PCL - SURFACE RECONSTRUCTION

TOYOTA CODE SPRINT

Alexandru-Eugen Ichim

Computer Graphics and Geometry Laboratory
• 3D revolution due to cheap RGB-D cameras (Asus Xtion & Microsoft Kinect)

• Affordability comes with poor quality:
  • high level of noise in both the depth and the color images
  • quantization artifacts
  • missing pixels
  • various color image distortions, specific to webcam sensors and optics
Incapability of the Kinect to record transparent or shiny objects
MOTIVATION

- 3D points are sampling surfaces - no dimension, orientation, etc. (see surflets)
- point neighborhoods and features capture some aspects of these surfaces
- in some cases we are interested in the actual surface
- Knowing the real surface enables:
  - more accurate feature extraction
  - smoothing and resampling (hole filling)
  - collision and occlusion checks
  - texture mapping
  - fitting and recognition
DATASET COLLECTION

• 30 realistic situations that a personal robot might face in an undirected human environment

• captured so that to simulate a robot movement and to record all the known sensor artifacts

• all available at http://svn.pointclouds.org/data/Toyota
CloudSurfaceProcessing
  - PointCloud to PointCloud for better surface approximation
  - e.g., MovingLeastSquares, BilateralUpsampling

MeshConstruction
  - PointCloud to PolygonMesh, convert cloud to mesh without modifying vertex positions
  - e.g., ConcaveHull, ConvexHull, OrganizedFastMesh, GreedyProjectionTriangulation

SurfaceReconstruction
  - PointCloud to PolygonMesh, generate mesh with a possibly modified underlying vertex set
  - e.g., GridProjection, MarchingCubes, SurfelSmoothing, Poisson

MeshProcessing
  - PolygonMesh to PolygonMesh, improve input meshes by modifying connectivity and/or vertices
  - e.g., EarClipping, MeshSmoothingLaplacianVTK, MeshSmoothingWindowedSincVTK, MeshSubdivisionVTK
MOVING LEAST SQUARES

1/2

- Approximation of the underlying surface with an analytical function
- Based on a (weighted) neighborhood
- Height-map along a tangent normal onto which the query point is projected
- Complexity of the used function (and of the tangent estimation) define accuracy/runtime
- Having a function enables accurate computation of normals, surface curves, and up/down-sampling
1. Fit plane to local surface (PCA)

2. Fit a polynomial function in the set of distances from the points to the surface.

$$\sum_{i=1}^{\infty} (p(x_i) - f_i)^2 \theta(||r_i - r||)$$

$$\theta(x) = e^{-\left(\frac{x}{\sigma r}\right)^2}$$

3. Project points back to surface
MOVING LEAST SQUARES
1. SMOOTHING 1/3

MLS applied on the Bottles dataset from left to right:
• original scan
• MLS smoothed with search radius = 3 cm and second order polynomial fitting
• MLS smoothed with search radius = 5 cm and second order polynomial fitting

Eliminates noise.
MOVING LEAST SQUARES
1. SMOOTHING 2/3

from left to right:
• original scan
• MLS smoothed with search radius = 5 cm and second order polynomial fitting
• MLS smoothed with search radius = 3 cm and second order polynomial fitting

Smother surfaces.
• Adding noise to original Stanford Bunny dataset with standard deviation 0.001 (resulting in RMSE of 0.00173)

• Best result at search_radius=0.005, improvement of RMSE with 15% (similarly for the Dragon dataset)

• Varying the order of the polynomial fitting: higher order beneficial when large surface/curvature variance in neighborhood
MOVING LEAST SQUARES
2. UPSAMPLING 1/6

SAMPLE_LOCAL_PLANE

Door Handle dataset

After

Before
Tupperware dataset

More grippable surfaces.

