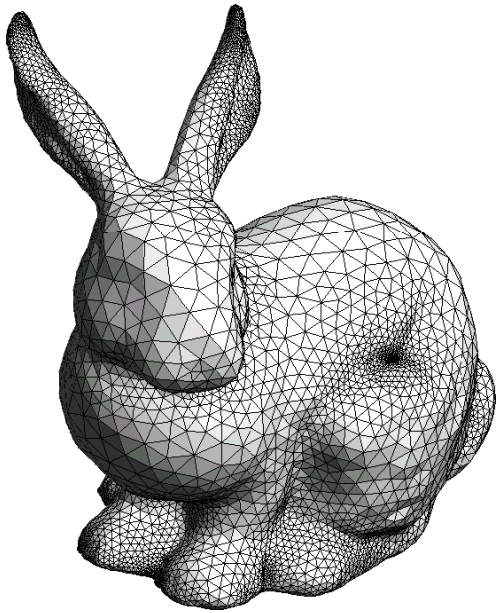


Geometric Modeling in Graphics

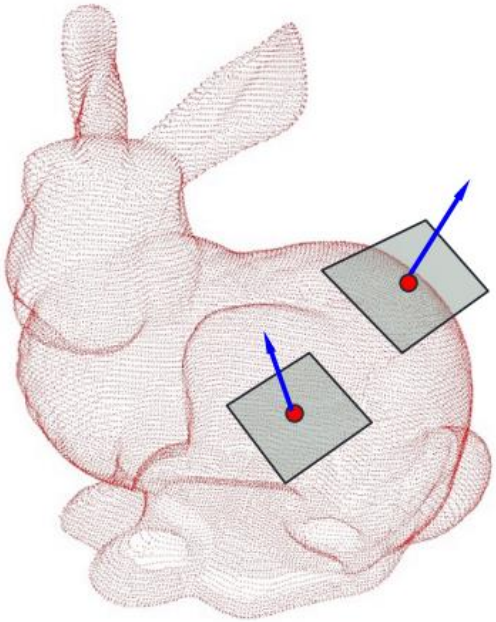
Part 9: Point Clouds

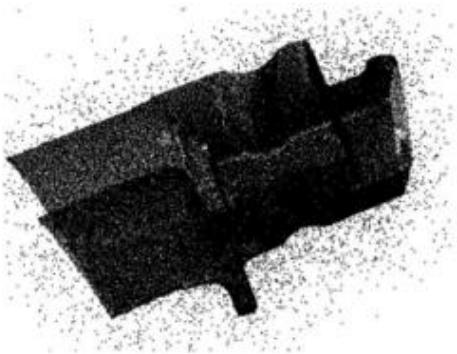
vis gravis
visualization, graphics & vision

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

Point cloud

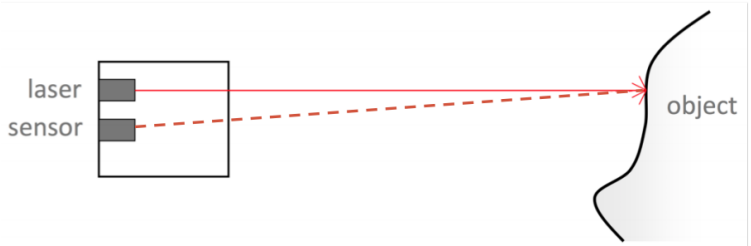
- Unorganized set of points in E^2, E^3
- Simplest representation, no connectivity, no topology
- Optional attributes: color, normal, intensity, …
- Introducing noise and outliers



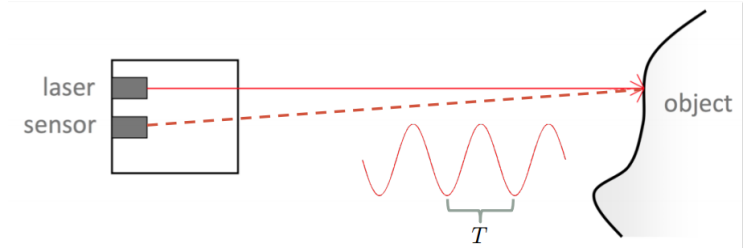


Point cloud sources

- Output from majority of 3D scanners
- Time-of-flight laser scanner
  - Compute point depth and position from travel time of laser beam



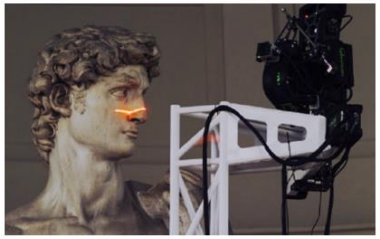
- Phase-based laser scanner
  - Compute point depth and position from phase shift of laser beam

