PointCloud, Mesh, Surface Reconstruction - Investigation Report

How to do surface reconstruction when modelling object using Kinect?

Lihang Li

March 2015

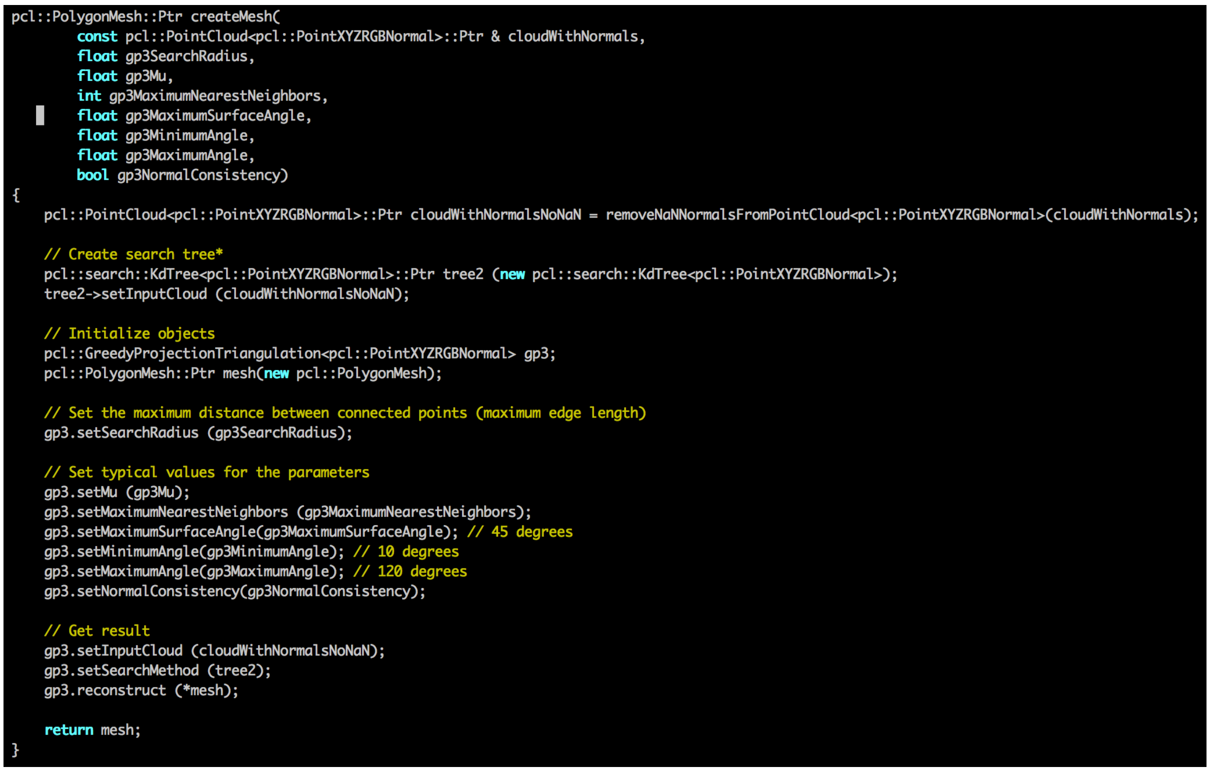
PointCloud, Mesh, Surface Reconstruction - Investigation Report

How to do surface reconstruction when modelling object using Kinect?

