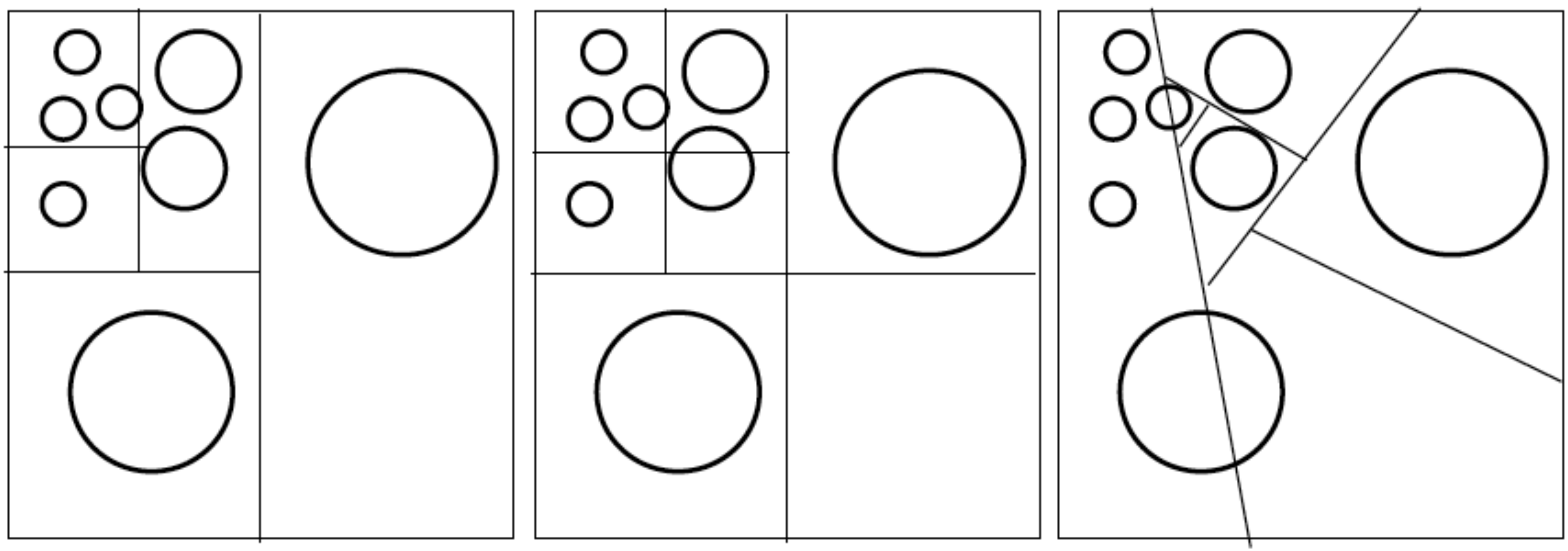
CS 419: Production Rendering

KD-Trees
BSP-Trees

Eric Shaffer

Some content taken from *Physically Based Rendering* by Pharr et al.

