Geometric Structures

2. Quadtree, k-d stromy

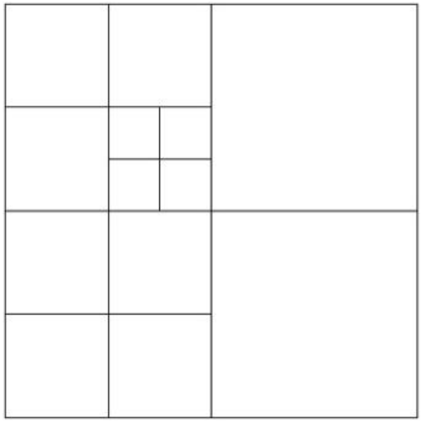
Martin Samuelčík
samuelcik@sccg.sk, www.sccg.sk/~samuelcik, I4

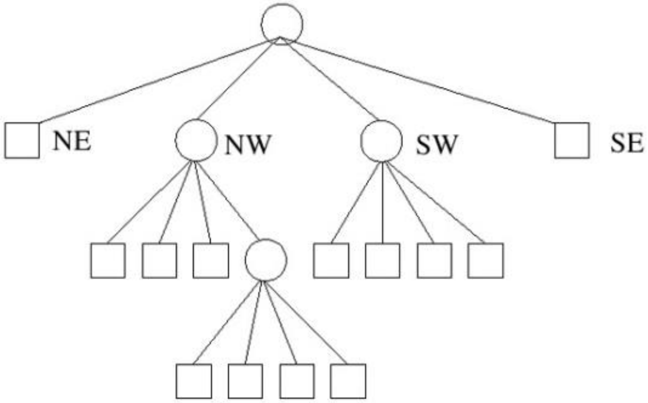
Window and Point query

- From given set of points, find all that are inside of given d-dimensional interval
- Solution using multi-dimensional range trees
  - Higher memory complexity
  - General for all dimensions
  - Adaptive construction based on given points
- Other solution– split using hyperplanes in one dimesion– slower, but lower memory complexity

Quadtree

- Each inner node of tree has exactly 4 siblings
- Each node represents area part of 2D space, usually square or rectangle, but other shapes are possible
- 4 children of node represents split of node area into 4 smaller equal areas
- 2D





Quadtree construction

- S – set of points in 2D
- Initial creation of bounding area for points in S
- Recursive

```
struct QuadTreeNode
{
   Point* point;
   float left, right, bottom, top;
   QuadTreeNode * parent;
   QuadTreeNode * NE;
   QuadTreeNode * NW;
   QuadTreeNode * SW;
   QuadTreeNode * SE;
}
```

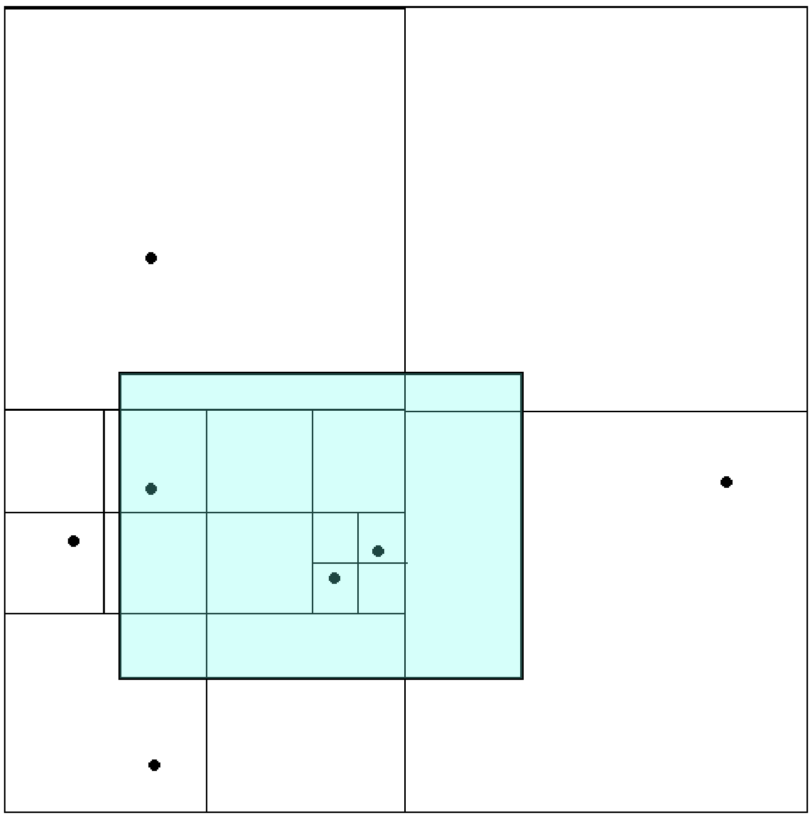
```
struct QuadTree
{
   QuadTreeNode* root;
}
```

```
QuadTreeConstruct(S)
{
   (left, right, bottom, top) = BoundSquare(S);
   QuadTree* tree = new QuadTree;
   tree->root = QuadTreeNodeConstruct(S, left, right, bottom, top);
   return tree;
}
```

```
QuadTreeNodeConstruct(P, left, right, bottom, top)
{
   v = new QuadTreeNode;
   v->left = left; v->right = right; v->bottom = bottom; v->top = top;
   v->NE = v->NW = v->SW = v->SE = v->parent = v->point = NULL;
   if (|P| == 0) return v;
   if (|P| == 1)
   {
      v->point = P.first;
      return v;
   }
   xmid = (left + right)/2; ymid = (bottom + top)/2;
   (NE, NW, SW, SE) = P.Divide(midx, midy);
   v->NE = QuadTreeNodeConstruct(NE, xmid, right, ymid, top);
   v->NW = QuadTreeNodeConstruct(NW, left, xmid, ymid, top);
   v->SW = QuadTreeNodeConstruct(SW, left, xmid, bottom, ymid);
   v->SE = QuadTreeNodeConstruct(SE, xmid, right, bottom, ymid);
   v->NE->parent = v; v->NW->parent = v;
   v->SW->parent = v; v->SE->parent = v;
   return v;
}
```

Quadtree search

- Finding point inside rectangle
  B=[left,right,bottom,top]



```
QuadTreeQuery(tree, B)
{
   return QuadTreeNodeQuery(tree->root, B)
}
```

```
QuadTreeNodeQuery(node, B)
{
   List result;
   if (node == NULL)
      return result;
   if (B->left > node->right || B->right < node->left ||
       B->bottom > node->top || B->top < node->bottom)
   {
      return result;
   }
   if (node->point)
      result.Add(point);

   result.Add(QuadTreeNodeQuery(v->NE, B));
   result.Add(QuadTreeNodeQuery(v->NW, B);
   result.Add(QuadTreeNodeQuery(v->SW, B);
   result.Add(QuadTreeNodeQuery(v->SE, B);
   return result;
}
```

Quadtree properties

- Maximal depth of quadtree is $log(s/c) + 3/2$, where c is smallest distance between points in S and s is length of one side of initial bounding square
- Quadtree od depth d with $|S|=n$ has $O(n.(d+1))$ nodes and can be constructed in time $O(n.(d+1))$
- Construction: $O(n^2)$
- Memory: $O(n^2)$
- Search: $O(n)$

Finding neighbor node

- For given node and corresponding area, find node and its area adjacent to given node in given direction
- Time complexity O(d), d – height of tree

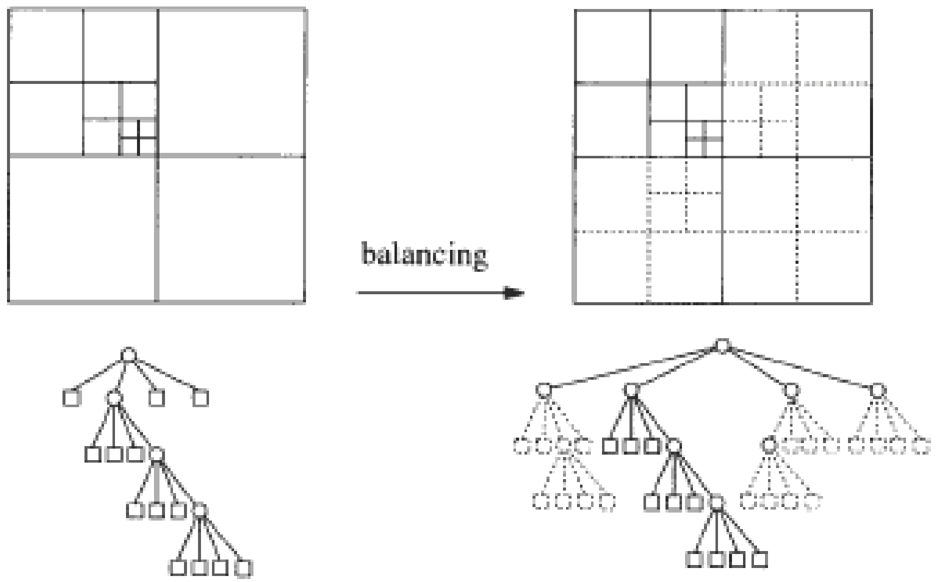
```
NorthNeighbor(v, T)
{
        if (v == T->root) return NULL;
        if (v == v->parent->SW) return v->parent->NW;
        if (v == v->parent->SE) return v->parent->NE;
        u = NorthNeighbor(v->parent, T);
        if (u == NULL || u->IsLeaf()) return u;
        if (v == v->parent->NW)
                return u->SW;
        else
                return u->SE;
}
```

```
SouthChilds(v)
{
        List result;
        if (v == NULL)
                return result;
        if (v->IsLeaf())
                result.Add(v);
        result.Add(SouthChilds(v->SE));
        result.Add(SouthChilds(v->SW));
        return result;
}
```

```
NorthNeighbors(v, T)
{
        List result;
        North = NorthNeighbor(v, T);
        if (North == NULL)
                return result;
        return SouthChilds(North);
}
```

Balancing quadtree

- Balanced quadtree – each two adjacent areas has almost same size
- Balancing - simple adding empty subtrees
- If T has m nodes, then balanced version has O(m) nodes and can be created in time O(m(d+1))



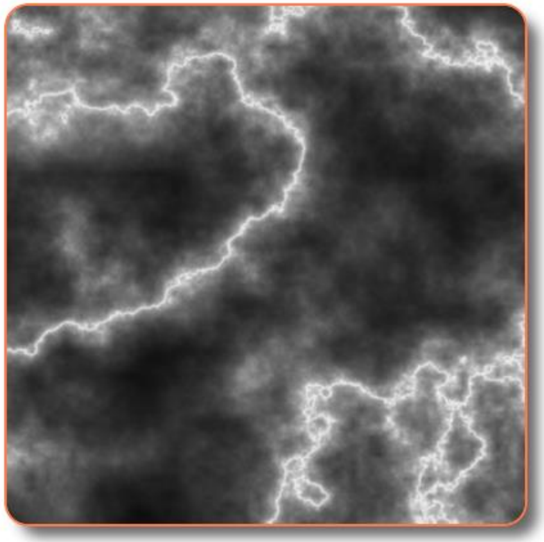
Balancing quadtree

- CheckDivide – checking, if node v has to be divided, finding if neighbor siblings are leaves or not
- Divide – dividing node of tree into four siblings, adding point from node to one sibling
- CheckNeighbours – if there unbalanced neighbours, then add neighbors to list L

```
BalanceQuadTree(T)
{
        if (v == T->root) return NULL;
        L = T.ListLeafs();
        while (!L.IsEmpty())
        {
                v = L.PopFirst();
                if (CheckDivide(v))
                {
                        Divide(v);
                        L.add(v->NE); L.add(v->NW);
                        L.add(v->SE); L.add(v->SW);
                        CheckNeighbors(v, L);
                }
        }
}
```