After

Before
MOVING LEAST SQUARES

2. UPSAMPLING 3/6

VOXEL_GRID_DILATION
MOVING LEAST SQUARES
2. UPSAMPLING 4/6
VOXEL_GRID_DILATION

Computer Screen dataset

Fills relatively large holes.
MOVING LEAST SQUARES

2. UPSAMPLING 5/6

Sample locally fitted polynomial in different ways.
MOVING LEAST SQUARES
2. UPSAMPLING 6/6

Plane fitting quality (images show inliers)

original
MLS, no upsampling
SAMPLE_LOCAL_PLANE
RANDOM_UNIFORM_DENSITY
VOXEL_GRID_DILATION

Best performer!
BILATERAL FILTERING
UPSAMPLING 1/4

• Kinect modes:
  • 640x480 RGB image + 640x480 depth image at 30Hz
  • 1280x1024 RGB image + 640x480 depth image at 15 Hz

Why not use the better quality RGB image to enhance the depth map?

\[ \tilde{S}_p = \frac{1}{k_p} \sum_{q_d \in \Omega} S_{q_d} f(||p_d - q_d||) g(||\tilde{I}_p - \tilde{I}_q||) \]

BILATERAL FILTERING
UPSAMPLING 2/4

4x the amount of points.
Smoother surfaces.

original
sampled
BILATERAL FILTERING
UPSAMPLING 3/4

Fills small holes.

More iterations - fill larger holes - needs more computation time
Still, this does not solve for the problem of transparent objects ...
MARCHING CUBES

• Have implementations of 2 classical methods:
  
  • Hoppe et al.: \( SDF(x_i) = \min_{x_j \in N}(n_j \cdot (x_i - x_j)) \)
  
  • RBF: \( SDF(x_i) = \sum_i w_i \Phi(||x - c_i||) \)

Sensitive to noise.
OTHER METHODS

• Mesh Operations from VTK
  • MeshSmoothingLaplacianVTK
  • MeshSmoothingWindowedSincVTK
  • MeshSubdivisionVTK
• OrganizedFastMesh (demo)
• ConcaveHull, ConvexHull - use QHull library
• GridProjection - work of WG intern, Rosie Li

Need to convert between data types.
#include <pcl/common/common.h>
#include <pcl/io/pcd_io.h>
#include <pcl/features/normal_3d_omp.h>
#include <pcl/surface/mls.h>
#include <pcl/surface/poisson.h>
#include <pcl/io/vtk_io.h>

using namespace pcl;

int main (int argc, char **argv)
{
  if (argc != 3)
  {
    PCL_ERROR ("Syntax: %s input.pcd output.ply\n", argv[0]);
    return -1;
  }

  PointCloud<PointXYZ>::Ptr cloud (new PointCloud<PointXYZ> ());
  io::loadPCDFile (argv[1], *cloud);
MovingLeastSquares<PointXYZ, PointXYZ> mls;
mls.setInputCloud (cloud);
mls.setSearchRadius (0.01);
mls.setPolynomialFit (true);
mls.setPolynomialOrder (2);
mls.setUpsamplingMethod (MovingLeastSquares<PointXYZ, PointXYZ>::SAMPLE_LOCAL_PLANE);
mls.setUpsamplingRadius (0.005);
mls.setUpsamplingStepSize (0.003);

PointCloud<PointXYZ>::Ptr cloud_smoothed (new PointCloud<PointXYZ> ());
mls.process (*cloud_smoothed);

NormalEstimationOMP<PointXYZ, Normal> ne;
ne.setNumberOfThreads (8);
ne.setInputCloud (cloud_smoothed);
ne.setRadiusSearch (0.01);
Eigen::Vector4f centroid;
compute3DCentroid (*cloud_smoothed, centroid);
ne.setViewPoint (centroid[0], centroid[1], centroid[2]);

PointCloud<Normal>::Ptr cloud_normals (new PointCloud<Normal> ());
ne.compute (*cloud_normals);

for (size_t i = 0; i < cloud_normals->size (); ++i)
{
    cloud_normals->points[i].normal_x *= -1;
    cloud_normals->points[i].normal_y *= -1;
    cloud_normals->points[i].normal_z *= -1;
}

PointCloud<PointNormal>::Ptr cloud_smoothed_normals (new PointCloud<PointNormal> ());
concatenateFields (*cloud_smoothed, *cloud_normals, *cloud_smoothed_normals);
RECONSTRUCTION PIPELINE
EXAMPLE 3/4
Poisson<PointNormal> poisson;
poisson.setDepth(9);
poisson.setInputCloud(cloud_smoothed_normals);
PolygonMesh mesh;
poisson.reconstruct(mesh);

io::saveVTKFile(argv[2], mesh);

return 0;
THANKS!