Lots of types of spatial hierarchies

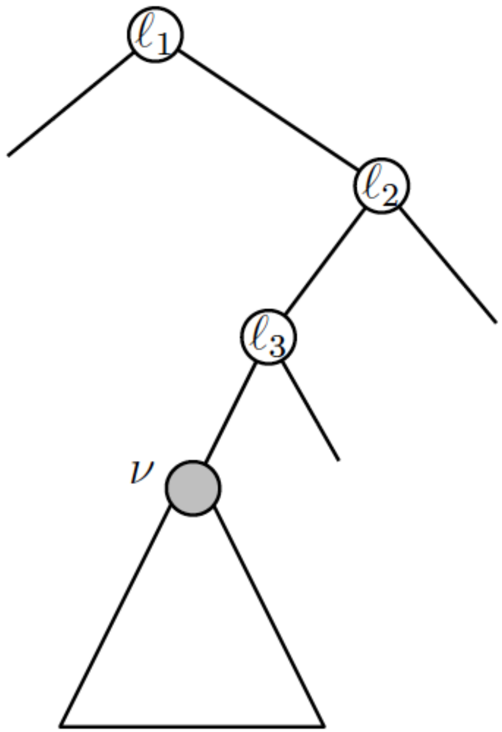


KD-Tree Oct-Tree BSP-Tree

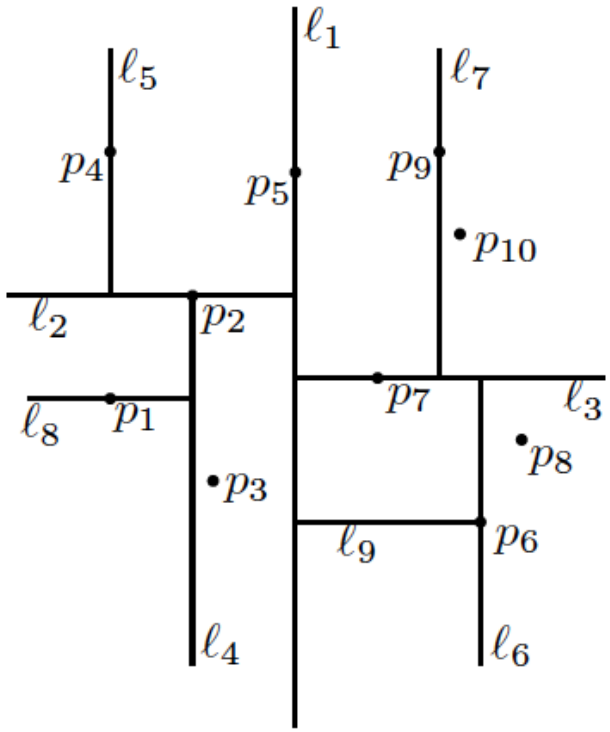
Taken from *Physically Based Rendering* by Pharr et al.

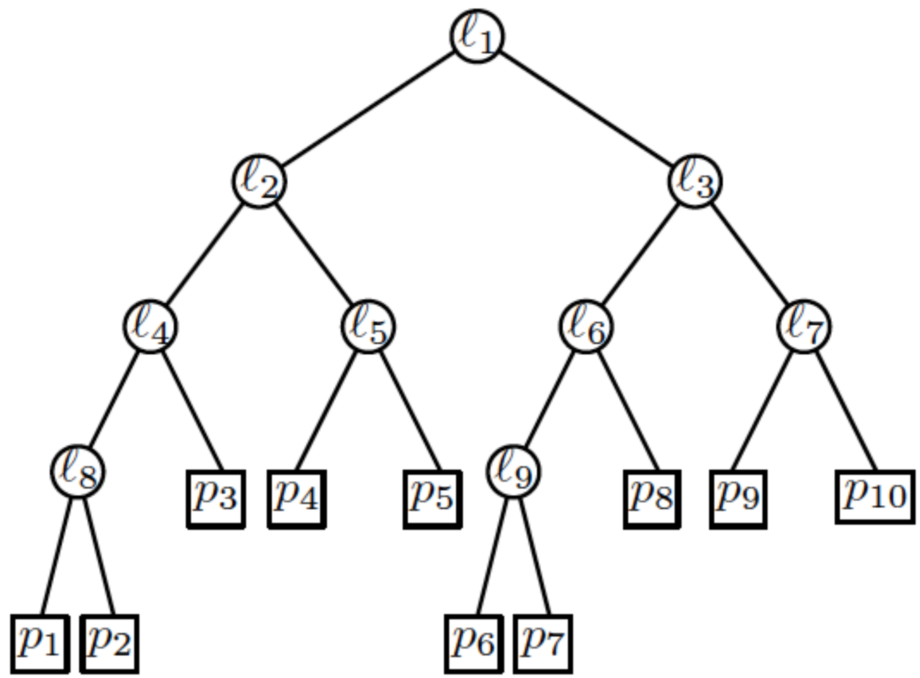
Building a kd-tree

- Splits are axis-aligned
- But we choose location of that split plane
- Alternate the axis that we split
- We want a ***balanced*** tree to decrease search time….each internal node prunes half the geometry



Building a 2D kd-tree - example





2D kd-tree example

Algorithm BUILDKDTREE($P, depth$)