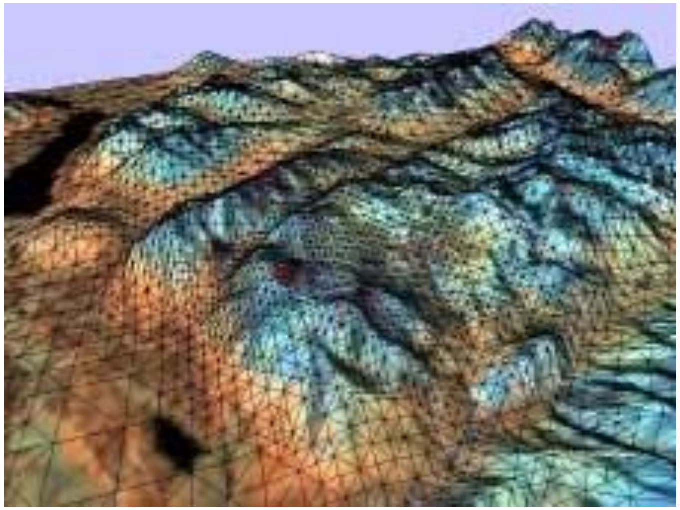
Terrain visualization

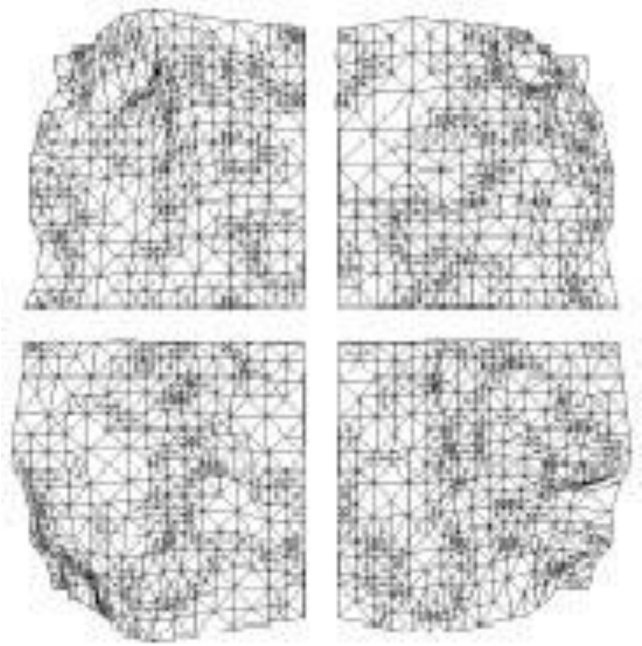
- View above terrain surface – some parts are close, some away – using LOD (Level Of Detail), each part of terrain is rendered in some detail based on distance from camera
- Needed structure for storing all levels of detail for each part of terrain
- Terrain
  - Height field



Terrain visualization

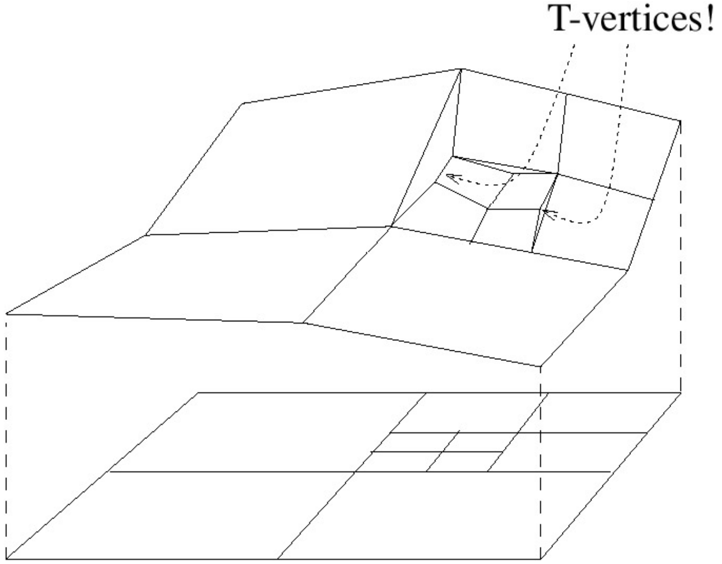
- Creating complete quadtree over height filed
- Traversing tree during rendering – based on distance of camera and node area, the traverse is stopped or contiinued

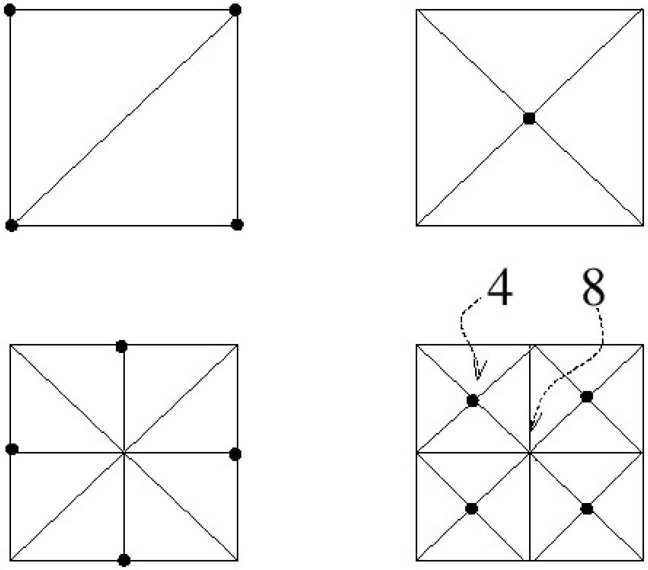


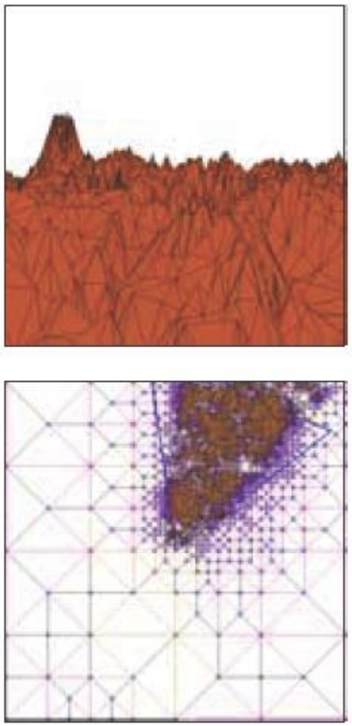


Terrain visualization

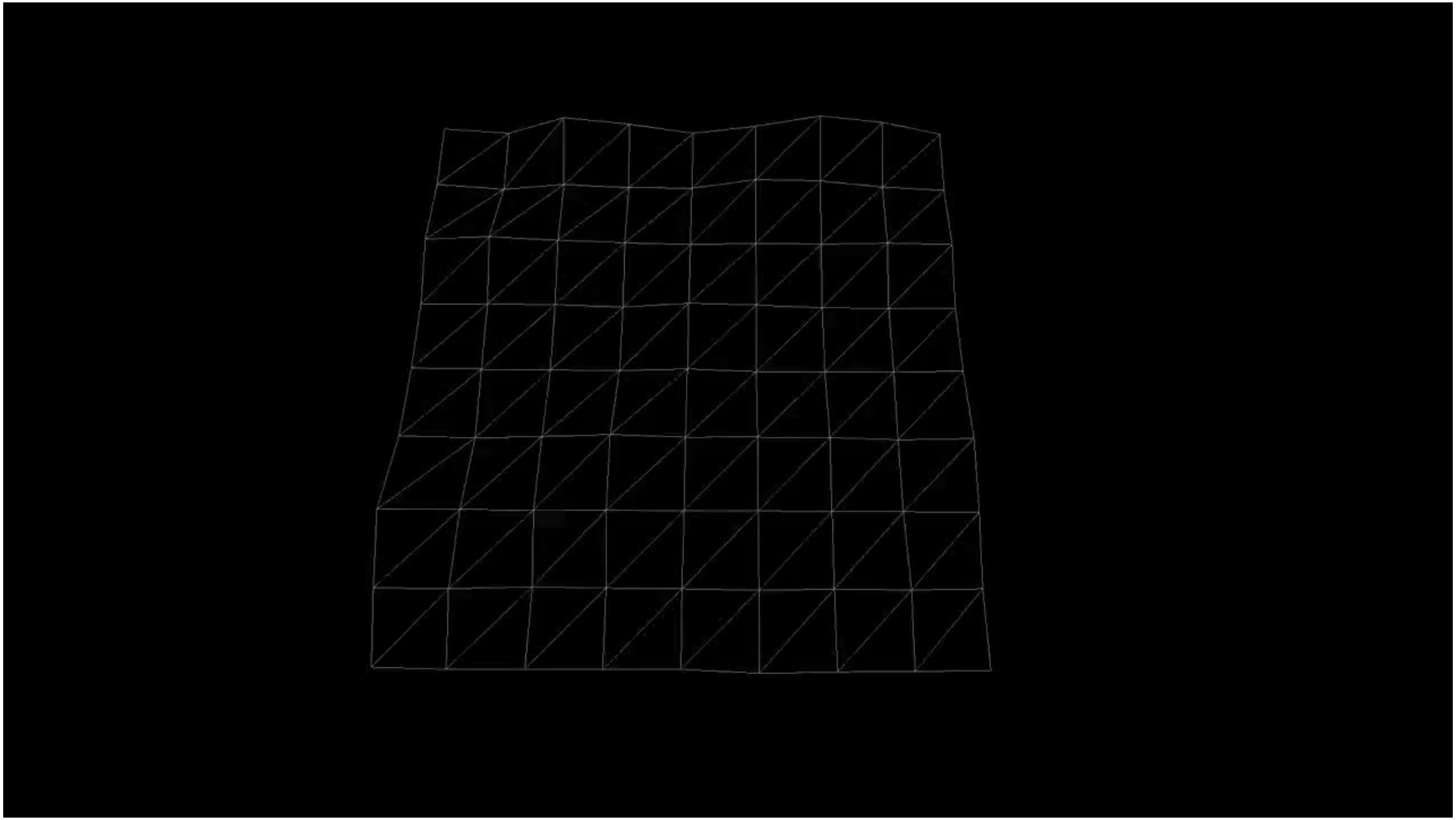
- Problems with the edge between areas on different levels in quadtree
- Solution using triangulation= connection of two consecutive quadtrees



