Robot Vision Library – Instructions for Using Point Cloud Segmentation Tool

Robert Cupec, Emmanuel Karlo Nyarko, Damir Filko

Robotics and 3D Vision Group, J. J. Strossmayer University of Osijek, Faculty of Electrical Engineering

1. Program overview

The program for segmentation of depth images into approximately convex subsets is a component of the *Robot Vision Library – RVL* provided here as a Microsoft Visual C++ solution consisting of three projects:

- *RVLCore* – library with basic classes needed for implementation of various image and point cloud processing tools and visualization of the results;
- *RVLPCS* – library with classes implementing algorithms for segmentation of depth images.
- *RVLPCSdemo* – console application representing a demo program for the developed segmentation tool.

The RVL library is primarily designed for Microsoft Windows. However, although we did not try to compile and test it in Linux, we assume that this should not be a problem, since the code is written using the standard C++ language. RVL requires OpenCV 2.0 library. In order to use *RVLPCSdemo* program to process depth images acquired on-line with a Microsoft Kinect sensor, OpenNI 2.0 SDK is required. If OpenNI 2.0 SDK is not available, the program can be used to process depth image files whose format is described in Appendix A. If you don't want to use OpenNI 2.0 SDK, the line `#define RVLOPENNI` in the *Platform.h* file of *RVLCore* library must be commented out and the corresponding library OpenNI2.lib must be excluded from *Linker/Input/Additional Dependencies* list of *RVLPCSdemo*. 3D visualization of the segmented triangular mesh is implemented using the Visualization Toolkit (VTK), which can be downloaded from http://www.vtk.org. If you don't need 3D visualization and you don't want to install VTK, the line `#define RVLVTK` in the *Platform.h* file of *RVLCore* library must be commented out and the corresponding libraries OpenNI2.lib must be excluded from *Linker/Input/Additional Dependencies* list of *RVLPCSdemo*. The VTK libraries which must be excluded are all libraries from *Linker/Input/Additional Dependencies* list whose name begins with "vtk" as well as the library "opengl32.lib". For both libraries OpenCV and VTK, environment variables must be defined. The required environment variables are given in Table 1.

Table 1. Environment variables which must be defined

Library	Variable name	Value
OpenCV	OPENCV20	path to the OpenCV folder
VTK	VTK_PATH	path to the VTK folder

How to define the environment variables on Windows 8 is explained in Appendix B.

Alternatively to using environment variables, the user can manually set relative (or absolute) paths to the required libraries in configuration options for all included projects in the solution. The user needs to modify the *compiler* (*C/C++*) configuration of all three projects, *RVLCore*, *RVLPCS*, *RVLPCSdemo*, by inserting paths to *Additional Include Directories* that correspond to the include directories of OpenCV, OpenNI and VTK libraries. Additionally, the user must modify *linker* configuration of *RVLPCSdemo* project and insert paths to *Additional Library Directories* that correspond to the library directories of OpenCV, OpenNI and VTK libraries. Furthermore before starting the *RVLPCSdemo* application users should copy library dll files to the application directory or the Windows system32 directory.

The highest resolution the current implementation of RVL supports is 640 × 480. It also supports lower resolutions. The default resolution is 320 × 240, which means that the image acquired by the Kinect sensor, whose original size is 640 × 480 is automatically subsampled to 320 × 240. The resolution can be specified by changing the program parameters *StereoVision.Width* and *StereoVision.Height* defined in the configuration file *RVLPCSdemo.cfg* located in the *RVLPCSdemo* directory. The resolution specified in the *RVLPCSdemo.cfg* file must be compatible with the parameter *StereoVision.Kinect.Scale* defined in the same file. The values of the parameter *StereoVision.Kinect.Scale* for different resolutions are given in Table 2.

Table 2. Values of the parameter *StereoVision.Kinect.Scale* for different image resolutions.

Resolution	StereoVision.Kinect.Scale
640 × 480	1
320 × 240	2
160 × 120	4

The application of RVL tools is explained in the following through the description of the *RVLPCSdemo* program. The structure of the program is show in Fig. 1.