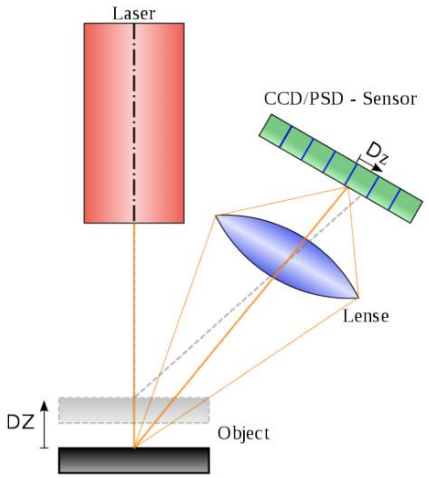


http://www.enseignement.polytechnique.fr/informatique/INF555/Slides/lecture7.pdf

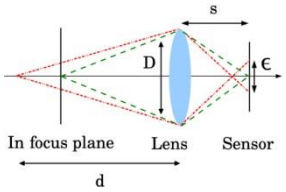
Point cloud sources

- Kinematic depth recovery
  - Recover depth from shift in photometric sensor

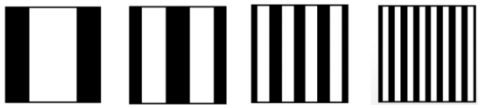




- Depth from blur
  - Estimate depth from blurriness of part of image

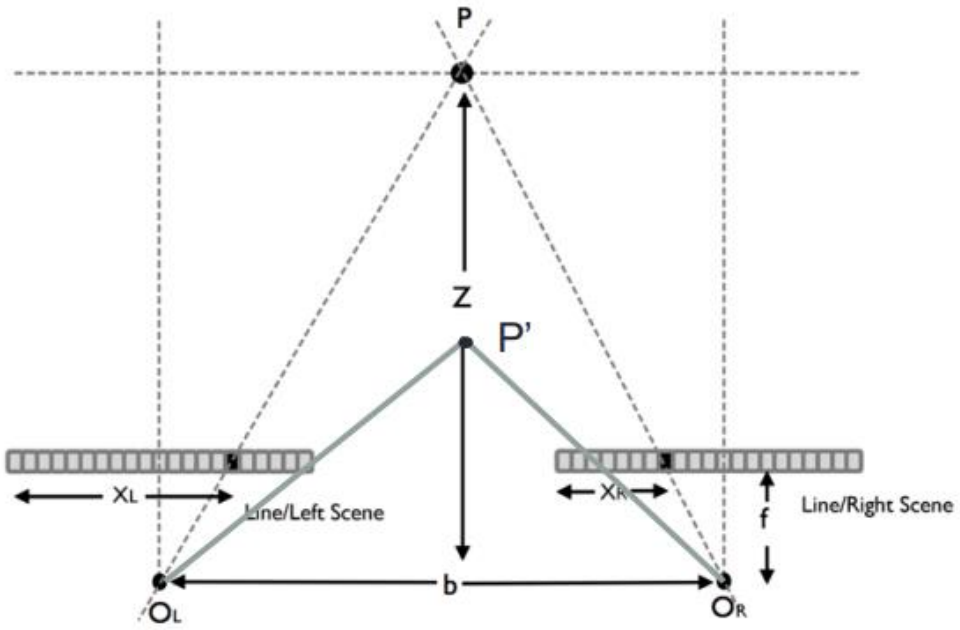


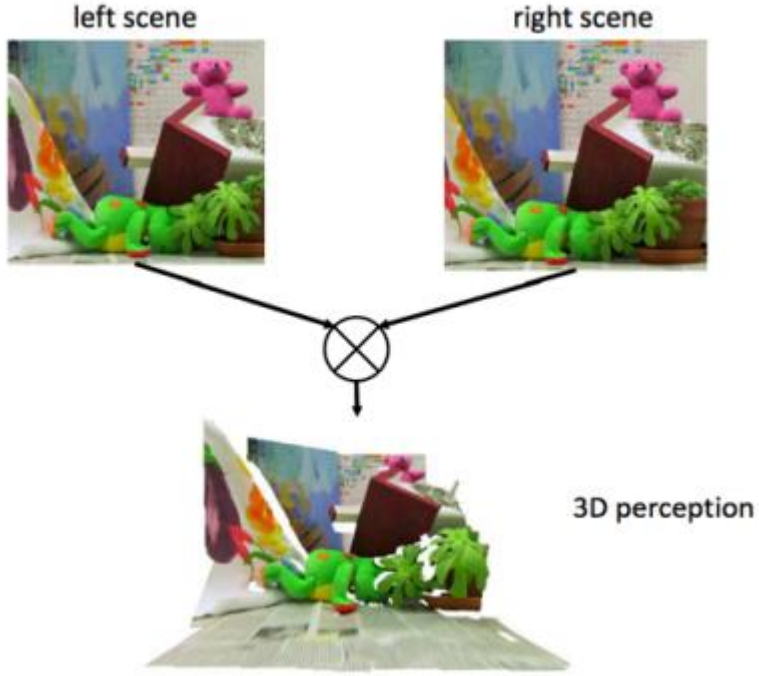
- Structured light scanner
  - Project different regular binary patterns on object
  - Estimate depth from recorded binary code
  - Using projector-camera correspondence
  - http://www.photoneo.com/



Point cloud sources

- Passive stereo
  - Use photogrammetry to reconstruct points from two or more images
  - Computing and using camera-camera correspondence
  - Finding feature corresponding points in images
  - https://www.capturingreality.com/





Point cloud sources

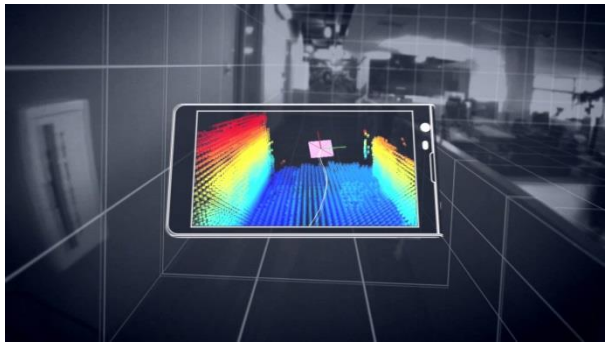
- Physical simulations, Eulerian, particle systems
- Fluid simulation