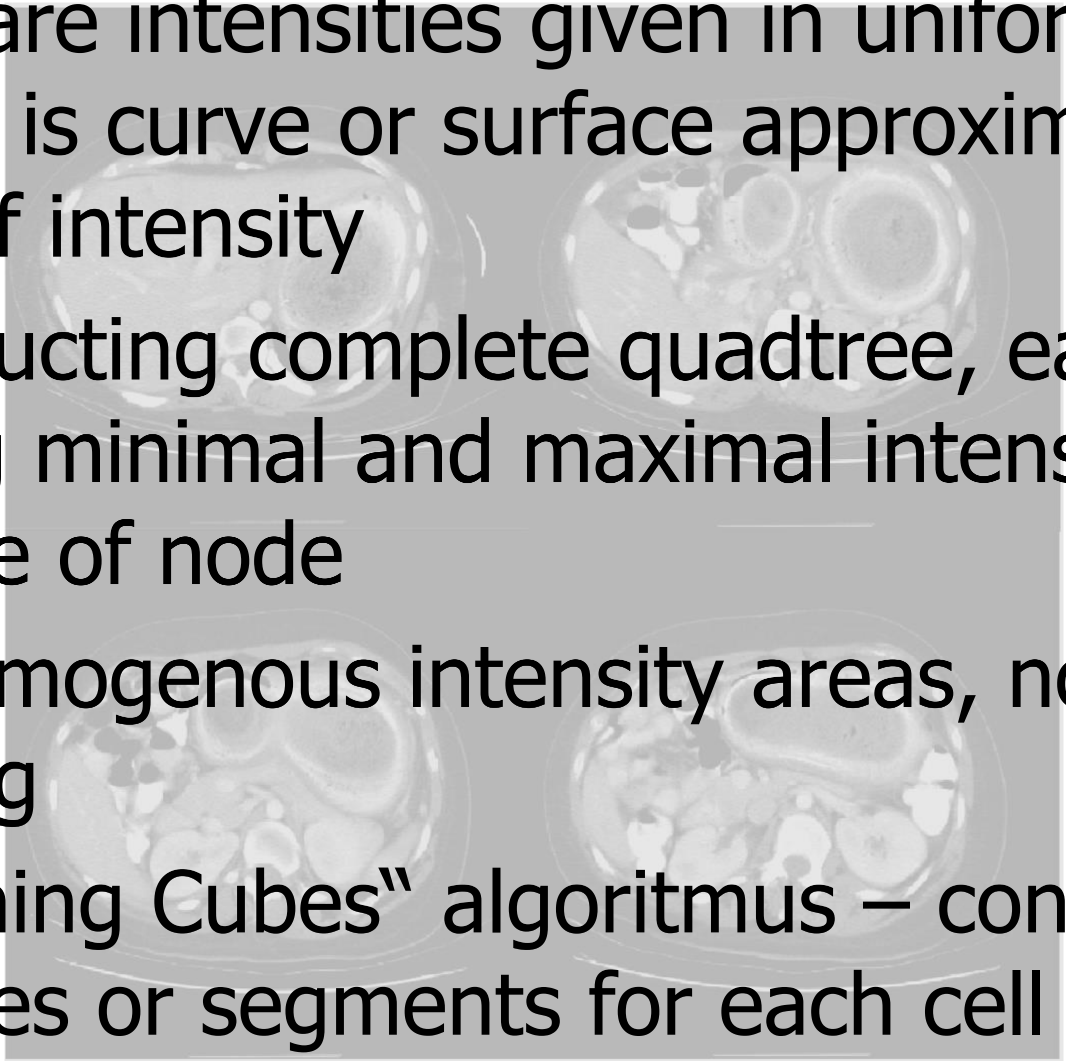




Terrain visualization

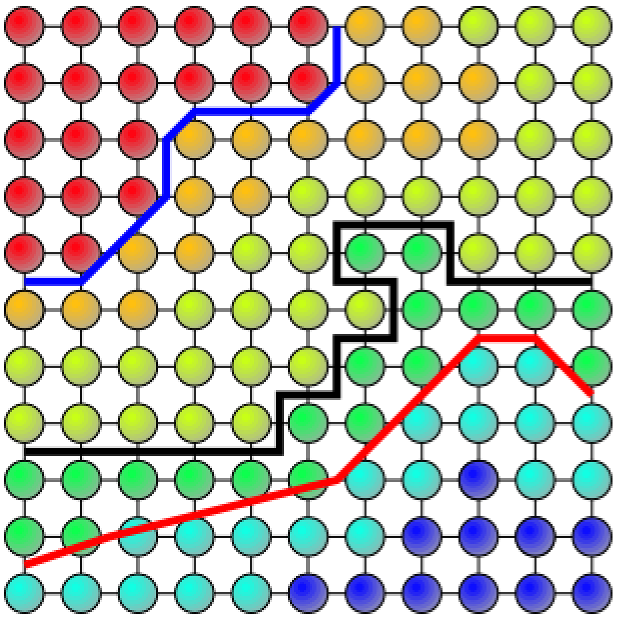


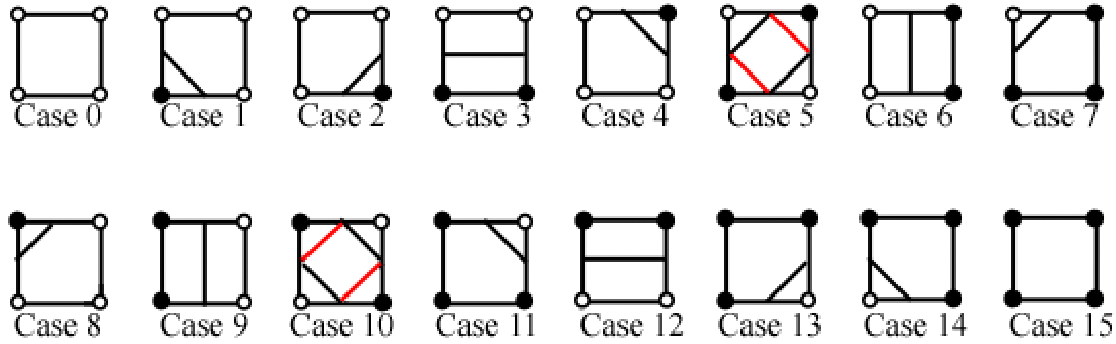
Isosurface generation

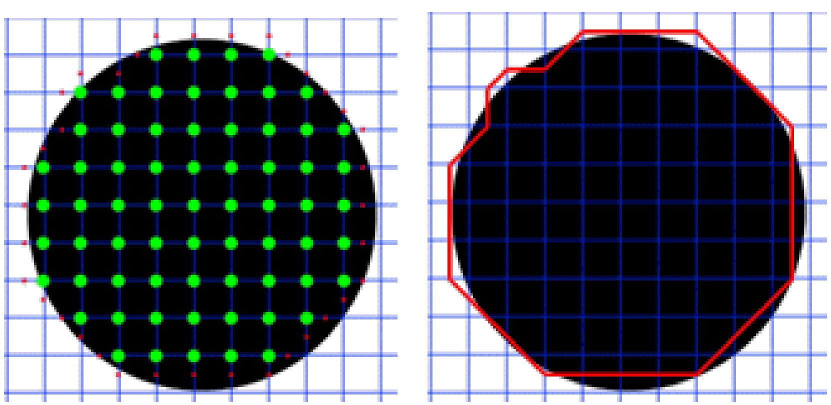


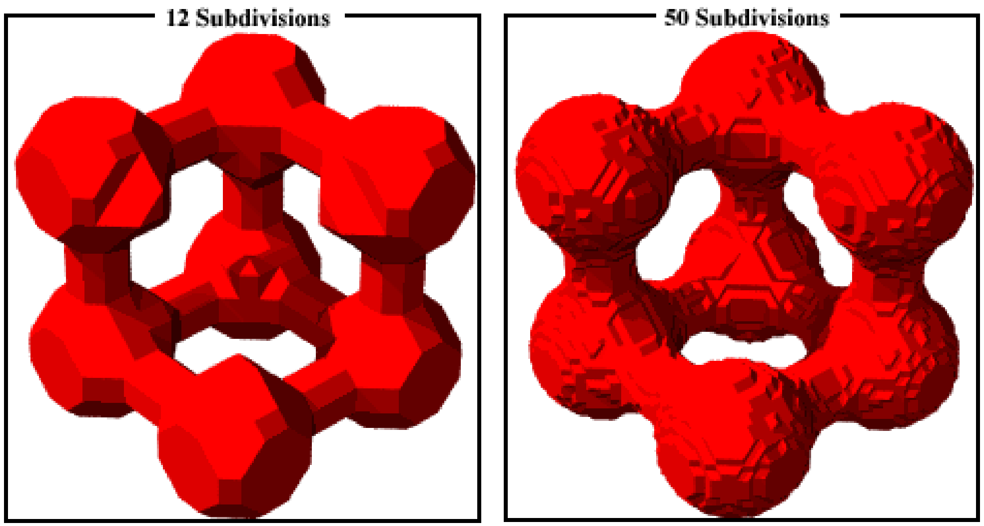
- Input are intensities given in uniform grid, output is curve or surface approximating one level of intensity
- Constructing complete quadtree, each node storing minimal and maximal intensity in subtree of node
- For homogenous intensity areas, no need for dividing
- „Marching Cubes“ algoritmus – constructing triangles or segments for each cell

Isocurve, isosurface generation



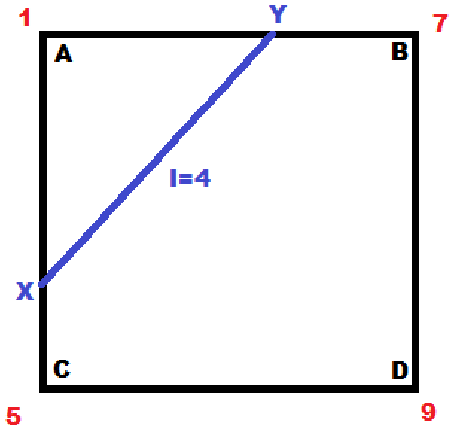






Bilinear interpolation in cell

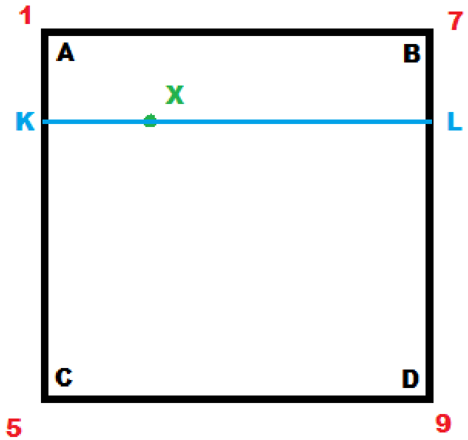
- Computing end points of segments



$$X = \frac{|5-4|}{|5-1|}A + \frac{|4-1|}{|5-1|}C = \frac{1}{4}A + \frac{3}{4}C$$

$$Y = \frac{|7-4|}{|7-1|}A + \frac{|4-1|}{|7-1|}B = \frac{1}{2}A + \frac{1}{2}B$$

- Computing intensity for point inside cell



$$I_K = \frac{|Xy-Ay|}{|Cy-Ay|}5 + \frac{|Cy-Xy|}{|Cy-Ay|}1 \qquad I_L = \frac{|Xy-By|}{|Dy-By|}9 + \frac{|Dy-Xy|}{|Dy-By|}7$$

$$I_X = \frac{|Xx-Ax|}{|Bx-Ax|}I_L + \frac{|Bx-Xx|}{|Bx-Ax|}I_K$$

Quadtree for Marching Squares

- Input is 2D array of intesities
  - $P[i,j];\ i = 0,\ldots,2^n\ ;\ j = 0,\ldots,2^m$

```
struct MCQuadTree
{
   MCQuadTreeNode* root;
}
```

```
struct MCQuadTreeNode
{
   float MinIntensity;
   float MaxIntensity;
   float Corner1, Corner2;
   float Corner3, Corner4;
   QuadTreeNode * parent;
   QuadTreeNode * NE;
   QuadTreeNode * NW;
   QuadTreeNode * SW;
   QuadTreeNode * SE;
}
```

