Ray Tracing with Spatial Hierarchies

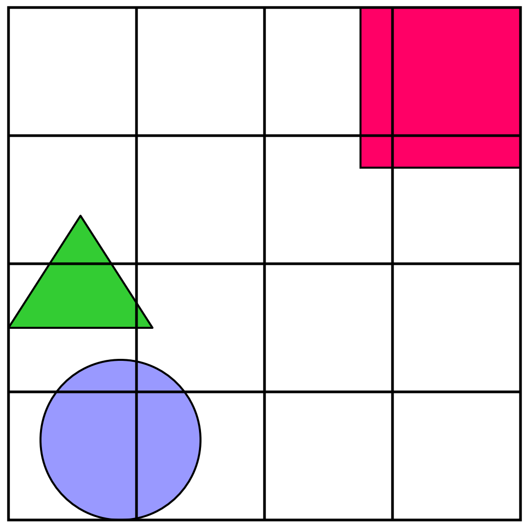
Jeff Mahovsky & Brian Wyvill

CSC 305

Ray Tracing

- Flexible, accurate, high-quality rendering
- Slow
- Simplest ray tracer:
  - Test every ray against every object in the scene
  - N objects, M rays → O(N * M)
- Using an *acceleration scheme*:
  - Acceleration scheme = sub-linear complexity of N
  - Grids and hierarchies
  - N objects, M rays → O(log(N) * M)
  - Log(N) is a theoretical estimate, in reality it depends on the scene
  - Speedups of over 100x for complex scenes are possible

Uniform Grid



- Ray steps through the grid and is tested against objects in the grid cells along the path of the ray
- Can avoid testing the vast majority of the objects for each ray
- Grid traversal overhead can negate savings…

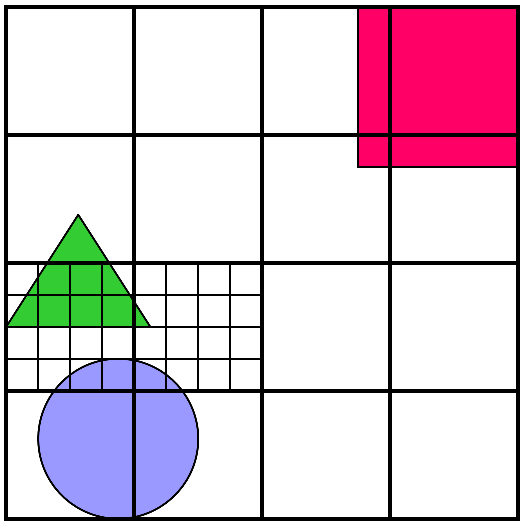
Uniform Grid: problems

- Grid does not adapt to empty space and local complexity
  - Works best for uniformly distributed objects (seldom happens in reality)
  - Typical scenes have areas of complex geometry with empty space between them
- Empty space:
  - Time is wasted tracing the ray through empty grid cells
- Local complexity:
  - Too many objects in each grid cell
  - Could increase grid resolution, but that makes the empty space problem worse
- Difficult to choose optimal grid resolution that minimizes rendering time: tradeoff between these two problems
- Despite this, a grid is still much better than nothing

Hierarchies

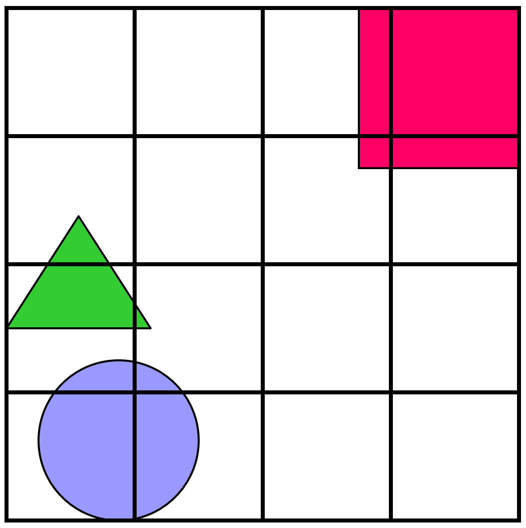
- Need a scheme that adapts to the distribution of objects in the scene
- Build a hierarchy or spatial tree
- The scene is recursively subdivided into nodes that enclose space and objects
  - Empty space is not subdivided
  - Complex areas are subdivided
  - Subdivide until criteria is met: e.g.
    - Number of objects in the node is below a certain threshold (4-8 works well)
    - Tree depth reaches a specified maximum
- Solves both empty space and local complexity problems
- Represented as a tree data structure in memory
- Examples…

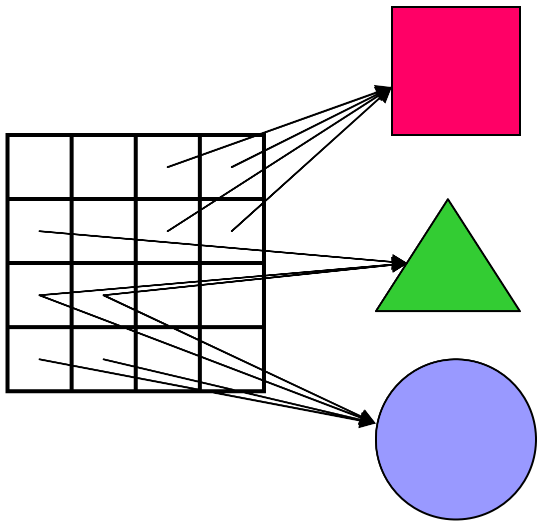
Hierarchy of Grids



- Grid cells/nodes may be empty, contain objects, or contain another grid (e.g. if a cell contains more than 1 object)
- An object may span multiple nodes or grid cells

