Spatial Subdivision for Ray tracing

Zoran Popović

Simple Ray Tracing is Slow

- *each* ray is tested for intersection with *each* object
- with many rays in a complex scene this becomes computationally prohibitive
- possible ways to speed it up:
    - fewer intersections by way of:
        - space subdivision
        - bounding volume hierarchy
    - fewer rays
    - generalized rays

Subdividing Space

- Spatial subdivision is an ubiquitous problem in computer graphics: ray tracing, illumination, hidden surface, collision detection, physically-based simulation
- Intersection between computational geometry and computer graphics
- Main idea: Subdivide space into more manageable chunks so that operations we want to perform on it are fast(er).
- Type of queries dictate the chosen data structure and algorithms. Ray tracing query is: find all objects that can possibly intersect with the ray .
- The space chunks are often referred to as voxels (from “volume elements”).

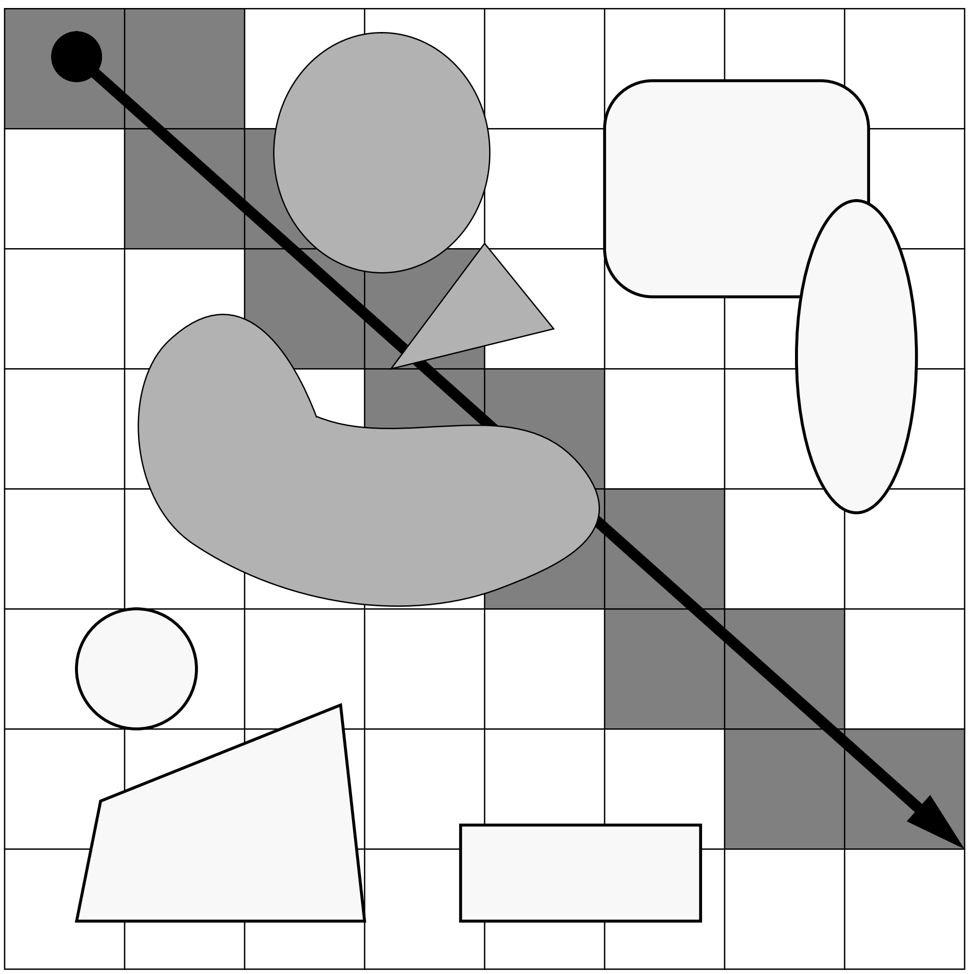
General Space Subdivision Algorithm

```
compute the voxel V containing ray.origin
repeat
   for each object in voxel V
      intersect(ray, object)
   if there was no intersection
      set V to a next voxel along ray.direction
until intersection is found or V is outside the
   voxel array
```

- Methods differ in how they implement two methods:
  - find voxel containing a point
  - given a voxel and a ray find next voxel pierced by a ray
- Algorithms can be roughly divided based on what data structures they use:
  - uniform
  - non-uniform