- Fracturing solids

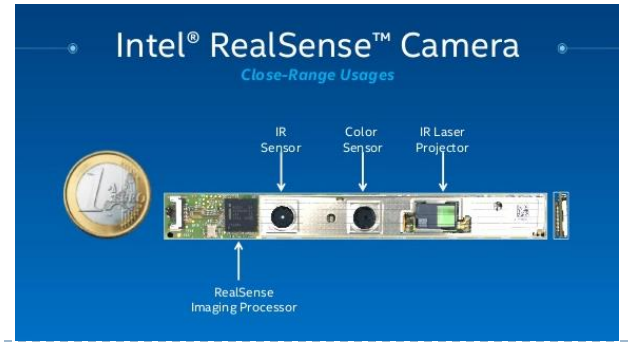
Point cloud sources

- Kinect
  - Low cost 3D scanner
  - V1 - 640 x 480 image and 77k 3D points at 30 FPS
  - V2 – 1080p image and 217k 3D points at 30 FPS

- Google Project Tango



- Intel RealSense

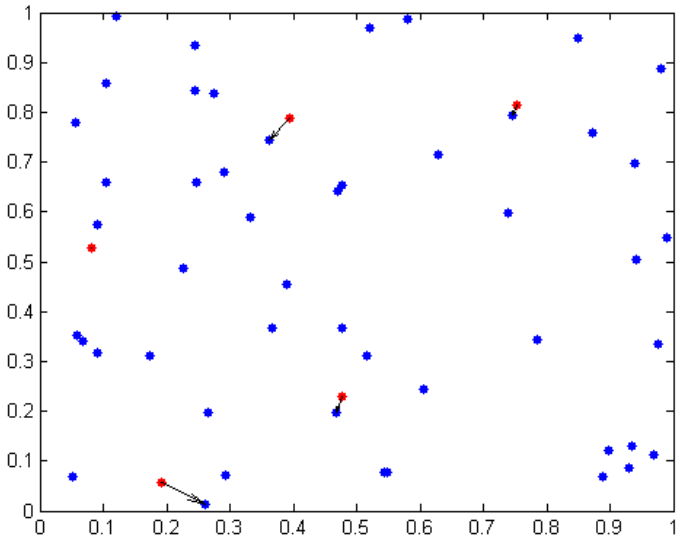


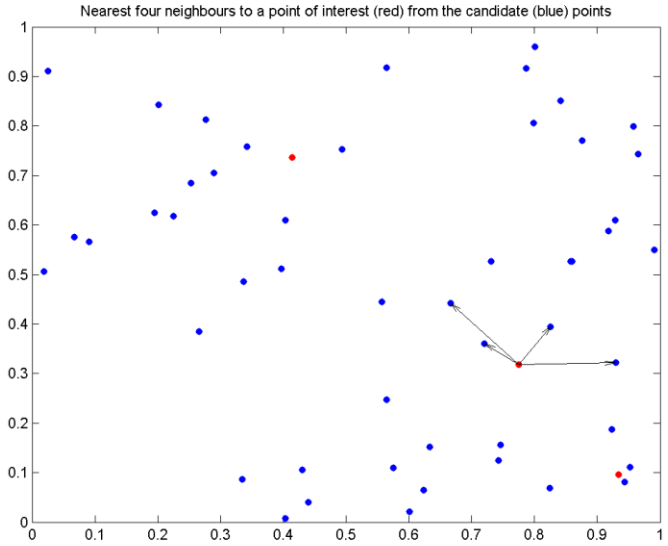
Point cloud processing

- Nearest neighbour search
- Outlier removal
- Normals estimation, orientation
- Registration, matching
- Smoothing, simplification
- Visualization
- Curve fitting, Surface reconstruction
- Point Cloud Library (PCL)
  - http://pointclouds.org/

Nearest neighbor

- For given set of points $\boldsymbol{S} = \{X; X \in E^2, E^3\}$and one arbitrary point P, find point $Q \in \boldsymbol{S}$ such that distance $d(P,Q)$ is minimal, $d(P,Q) = \min_{X \in \boldsymbol{S}} d(P,X)$
- Find k-nearest neighbors to point P
- Find all neighbors to point P closer then radius r

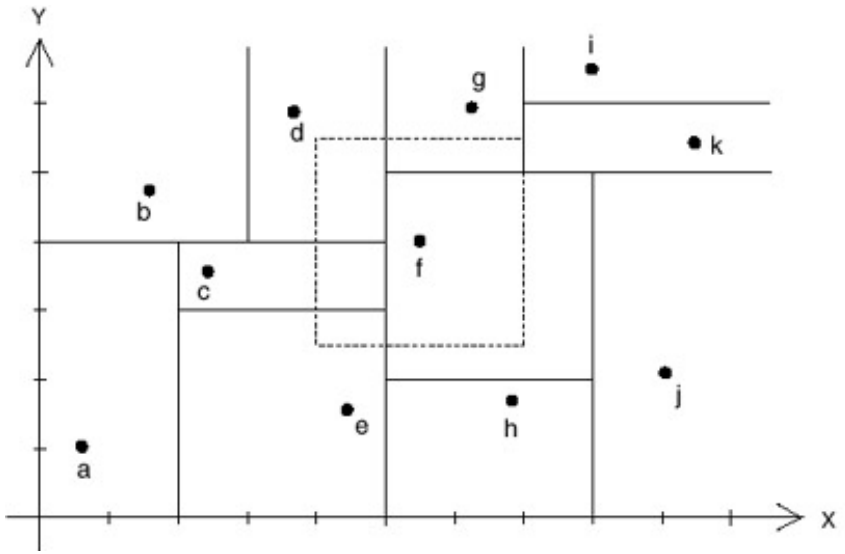




K-d tree

- Hierarchical space partitioning
- Represented as binary tree
- Each node of tree correspond to one splitting hyperplane parallel to coordinate axle
- All points from left subtree of node are on the left side of node hyperplane, all points from right subtree of node are on the right side of node hyperplane
- Each subtree of tree correspond to convex region
- Given points can be stored in each node or only in leafs
- Splitting hyperplane divides points in that subtree on two almost equally sized subsets