1. Meshing methods used in RTABMAP & RGBDMapping

(1) RTABMAP

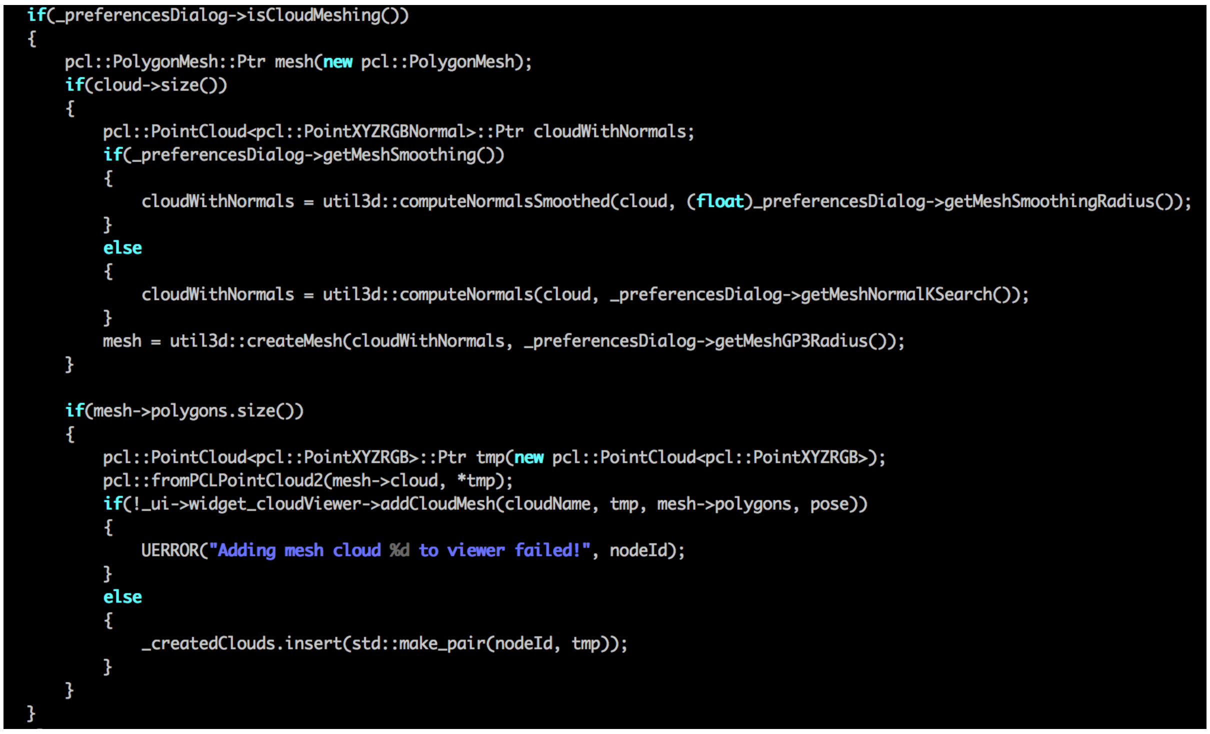
corelib/src/util3d.cpp



```
pcl::PolygonMesh::Ptr createMesh(
        const pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr & cloudWithNormals,
        float gp3SearchRadius,
        float gp3Mu,
        int gp3MaximumNearestNeighbors,
        float gp3MaximumSurfaceAngle,
        float gp3MinimumAngle,
        float gp3MaximumAngle,
        bool gp3NormalConsistency)
{
    pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloudWithNormalsNoNaN = removeNaNNormalsFromPointCloud<pcl::PointXYZRGBNormal>(cloudWithNormals);

    // Create search tree*
    pcl::search::KdTree<pcl::PointXYZRGBNormal>::Ptr tree2 (new pcl::search::KdTree<pcl::PointXYZRGBNormal>);
    tree2->setInputCloud (cloudWithNormalsNoNaN);

    // Initialize objects
    pcl::GreedyProjectionTriangulation<pcl::PointXYZRGBNormal> gp3;
    pcl::PolygonMesh::Ptr mesh(new pcl::PolygonMesh);

    // Set the maximum distance between connected points (maximum edge length)
    gp3.setSearchRadius (gp3SearchRadius);

    // Set typical values for the parameters
    gp3.setMu (gp3Mu);
    gp3.setMaximumNearestNeighbors (gp3MaximumNearestNeighbors);
    gp3.setMaximumSurfaceAngle(gp3MaximumSurfaceAngle); // 45 degrees
    gp3.setMinimumAngle(gp3MinimumAngle); // 10 degrees
    gp3.setMaximumAngle(gp3MaximumAngle); // 120 degrees
    gp3.setNormalConsistency(gp3NormalConsistency);

    // Get result
    gp3.setInputCloud (cloudWithNormalsNoNaN);
    gp3.setSearchMethod (tree2);
    gp3.reconstruct (*mesh);

    return mesh;
}
```

guilib/src/MainWindow.cpp

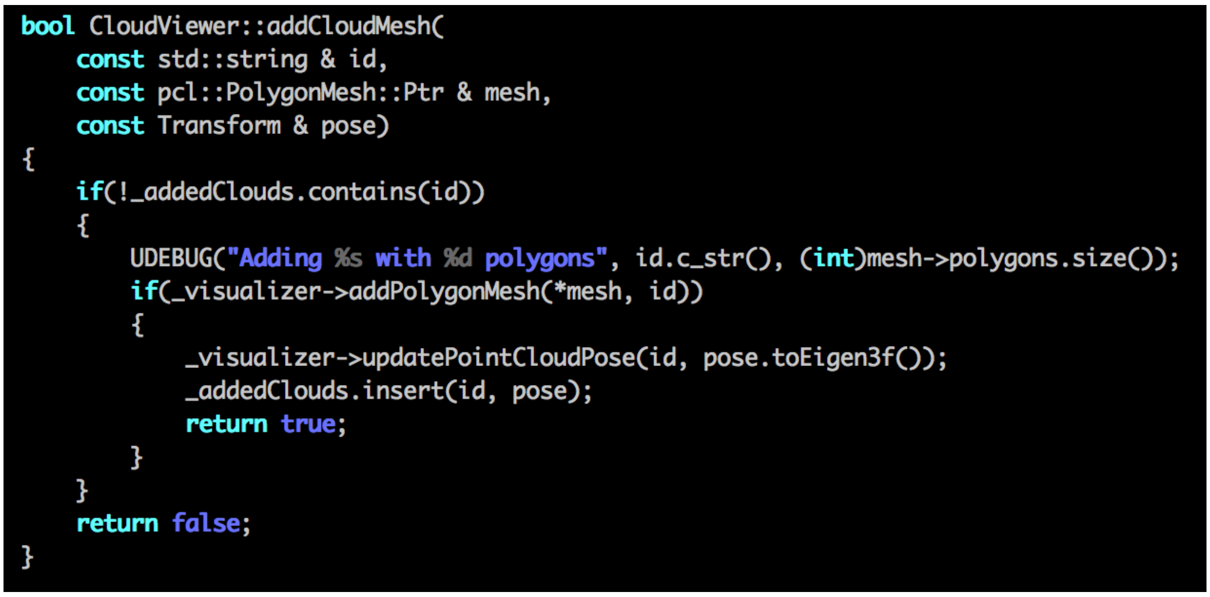
void MainWindow::createAndAddCloudToMap(int nodeId, const Transform & pose, int mapId);



```
if(_preferencesDialog->isCloudMeshing())
{
    pcl::PolygonMesh::Ptr mesh(new pcl::PolygonMesh);
    if(cloud->size())
    {
        pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloudWithNormals;
        if(_preferencesDialog->getMeshSmoothing())
        {
            cloudWithNormals = util3d::computeNormalsSmoothed(cloud, (float)_preferencesDialog->getMeshSmoothingRadius());
        }
        else
        {
            cloudWithNormals = util3d::computeNormals(cloud, _preferencesDialog->getMeshNormalKSearch());
        }
        mesh = util3d::createMesh(cloudWithNormals, _preferencesDialog->getMeshGP3Radius());
    }

    if(mesh->polygons.size())
    {
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr tmp(new pcl::PointCloud<pcl::PointXYZRGB>);
        pcl::fromPCLPointCloud2(mesh->cloud, *tmp);
        if(!_ui->widget_cloudViewer->addCloudMesh(cloudName, tmp, mesh->polygons, pose))
        {
            UERROR("Adding mesh cloud %d to viewer failed!", nodeId);
        }
        else
        {
            _createdClouds.insert(std::make_pair(nodeId, tmp));
        }
    }
}
```