Uniform Spatial Subdivision

- 3-D grid of identical voxels. Uniformity of voxels results in:
    - subdivision independent of the environment (**bad**)
    - voxels pierced by rays are easily accessed by incremental calculation (**good**)
    - line rasterization in 3-D determines the voxels the ray goes through



Uniform Subdivision Caveats

- voxels quickly take up all available memory , often with many empty voxels
  - allocate only voxels that are full, and use voxel lookup-scheme similar to hashing
- difficult to find the optimal voxel size: small voxels result in too many voxels to traverse; large voxels leaves too many objects within a single voxel
  - precompute a spatial distribution histogram
- same object can be tested for intersection in different voxels
  - keep a "mailbox" for each object where the unique id of each ray is stored
- ray intersects an object but not within the current cell — could result in an error
  - accept only intersections that occur within the current voxel

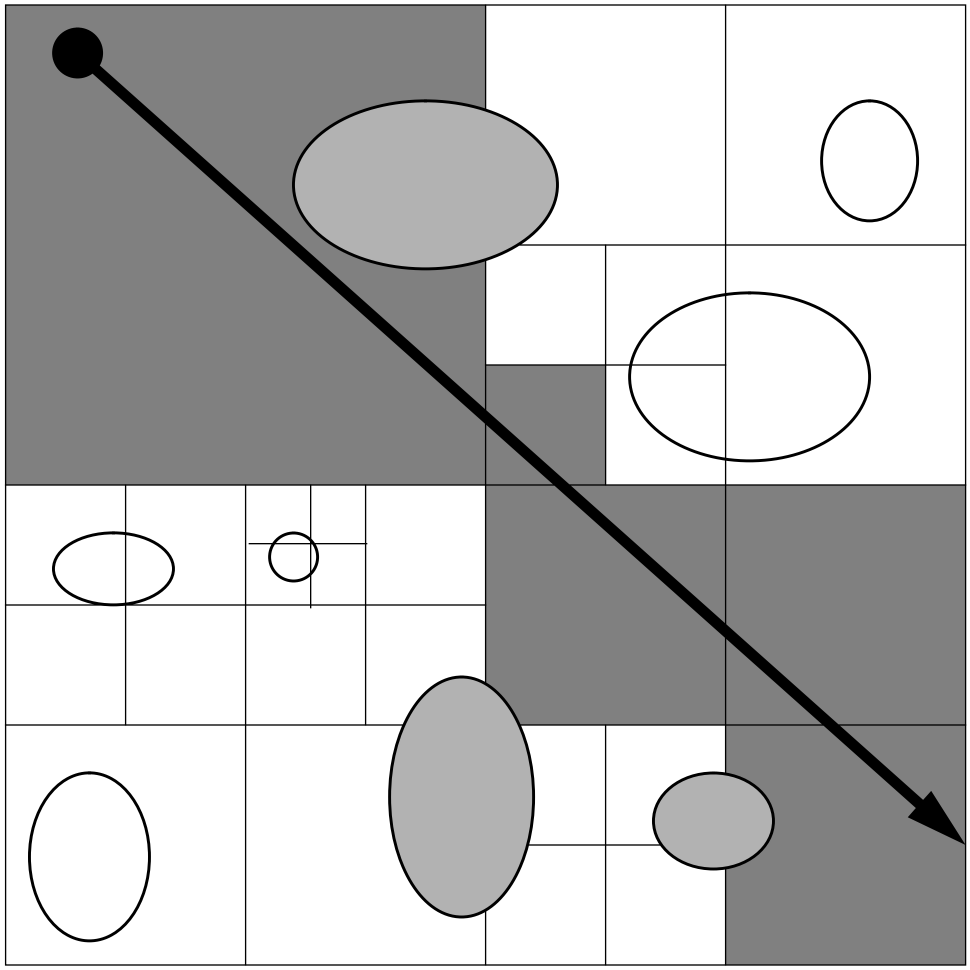
Non-uniform Spatial Subdivision

- Basic idea: need more voxels in the densely populated area and fewer in the relatively empty regions
- Computational geometry offers a number of possibilities: octrees, bsp trees, k-d trees
- Non-uniformity makes it much harder to:
  - locate a voxel which contains a given point
  - locate the next voxel that a ray pierces

Octrees

- at each level of the hierarchy a voxel is subdivided into 8 smaller voxels
- subdivide a voxel if it contains more than a specified number of elements
- do not subdivide if: voxel fewer than some number of elements, or voxel is too small (smaller than the threshold size)
- to find a voxel that contains a given point:
    - beginning at the root node determine the octant with the point and recurse
    - stop when we reach the leaf node
- needs 8 pointers at each node. Alternative:
    - assign each voxel octary number id that encodes the path to it.
    - to recurse construct the name of the child by appending the octant digit and consult the hash to find the child. Leaf is reached when constructed id is non-existent.
    - neighbor is found by searching for a point which is in the neighboring voxel