K-d tree construction

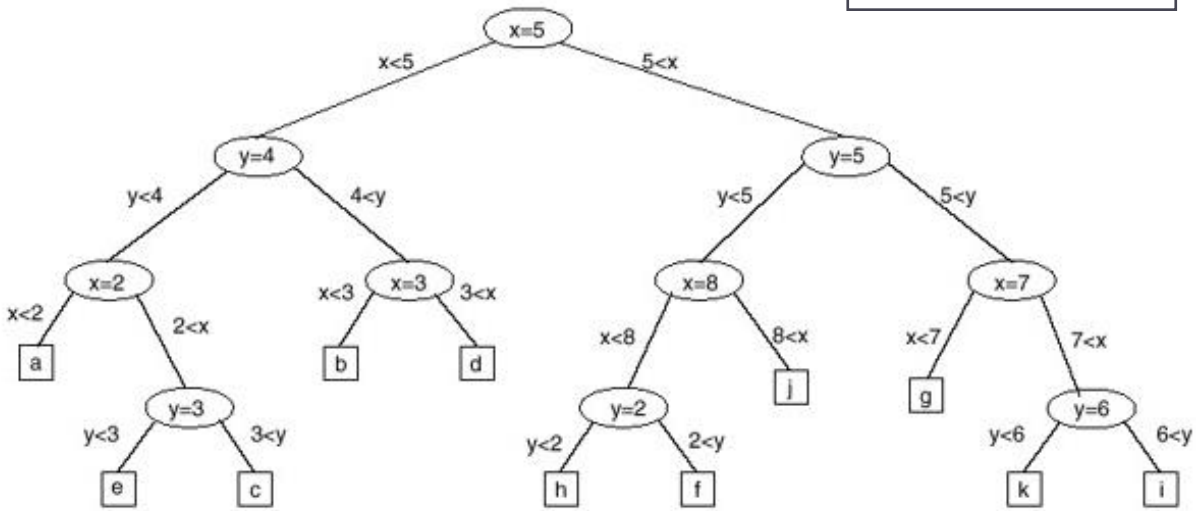



```
struct KdTreeNode
{
   float  split;
   int dim;
   Point* point;
   KdTreeNode * left;
   KdTreeNode * right;
   KdTreeNode * parent;
}
```

```
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;
}
```



Nearest neighbor search

```
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;
}
```

```
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;
}
```

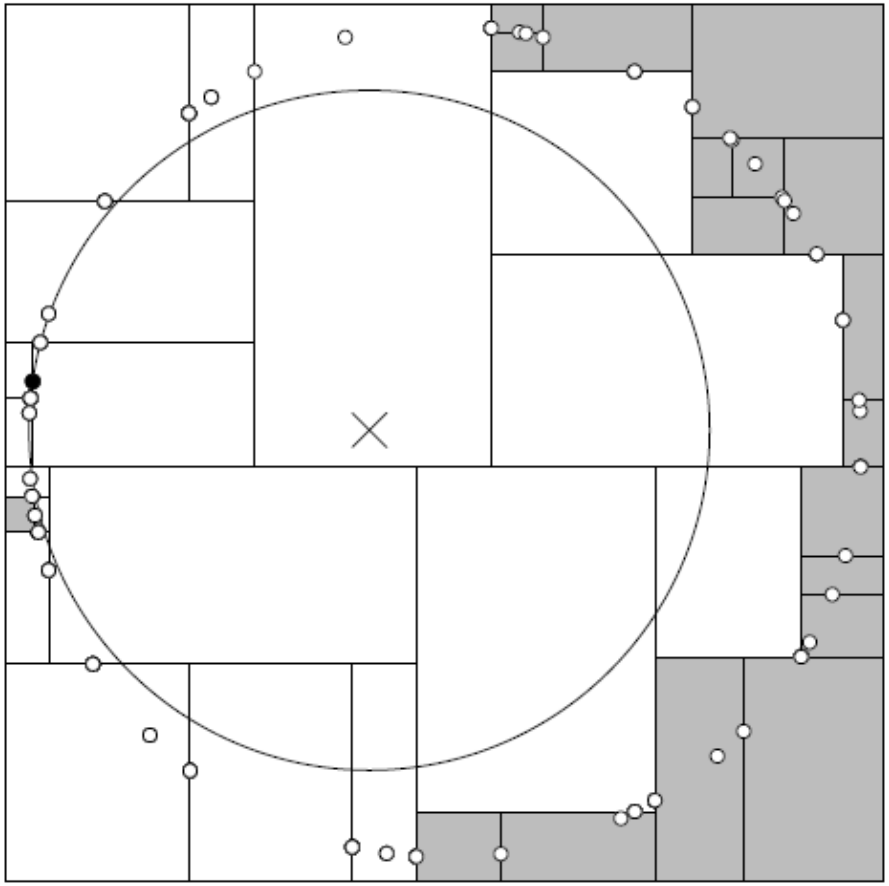
```
FindNearPoint(v, P)
{
   if (v->IsLeaf() &&  (v->point))
      return v;
   if (InLeftPart(P, v->dim, v->split))
      return FindNearPoint(v->left, P);
   else
      return FindNearPoint(v->right, P);
}
```