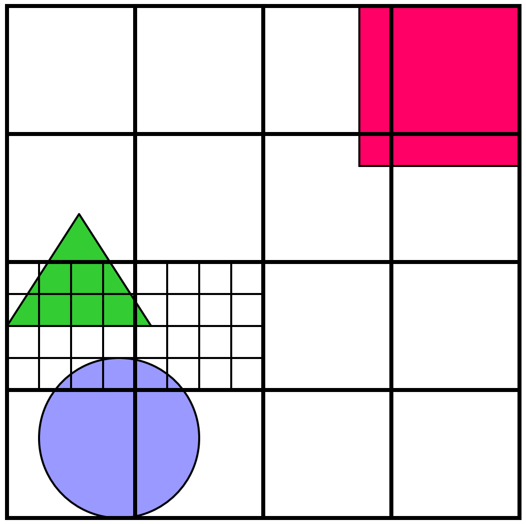
Hierarchy of Grids 2

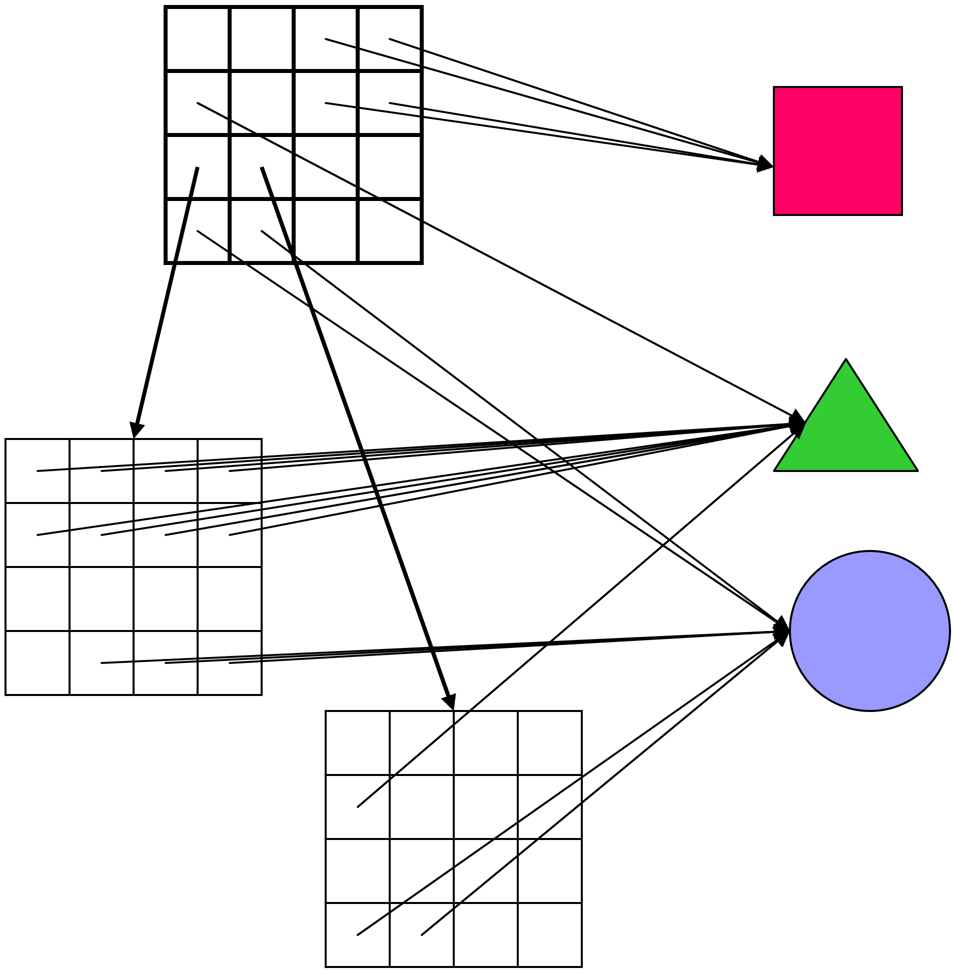

Scene


Data structure

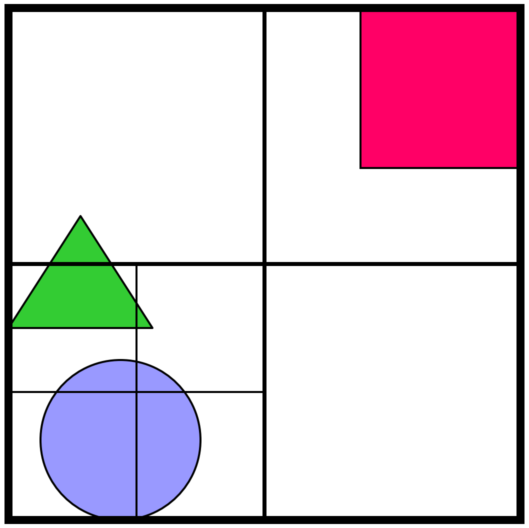
- Start with a normal grid just large enough to enclose all the objects
- Cells contain pointers to the objects
- Cells with more than a certain number of objects (2 in the example) are subdivided into another grid

Hierarchy of Grids 3



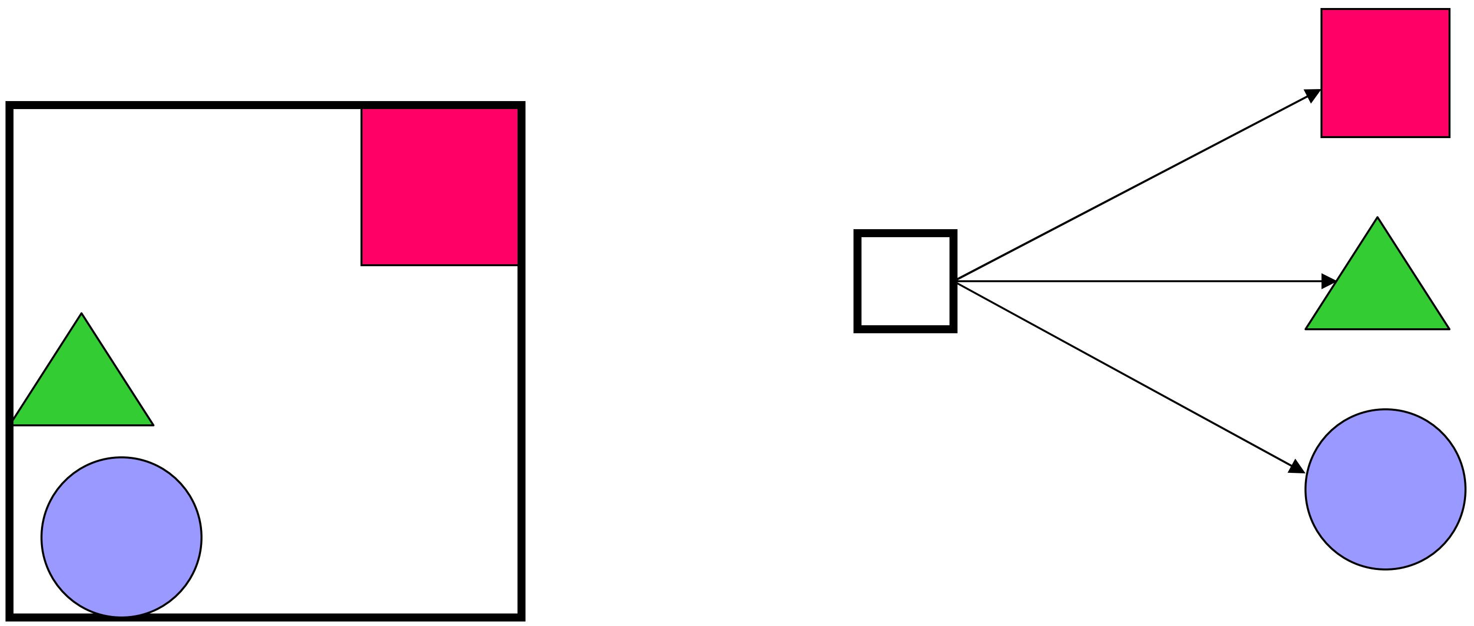


Octree



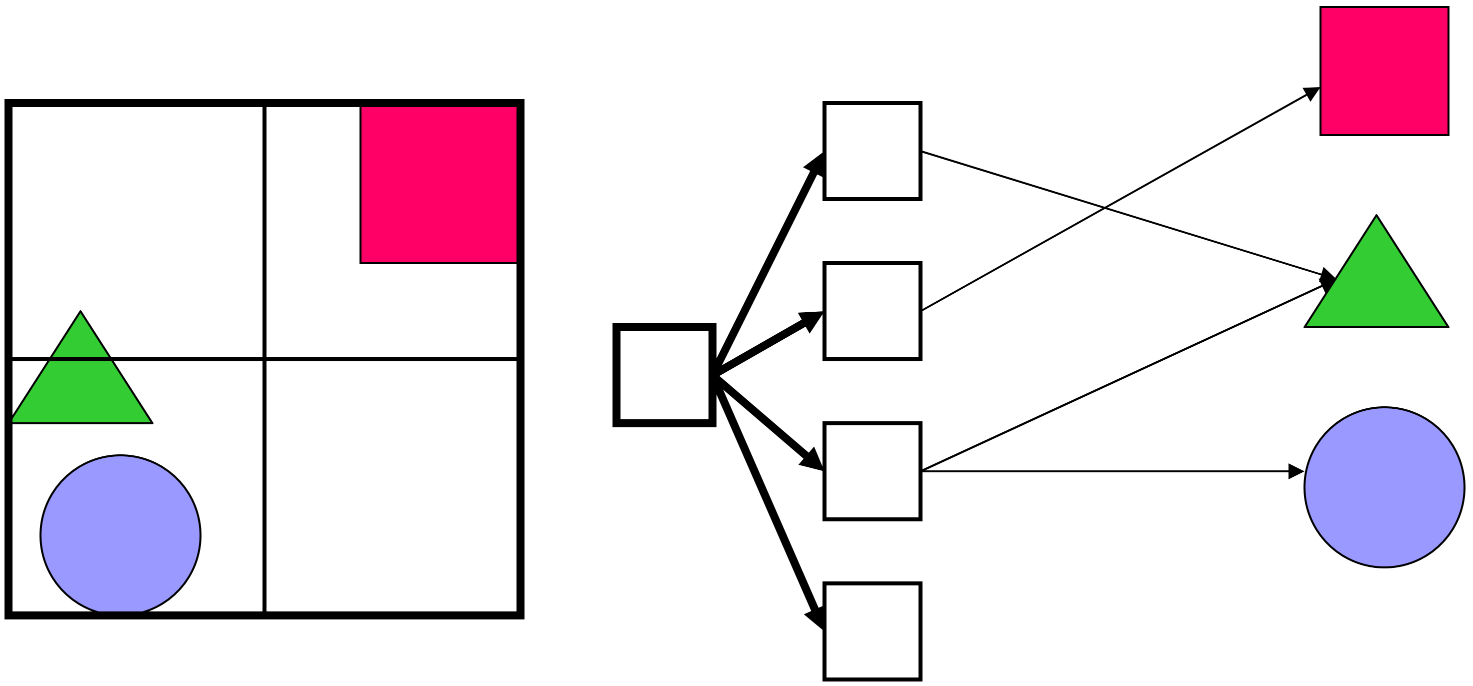
- 3D space is recursively subdivided into 8 (oct) equal boxes/cubes (child nodes)
- Example is a 2D quadtree that subdivides if node has >1 object
- A node’s child nodes may not overlap in space
- An object may span multiple nodes
- Technically a special case of a Hierarchy of Grids where the sub-grids are 2x2x2 cells