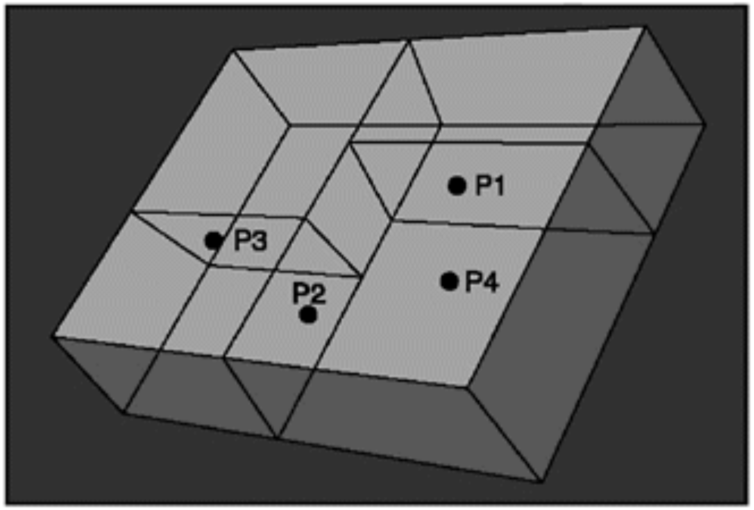
if P contains only one point
  then return a leaf storing this point
  else if $depth$ is even
    then Split P with a vertical line ℓ through the median x-coordinate into P_1 (left of or on ℓ) and P_2 (right of ℓ)
    else Split P with a horizontal line ℓ through the median y-coordinate into P_1 (below or on ℓ) and P_2 (above ℓ)
   $\nu_{\text{left}} \leftarrow$ BUILDKDTREE($P_1, depth + 1$)
   $\nu_{\text{right}} \leftarrow$ BUILDKDTREE($P_2, depth + 1$)
   Create a node ν storing ℓ, make ν_{left} the left child of ν, and make ν_{right} the right child of ν.
   return ν

2D KD-tree Build Time

- Median finding among n numbers takes $O(n)$ time
- What then is the computational cost $T(n)$ for n points?
  - $T(n) = 2T(n/2)+O(n)$
  - $T(1) = O(1)$
- Total build time will be $O(n \lg n)$

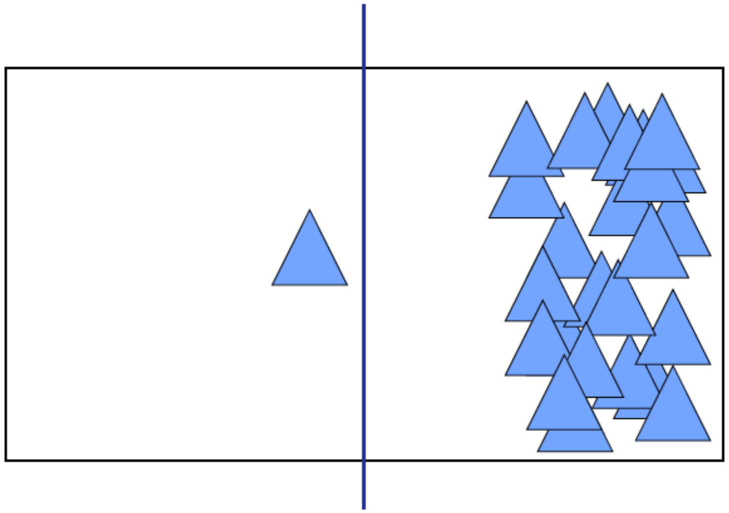
3D kd-tree

- Very similar…3 alternating axes instead of 2
- Point location done recursively
- For n points
  - O(n) size structure
  - O(lg n) point location…for balanced tree…



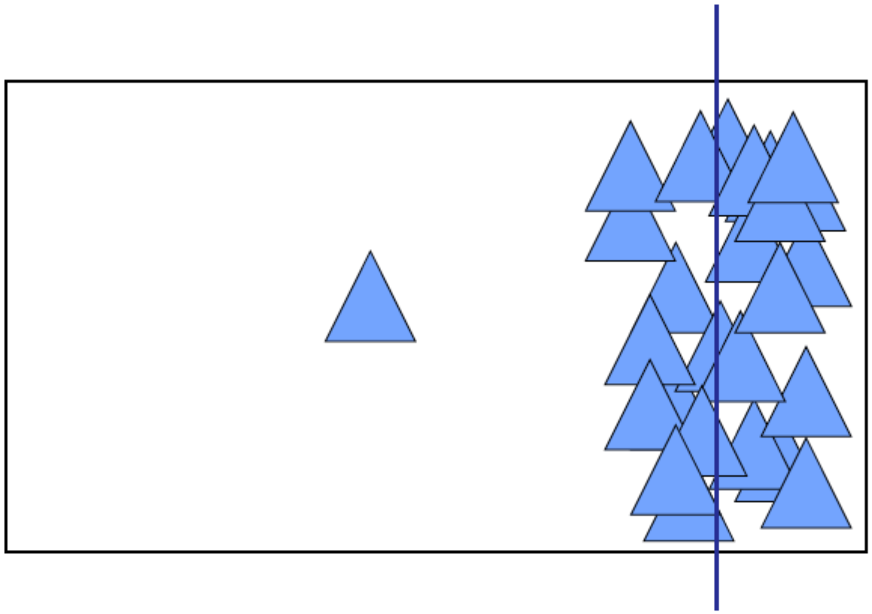
Splitting in 3D

Split In The Middle: Bad!