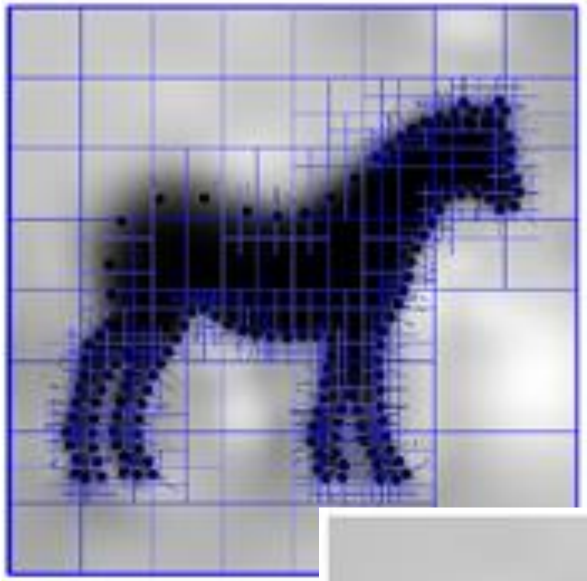
```
MCQuadTreeConstruct(P, n, m)
{
   MCQuadTree* tree = new MCQuadTree;
   tree->root = MCQuadTreeNodeConstruct(
                  P, 0, 2^n, 0, 2^m);
   return tree;
}
```

```
MCQuadTreeNodeConstruct(P, MinXIndex, MaxXIndex, MinYIndex, MaxYIndex)
{
   v = new MCQuadTreeNode;
   v->Corner1 = P[MinXIndex, MinYIndex]; v->Corner2 = P[MaxXIndex, MinYIndex];
   v->Corner3 = P[MaxXIndex, MaxYIndex]; v->Corner4 = P[MinXIndex, MaxYIndex];
   v->NE = v->NW = v->SW = v->SE = v->parent = NULL;
   (Min, Max) = GetMinMaxIntensities(P, MinXIndex, MaxXIndex,
                                     MinYIndex, MaxYIndex);
   v->MinIntensity = Min;
   v->MaxIntensity = Max;
   if (Min == Max)
      return v;
   MidXIndex = (MinXIndex + MaxXIndex) / 2;
   MidYIndex = (MinYIndex + MaxYIndex) / 2;
   v->NE = MCQuadTreeNodeConstruct(P, MidXIndex, MaxXIndex, MidYIndex, MaxYIndex);
   v->NW = MCQuadTreeNodeConstruct(P, MinXIndex, MidXIndex, MidYIndex, MaxYIndex);
   v->SW = MCQuadTreeNodeConstruct(P, MinXIndex, MidXIndex, MinYIndex, MidYIndex);
   v->SE = MCQuadTreeNodeConstruct(P, MidXIndex, MaxXIndex, MinYIndex, MidYIndex);
   v->NE->parent = v; v->NW->parent = v;
   v->SW->parent = v; v->SE->parent = v;
   return v;
}
```

Marching Squares

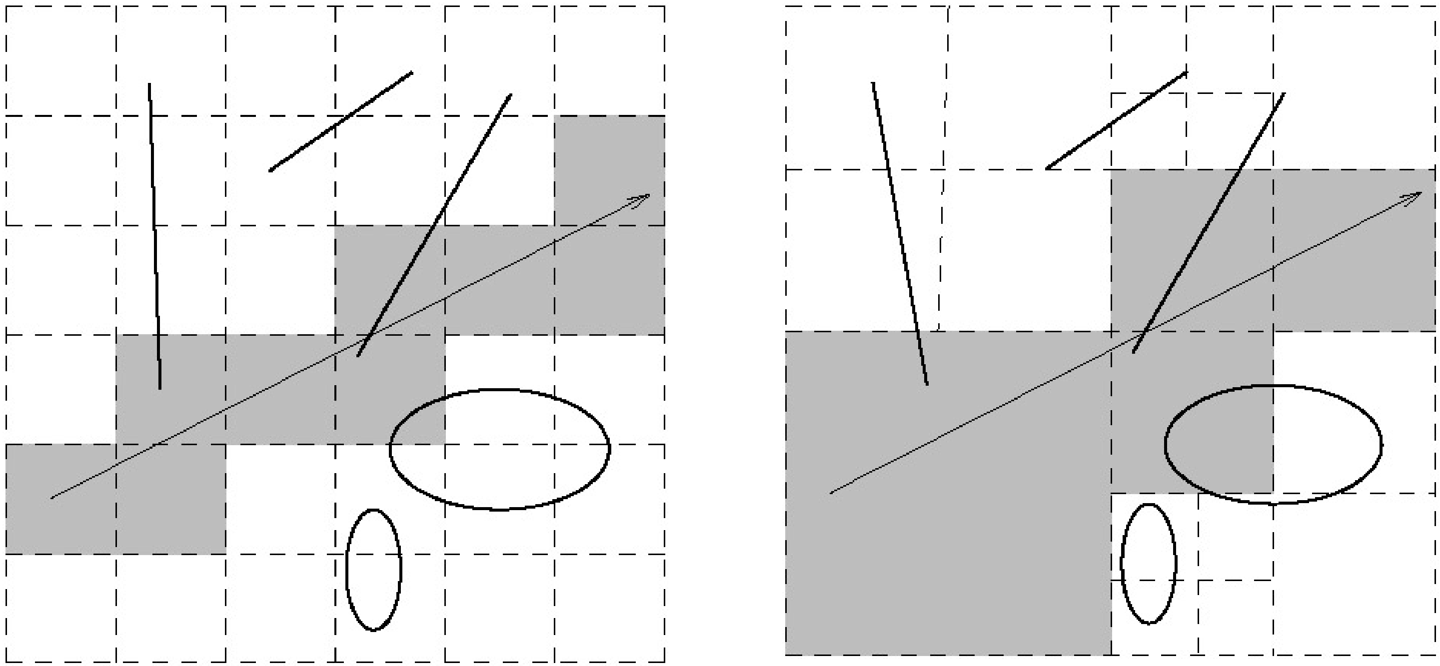
```
MarchCubes(T, intensity, left, right, bottom, top)
{
   MarchCubesNode(T->root, intensity, left, right, bottom, top);
}
```

```
MarchCubesNode(v, intensity, left, right, bottom, top)
{
   if (intensity < MinIntensity || intensity > MaxIntensity)
      return;
   if (MinIntensity == MaxIntensity)
      return;
   if (v->IsLeaf())
   {
      CreateLinesInRextangle(left, right, bottom, top,
         v->Corner1, v->Corner2, v->Corner3, v->Corner4);
      return;
   }
   float midx = (left+right) / 2;
   float midy = (bottom+top) / 2;
   MarchCubesNode(v->SW, intensity, left, midx, bottom, midy);
   MarchCubesNode(v->SE, intensity, midx, right, bottom, midy);
   MarchCubesNode(v->NE, intensity, midx, right, midy, top);
   MarchCubesNode(v->NE, intensity, left, midx, midy, top);
}
```



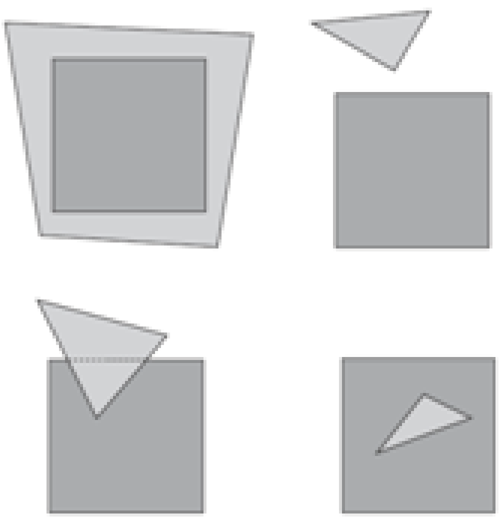
Raytracing

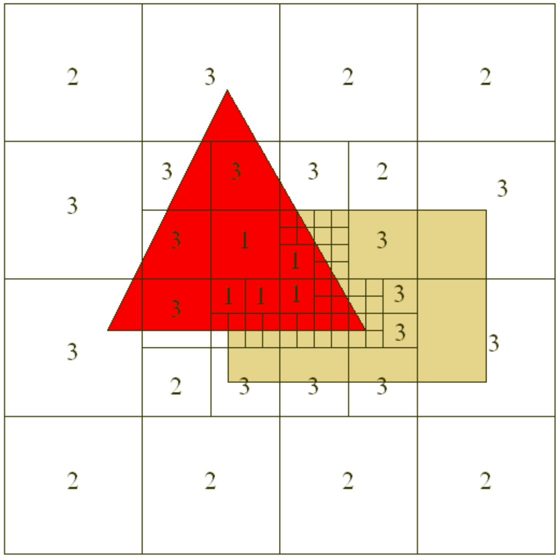
- Storing pointers to objects inside quadtree
- Finding neighbor cells in quadtree when traversing along ray



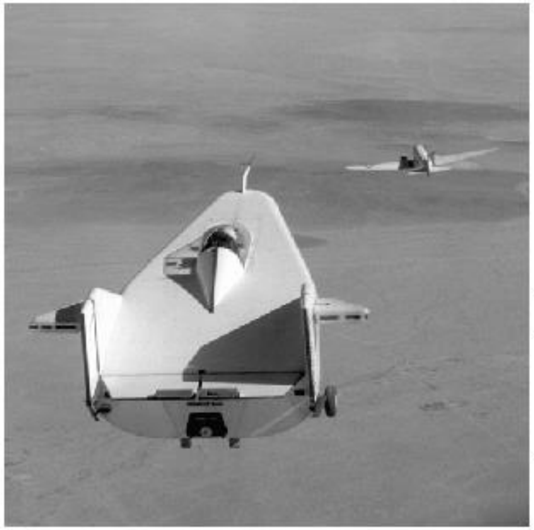
Visibility computation (Warnock)

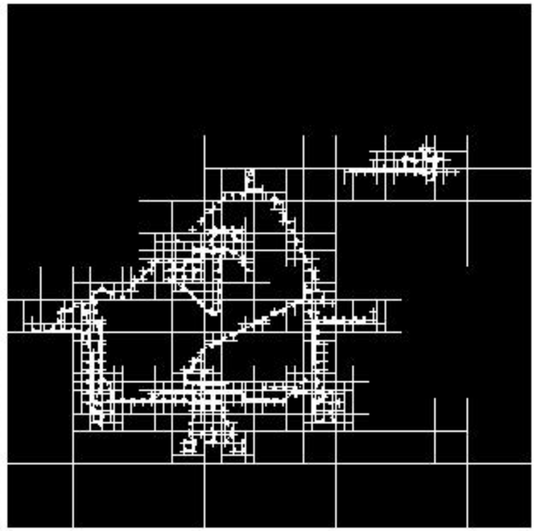
- Computation in screen space
- Divide parts of screen using quadtree until simple cases occur
- In each leaf of quadtree, compute color of all pixels in node ares from nearest polygon