The program starts by creation and initialization of an instance of the *CRVLPCSVS* class using the commands

```
CRVLPCSVS VS;

VS.CreateParamList();

VS.Init("RVLPCSdemo.cfg");
```

This class initializes all tools needed for the considered task and performs memory management. During this initialization, the parameters of all tools are read from the *RVLPCSdemo.cfg* file. Furthermore, Kinect sensor is initialized by the following command

```
VS.m_Kinect.Init();
```

After the initialization, the depth image acquisition and processing loop is performed cyclically in a loop which stops when the *Esc* key is pressed. A cycle starts by depth image acquisition. If a Kinect sensor is connected, the depth image is provided by this sensor.

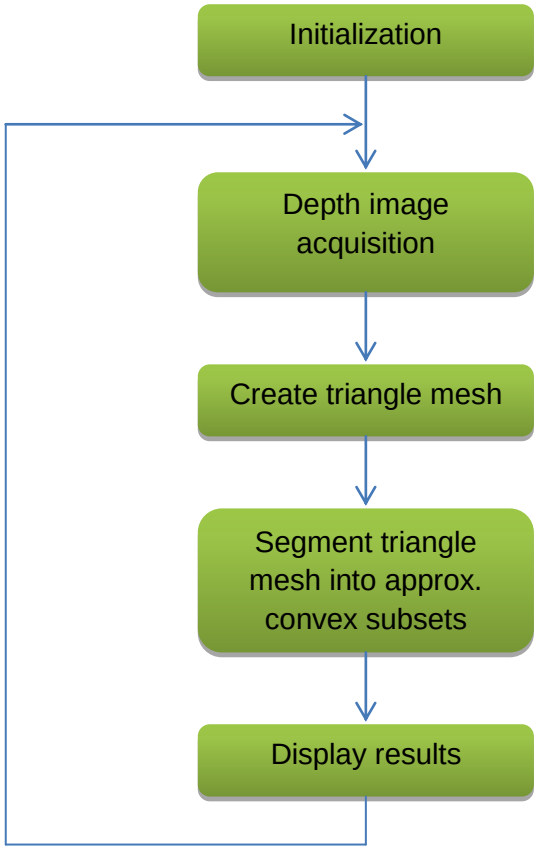

Fig. 1 Structure of *RVLPCSdemo*.

The image acquisition is performed using the following commands

```
RVLDISPARITYMAP *pDepthImage = &(VS.m_StereoVision.m_DisparityMap);

int w = pDepthImage->Width;

int h = pDepthImage->Height;

IplImage *pRGBImage = cvCreateImage(cvSize(w, h), IPL_DEPTH_8U, 3);

IplImage *pGSImage = cvCreateImage(cvSize(w, h), IPL_DEPTH_8U, 1);

VS.m_Kinect.GetImages(pDepthImage->Disparity, pRGBImage, NULL, pGSImage);
```

If Kinect is not connected, the depth image is read from a file using the following command

```
RVLImportDisparityImage(VS.m_ImageFileName, pDepthImage);
```

The depth image file name is initially read from the *RVLPCSdemo.cfg* file, where it can be specified by the user by setting *VS.ImageFileName* parameter. From the input depth image, a triangle mesh is created using the recursive Delaunay triangulation refinement procedure proposed by Schmitt and Chen [1]. The resulting mesh is stored in `VS.m_AImage` representing an instance of the *CRVLAImage* class used in RVL for storing the results of various image processing tools. Before executing the mesh building process, `VS.m_AImage` must be cleared from the data obtained in the previous cycle using the command

```
VS.m_AImage.Clear();
```

The depth information stored in `VS.m_StereoVision.m_DisparityMap` is then converted into an organized 3D point cloud using the command

```
VS.m_PSD.GetPointsWithDisparity(pDepthImage);
```

After that, mesh building is performed by executing the following command

```
VS.m_PSD.Segment(&(VS.m_AImage.m_C2DRegion),&(VS.m_AImage.m_C2DRegion2),&(
VS.m_AImage.m_C2DRegion3),&(VS.m_Mem));
```