Octree 2



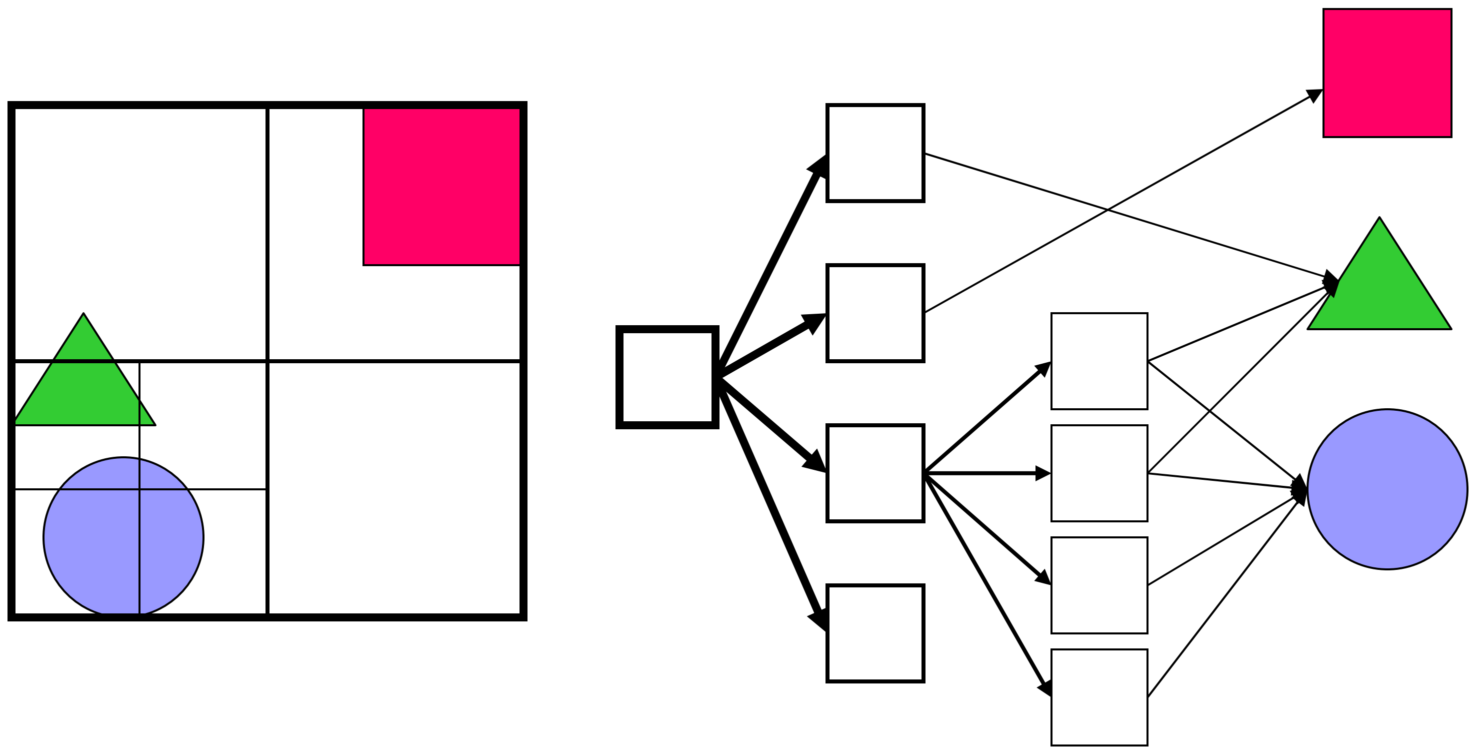
- Start with all the objects in the root node
- The root node's size is the minimum bounding cube that encloses all of the objects

Octree 3



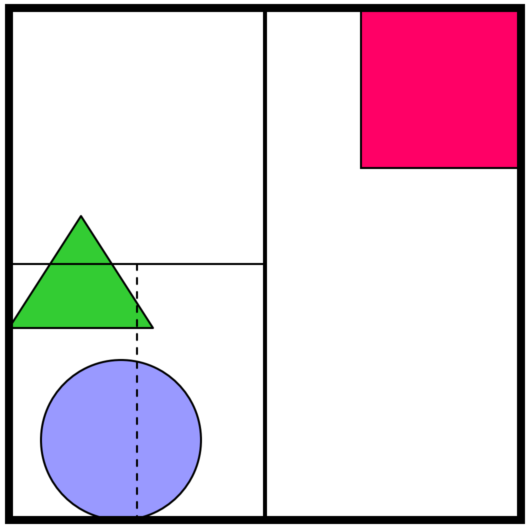
- Since the root node contains more than 1 object, it is subdivided

Octree 4



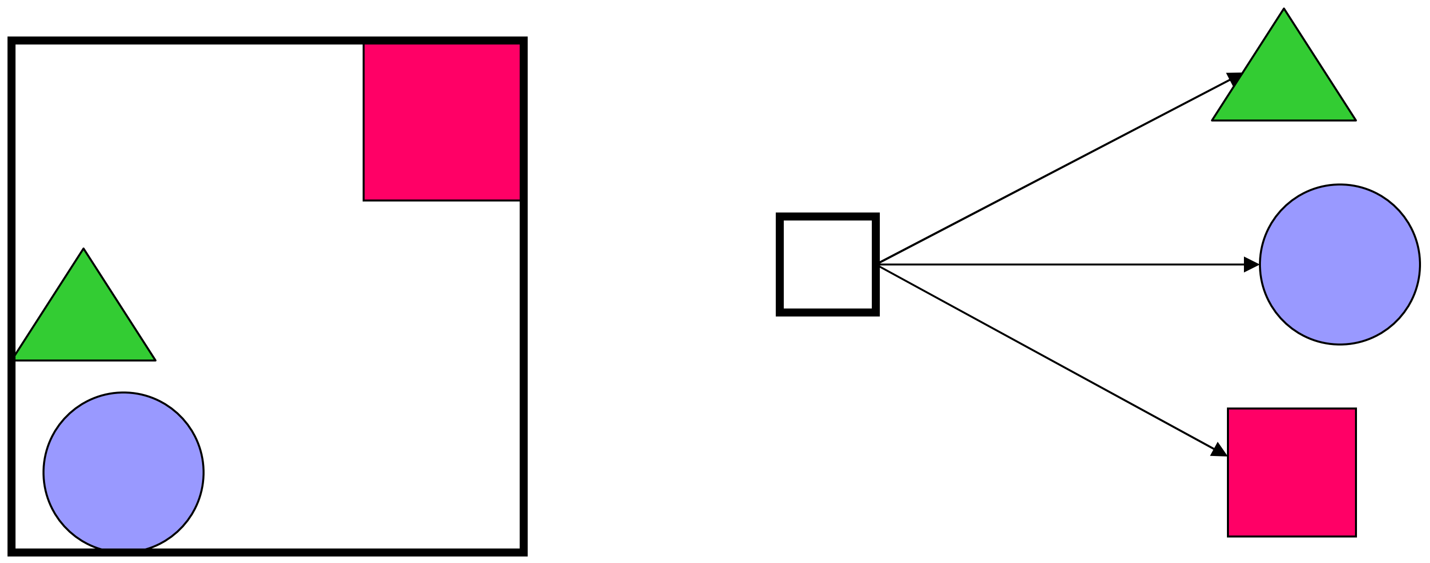
- One child has more than 1 object, so it is subdivided
- Still need to keep going…