Midpoint: makes left and right probabilities equal
Cost of R greater than cost of L

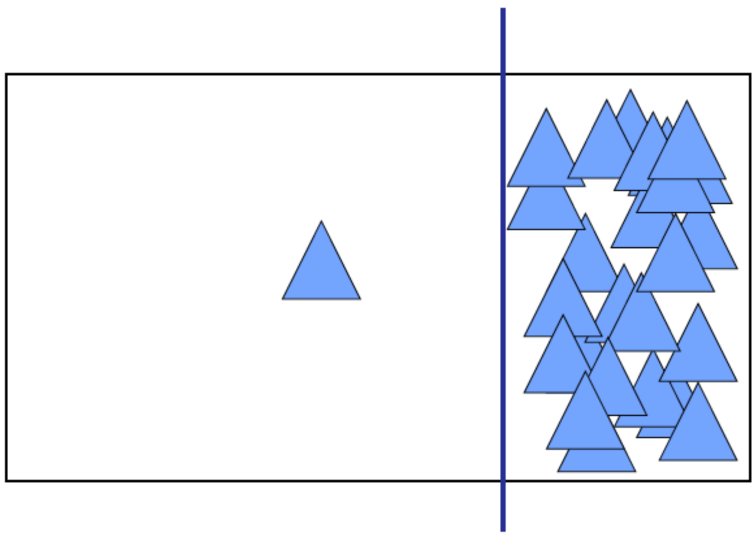
Looking at costs



Median: makes left and right costs equal
Probability of hitting L greater than R

Looking at costs

Cost-Optimized Split



Cost(node) = C_{visit} + Prob(hit L) * Cost(L) + Prob(hit R) * Cost(R)

Computing costs for kd-trees

$$\text{Cost(node)} = C_{\text{visit}} + \text{Prob(hit L)} * \text{Cost(L)} + \text{Prob(hit R)} * \text{Cost(R)}$$

C_{visit} = cost of visiting a note

Cost(L) = cost of traversing left child

Cost(R) = cost of traversing right child

Computing costs for kd-trees

- Need the probabilities
  - Turn out to be proportional to the surface area
- Need the child cell costs
  - Triangle count is a good approximation

$$\text{Cost(cell)} = C_{\text{visit}} + \text{SurfArea(L)} * \text{TriCount(L)} + \text{SurfArea(R)} * \text{TriCount(R)}$$

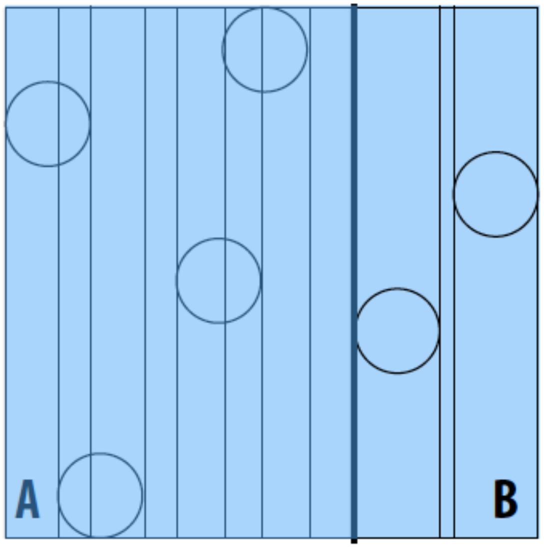
C_{trav} is the ratio of the cost to traverse to the cost to intersect

$C_{\text{trav}} = 1{:}80$ in PBRT

$C_{\text{trav}} = 1{:}1.5$ in a highly optimized version

Build Algorithm

1. Pick an axis, or optimize across x, y, z
2. Build a set of candidate split locations
    - Note: cost extrema must be at bbox vertices
        - Vertices of triangle
        - Vertices of triangle clipped to node bbox
3. Sort the triangles into intervals
4. Sweep to incrementally track L/R counts, costs
5. Output position of minimum cost split