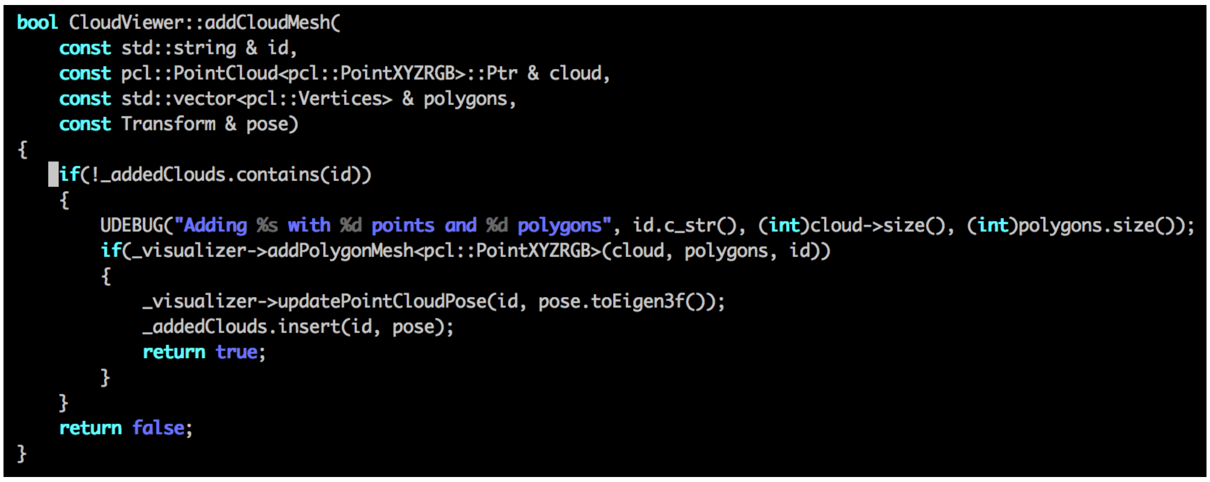
guilib/src/CloudViewer.cpp

bool CloudViewer::addCloudMesh(const std::string & id, const pcl::PointCloud<pcl::PointXYZRGB>::Ptr & cloud, const std::vector<pcl::Vertices> & polygons, const Transform & pose);



```
bool CloudViewer::addCloudMesh(
    const std::string & id,
    const pcl::PolygonMesh::Ptr & mesh,
    const Transform & pose)
{
    if(!_addedClouds.contains(id))
    {
        UDEBUG("Adding %s with %d polygons", id.c_str(), (int)mesh->polygons.size());
        if(_visualizer->addPolygonMesh(*mesh, id))
        {
            _visualizer->updatePointCloudPose(id, pose.toEigen3f());
            _addedClouds.insert(id, pose);
            return true;
        }
    }
    return false;
}
```

bool CloudViewer::addCloudMesh(const std::string & id, const pcl::PolygonMesh::Ptr & mesh, const Transform & pose);

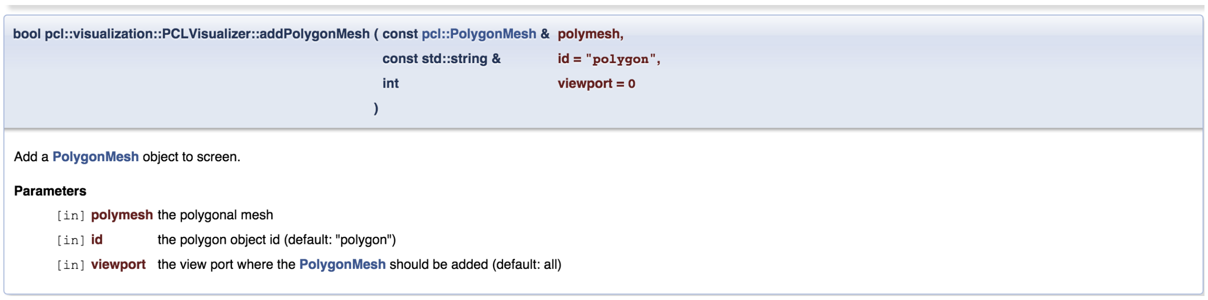


```
bool CloudViewer::addCloudMesh(
    const std::string & id,
    const pcl::PointCloud<pcl::PointXYZRGB>::Ptr & cloud,
    const std::vector<pcl::Vertices> & polygons,
    const Transform & pose)
{
    if(!_addedClouds.contains(id))
    {
        UDEBUG("Adding %s with %d points and %d polygons", id.c_str(), (int)cloud->size(), (int)polygons.size());
        if(_visualizer->addPolygonMesh<pcl::PointXYZRGB>(cloud, polygons, id))
        {
            _visualizer->updatePointCloudPose(id, pose.toEigen3f());
            _addedClouds.insert(id, pose);
            return true;
        }
    }
    return false;
}
```

guilib/include/rtabmap/gui/CloudViewer.h

pcl::visualization::PCLVisualizer * _visualizer;

See: http://docs.pointclouds.org/1.7.0/classpcl_1_1visualization_1_1_p_c_l_visualizer.html