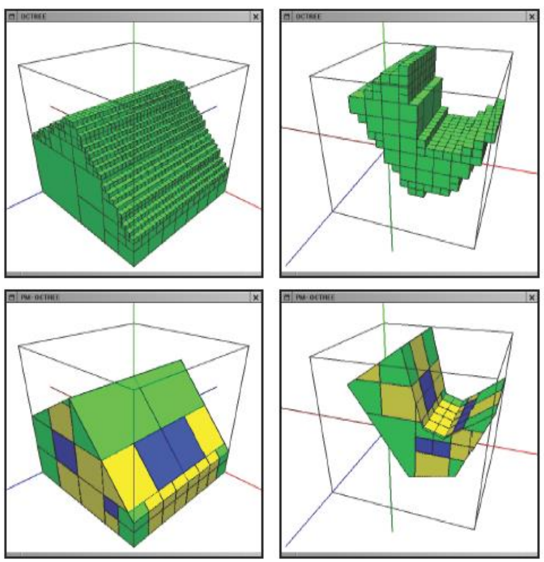


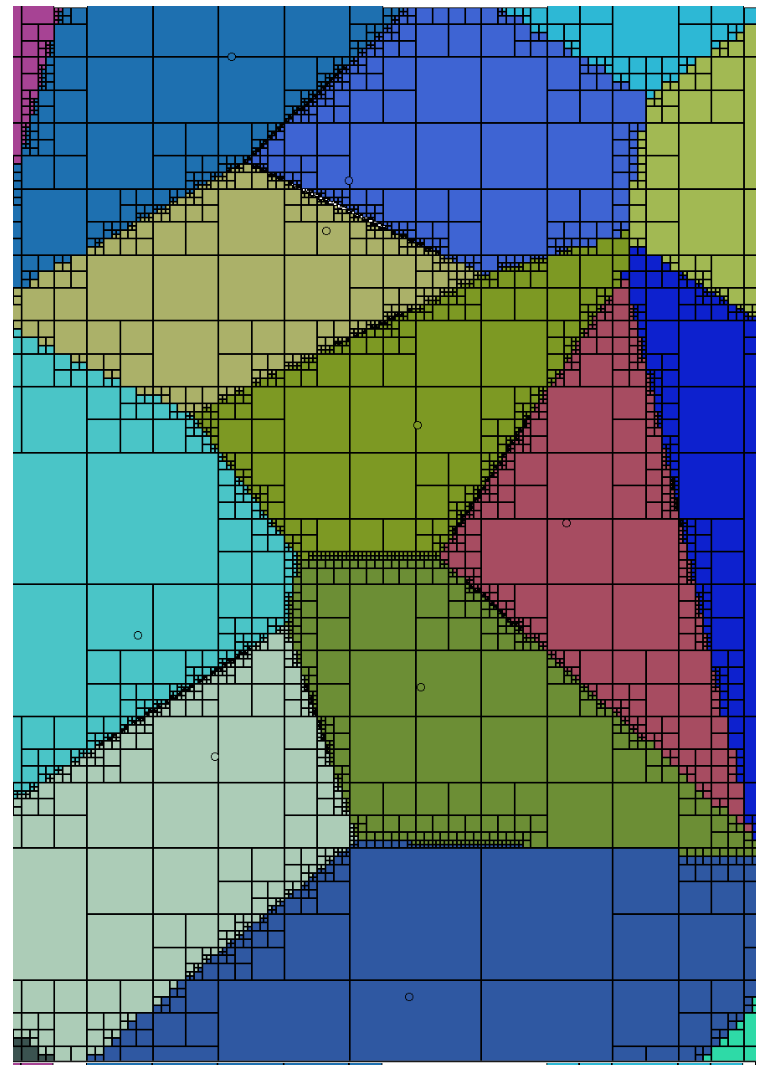


Representations



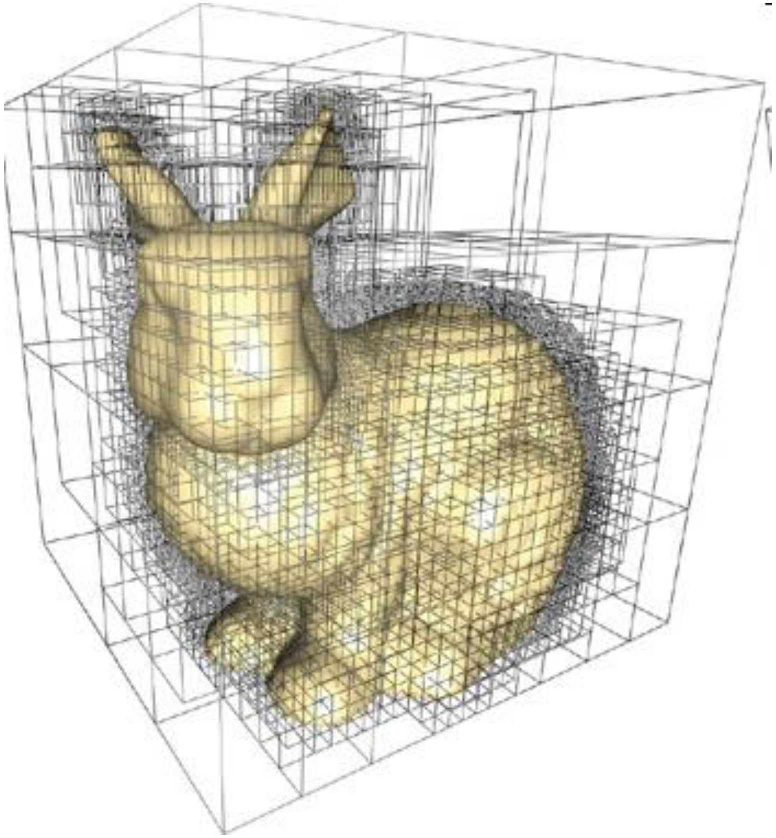


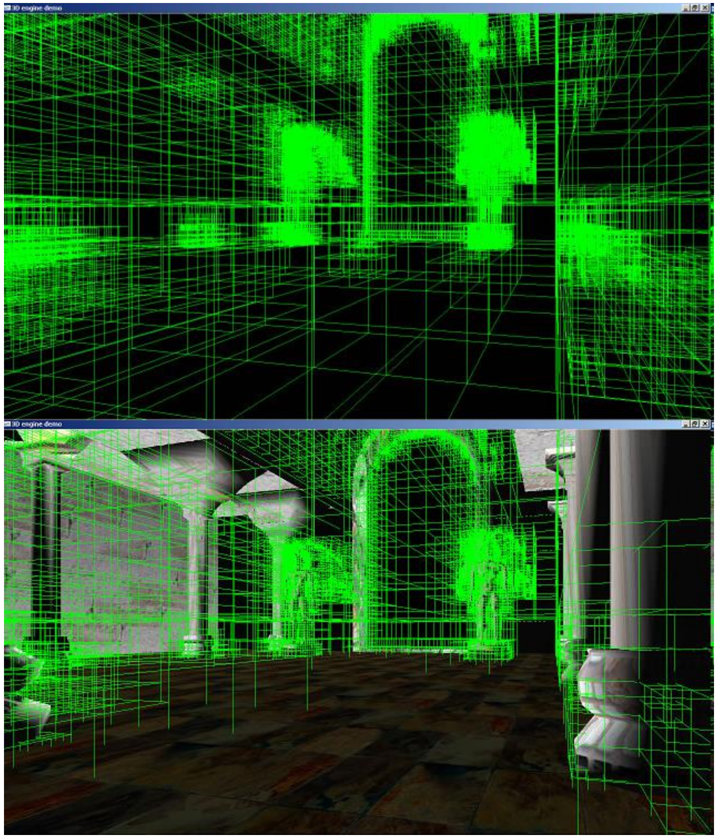




Octree

- Extension of quadtree in 3D space, solution of same or similar problems





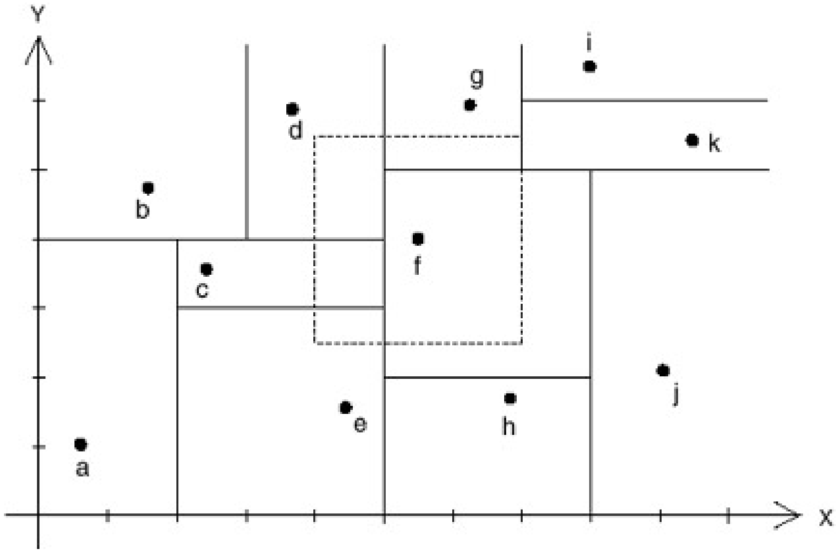
K-d tree

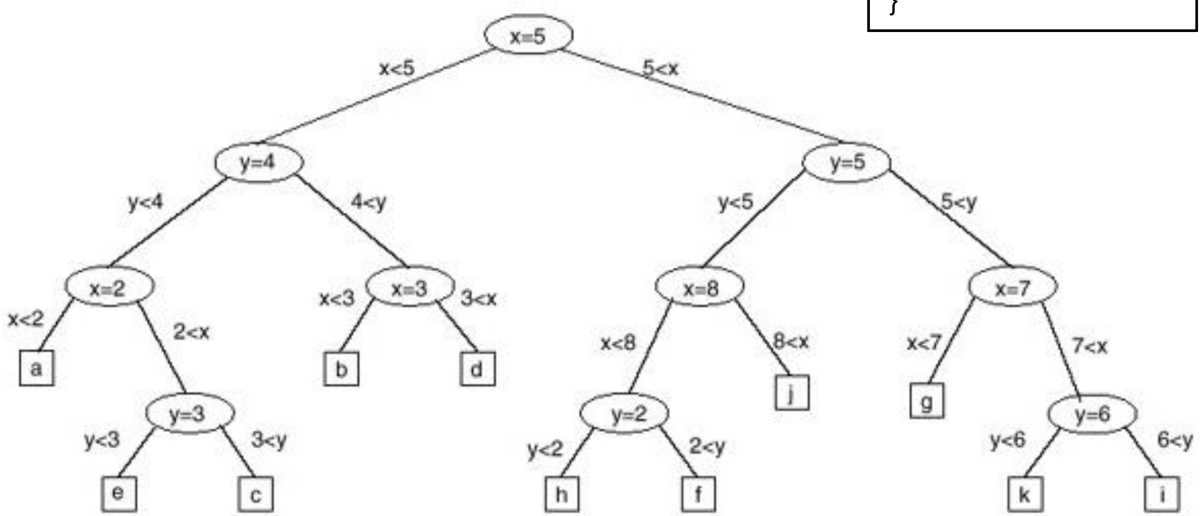
- Input: set of points S from R^d
- Query: d-dimensional interval B
- Output: set of points from S, that are inside set B
- Recursive construction:

$$D_{<s_i} = \{(x_1, \ldots, x_i, \ldots, x_n) \in D | x_i < s\},$$
$$D_{>s_i} = \{(x_1, \ldots, x_i, \ldots, x_n) \in D | x_i > s\};$$

  - Given set of points D from R^d and split coordinate i
  - If D is empty, return empty node
  - If D contains 1 point, current node becomes leaf
  - Else compute split value s in i-th coordinate and based on this value divide D into two sets $D_{<s_i}$, $D_{>s_i}$ and for these two sets recursively construct two sibling nodes with increase coordinate i by 1

K-d tree construction






```
struct KdTreeNode
{
   float  split;
   int dim;
   Point* point;
   KdTreeNode * left;
   KdTreeNode * right;
   KdTreeNode * parent;
}
```

```
struct KdTree
{
   KdTreeNode * root;
}
```

```
KdTreeConstruct(S, d)
{
   T = new KdTree;
   T->root = KdTreeNodeConstruct(S, 0, d);
   return T;
}
```

```
KdTreeNodeConstruct(D, dim, d)
{
   if (|D| = 0)  return NULL;
   v = new KdTreeNode;
   v->dim = dim;
   if (|D| = 1)
   {
      v->point = D.Element;
      v->left = NULL;
      v->right = NULL;
      return v;
   }
   v->point = NULL;
   v->split = D.ComputeSplitValue(dim);
   D_<s = D.Left(dim, v->split);
   D_>s = D.Right(dim, v->split);
   j = (dim + 1) mod d;
   v->left = KdTreeNodeConstruct(D_<s, j);
   v->right = KdTreeNodeConstruct(D_>s, j);
   return v;
}
```

Query

- When searching for points inside given d-dimensional interval B, we are working with areas representing each node of k-d tree (Q)

```
KdTreeNodeQuery(v, Q, B)
{
   List L;
   if (v->IsLeaf() &&  (v->point in B))
   {
      L.add(v->point);
      return L;
   }
   v_l := v->left;
   v_r := v->right;
   Q_l := Q.LeftPart(v->dim, v->split);
   Q_r := Q.RightPart(v->dim, v->split);
   if (Q_l in B)
      L.add(Report(v->left));
   else if (Q_l ∩ B != 0)
      L.add(KdTreeQuery(v->left,Q_l ,B));
   if (Q_r in B)
      L.add(Report(v->right));
   else if (Q_r ∩ B ! = 0)
      L.add(KdTreeQuery(v->right, Q_r, B));
   return L;
}
```

```
Report(v)
{
   List L;
   if (v->IsLeaf() &&  (v->point))
   {
      L.add(v->point);
      return L;
   }
   L.add(Report(v->left));
   L.add(Report(v->right));
   return L;
}
```