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

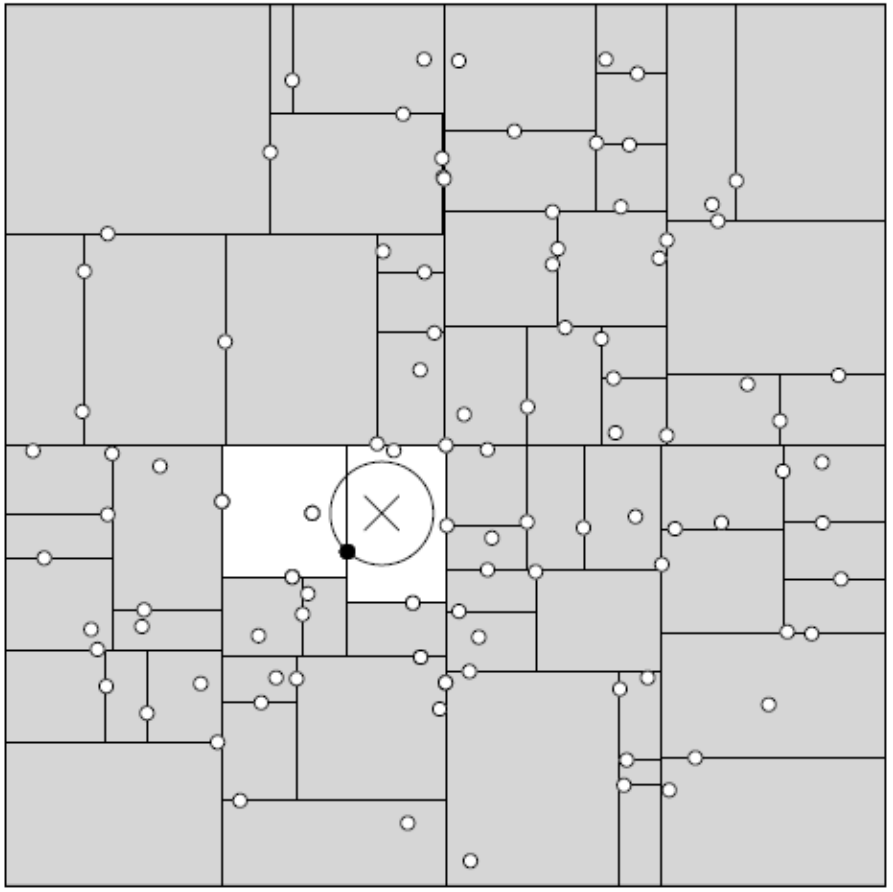
Nearest neighbor search

- Initially find leaf of k-d tree where given point P is positioned – point in leaf is first candidate
- Tracing back from leaf to root and trying to find closer point from $\boldsymbol{S}$ in opposite subtrees
- Time complexity: $O(d.n^{1-\frac{1}{d}})$
- Time complexity for random distribution: $O(\log(n))$
- 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



Many visited tree nodes



Few visited tree nodes

k-nearest neighbors search

- Similar to previous algorithm
- Storing sphere containing temporary set of k-near points from S
- Two traversals of tree

```
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 neighbors 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;
}
```

Radius neighbors search

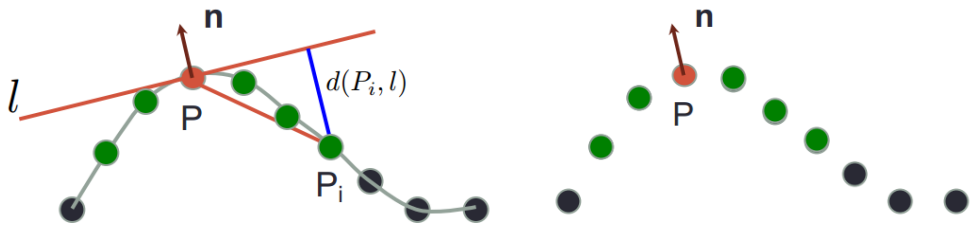
- Finding all points inside sphere with center P and radius r
- One traversal of tree

```
KdTreeRadiusNearestNeighbors(T, P, r)
{
   return RadiusSearch(T->root, P, k, &result);
}
```

```
RadiusSearch(v, P, r, result)
{
   List result;
   if (v->IsLeaf() &&  (v->point))
   {
      if (Distance(P, v->point) < r)
         result.Add(v);
   }
   else if (InLeftPart(P, v->dim, v->split))
   {
      result.Add(RadiusSearch(v->left, P, r));
      hyperplane_distance = Distance(P, v->dim, v->split);
      if (hyperplane_distance < r)
         result.Add(RadiusSearch(v->right, P, r));
   }
   else
   {
      result.Add(RadiusSearch(v->right, P, r));
      hyperplane_distance = Distance(P, v->dim, v->split);
      if (hyperplane_distance < r)
         result.Add(RadiusSearch(v->left, P, r));
   }
   return result;
}
```