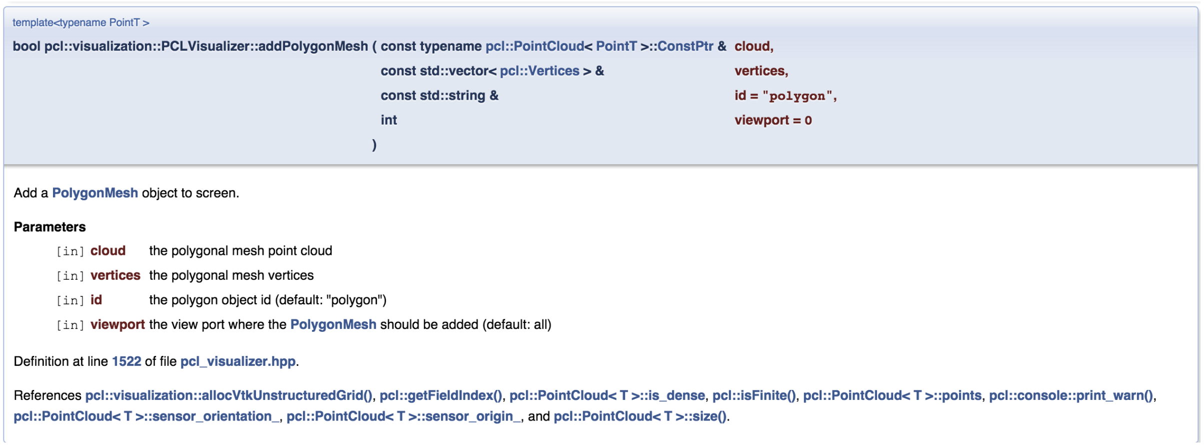


bool pcl::visualization::PCLVisualizer::addPolygonMesh (const pcl::PolygonMesh & polymesh,
const std::string & id = "polygon",
int viewport = 0
)

Add a PolygonMesh object to screen.

Parameters

[in]	polymesh	the polygonal mesh
[in]	id	the polygon object id (default: "polygon")
[in]	viewport	the view port where the PolygonMesh should be added (default: all)



template<typename PointT >
bool pcl::visualization::PCLVisualizer::addPolygonMesh (const typename pcl::PointCloud< PointT >::ConstPtr & cloud,
const std::vector< pcl::Vertices > & vertices,
const std::string & id = "polygon",
int viewport = 0
)

Add a PolygonMesh object to screen.

Parameters

[in]	cloud	the polygonal mesh point cloud
[in]	vertices	the polygonal mesh vertices
[in]	id	the polygon object id (default: "polygon")
[in]	viewport	the view port where the PolygonMesh should be added (default: all)

Definition at line 1522 of file pcl_visualizer.hpp.

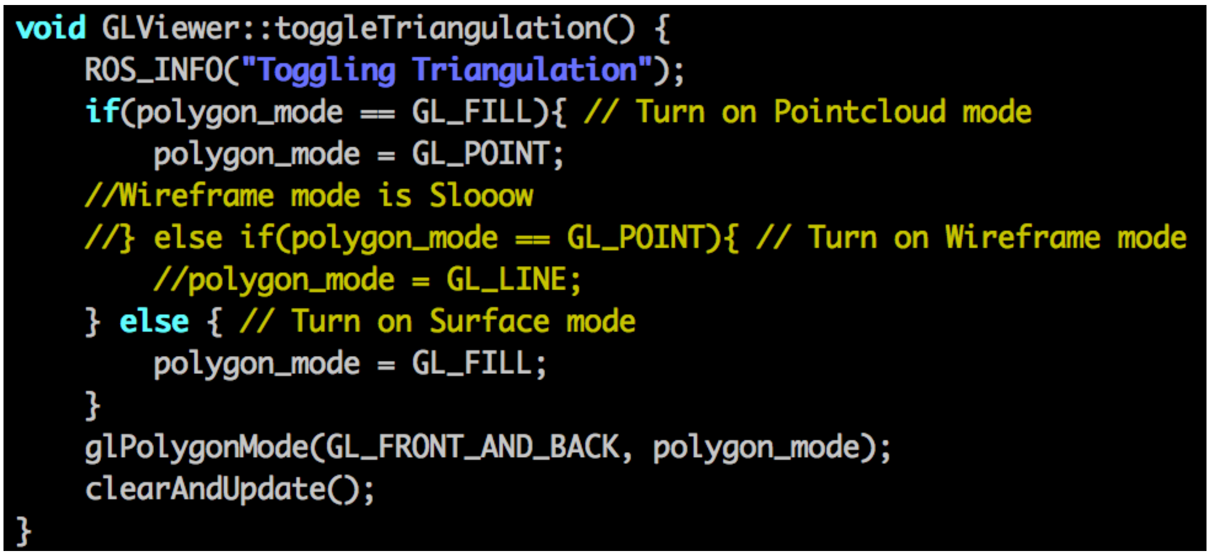
References pcl::visualization::allocVtkUnstructuredGrid(), pcl::getFieldIndex(), pcl::PointCloud< T >::is_dense, pcl::isFinite(), pcl::PointCloud< T >::points, pcl::console::print_warn(), pcl::PointCloud< T >::sensor_orientation_, pcl::PointCloud< T >::sensor_origin_, and pcl::PointCloud< T >::size().

