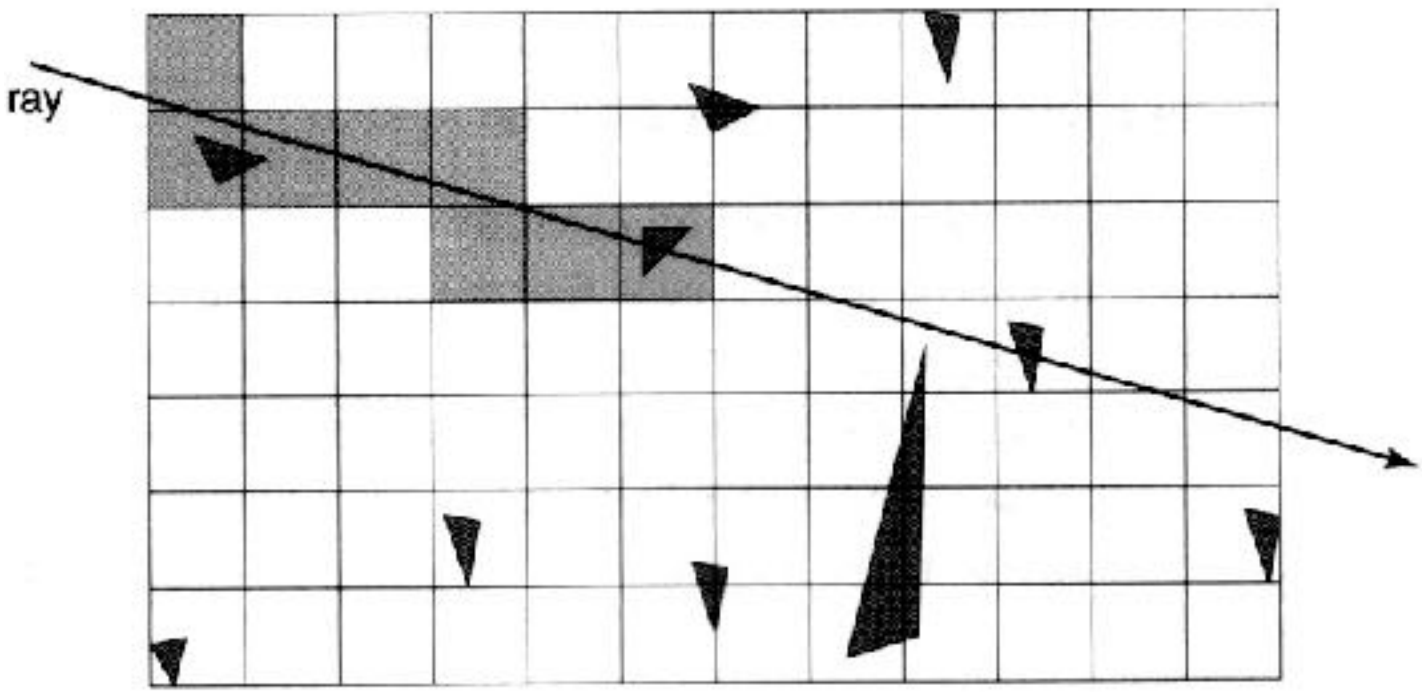
- What makes a good split?
  - it should be on average cheap to ray trace
  - very simple cost model: cost proportional to #objects
    Pr{hit left box} (# left) + Pr{hit right box} (# right) + C
  - for random lines in space, the probability of intersecting a convex shape is proportional to the surface area
- How to find the best split?
  - assume we will divide primitives at some axis-aligned plane
  - try a reasonable number of such planes
    - for each one compute the bboxes and their SA
    - choose smallest SA(left) (# left) + SA(right) (# right)
  - this is not perfect but it is a good balance of simple / good

Using the hierarchy

- Input: ray and tree (could be subtree)
- Output: smallest t, corresponding hit data
- Strategy:
  - Ray hits this tree's bbox? No $\Longrightarrow$ miss
  - For leaf node: intersect all triangles, return first hit
  - For internal node: intersect both children, return first hit