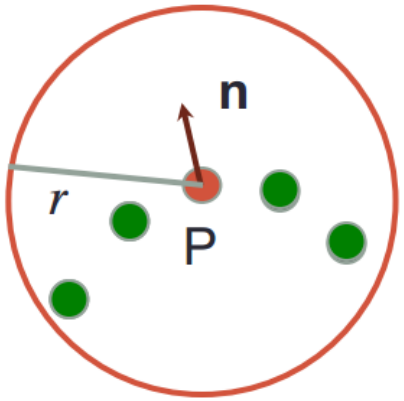
Normals estimation

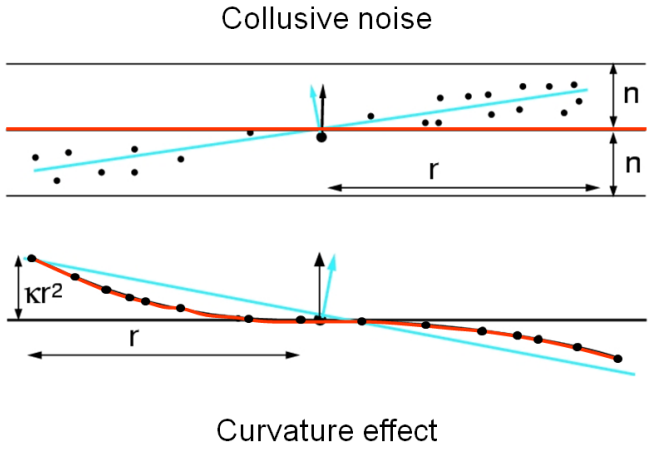
- Estimating normal for each point P from point cloud $\boldsymbol{S}$
- Choosing k-nearest neighbors of P
- Fitting line or plane that it minimizes sum of squared distances from k neighbors to line or plane, normal is vector perpendicular to line or plane
- Principal Component Analysis (PCA)
- Computing covariance matrix of k-points, use P or barycenter of k neighbors as mean position
- Normal is eigenvector corresponding to smallest eigenvalue



Normals estimation

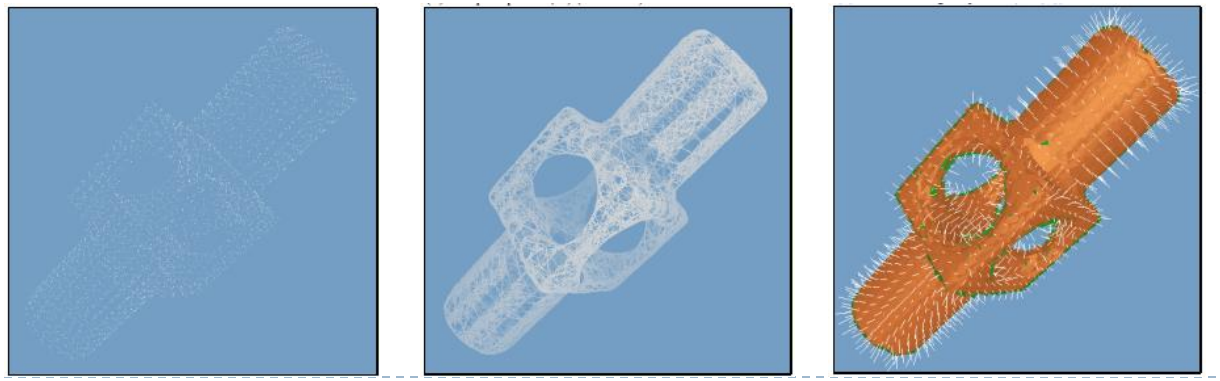
- Critical parameter k
- Due to uneven distribution of points, using radius search with distance r
- Automatic computation of radius r based on sampling density, distribution, local curvature, …
- http://vecg.cs.ucl.ac.uk/Projects/SmartGeometry/normal_est/paper_docs/normal_estimation_socg_03.ppt





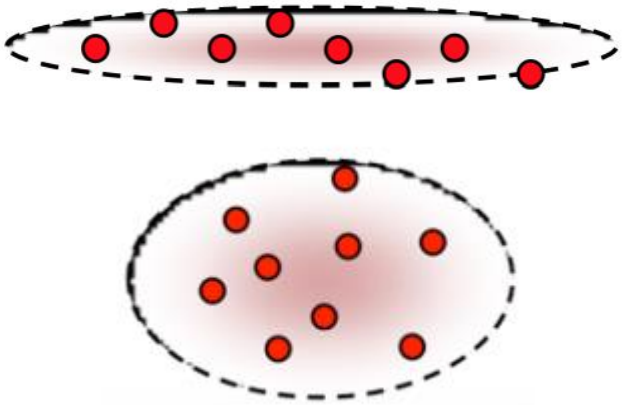
Normals orientation

- http://research.microsoft.com/en-us/um/people/hoppe/recon.pdf
- Set normal of point with maximal z coordinate as consistent pointing in +z axis
- Propagate good normal orientation from consistent to non-consistent points, flipping normals if necessary
- Two points are connected for propagation if are close enough – Riemann graph