(2) RGBDMapping

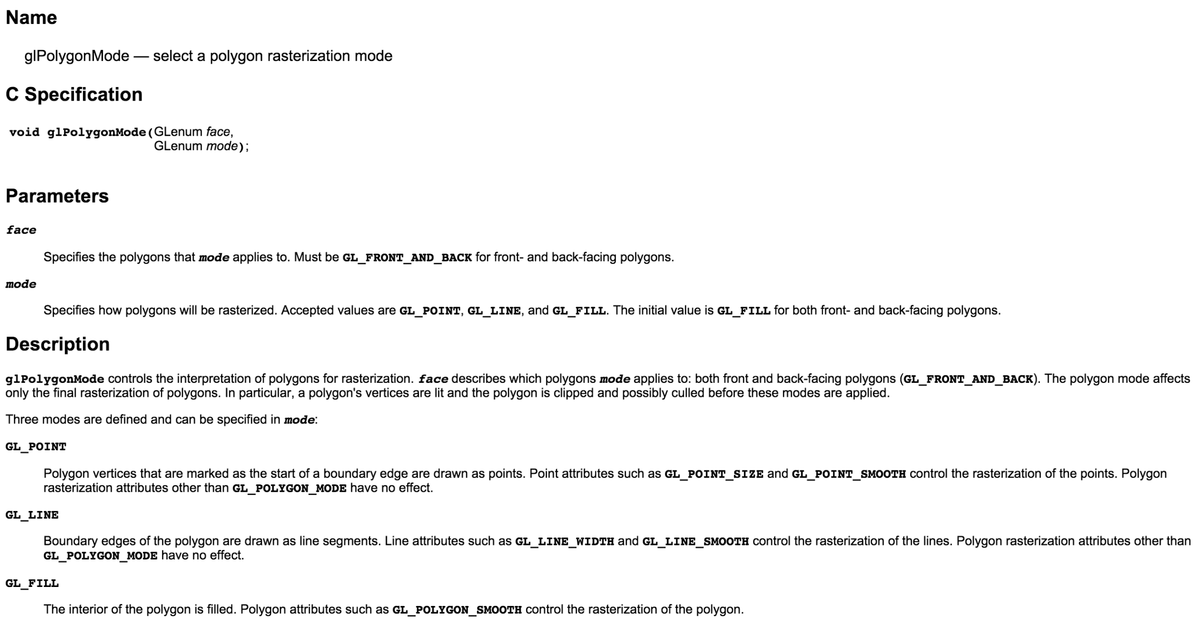
glviewer.cpp

void GLViewer::toggleTriangulation();

See: https://www.opengl.org/sdk/docs/man3/xhtml/glPolygonMode.xml



```
void GLViewer::toggleTriangulation() {
    ROS_INFO("Toggling Triangulation");
    if(polygon_mode == GL_FILL){ // Turn on Pointcloud mode
        polygon_mode = GL_POINT;
    //Wireframe mode is Slooow
    //} else if(polygon_mode == GL_POINT){ // Turn on Wireframe mode
        //polygon_mode = GL_LINE;
    } else { // Turn on Surface mode
        polygon_mode = GL_FILL;
    }
    glPolygonMode(GL_FRONT_AND_BACK, polygon_mode);
    clearAndUpdate();
}
```



Name

glPolygonMode — select a polygon rasterization mode

C Specification

```
void glPolygonMode(GLenum face,
                   GLenum mode);
```

Parameters

face

Specifies the polygons that ***mode*** applies to. Must be **GL_FRONT_AND_BACK** for front- and back-facing polygons.

mode

Specifies how polygons will be rasterized. Accepted values are **GL_POINT**, **GL_LINE**, and **GL_FILL**. The initial value is **GL_FILL** for both front- and back-facing polygons.

Description

glPolygonMode controls the interpretation of polygons for rasterization. ***face*** describes which polygons ***mode*** applies to: both front and back-facing polygons (**GL_FRONT_AND_BACK**). The polygon mode affects only the final rasterization of polygons. In particular, a polygon's vertices are lit and the polygon is clipped and possibly culled before these modes are applied.

Three modes are defined and can be specified in ***mode***:

GL_POINT

Polygon vertices that are marked as the start of a boundary edge are drawn as points. Point attributes such as **GL_POINT_SIZE** and **GL_POINT_SMOOTH** control the rasterization of the points. Polygon rasterization attributes other than **GL_POLYGON_MODE** have no effect.

GL_LINE

Boundary edges of the polygon are drawn as line segments. Line attributes such as **GL_LINE_WIDTH** and **GL_LINE_SMOOTH** control the rasterization of the lines. Polygon rasterization attributes other than **GL_POLYGON_MODE** have no effect.

GL_FILL

The interior of the polygon is filled. Polygon attributes such as **GL_POLYGON_SMOOTH** control the rasterization of the polygon.

2. Popular surface reconstruction methods survey

(1) PCL-Surface Reconstruction

• 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.,

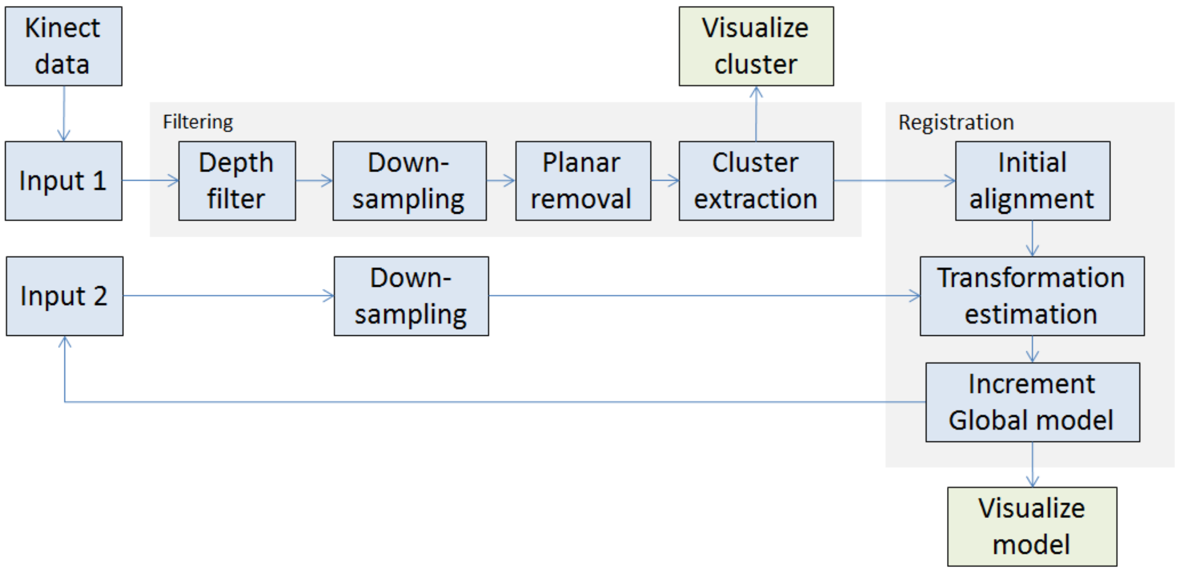
GridProjection, MarchingCubes, SurfelSmoothing, Poisson