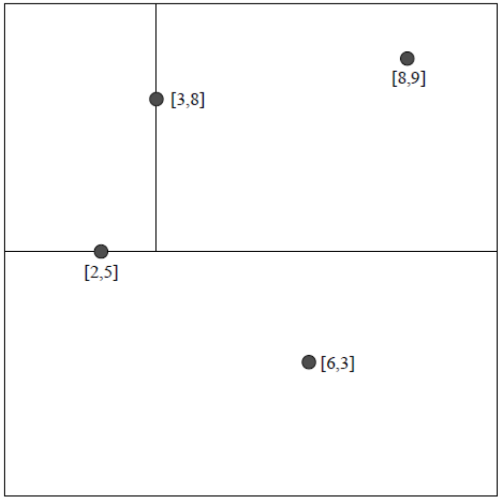
```
KdTreeQuery(T, B)
{
   Q = WholeSpace();
   return KdTreeNodeQuery(T->root, Q, B);
}
```

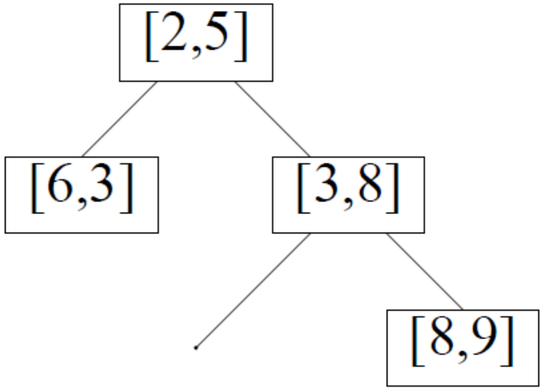
K-d tree properties

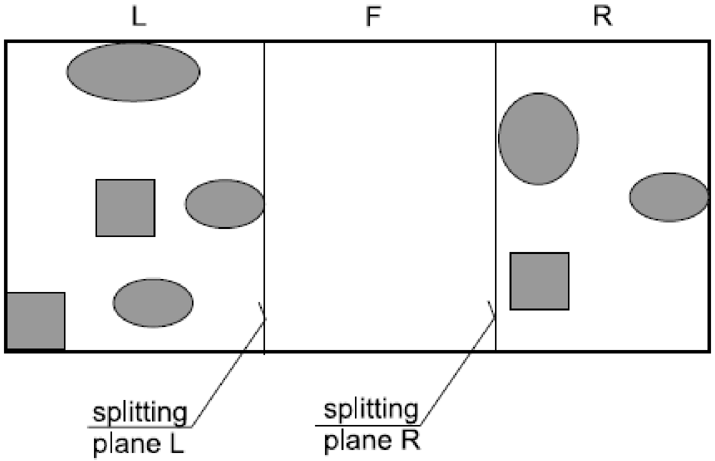
- If split sets $D_{<s_i}$, $D_{>s_i}$ are almost equal (using for example median), then tree is balanced
- Balanced k-d tree in R^d can be constructed in time $O(n.\log(n))$ with memory complexity $O(n)$
- Query for searching using k-d tree in R^d hjas time complexity $O(n^{(1-1/d)} + k)$, where k is cardinality of output
- High time complexity in worst cases (bad divide), expected complexity is $O(\log(n) + k)$
- Point insertion: $O(\log(n))$
- Point removal: $O(\log(n))$

K-d trees

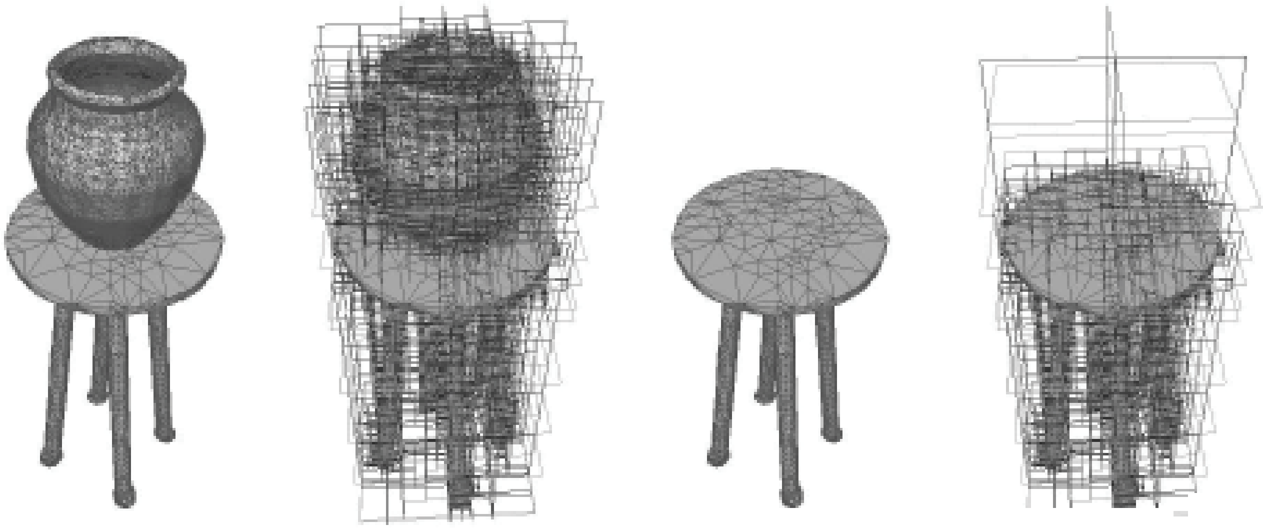
- Trees variations: points stored not only in leafs, non-periodic change od split hyperplane, different ways for split and termination, two split hyperplanes



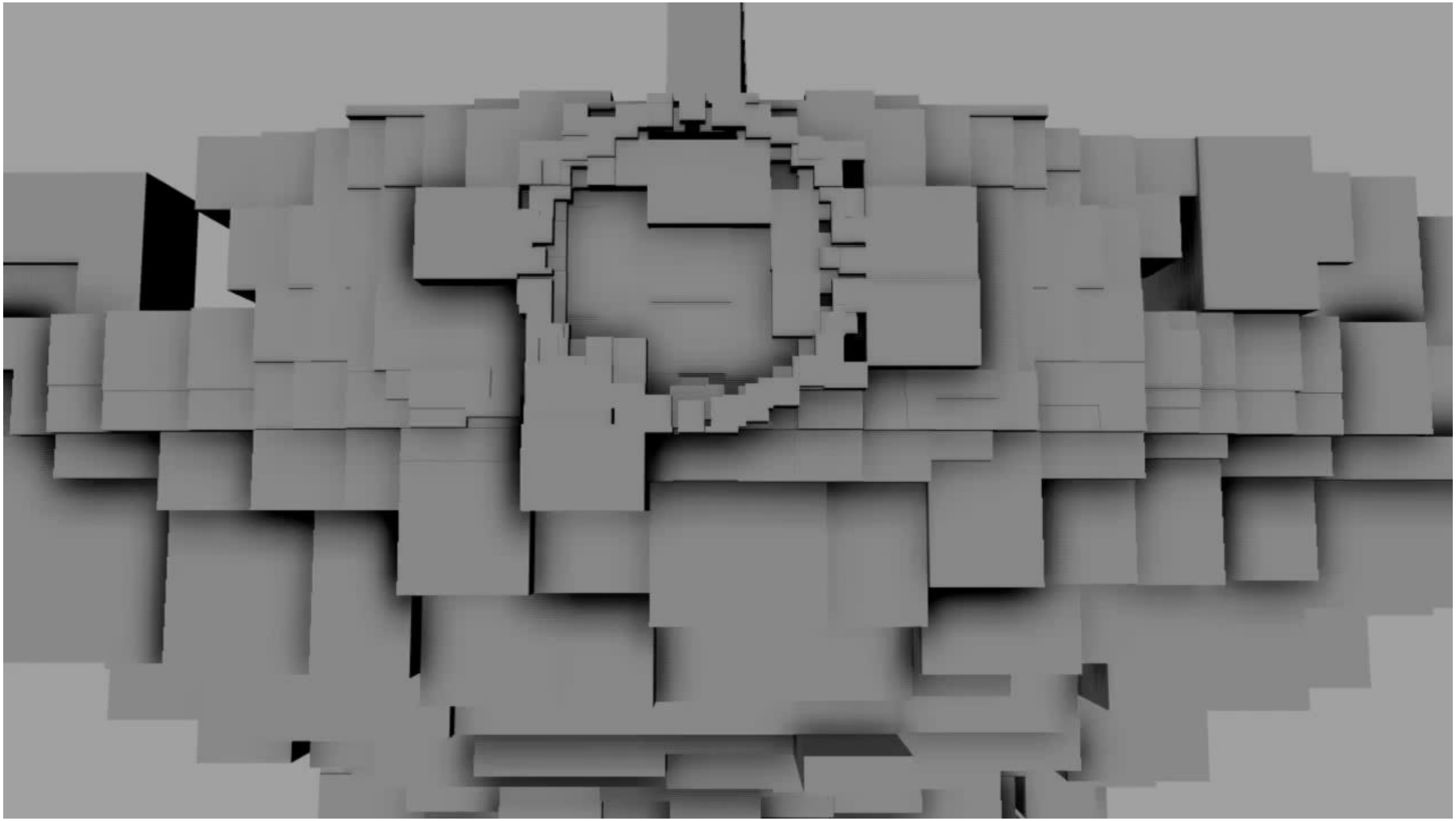




K-d trees

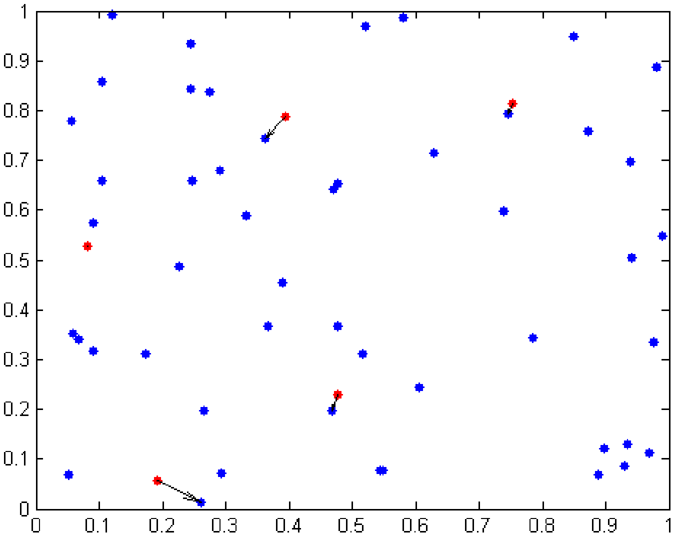


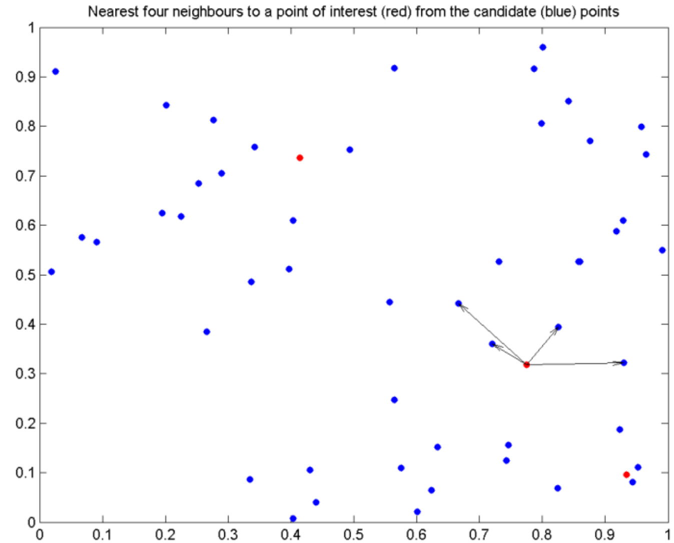
K-d trees



Nearest neighbor search

- For set of points *S* from R^d and one other point *P* from R^d, find one point *Q* from *S* such that distance $|PQ|$ is minimal
- Extension – find *k* nearest neighbors, or alternatively *k* approximate nearest neighbors