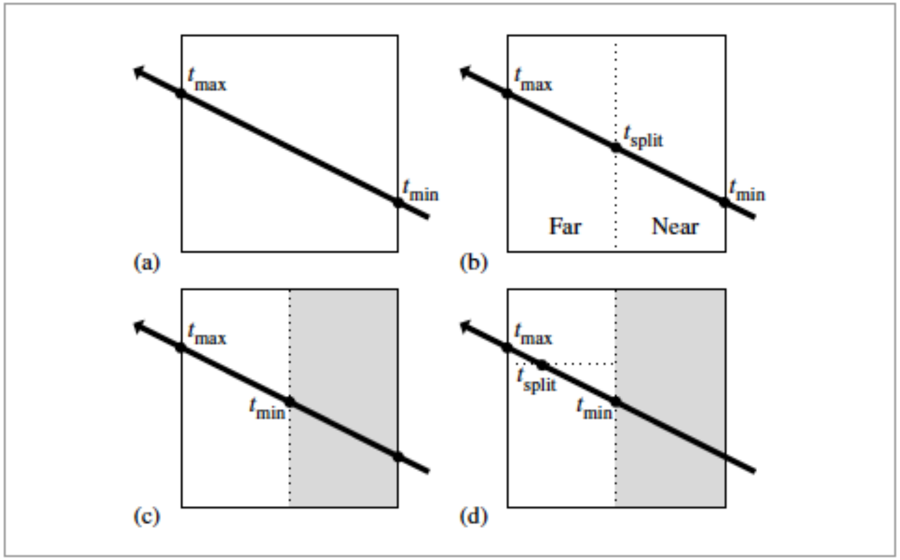


Termination Criteria

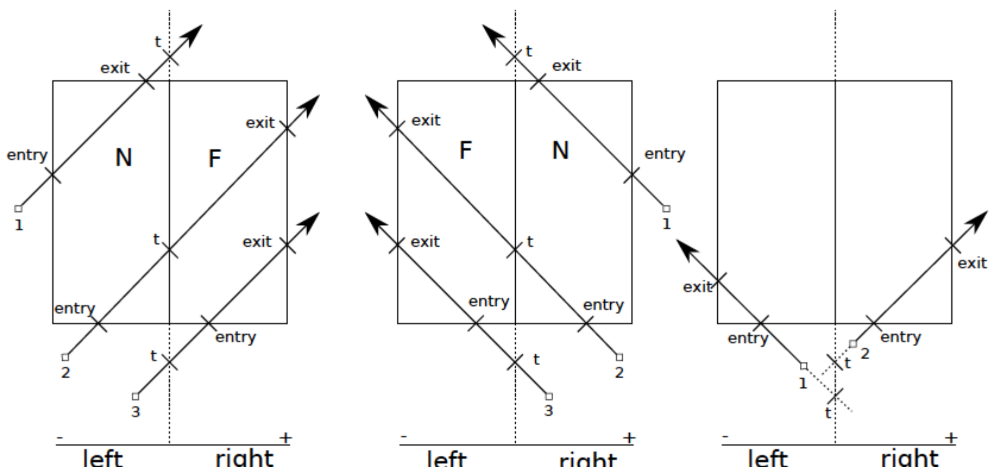
- **When should we stop splitting?**
  - **Bad: depth limit, number of triangles**
  - **Good: when split does not lower the cost**
- **Threshold of cost improvement**
  - **Stretch over multiple levels–e.g., terminate if cost doesn't go down after three splits in a row**
- **Threshold of cell cize**
  - **Absolute probability SA(node)/SA(scene) low**

Simple Traversal

- Simple sequential traversal
  - Find ray entry point to top node bounding box
  - Traverse kd-tree doing point location
  - At leaf, test ray against primitives
  - If no hit, find leaf bbox exit point and repeat search
- How is this inefficient?