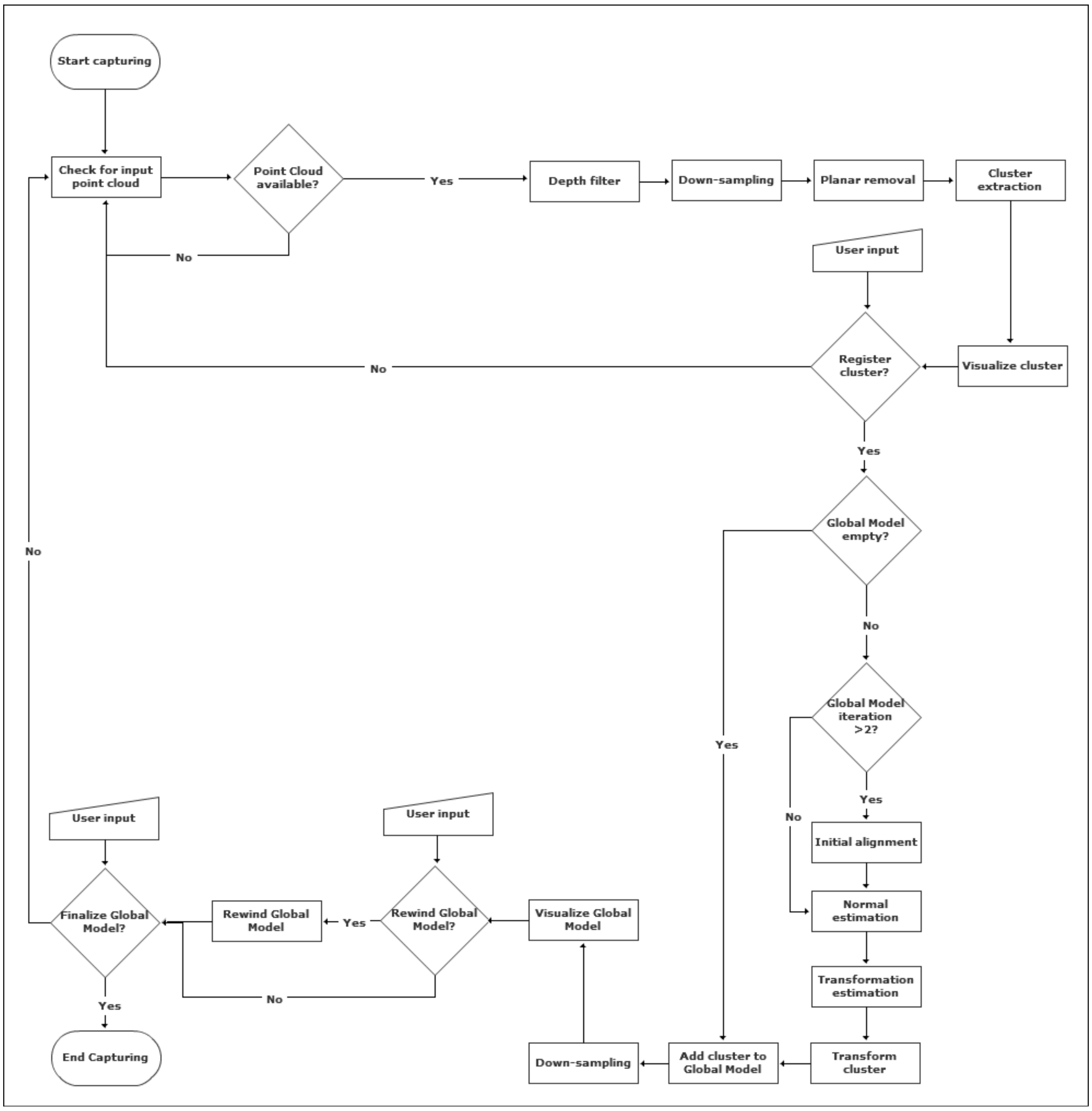
• MeshProcessing

PolygonMesh to PolygonMesh, improve input meshes by modifying connectivity and/or vertices. e.g.,