Binary Space Partitioning (BSP) Tree



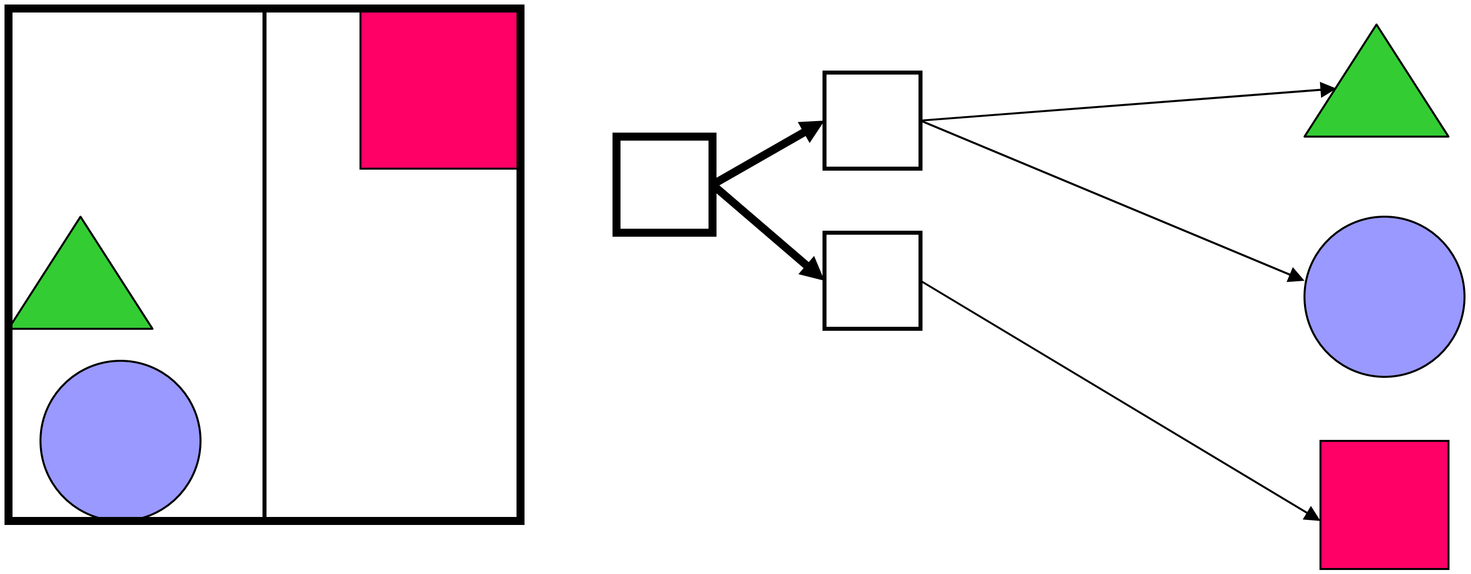
- Similar to an octree, but divides nodes into two children instead of eight
- Alternates splitting along the x, y, and z axes
- Example is a 2D BSP that subdivides if a node has > 1 object
- Preferred over octrees because tracing a ray through a BSP tree is simpler

Binary Space Partitioning (BSP) Tree 2



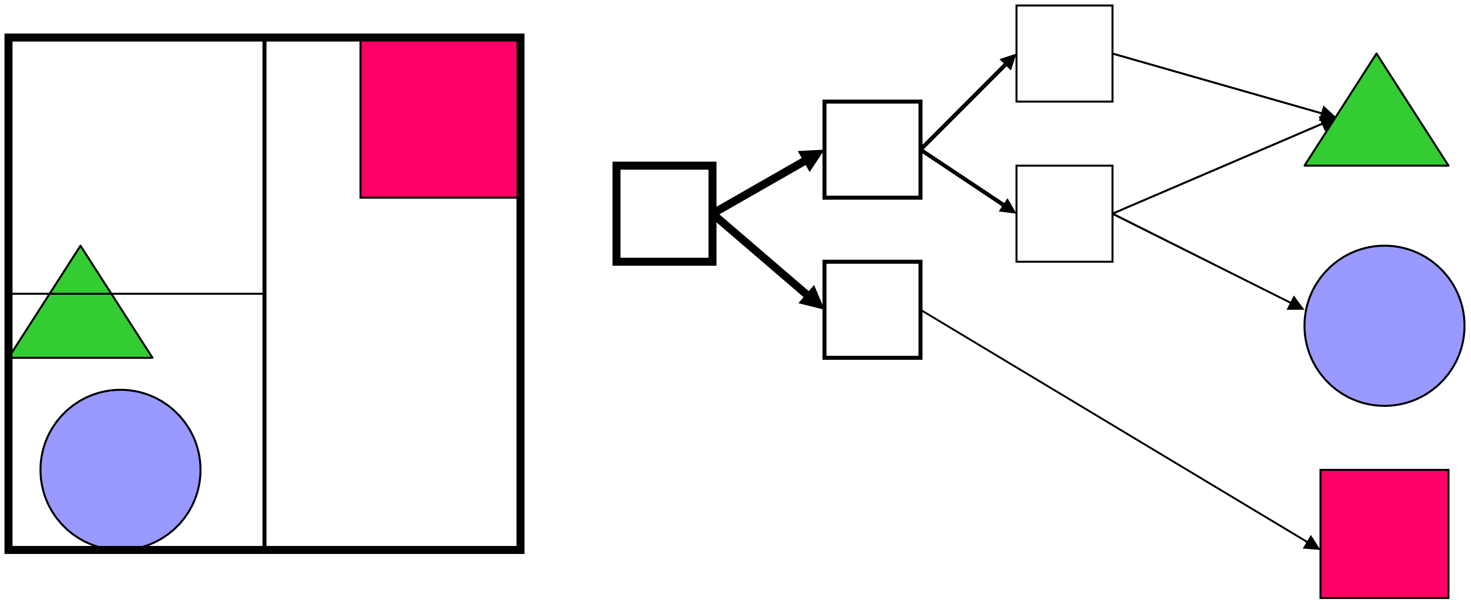
- Start with all the objects in the root node
- Root node's size is the minimum bounding cube or box that encloses all the objects
- Root node contains more than 1 object so it is subdivided in the x-axis

Binary Space Partitioning (BSP) Tree 3



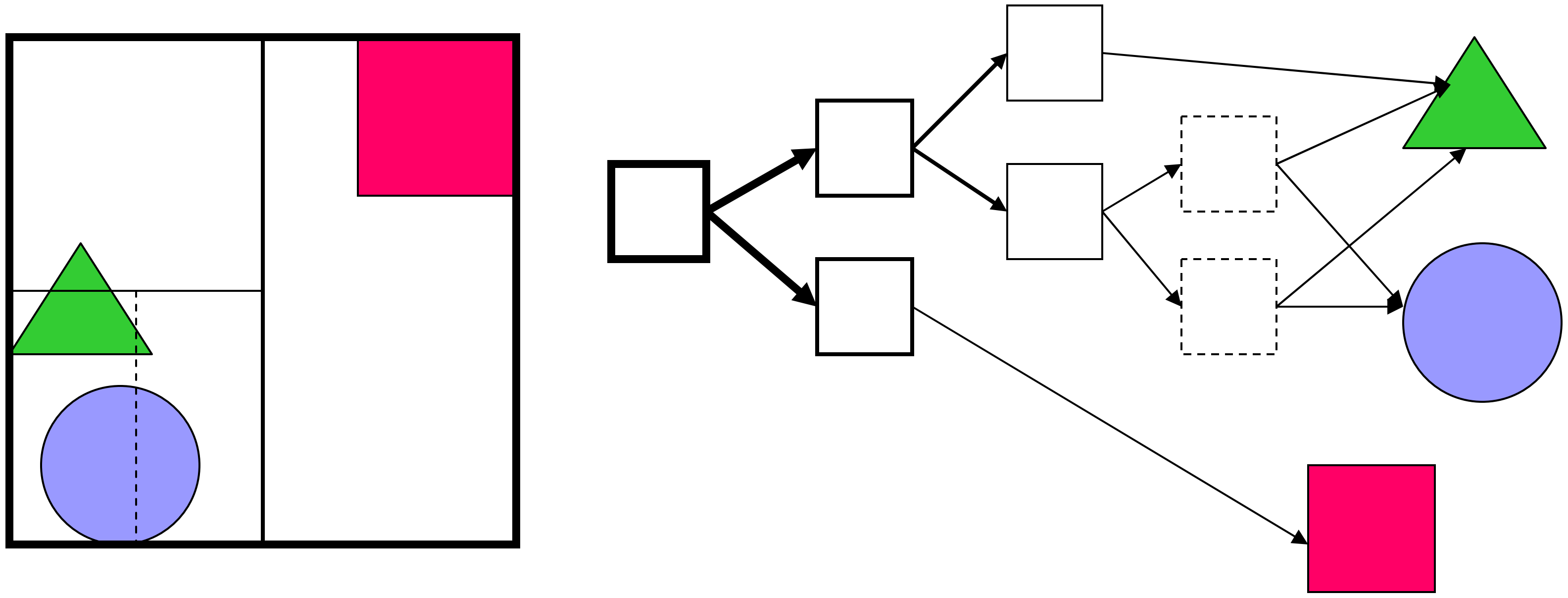
- One child contains more than 1 object so it is subdivided in the y-axis