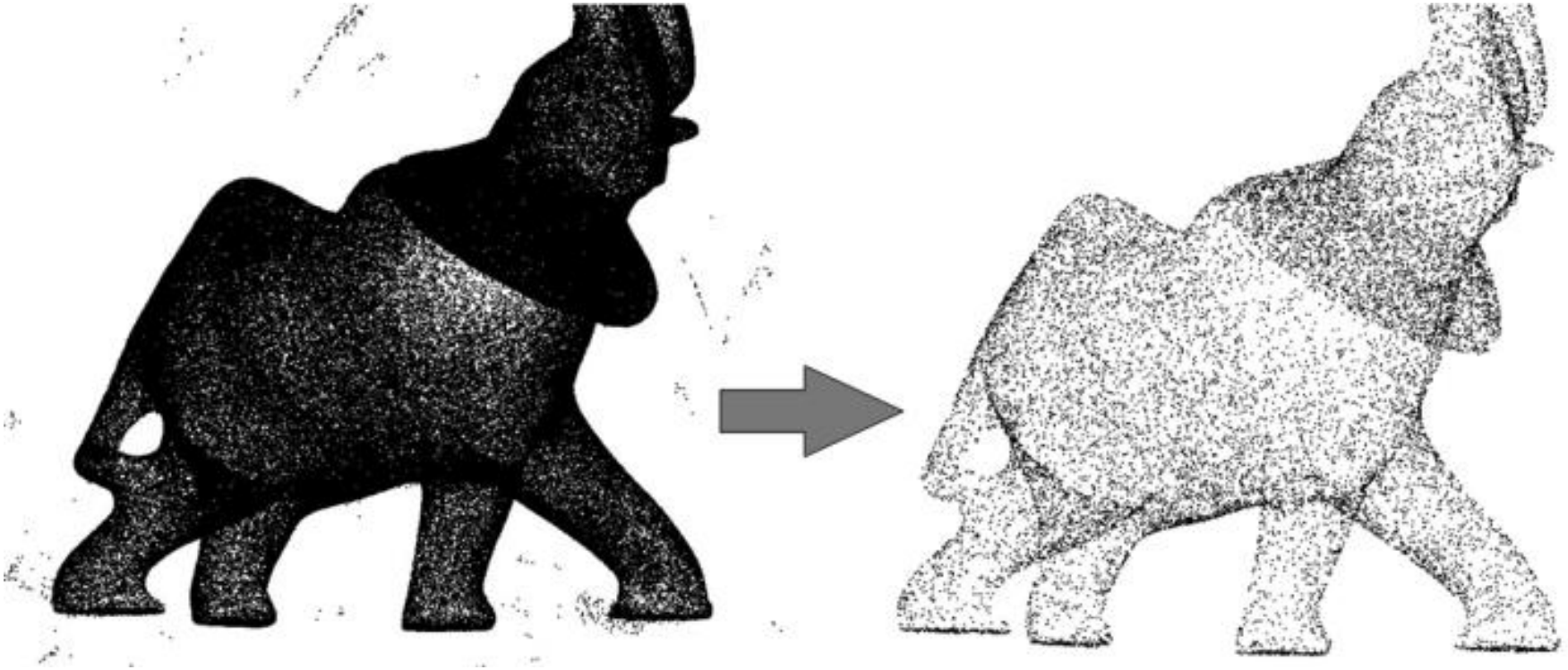
Outlier removal

- Remove points that do not lie close to a surface
- Checking for each point P
- Assumption - surface is almost plane in close vicinity of P
- Compute bounding box as in normal estimation using PCA
- Flatten bounding box in normal direction using parameter
- Remove points outside flattened box



Outlier removal

- Sorting input points in increasing order of average squared distances to their k-nearest neighbors
- Deleting points with largest value