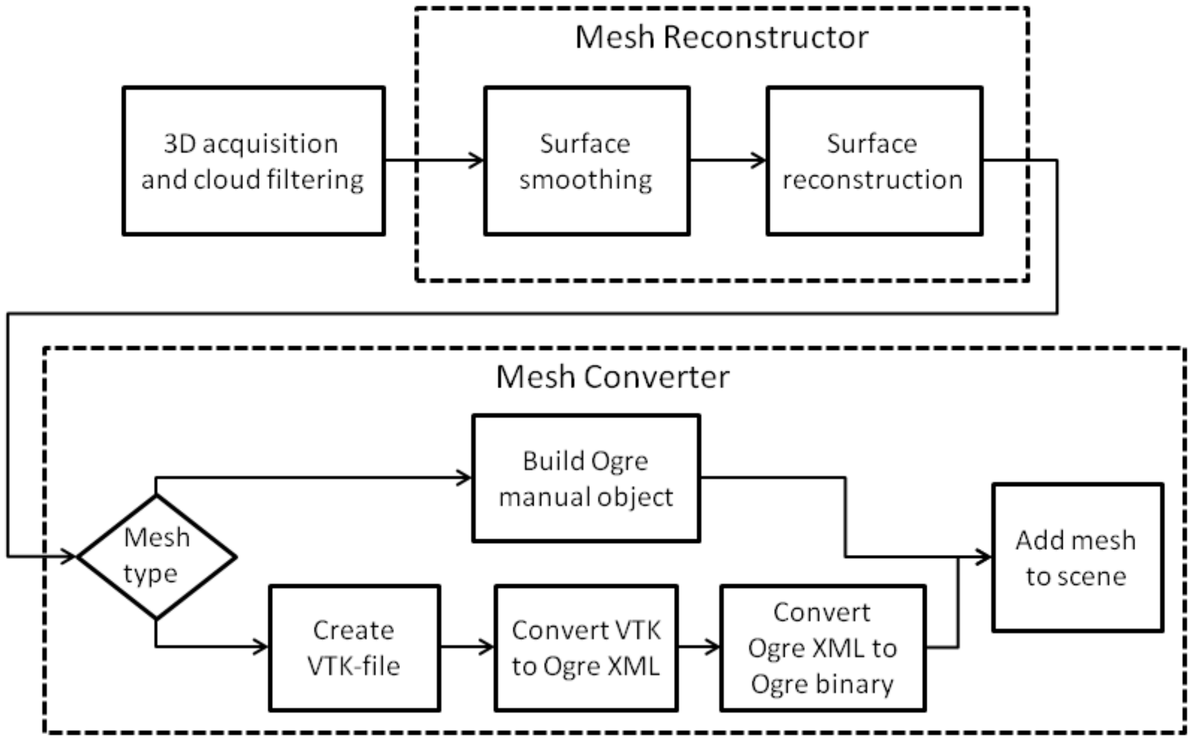
EarClipping, MeshSmoothingLaplacianVTK, MeshSmoothingWindowedSincVTK, MeshSubdivisionVTK

(2) 3D Content Capture and Reconstruction using Microsoft Kinect Depth Camera





(3) Surface Reconstruction of Point Clouds Captured with Microsoft Kinect

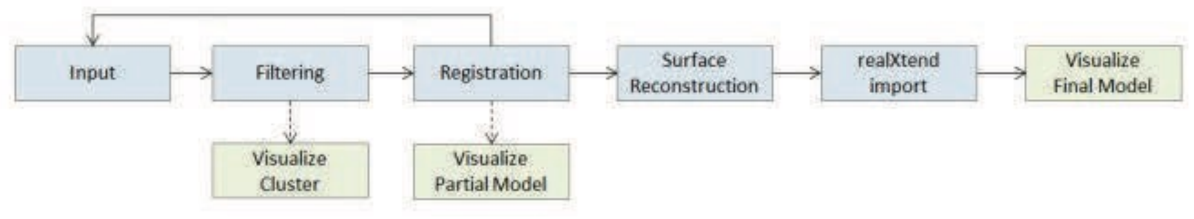


1) Cloud Filtering
- Filtering by depth, discard the cloud point with Z > threshold
- Filtering by density, down-sampling using voxel grid sampling
- Cluster extraction, detect and segment object-of-interest by clustering the point clouds

2) Surface Smoothing
MLS(Moving Least Squares)

3) Surface Reconstruction
Greedy Projection Triangulation.

(4) 3D Object Reconstruction Processing Chain for Extensible Virtual Spaces



3. A typical point cloud meshing pipeline

Reference: http://meshlabstuff.blogspot.com/2009/09/meshing-point-clouds.html

Subsampling

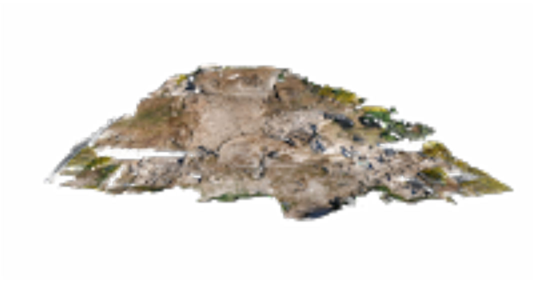
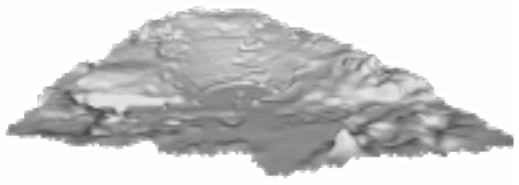
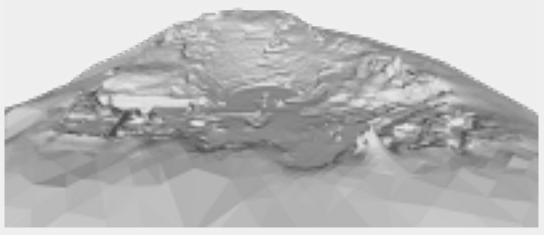
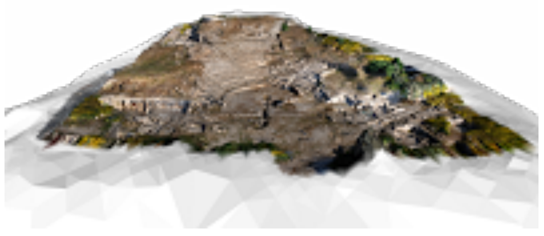
Normal Reconstruction