Binary Space Partitioning (BSP) Tree 4



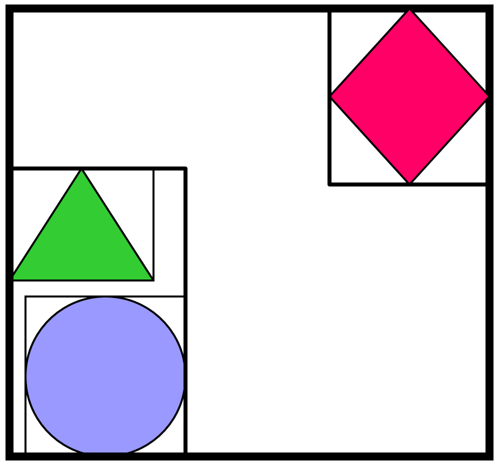
- One child contains more than 1 object so it is subdivided in the x-axis

Binary Space Partitioning (BSP) Tree 5



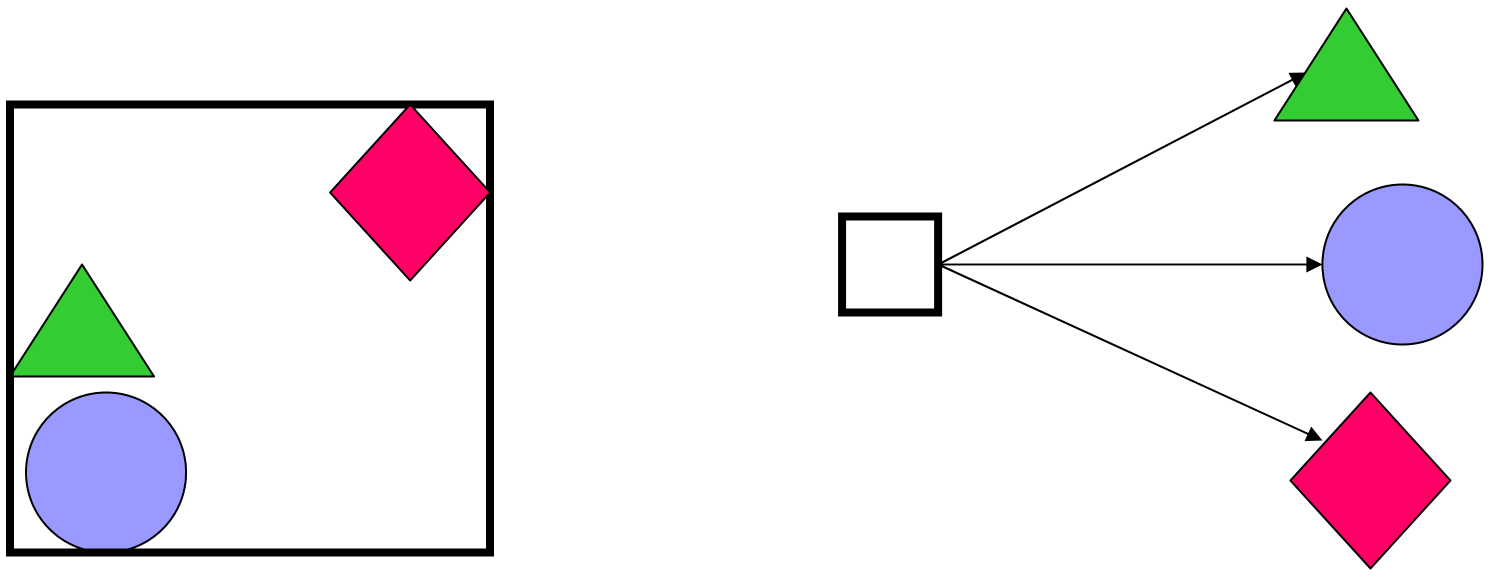
- That didn't help at all! (need to keep going or try a different splitting plane)
- In some situations you can keep subdividing infinitely and never have fewer than 2 objects (also happens with octrees and grid hierarchies)
  - The higher the maximum number of objects per node, the less chance of runaway subdivision (I use a maximum of 8, which seems to work well.)
  - Stop runaway subdivision by having a maximum tree depth or other heuristic that detects it

Bounding Volume Hierarchy (BVH)



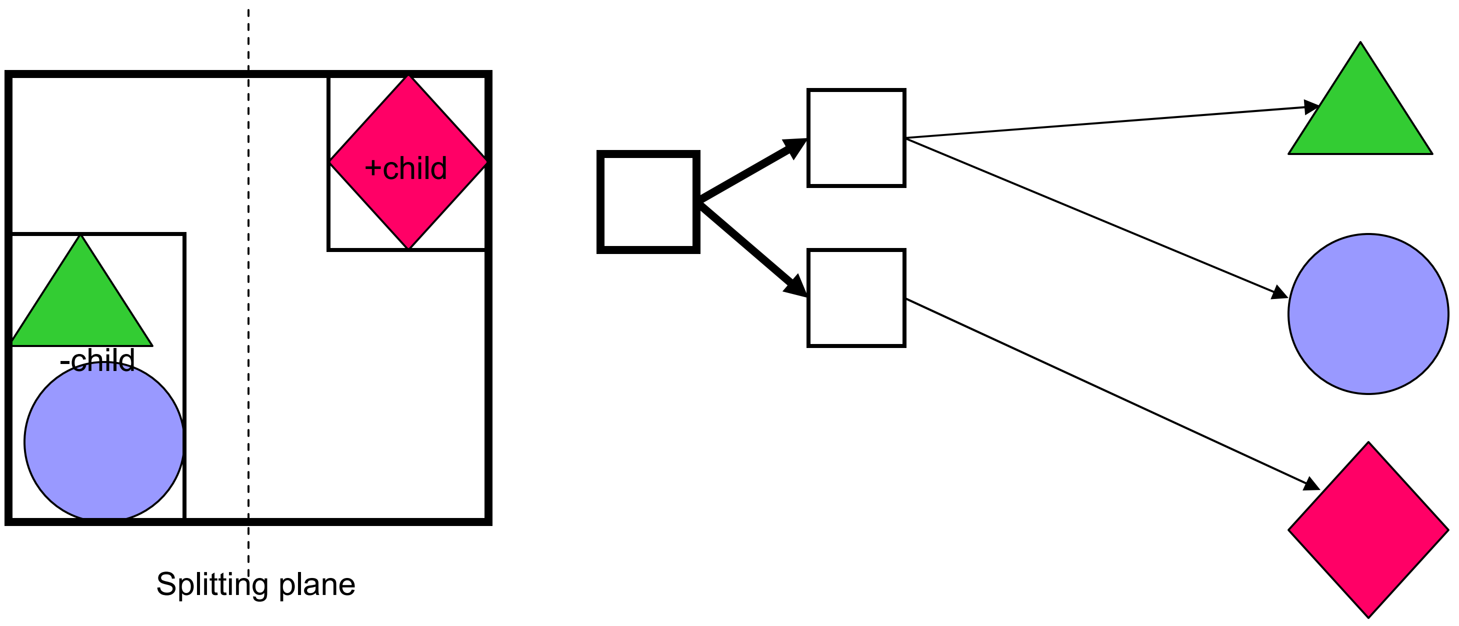
- Subdivides the set of objects instead of subdividing 3D space (opposite of the others)
- Compare the objects' center points to the splitting plane
  - Those on + side belong to one child, those on the – side belong to the other
  - The two children are then each 'shrink wrapped' with a minimum bounding volume
- Alternates splitting nodes in half along the x, y, and z axes (like the BSP tree)
- A node's children MAY overlap each other and the splitting plane that created them
- Objects MAY NOT span multiple nodes