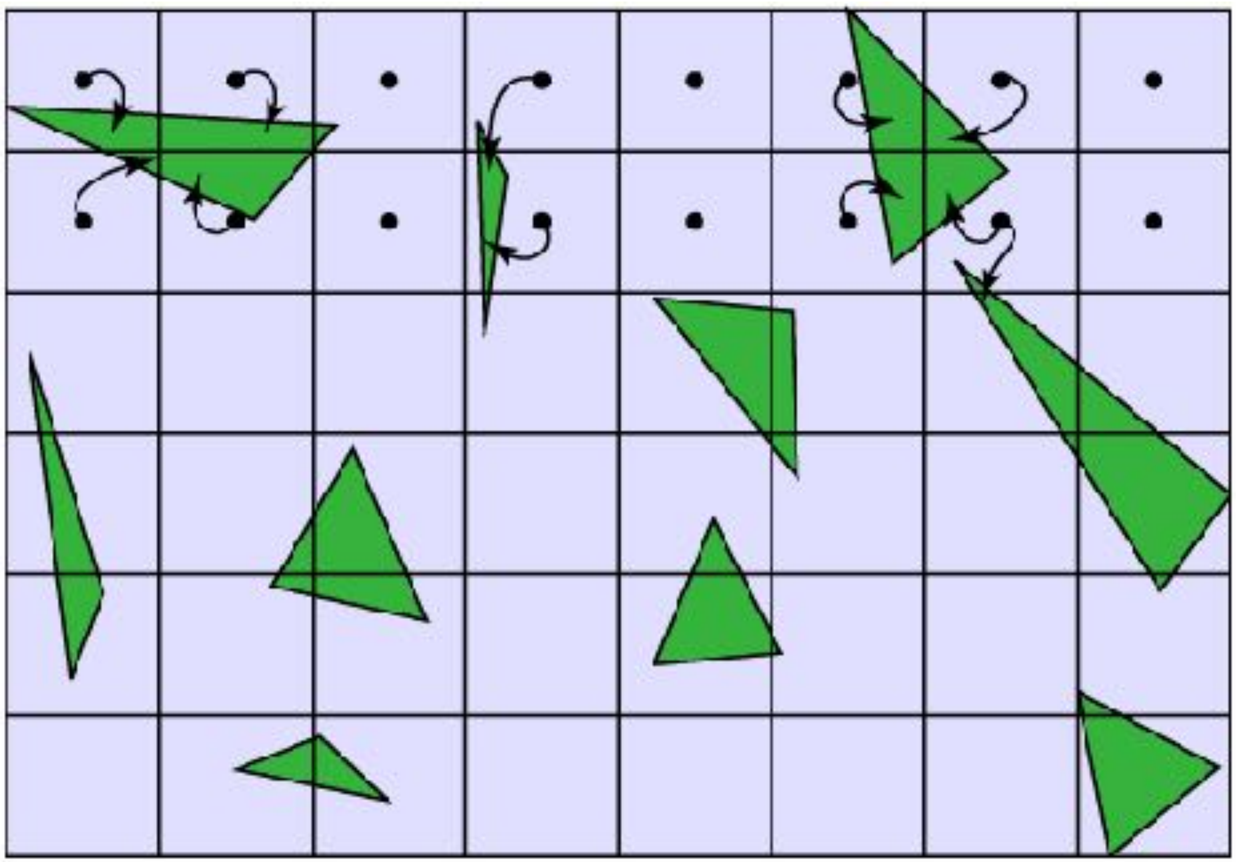
Regular space subdivision

- An entirely different approach: uniform grid of cells

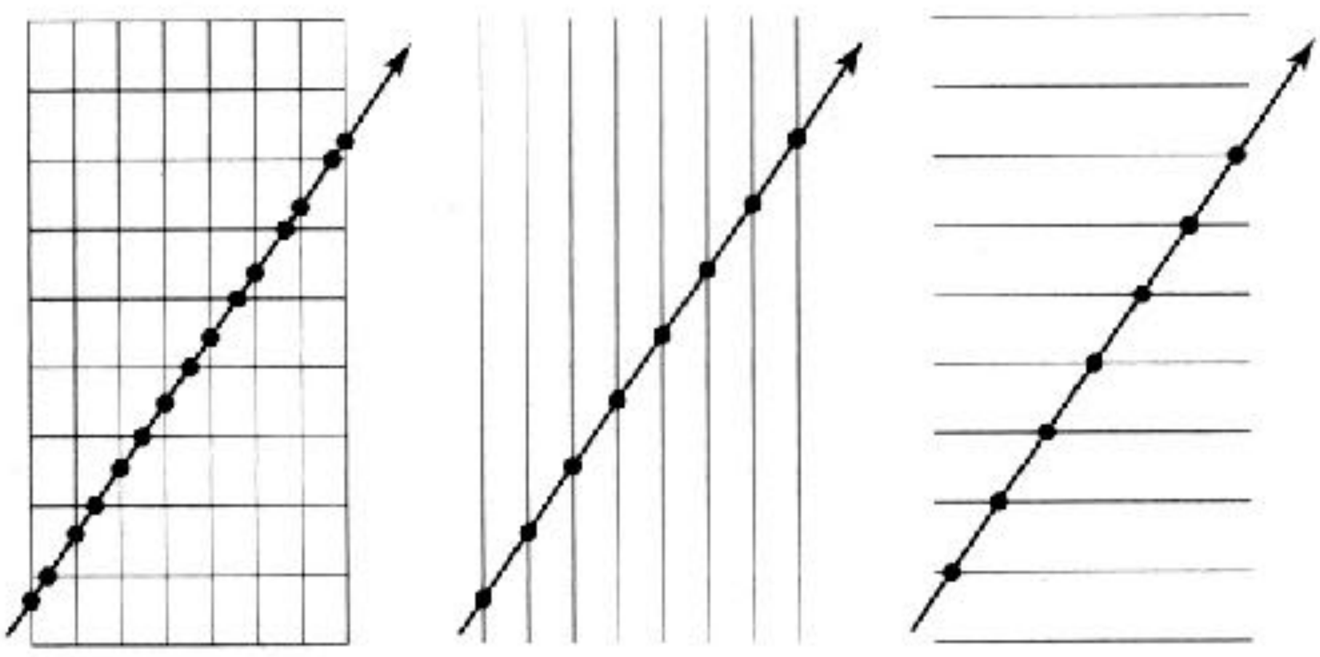


Regular grid example

- Grid divides space, not objects

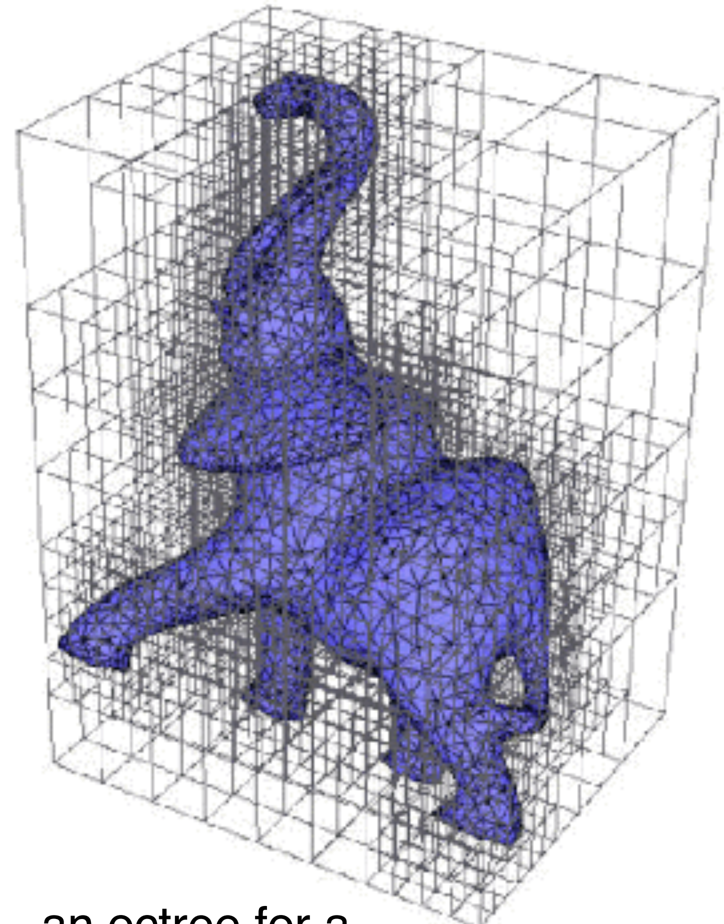


Traversing a regular grid



Nested grids

- Within each cell you can also organize objects in a grid