Stack-based Traversal

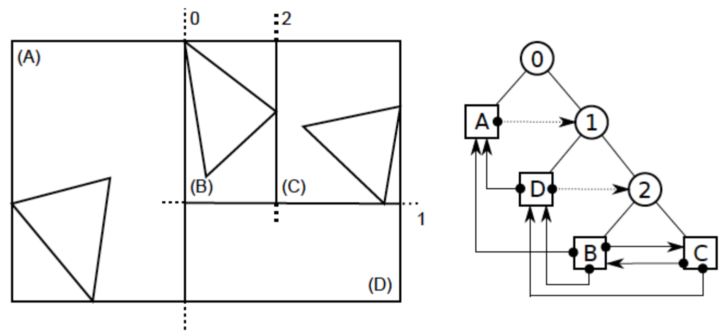


Use a stack of nodes to visit to limit repeated visits

```
Kd-tree Recursive Traversal:
begin
    (entry distance, exit distance) ← intersect ray with root's
    AABB;
    if ray does not intersect AABB then
        return no object intersected;
    end
    push ( tree root node, entry distance, exit distance) to
    stack ;
    while stack is not empty do
        (current node, entry distance, exit distance) ← pop
        stack;
        while current node is not a leaf do
            a ← current node's split axis;
            t ← (current node's split position.a - ray origin.a)
            / ray dir.a;
            (near, far) ← classify near/far with (split
            position.a > ray origin.a);
            if t ≥ exit distance or t < 0 then
                current node ← near;
            else if t ≤ entry distance then
                current node ← far;
            else
                push ( far, t, exit distance) to stack;
                current node ← near;
                exit distance ← t;
            end
        end
        if current node is not empty leaf then
            intersect ray with each object;
            if any intersection exists inside the leaf then
                return closest object to the ray origin;
            end
        end
    end
    return no object intersected;
end
```

Other Speedups

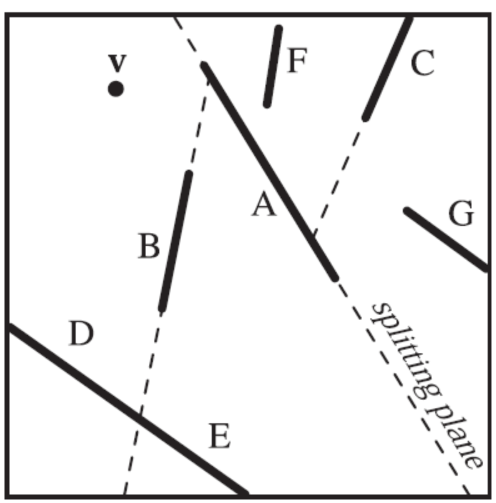
- Neighbor links (ropes) to reference sibling cells

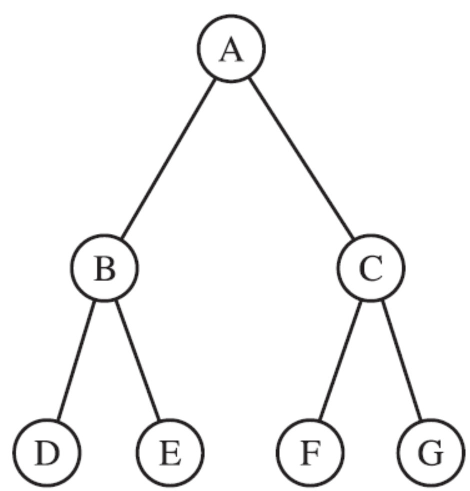


- Packet tracing: rays with similar origin and direction traced together through the structure

BSP Tree

- Cutting planes have arbitrary orientation
- Splitting can be done along ploygon
  - Choose subset (5?) to test…pick one that yields best balance





Constructing BSP for Point Location

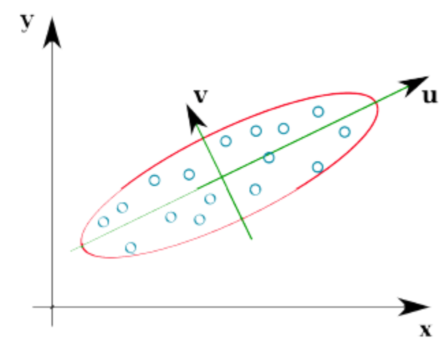
- Build using Principal Component Analysis (PCA)

The scatter matrix is computed by the following equation:

$$S = \sum_{k=1}^{n} (\boldsymbol{x}_k - \boldsymbol{m})(\boldsymbol{x}_k - \boldsymbol{m})^T$$

where $\boldsymbol{m}$ is the mean vector

$$\boldsymbol{m} = \frac{1}{n} \sum_{k=1}^{n} \boldsymbol{x}_k$$



Compute eigenvalues and eigenvectors
Largest eigenvalue ➔ eigenvector indicating direction of greatest variation
Cut at mean perpendicular to that vector