Bounding Volume Hierarchy (BVH) 2



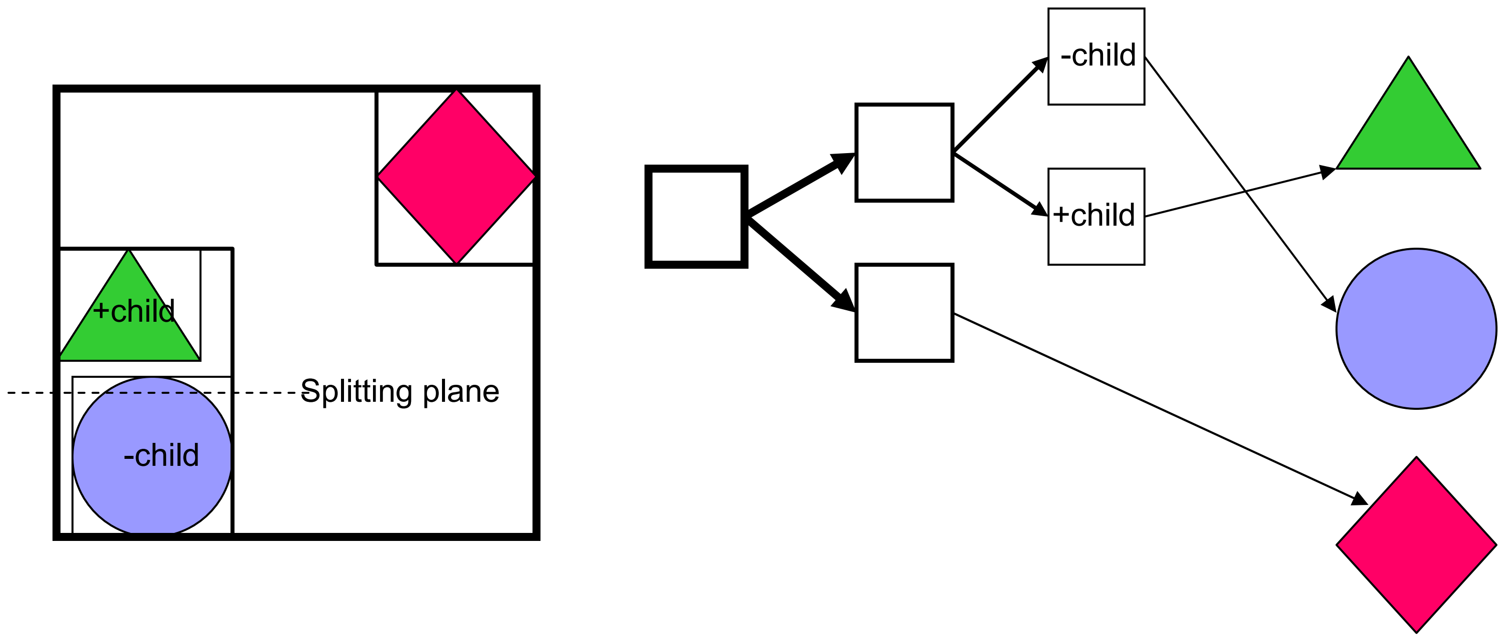
- Start with all the objects in the root node
- The root node’s size is the minimum bounding volume that encloses all of the objects

Bounding Volume Hierarchy (BVH) 3



- Split the root node's bounding volume in half and compare the object center points to the splitting plane, giving two sets of objects
- Two children are created (-child and +child), one for each set of objects
- The dimensions of the child nodes are the minimum bounding boxes that enclose each child's objects

Bounding Volume Hierarchy (BVH) 4



- The node with two objects is split in half and the process repeats
- Note that the –child’s bounding box overlaps the splitting plane: this is OK
- The children may overlap too, this is also OK

Which is better?

- Performance partially depends on the scene properties and distribution of objects
- Most widely used (recently) are the BSP tree and the BVH
  - Simplest to implement
  - Good Performance
- Hierarchy of Grids is not well-used
  - Fairly complicated to implement
  - Similar performance to the simpler methods
- Octrees were extensively used in the past, but not so much today
  - Somewhat complicated to implement
  - Similar performance to the simpler methods
- BVHs tend to need fewer nodes to get the same performance

- (In my experience) BSP trees have the advantage in ray tracing speed for complex scenes, but BVHs take less time/computation to construct
  - BVH construction is simpler and faster because an object can only belong to one hierarchy node

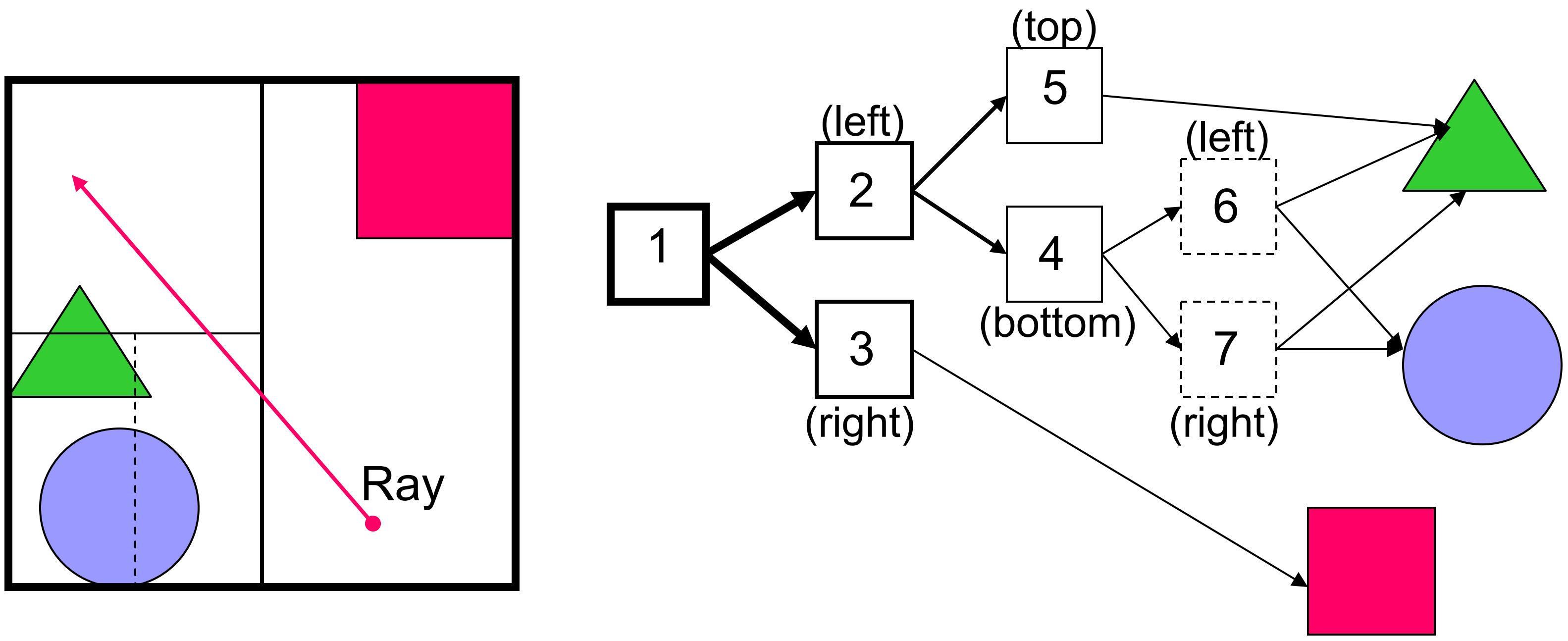
Tracing Rays through a Hierarchy

- Tracing a ray through a hierarchy is a depth-first tree search
- Start at the root node of the hierarchy, and test the ray against the children
  - Fundamentally, this is a ray-box overlap test
  - The box is the region of space enclosed by the child
  - A full ray-box test is only necessary for the BVH
    - Other schemes have optimized traversal algorithms that can take shortcuts
  - Only the BVH requires complete box coordinates to be stored with each node
    - Others can get away without storing any coordinates because the traversal algorithms don't need them or can figure them out
- If the ray overlaps a child, test its children, and so forth
- Only leaf nodes contain objects
- When a leaf node is encountered, test the objects within the node for intersection with the ray

