Space-based Partitioning Data Structures are an efficient way of organizing data that lies in an n-dimensional space:
- 2D, 3D, even 4D and beyond

Reasoning about the data is usually much easier.

Bottom Line: Cut down on the search space.
<table>
<thead>
<tr>
<th>X Values</th>
<th>Y Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
• Quadtrees
  – Octrees
• Binary Space Partitions (BSP)
  – Kd-Trees
• R-Tree
• Bounded Volume Hierarchy (BVH)
Outline

- **Quadtrees**
  - Octrees
- **Binary Space Partitions (BSP)**
  - Kd-Trees
- **R-Tree**
- **Bounded Volume Hierarchy (BVH)**
Quadtree Origins

- The term Quadtree was coined by Raphael Finkel and J.L. Bentley in 1974
- A data structure to store data in two dimensions
- Follows the same intuition as a binary tree
  - Binary Tree: I have one range of data I want to partition
  - Quadtree: I have two ranges of data I want to partition
Quadtree

- Quadtrees represent finite, square areas
  - The internal nodes of a quadtree are also finite, square areas that make up a quadrant of their parent node's region
    - As such, each internal node has four children, one for each quadrant
  - The leaf nodes of a quadtree hold a finite number of elements
- Filled leaf nodes will break into four quadrants and the data is inserted into the quadrants
Quadtree
quadtree_insert(quadtree node, data_type data)

    if(node has quadrants)
        if in first quadrant
            quadtree_insert(node.firstQuad, data)
        if in second quadrant
            quadtree_insert(node.secondQuad, data)
        ...  
    else
        if node.data.size < threshold
            node.data.insert(data)
        else
            break up into quadrants
            insert existing data into quadrants
            insert data into quadrant
Quadtree

- Alternative Implementations
  - Rectangular Quadrants
    - Point Quadtree
- Quadtrees are flexible, implementations can vary to work for many situations
  - Conway's Game of Life
  - Bitmap Image Representation
- Not limited to spacial data
Octree

- Octrees are the 3D analog to Quadtrees
- Octree internal nodes contain 8 octants
  - Aside from more coordinate comparisons, quadtrees and octrees are identical in code
Quadtree/Octree Uses

- Ray tracing
  - Line of Sight
- Hidden Surface Removal
  - Occlusion Culling
- Collision Detection
Outline

- Quadtrees
  - Octrees
- Binary Space Partitions (BSP)
  - Kd-Trees
- R-Tree
- Bounded Volume Hierarchy (BVH)
Binary Space Partition Trees

- Binary Space Partition Trees (BSPs) are binary trees
- Each internal node in a BSP has a partition, oriented any arbitrary direction
  - The two branches of the BSP, “Front” and “Back”
  - All data down either side of the node is guaranteed to be either in front of the partition or behind
```cpp
bsp_tree create_bsp(data_type data[]) {
    node = new bsp_tree;
    if data[].size == 1
        node.data = data
    else
        good_partition = find the partition that cuts data[] into two near equal sets
        node.partition = good_partition
        node.front = create_bsp(data[] in front of partition
        done.back = create_bsp(data[] behind partition
    return node;
}
```
Binary Space Partition Uses

- Rendering Scenes
  - Painter's Algorithm
    - Back to Front Tree Traversal

- Ray Tracing
Back to Front Traversal

```c
traverse_tree(bsp_tree node, point eyepoint)
{
    location = determine which side of the partition eyepoint is on
    if location is in front
        traverse_tree(bsp.back, eyepoint)
        show(node.data)
        traverse_tree(bsp.front, eyepoint)
    else // location in behind or eyepoint is on the partition
        traverse_tree(bsp.front, eyepoint)
        show(node.data)
        traverse_tree(bsp.back, eyepoint)
}
```
Binary Space Partition

- Has now been replaced for graphical applications
  - Only works in 2D/Pseudo-3D spaces
- Historically solved computational barriers
  - BSPs utilized in *Doom* (1993)
    - *Quake* (1996) as well
- Other Applications
  - Image compression
In a paper released by Daniel Gordon and Shuhong Chen in September 1991, the Doom BSP was utilized by John Carmack. It uses walls and other naturally flat objects in the game as the partition. The method does a front to back traversal of the tree, keeping track of what areas of the screen have been drawn to already.
Kd-tree

- A special form of a BSP
  - Instead of arbitrarily directed partitions, Kd-tree partitions are parallel to the x and y axis

- Works just like a binary tree
  - Each internal node has 2 children
  - Each internal node either splits the x axis or the y axis
Kd-tree Construction

KdTreeNode construct_kdtree(int axis, data_type data[])
{
    find the datapoint closest to the average/median value for the given axis as the pivot point
    store the data in the node
    leftNode = construct_kdtree(next axis, data less than pivot point)
    rightNode = construct_kdtree(next axis, data greater than pivot point)
}
Nearest Neighbor Search

1. Traverse down the tree to a leaf node that contains the query point

2. Return current leaf node as best choice

3. When returning up the tree, check to see if a circle centered on the query point with radius equal to the distance to the current best choice will cross the pivoting axis at that node

   If so, following this procedure again down the other branch and compare the result to the current best choice

   Otherwise, continue up the tree continuing to check if another branch should be considered.
Outline

- Quadtrees
  - Octrees
- Binary Space Partitions (BSP)
  - Kd-Trees
- R-Tree
- Bounded Volume Hierarchy (BVH)
R-tree

- Stands for Rectangle-Tree
  - Variation of a B-tree
- Made up of minimum bounding rectangles (MBRs)
  - Unlike quadtrees quadrants, MBRs can overlap
- For large datasets
R–Tree
R-Tree Variations

- **R*-tree**
  - Indexes spatial information more efficiently than a R-tree by minimizing MBR overlap

- **R+ tree**
  - No overlapping MBRs, it will insert an object in multiple nodes if needed

- **Priority R-tree**
  - Better overall R-tree, but more efficient for extreme datasets
Outline

- Quadtrees
  - Octrees
- Binary Space Partitions (BSP)
  - Kd-Trees
- R-Tree
- **Bounded Volume Hierarchy (BVH)**
Bounded Volume Hierarchy

• Similar to a 3D R-Tree
  – Not limited to simple bounding rectangles
• Any volume to surround objects can be utilized
  – Sphere, Cone, etc.
• Tries to maximize “tightness” property of spacial data structures
Querying any of these data structures is only as efficient as how tight the data structure fits the data set

- Minimize the amount of unused space bounded by the bounding shape
- While minimizing the height of the tree
Fig. 4: Six levels of a bounding volume hierarchy
Conclusions