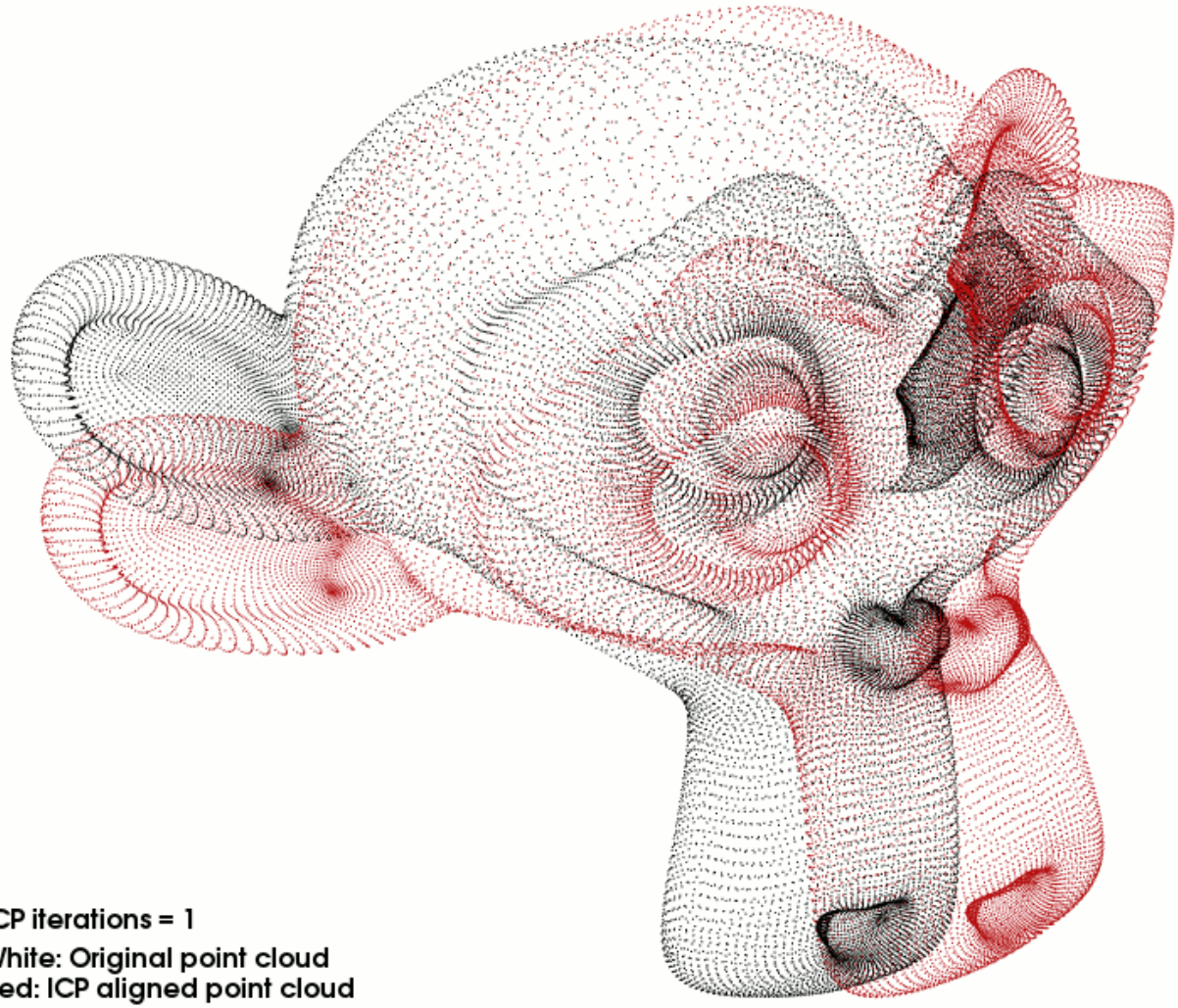
Registration

- Aligning two point clouds
- Registering both clouds in one space
- Minimalizing distance between points of two clouds

Iterative Closest Point

- Finding transformation from space of target point cloud to space of reference point cloud
- 1. Compute initial estimation of transformation T
  - Alignment of bounding boxes
- 2. For subset $\boldsymbol{S}$ of transformed points from target point cloud, find the closest point in reference point cloud
  - Choose $\boldsymbol{S}$ as whole reference point cloud
  - Choose $\boldsymbol{S}$ as random subset
  - Choose $\boldsymbol{S}$ as representations from clusterization
  - Choose $\boldsymbol{S}$ from keypoints detection
- 3. Estimate new transformation that minimizes mean squared error of newly transformed points from $\boldsymbol{S}$ and its closest points from reference point cloud
- 3. Repeat 2. until change in transformation is small

Iterative Closest Point

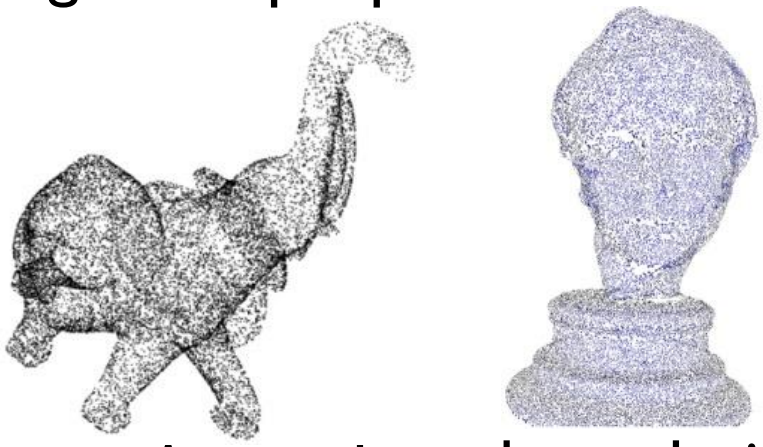


Feature based registration

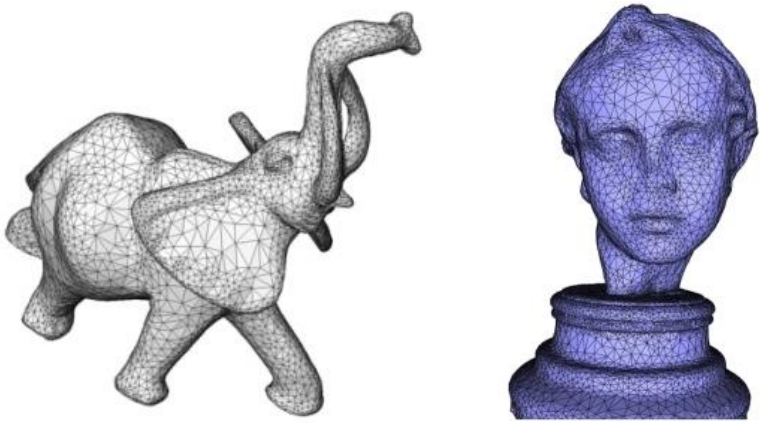
- 1. Detect keypoints, feature points
  - NARF, SIFT, FAST
- 2. Compute feature descriptor vector for each keypoint
  - NARF, FPFH, BRIEF, SIFT
- 3. Find correspondences between keipoints of both point clouds using nearest neighbor search in feature vectors space
- 4. Reject keypoints with bad correspondence
- 5. Estimate transformation that minimizes total distance between keypoints of reference point cloud and transformed keypoints of target point cloud

Point cloud visualization

- Direct rendering of simple points with its attributes



- Surface reconstruction and mesh rendering



Point cloud visualization

- Splats rendering
  - Represent each point as planar disc or ellipse
  - Size and orientation of splat is given by normal and k-neighborhood of point
  - http://graphics.ucsd.edu/~matthias/Papers/HighQualitySplattingOnGPUs.pdf
  - https://www.cg.tuwien.ac.at/research/publications/2012/preiner_2012_AS/preiner_2012_AS-draft.pdf
  - http://graphics.stanford.edu/software/qsplat/

The End

for today