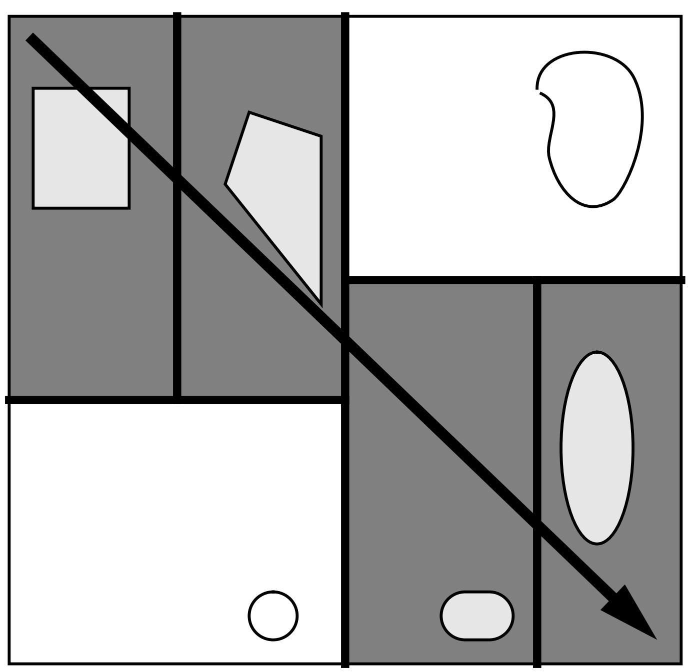
Octrees



- Disadvantages:
  - finding neighbors involves tree traversal each time
  - performance degrades if objects are near the center of the subdivided space

k-D trees (Orthogonal BSP Trees)

- same idea as octrees, only use binary trees instead
- use planes to subdivide space: nodes in the tree are space separators not regions
- still need to traverse up the tree to find neighbors
- recursive traversal offers an alternative:
  - at each node intersect a ray with the plane and traverse down the “closer” half-space
  - recurse into the “farther” half-space only if no intersection found in the “closer” half-space



Bounding Volumes

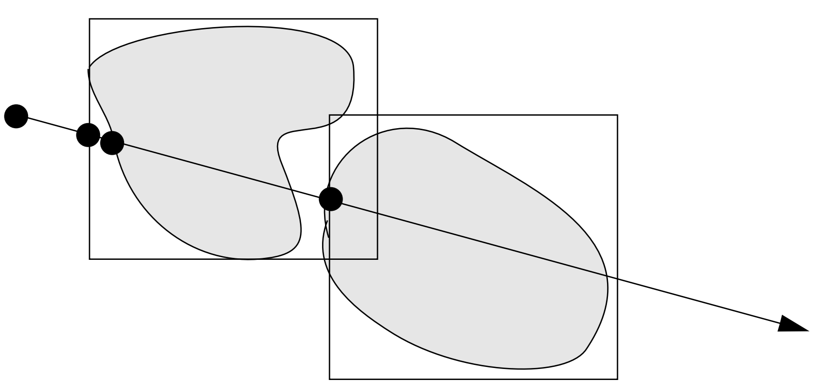
- simple primitives that contain the object and can quickly intersect with a ray:
  - spheres
  - axis aligned boxes
  - boxes
  - CSGs
  - intersection of half spaces
- necessary but not sufficient condition for intersection
- actually increase the intersection time, but the hope is that many will fail
- simple list of bounding volumes would decrease intersection time only by a constant

Hierarchical Bounding Volumes

- each bounding volume contains a set of smaller bounding volumes (list -> tree)
- tree is constructed bottom up from primitives rather than top down as in spatial subdivision

```
intersect_bvh(ray, node)
{
    if node is a leaf
        intersect(ray, node.object)
    else if intersect_bv(ray, node.bv)
        for each child of node
            intersect_bvh(ray, child)
}
```