Nearest neighbor search

```
SearchSubtree(v, nearest_node, P)
{
   List nodes;
   nodes.Add(v);
   current_nearest = nearest_node;
   while (nodes.size() > 0)
   {
      current_node = nodes.PopFirst();
      if (current_node->IsLeaf() &&  (current_node->point))
      {
          if (Distance(current_node->point, P) < Distance(current_nearest->point, P))
             current_nearest = current_node;
          continue;
      }
      hyperplane_distance = Distance(P, current_node->dim, current_node->split);
      if (hyperplane_distance > Distance(current_nearest->point, P))
      {
          if (InLeftPart(P, current_node->dim, current_node->split)) nodes.Add(current_node->left);
          else nodes.Add(current_node->right);
      }
      else
      {
          nodes.Add(current_node->left);
          nodes.Add(current_node->right);
      }
   }
   return current_nearest;
}
```

```
FindNearestPoint(T, near_node, P)
{
   nearest_node = current_node = near_node;
   while (current_node != T->root)
   {
      hyperplane_distance = Distance(P, current_node->parent->dim, current_node->parent->split);
      if (hyperplane_distance < Distance(P, nearest_node))
      {
         if (current_node == current_node->parent->left)
            nearest_node = SearchSubtree(current_node->parent->right, nearest_node, P);
         else
            nearest_node = SearchSubtree(current_node->parent->left, nearest_node, P);
      }
      current_node = current_node->parent;
   }; return nearest_node;
}
```

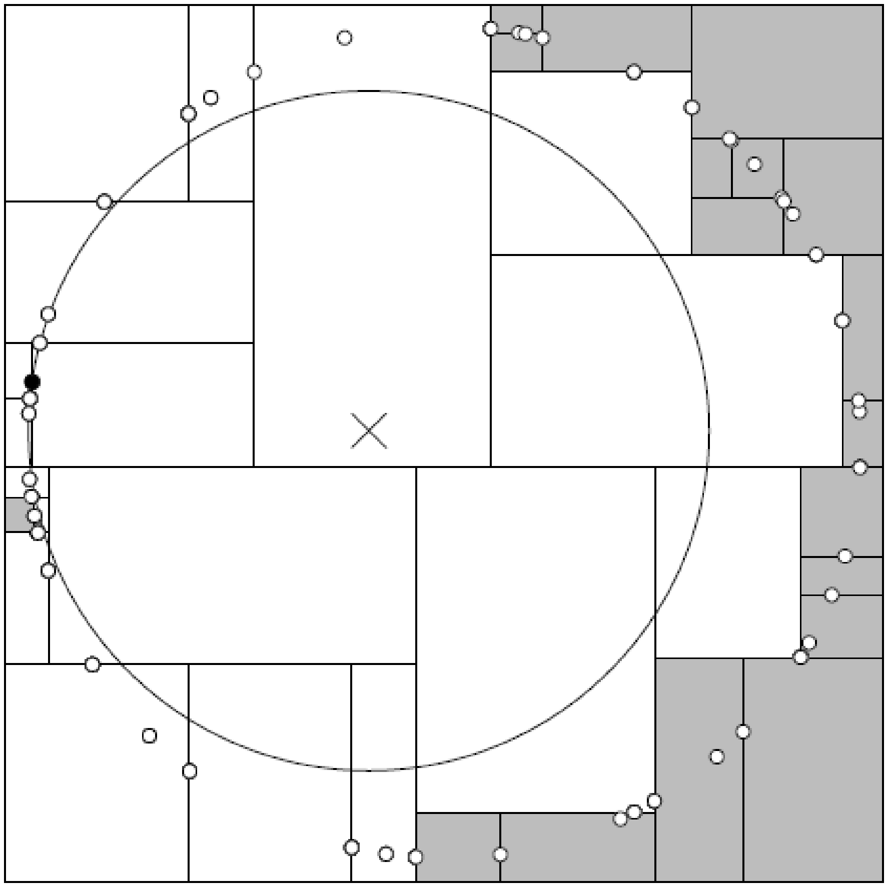
```
FindNearPoint(v, P)
{
   if (v->IsLeaf() &&  (v->point))
      return v;
   if (InLeftPart(P, v->dim, v->split))
      return FindNearPoint(v->left, P);
   else
      return FindNearPoint(v->right, P);
}
```

```
KdTreeNearestNeighbor(T, P)
{
   near = FindNearPoint(T->root, P);
   return FindNearestPoint(T, near, P);
}
```

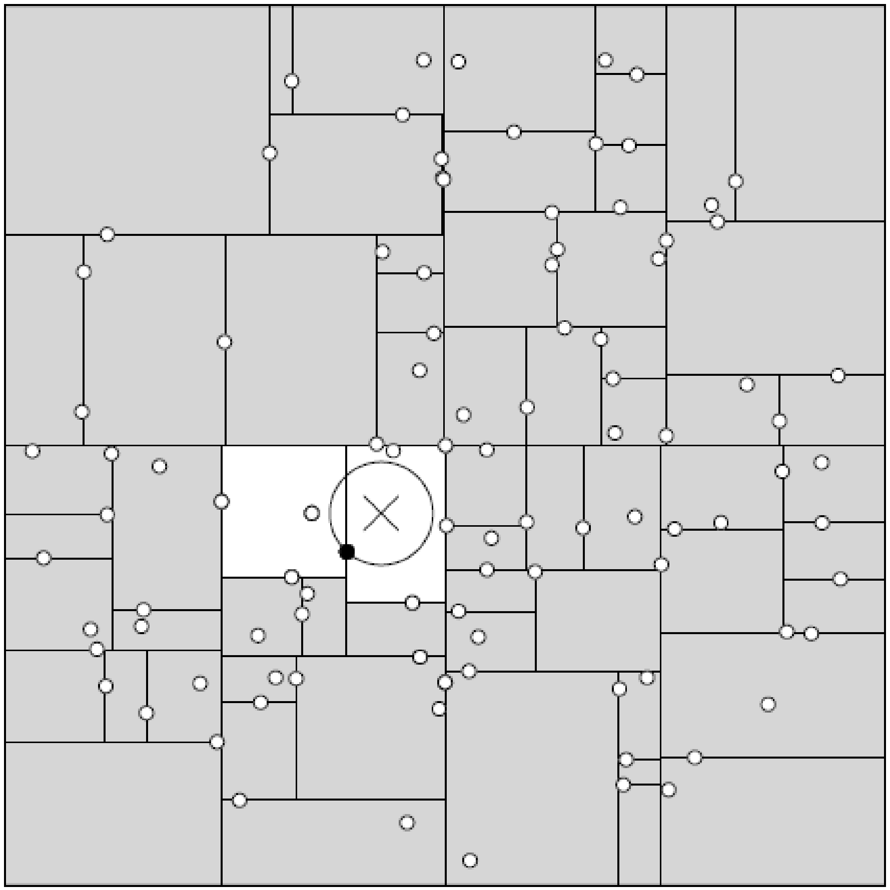
Nearest neighbor search

- First step: find node (leaf of k-d tree) containig point that is near to point P
- Second step: From this leaf, traverse tree back to root and search for nearer points stored in opposite subtrees
- Time complexity: $O(d.\ n^{(1-1/d)})$
- For random distribution of points, expected time complexity is $O(\log(n))$
- http://dl.acm.org/citation.cfm?id=355745
- http://dimacs.rutgers.edu/Workshops/MiningTutorial/pindyk-slides.ppt
- http://www.ri.cmu.edu/pub_files/pub1/moore_andrew_1991_1/moore_andrew_1991_1.pdf

Nearest neighbor search



Mnoho navštívených vrcholov



Málo navštívených vrcholov

k nearest neighbors

- Extension of previous algorithm
- Instead of sphere with 1 actually nearest point, we have sphere containing *k* actually nearest points
- If the sphere in one moment contains less than *k* points, its radius is infinite
- In first step, we find *k* potentially nearest points instead of 1

k nearest neighbor search

```
Struct SearchRecord
{
   vector<KdTreeNode> points;
   float radius;
}
```

```
KdTreeNearestNeighbors(T, P, k)
{
   SearchRecord result;
   FindNearPoints(T->root, P, k, &result);
   FindNearestPoints(T, P, k, &result);
   return result;
}
```

```
FindNearPoints(v, P, k, result)
{
   if (v->IsLeaf() &&  (v->point))
   {
      result->points.Add(v);
      result->UpdateRadius(P);
      return;
   }
   if (InLeftPart(P, v->dim, v->split))
   {
      FindNearPoints(v->left, P, k, result);
      if (result->points.size < k)
         FindNearPoints(v->right, P, k, result);
   }
   else
   {
      FindNearPoints(v->right, P, k, result);
      if (result->points.size < k)
         FindNearPoints(v->left, P, k, result);
   }
}
```

k nearest neighbor search

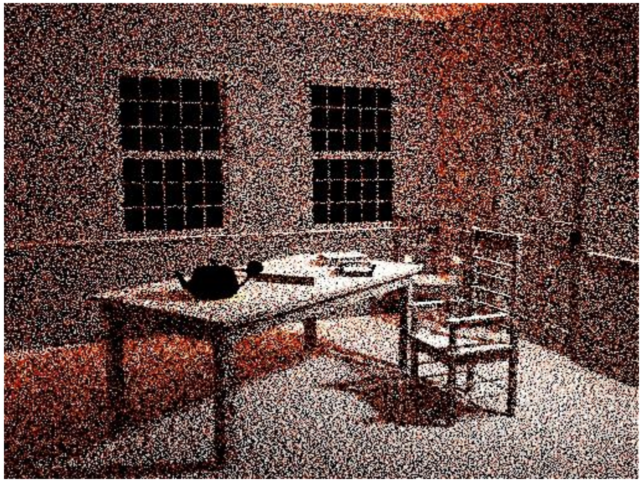
```
FindNearestPoints(T, P, k, result)
{
   current_node = result->points[0];
   while (current_node != T->root)
   {
      hyperplane_distance = Distance(P, current_node->parent->dim,
                                                 current_node->parent->split);
      if (hyperplane_distance < result->radius)
      {
         if (current_node == current_node->parent->left)
            SearchSubtree(current_node->parent->right, P, k, result);
         else
            SearchSubtree(current_node->parent->left, P, k, result);
       }
       current_node = current_node->parent;
   };
   return;
}
```