Tracing Rays through a Hierarchy 2

- With Hierarchy of Grids, Octrees and BSP trees, the ray *always* traverses the child nodes in the order of their distance along the ray
  - This order is easy to determine because the children don't overlap
  - For each node, determine the closest child and traverse it first
  - If an object is hit within a node, the traversal is finished because the rest of the nodes that would have been tested are further away than the intersection point
  - The intersection point MUST be within the node, or it is ignored (remember that an object may overlap multiple nodes, and the intersection point may lie OUTSIDE the current node and inside a different one!)
  - This often results in the same object being tested against the ray multiple times
  - This problem also happens with uniform grids

Tracing Rays through a Hierarchy 3



- BSP tree example: Traversal order is 1, 3 (and test red square), 2, 4, 7 (and test triangle and circle), 5 (and test triangle)
- Node 6 is not hit by the ray and is ignored
- Remember that this is a depth-first tree search, and always traverse the closest child first

Tracing Rays through a Hierarchy 4

- In an BVH, the children often overlap (sometimes in strange ways)
  - Always need to test both children, even if a ray-object intersection happens in one (because they can overlap each other)
  - Still try to test them in order of distance along the ray
    - Compute actual distances to the children's bounding boxes (can be expensive)
    - Use a heuristic that's simple but correct most of the time
  - Don't need to check if a ray-object intersection point is within the current node, because it is guaranteed to be (objects do not overlap nodes)
  - Duplicate ray-object tests are avoided because objects do not span multiple nodes
  - Need to trim the length of the ray as objects are hit, or unnecessary traversal will occur

Tracing Groups of Rays through a Hierarchy

- Can nearly double performance by tracing groups of rays (e.g. 64 at a time) instead of individual rays
- Same hierarchy traversal rules apply, except a list of rays is used instead of a single ray
- In modern computers, loads from memory can limit performance, especially if data is not in cache
- Group of rays = small, frequently accessed
  - Resides in CPU cache memory
- Hierarchy = large, many nodes infrequently accessed
  - Mostly in slower main memory
- Nodes and objects within are loaded ONCE for each group of rays, instead of once for each individual ray
  - Can reduce main memory fetches by up to a factor of 64, if using groups of 64 rays

Tracing Groups of Rays through a Hierarchy 2

- Implement group/list of rays as a stack of pointers to rays
- Rays are tested against a child node’s bounding box, and those rays that overlap the child node have their pointers pushed onto the stack
- The child node is traversed using these stacked ray pointers
- The pointers are popped off the stack after the traversal returns

- Can also use Pentium 4 SSE or G4/G5 Altivec vector operations to test 4 rays at a time against a child node, for a maximum 4x speedup

Speeding Up Ray Tracing

In the naive algorithm each ray has to be tested against each object.

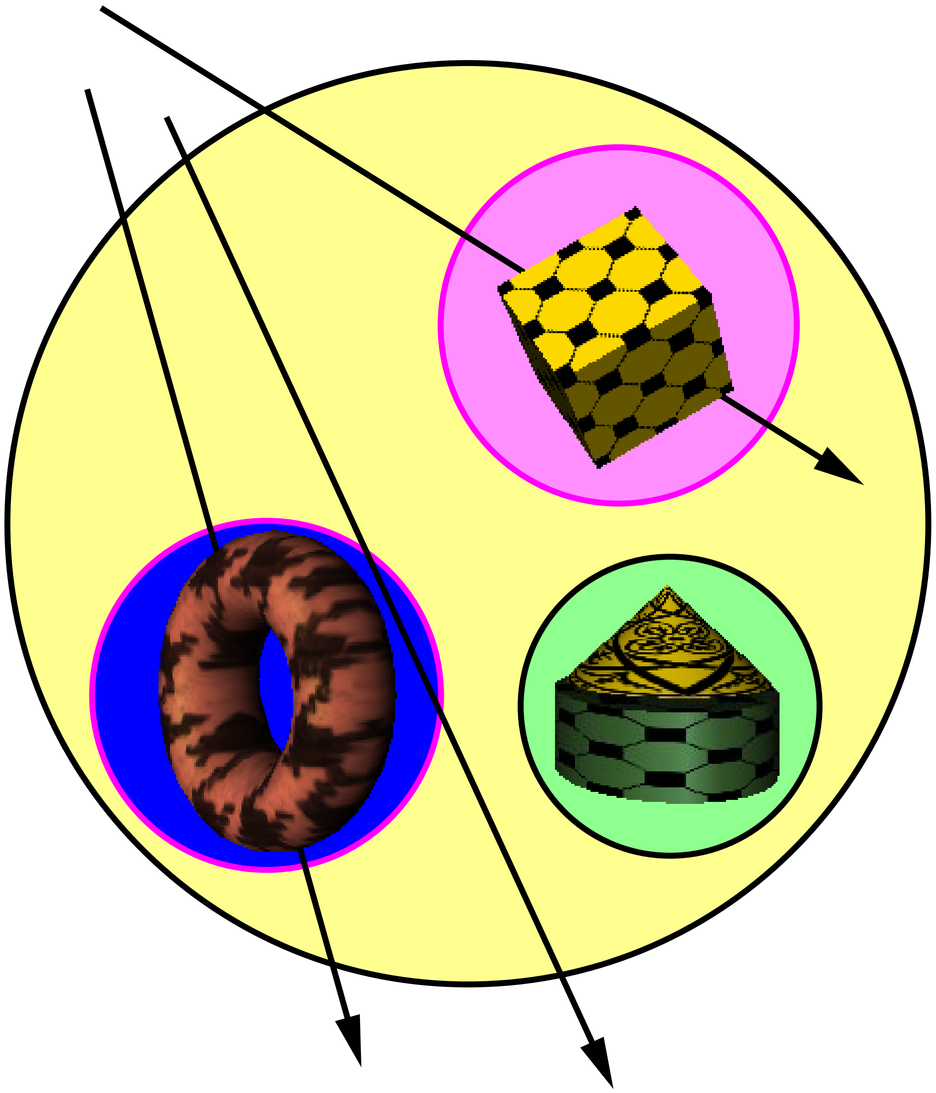
O (m*n)   m rays   and n objects.

Most rays miss most objects.

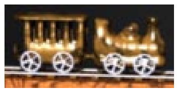
Exploit this:

1. Bounding volumes or boxes on hierarchical groups of objects.

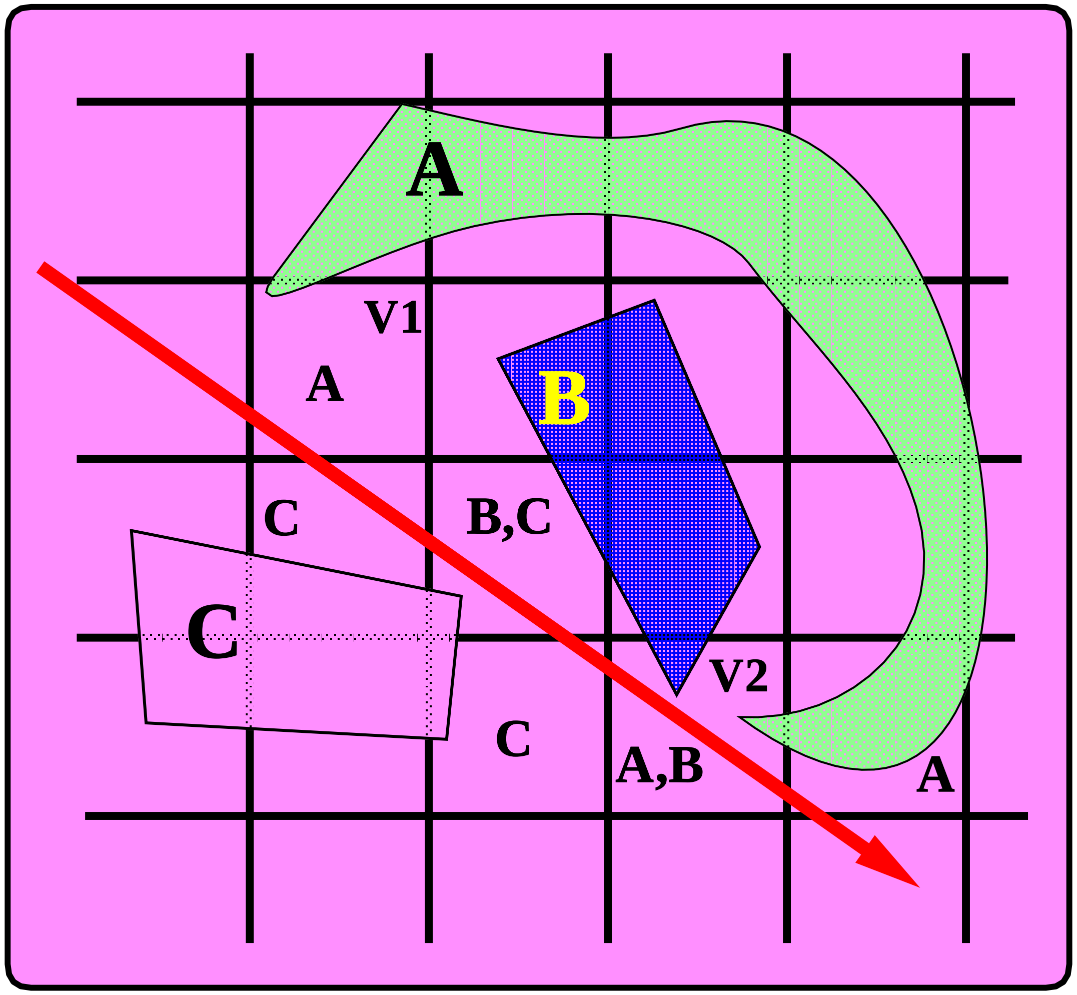
2. Space Sub-division.



Bounding Spheres



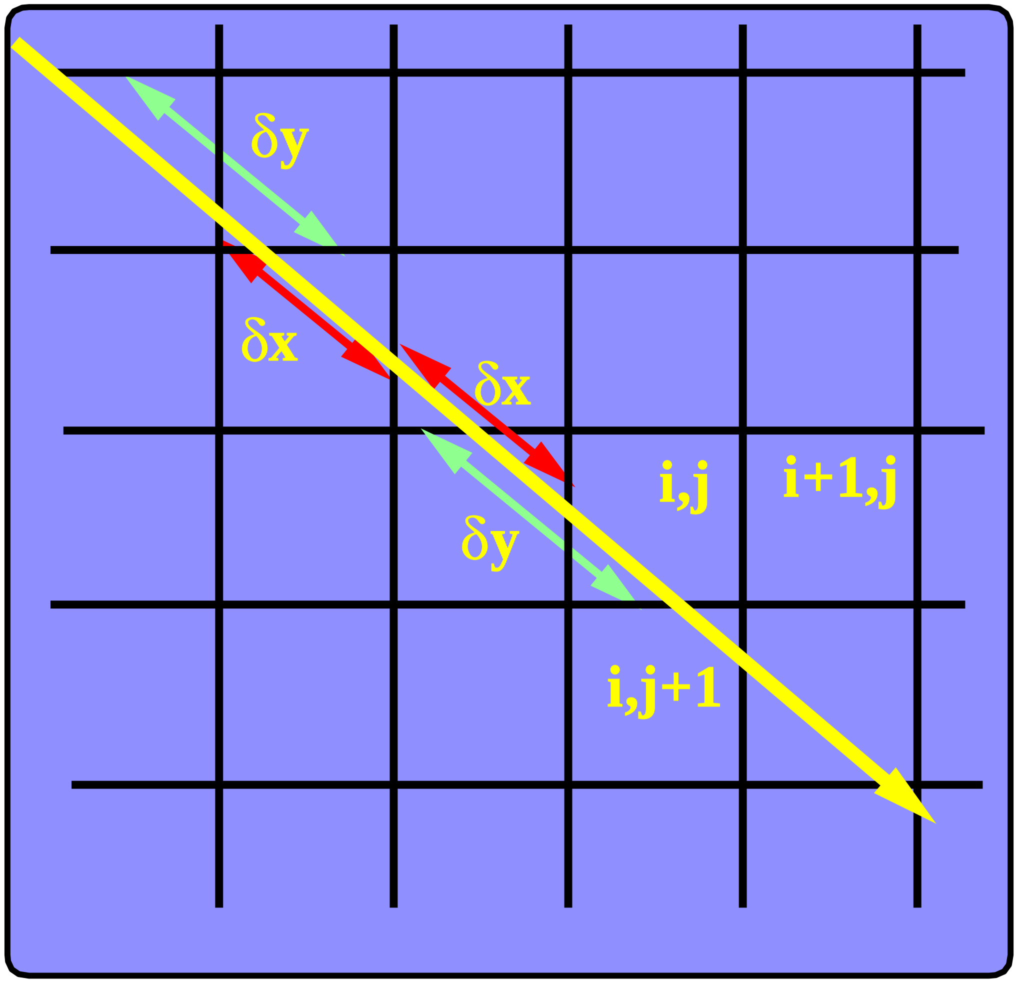
Uniform Space Sub-Division



Each ray is checked against each voxel. In the figure the ray intersection for object A is found in voxel v1.

Each ray has a unique number or signature. This is stored with the object so that when the ray is intersected with A in voxel v2 the intersection information is retrieved and the object intersection test is not repeated.

Uniform Space Sub-Division Voxel traversal (Cleary et al)



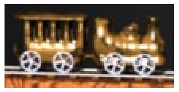
dx is the distance between voxels yz faces

dx and dy record the total distance along the ray.

The Algorithm

dx and dy are the distance from the x and y axis. px,py,pz have values of 1 or -1 depending on direction of ray.

```
repeat
    if dx<=dy {
        i:=i+px
        dx:=dx+δx
    } else {
        j:=j+py
        dy:=dy+δy
    }
Until an intersection is
found in cell i,j
```



Next Voxel Algorithm

Suppose voxels stored as a 3D array n*n*n

voxel[i,j,k]   address p = i*n*n + j*n +k

Multiplications in next voxel loop.

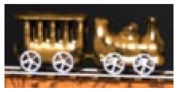
But n*n constant

Each time i is incremented p incemented by + or − n^2

Each time j is incremented p incemented by + or − n

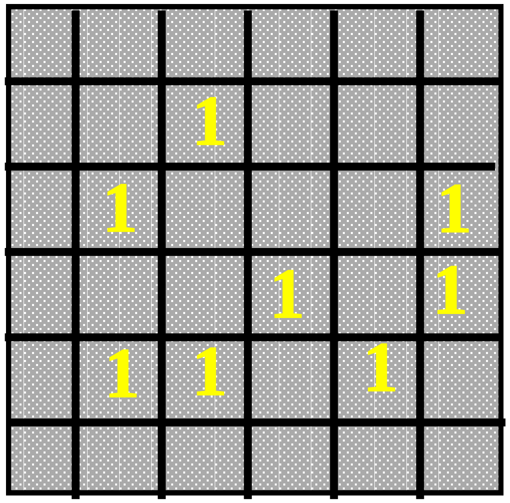
n should be large enough such that most cells are empty

use a hash table instead of 3D array of voxels.
p mod M index into table length M
avoid division by checking p against M at the end of each loop



Termination of Next Voxel Algorithm

Detect when ray leaves bounding volume for scene. Distances to boundary faces are given by sx,sy,sz Compare dx,dy,dz to sx,sy,sz once per loop.



Bit array one bit per voxel indicates if any object overlaps that voxel. Hash table only accessed if bit on.

Voxel Traversal Algorithm in 3D with termination

initialize values of px,py,pz, δx,δy,δz,dx,dy,dz and p

```
repeat
    if (dx<=dy) and (dx<=dz) {
        if dx>=sx exit;
        p:=p+px
        dx:=dx+δx
    } else if (dy<=dx) and (dy<=dz) {
        if dy>=sy exit;
        p:=p+py
        dy:=dy+δy
    } else if (dz<=dy) and (dz<=dx) {
        if dz>=sz exit;
        p:=p+pz
        dz:=dz+dz
    }
    if p>M  p:=p−M
 until an intersection foundin cell with hash key p
```