Surface Reconstruction

Recovering Original Color

Cleaning up and assessing

Point Cloud Meshing Pipeline

Pipeline	Demo
Original	
Subsampling	
Normal Reconstruction	
Surface Reconstruction	
Recovering Original Color	

Pipeline	**Demo**
Cleaning up and assessing	

(1) Subsampling

As a first step we reduce a bit the dataset in order to have amore manageable dataset. Many different options here. Having a nicely spaced subsampling is a good way to make some computation in a faster way. The Sampling->Poisson Disk Sampling filter is a good option. While it was designed to create Poisson disk samples over a mesh, it is able to also compute Poisson disk subsampling of a given point cloud (remember to check the 'subsampling' boolean flag). For the curious ones, it uses an algorithm very similar to the dart throwing paper presented at EGSR2009 (except that we have released code for such an algorith long before the publication of this article :)). In the invisible side figure a Poisson disk subsampling of just 66k vertices.

(2) Normal Reconstruction

Currently inside MeshLab the construction of normals for a point cloud is not particularly optimized (I would not apply it over 9M point cloud) so starting from smaller mesh can give better, faster results. You can use this small point cloud to issue a fast surface reconstruction (using Remeshing->Poisson surface reconstruction) and then transfer the normals of this small rough surface to the original point cloud. Obviously in this way the full point cloud will have a normal field that is by far smoother than necessary, but this is not an issue for most surface reconstruction algorithms (but it is an issue if you want to use these normals for shading!).

(3) Surface Reconstruction

Once rough normals are available Poisson surface reconstruction is a good choice. Using the original point cloud with the computed normals we build a surface at the highest resolution (recursion level 11). Roughly clean it removing large faces filter, and eventually simplify it a bit (remove 30% of the faces) using classical Remeshing->Quadric edge collapse simplification filter (many implicit surface filters rely on marching cube like algorithms and leave useless tiny triangles).

(4) Recovering original color

Here we have two options, recovering color as a texture or recovering color as per-vertex color. Here we go for the latter, leaving the former to a next post where we will go in more details on the new automatic parametrization stuff that we are adding in MeshLab. Obviously if you store color onto vertexes you need to have a very dense mesh, more or less of the same magnitudo of the original point cloud, so probably refining large faces a bit could be useful. After refining the mesh you simply transfer the color attribute from the original point cloud to the reconstructed surface using the vertex attribute transfer filter.

(5) Cleaning up and assessing

The vertex attribute transfer filter uses a simple closest point heuristic to match the points between the two meshes. As a side product it can store (in the all-purpose per-vertex scalar quality) the distance of the matching points. Now just selecting the faces having vertices whose distance is larger than a given threshold we can easily remove the redundant faces created by the Poisson Surface Reconstruction.

4. Things learned

• RTABMAP

RTABMAP is heavily using PCL both for the processing backend and the viewing frontend. For point cloud and polygon mesh, there is a module called **visualisation** which is based on VTK, the **PCLVisualizer** class can handle the integration and presentation of both point cloud and polygon mesh. The core methods can **add, update, remove** point cloud and mesh. See http://docs.pointclouds.org/1.7.0/classpcl_1_1visualization_1_1_p_c_l_visualizer.html for more details.

For creating meshes, RTABMAP utilises Greedy Projection Triangulation and for better results, it uses Moving Least Squares(MLS) to smooth the point cloud.

• RGBDSLAM_v2

It's quite trivial that RGBDSLAM_v2 uses OpenGL for rendering point cloud and meshes, for point cloud, it uses GL_POINT, for mesh, it justs uses GL_FILL. See https://www.opengl.org/sdk/docs/man3/xhtml/glPolygonMode.xml for more details.

So technically speaking, RGBDSLAM_v2 does not create meshes explicitly, however it uses a trick when rendering using OpenGL to get a triangulation view. For aggregating point cloud, there is a function called **transformAndAppendPointCloud** in **misc.cpp**. The source code says there is no special policy but to add the transformed point cloud to the aggregated one. So here there are big room for optimization.

• Chiru

Chiru is a project aiming 3D Object Capture and the website is http://www.chiru.fi/?udpview=12357&sid=3&src=db23044. Kinect based sensor system was used to detect and record the physical shape, and a 3D object capturing module for RealXtend Tundra was created to capture the digitalized object to the 3D virtual world. The work included the development of a user interface for uploading the captured objects to a web server and viewing them with a web browser using a WebGL-based viewer.

There are many assumptions for achieving good capture using Chiru, which I listed below:

- The object should be placed on a planar scene, such as floor, table, etc
- The user should move the Kinect in a pre-defined manner, like you should move in clockwise and make sure the transformation between each move is identity or so since the alignment using ICP uses this assumption for quick calculation
- The object-of-interest is recommended to place away from the Kinect about 1.0 meter, since the depth filtering process will discard all the point cloud which Z is bigger than 1.5 m

For surface reconstruction, like RTABMAP, it uses MLS to smooth the point cloud and Greedy Projection Triangulation to create meshes.

• Our methods

Try MLS and Greedy Projection Triangulation.

5. Good references

• A Benchmark for Surface Reconstruction-a20-berger

• State of the Art in Surface Reconstruction from Point Clouds-reconstar_eg14

• Surface Reconstruction Algorihtms-Review and Comparison-ISDE2013

• An As-Short-As-Possible Introduction to the Least Squares, Weighted Least Squares and Moving Least Squares Methods for Scattered Data Approximation and Interpolation-asapmls