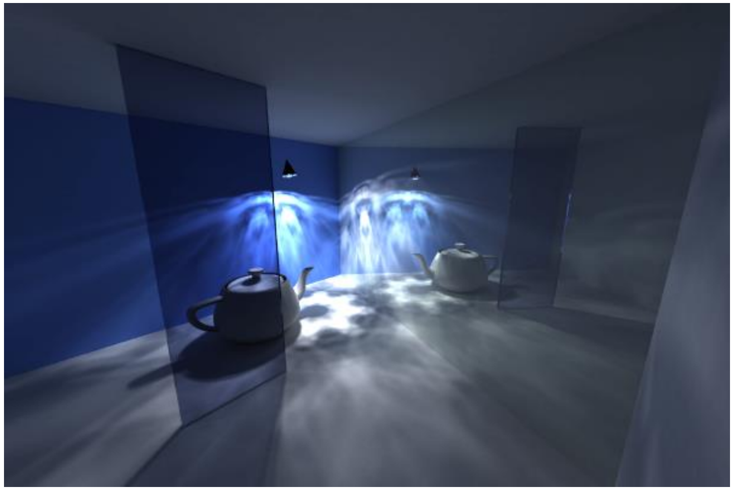
```
SearchSubtree(v, P, k, result)
{
   List nodes;
   nodes.Add(v);

   while (nodes.size() > 0)
   {
      current_node = nodes.PopFirst();
      if (current_node->IsLeaf() &&  (current_node->point))
      {
          if (Distance(current_node->point, P) < result->radius)
          {
             result->points.AddNewAndRemove(current_node, P, k);
             result->UpdateRadius(P);
          }
          continue;
      }
      hyperplane_distance = Distance(P, current_node->dim, current_node->split);
      if (hyperplane_distance > result->radius)
      {
          if (InLeftPart(P, current_node->dim, current_node->split))
              nodes.Add(current_node->left);
          else
              nodes.Add(current_node->right);
      }
      else
      {
          nodes.Add(current_node->left);
          nodes.Add(current_node->right);
       }
   }
   return;
}
```

Photon mapping

- 1. pass:
  - Shooting photons from light source in random directions
  - Computing intersections and bounces of photons in scene
  - Storing intersection points in map
- 2. pass:
  - Rendering from camera
  - Using light data from map (searching for k closest photons in map from surface point) for global illumination computation
- Photon map structure = k-d tree



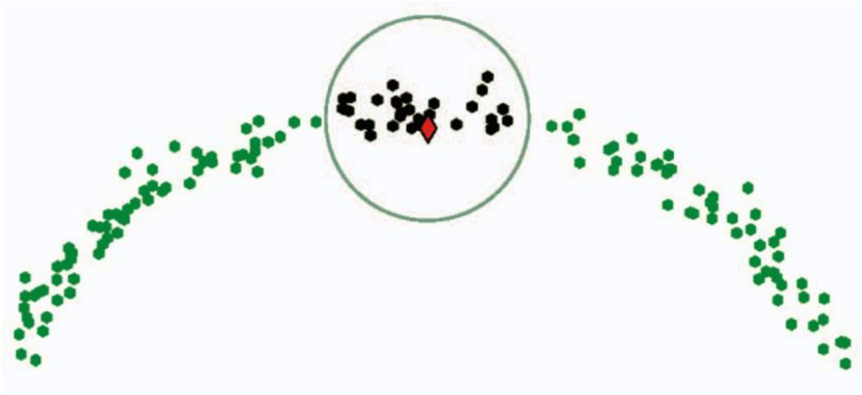


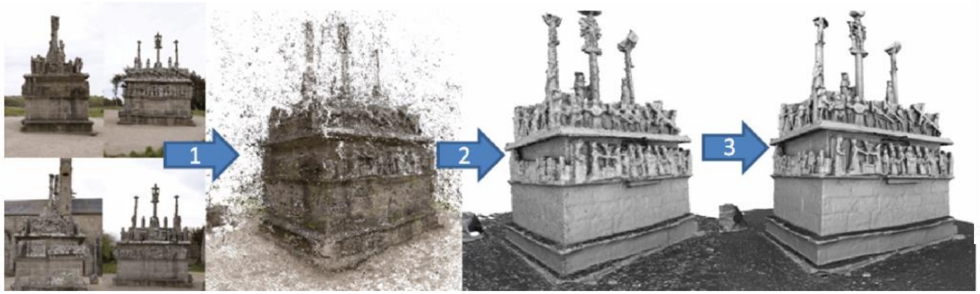
Point clouds

- Many construction possibilities: laser scanning, Kinect, structured light, …
- Surface reconstruction – find continuous surface based on points

Surface reconstruction

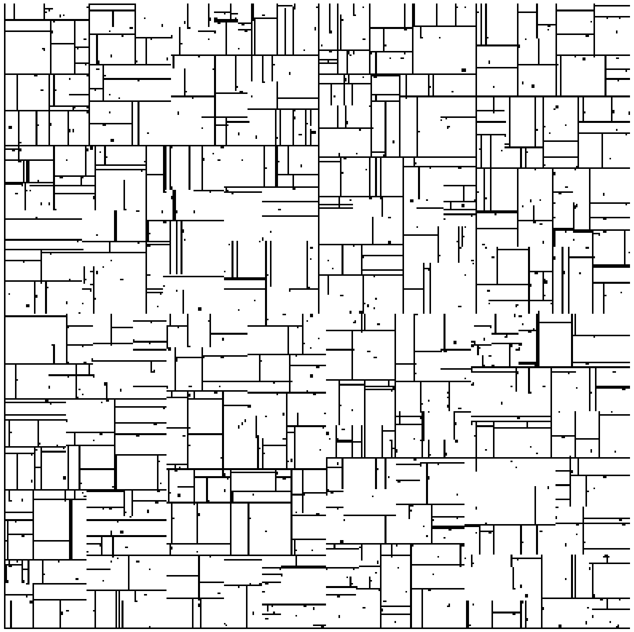
- Searching for points inside given sphere
- Small modification of nearest neighbor search

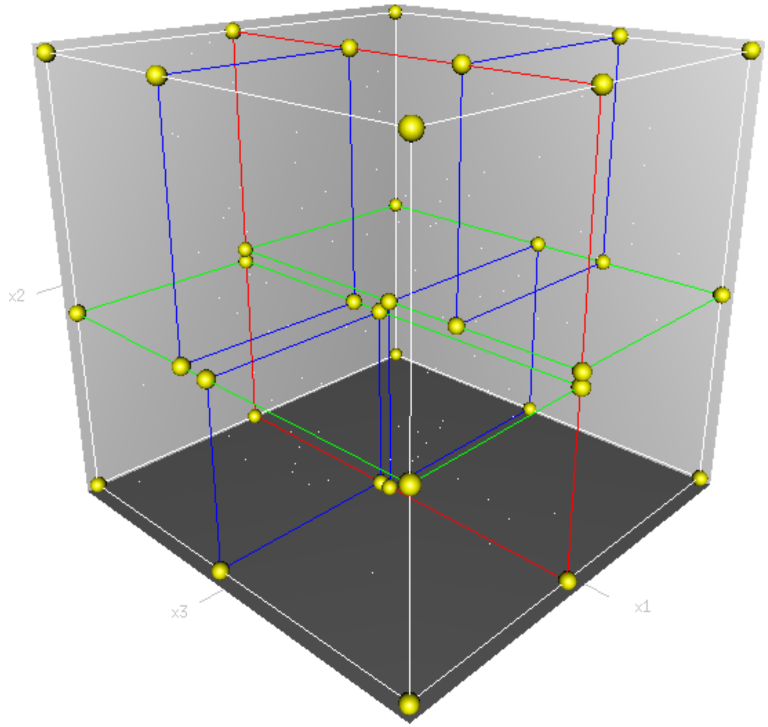




Database

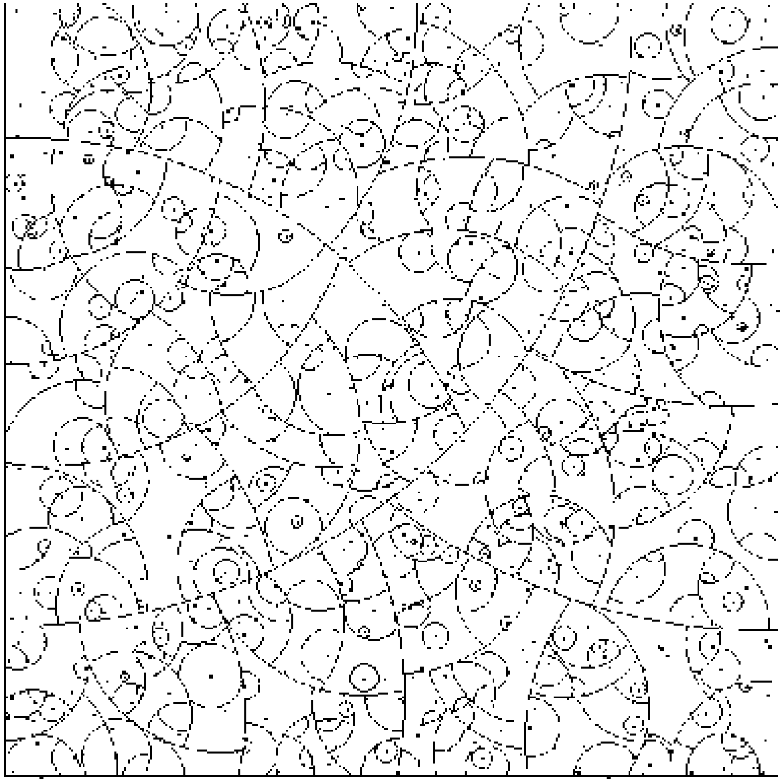
- Record in database = d-dimensional vector
- For input record, find most similar record in database = nearest neighbor search





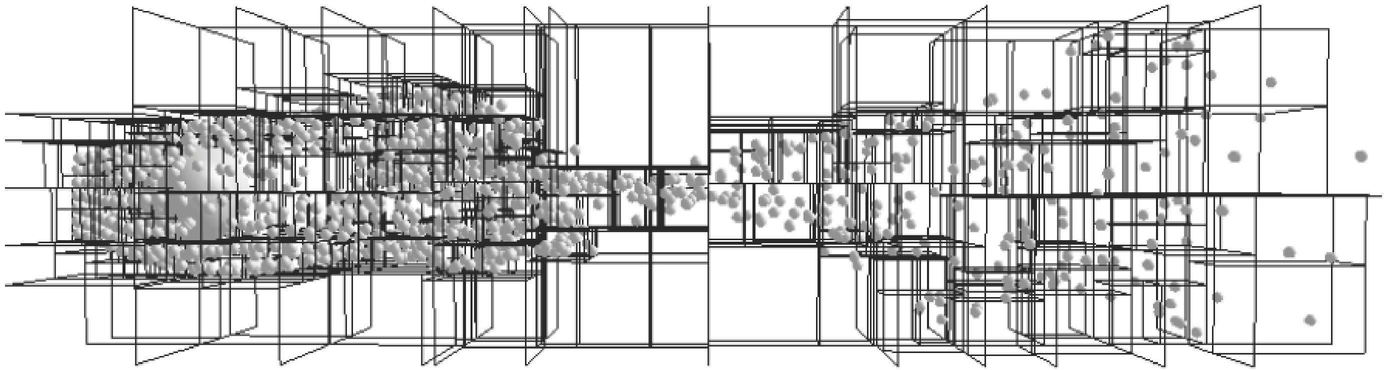
vp (Vantage-point) tree

- Binary tree, each node contains center point **P** and radius *r*, in left subtree are points with distance **P** less than *r*, in right subtree are all other points
- Pick **P** – random point
- Pick *r* – median of distances of **P** and all other points



Raytracing

- K-d tree is best space partition for raytracing (minimalizing ray-object intersection count)
- http://dcgi.felk.cvut.cz/home/havran/phdthesis.html
- Adaptively divide based on surface
- Traversing structure along ray



Raytracing

- Divide in node
  - Spatial median
  - Object median
  - Direction – largest variation
  - In center in direction of "longest" dimension
  - Cost techniques
    - Computing split cost based on ray-area hit probability (ordinary surface area heuristic)

$$C_{\nu^G} = \frac{1}{SA(\mathcal{AB}(\nu^G))}[SA(\mathcal{AB}(lchild(\nu^G))).(N_L + N_{SP}) + SA(\mathcal{AB}(rchild(\nu^G))).(N_R + N_{SP})],$$

Questions?