- Logical extreme is an octree
  - grid is 2x2x2
  - every cell contains a 2x2x2 grid or a small number of primitives

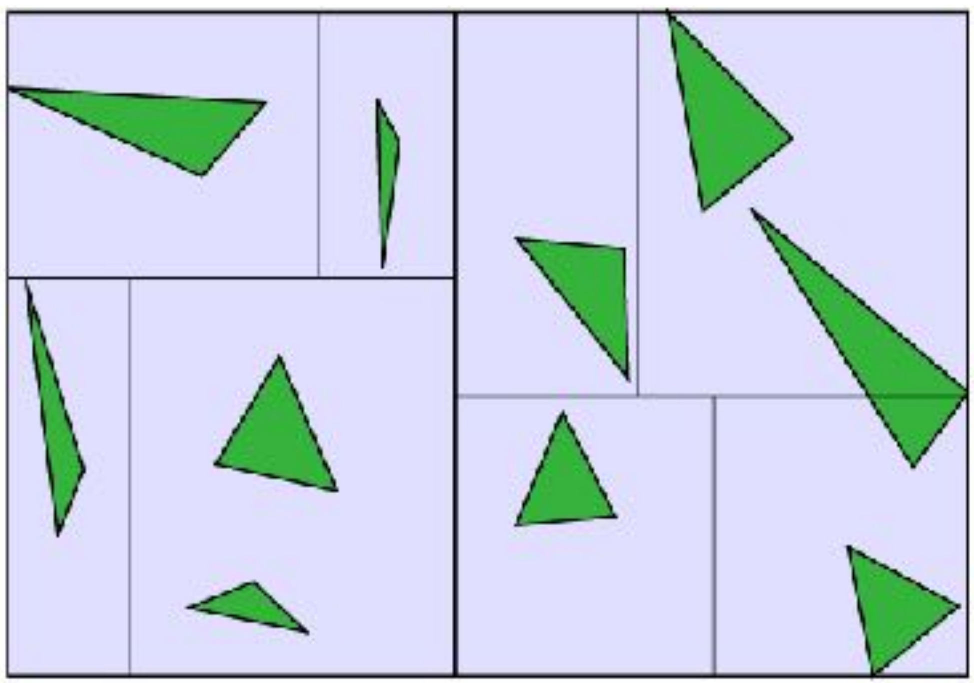


CGAL user manual

an octree for a triangle mesh

Non-regular space subdivision

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



Ray tracing acceleration in practice

- High RT performance is a major engineering task
  - most common to rely on external libraries
  - Intel Embree: CPU library optimized for Intel processors
  - NVIDIA RTX: hardware accelerated ray tracing for recent generation GPUs
  - ray tracing is moving into graphics APIs
- Fastest current systems:
  - CPU: tens to hundreds of megarays / sec
  - GPU: several gigarays / sec
  - 1 gigaray / 60 frames / 1M pixels ≈ 16 rays/pixel/frame