- distance interval clipping further prunes the search

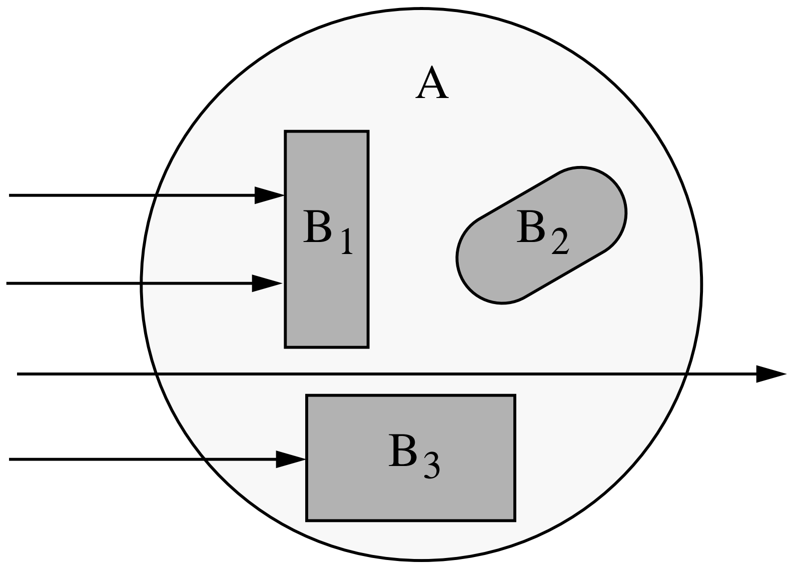


Effectiveness of a Bounding Volume

- Effectiveness depends on:
    - the external cost of intersecting with the bounding volume — EC(V)
    - and the internal cost of intersecting with an object — IC(V)
- metric of effectiveness is **projected void area**
    - difference in the projected areas of the bounding volume and the enclosed object
    - for a convex body **average** projected void area is a quarter of its surface area
- Cost(V) = n EC(V) + m IC(V)
- we can reduce EC(V) and m

Effectiveness of a Hierarchy

- determine when it is feasible to create a parent bounding volume
- examine how the child bounding volume affects the parent



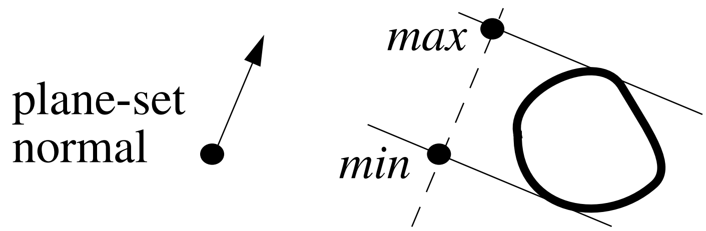
- Pr(ray hits B | ray hits A) = S(B) / S(A)
- $IC(A) = \sum_i \left(EC(Bi) + \frac{S(Bi)}{S(A)} IC(Bi) \right)$
- assumptions: proper nesting, convexity ,
randomness of rays, occlusion, EC is constant

Choosing the Right Hierarchy

- construction tree
  - large branching factors
- interactive tree construction
  - we’re lazy
- incremental insertion
  - start and the root and select a subtree which incurs the smallest increase in surface area if the new object were to become the child of it
  - continue until the leaf is reached and insert the object
  - order of insertion is crucial
    - sorting along a line produces poor results
    - random shuffling is best

Approximate Convex Hulls

- construct bounding volumes with a collection of slabs that form a convex hull
- a slab is a set of two parallel planes defined by a plane-set normal N and an interval (min, max)



- ray intersect algorithm:
  - compute intervals where ray intersects the planes
  - if intersection of those intervals is empty -> no intersection
- rotation affects performance (**bad**)
- common subexpressions for each plane set (**good**)
- to create a parent bounding hull, find *min* and *max* of the slab intervals of the corresponding plane-sets
- distance interval pruning is particularly effective since objects can be sorted for each ray with respect to the first plane intersection

Tradeoffs

<table>
<tr><th>Method</th><th colspan="2">Disadvantages</th><th>Advantages</th></tr>
<tr><td>Grid</td><td colspan="2">poor on non-uniform object distribution,
a memory hog</td><td>fast neighbor-finding,
simple to implement</td></tr>
<tr><td>Octree</td><td rowspan="2">hard to implement,
neighbor-finding
requires tree-traversal</td><td>sensitive to object
arrangement,
large branching factor (8)</td><td>fast tree construction</td></tr>
<tr><td>k-D tree,
BSP tree</td><td>constructing a balanced
tree is hard</td><td>on average faster than Octrees,
recursive traversal is possible</td></tr>
<tr><td>Hier. BV</td><td colspan="2">hard to find tight BVs and optimal hierarchy,
extra BV check for each object</td><td>simple to implement in most cases,
independent of spatial distribution</td></tr>
</table>