The result of this command is a triangle mesh representing the input depth image, where the maximum deviation of the image points from the triangles of the obtained mesh is given by the triangulation tolerance τ specified by the *PSD.STRM.uvdTol* parameter in the *RVLPCSdemo.cfg* file. This triangle mesh is a useful representation itself and it can be used not only to detect approximately convex objects, as discussed herein, but also for some other segmentation approaches. The format of the triangle mesh provided by RVL is given later in this document. The computed mesh is then segmented into approximately convex sub-meshes, referred to in this document as *segments*, by applying the algorithm proposed in [2]. This algorithm is executed by the following command

```
nObjects = RVLSegmentToConvex(&(VS.m_AImage.m_C2DRegion), NULL,
&(VS.m_AImage.m_C2DRegion2), VS.m_ConvexSegmentThr, w, h,
VS.m_PSD.m_Point3DMap, &(VS.m_Mem));
```

This command assigns a label to each triangle of the mesh. Triangles which are assigned the same label belong to the same segment. The segmentation depends on the parameter ε representing the maximum allowed distance of mesh vertices from the convex hull of a segment. This parameter can be specified by the user by setting the value *Segmentation.Convex.Thr* in the *RVLPCSdemo.cfg* file. The results are displayed using the graphical user interface of RVL implemented by the class *CRVLGUI*.

2. Display Control

RVLPCSdemo allows the user to control the display of the results and change the parameters of the algorithm by pressing certain keys. The functions assigned to a particular key pressed are

- m - switch the mesh display on/off;
- s - switch the segmentation display on/off;
- v - activate 3D visualization;
- p - save 3D model created using VTK to a file in PLY-format in the *RVLPCSdemo* folder;
- b - switch between depth, RGB and grayscale image (only in the on-line mode with a Kinect connected to the PC);

c - switch between the continuous and the image-by-image mode (in the image-by-image mode the user must press a key in order for the next image to be processed);

z - switch between the original image size and the double size;

r - switch the record mode on/off (in the record mode, the images acquired by the Kinect sensor are stored in files; no processing is performed);

↑ - increase τ;

↓ - decrease τ;

→ - increase ε;

← - decrease ε.

Esc - quit the demo.

In addition to the execution time, the values of the parameters τ and ε are also displayed. The parameter τ is denoted by "TT" on the display window and ε by "CT".

3. Output data

Triangle Mesh

The triangle mesh created by the function `VS.m_PSD.Segment` consists of triangles stored in the list `VS.m_AImage.m_C2DRegion.m_ObjectList`. The mesh triangles are represented by instances of the class `CRVL2DRegion2` which can be accessed as illustrated by the following sample code.

```
CRVL2DRegion2 *pTriangle;
VS.m_AImage.m_C2DRegion.m_ObjectList.Start();
while(VS.m_AImage.m_C2DRegion.m_ObjectList.m_pNext)
{
      pTriangle = (CRVL2DRegion2 *)(
            VS.m_AImage.m_C2DRegion.m_ObjectList.GetNext());
      // pTriangle is a pointer to an instance of the class
      // CRVL2DRegion2 representing a mesh triangle.
      // Now you can do whatever you want with the triangle.
}
```

The parameters of the class *CRVL2DRegion2* which are relevant for the considered application are explained in the following.

m_Flags - Some triangles generated in the mesh building process are marked as rejected by setting the *RVLOBJ2_FLAG_REJECTED* flag. This flag is set in the following cases:

- if the percentage of the pixels within the triangle with assigned depth is lower than the threshold which can be specified by setting the *PSD.STRM.MinTriangleFillPerc* parameter in the *RVLPCSdemo.cfg* file;
- if the triangle stretches over a depth discontinuity.

m_PtArray - a pointer to a reference mesh link of the triangle;

m_Label - the label of the segment to which the triangle belongs;

The flag *RVLOBJ2_FLAG_REJECTED* can be evaluated as in the following example.

```
If(pTriangle->m_Flags & RVLOBJ2_FLAG_REJECTED)

{

      // the code which should be executed for a triangle marked as
      // rejected

}
```

The triangles of a mesh generated by the considered mesh building process are represented by *links*. Each triangle is defined by three *vertices* connected by *edges*. Each edge is assigned two vectors referred to in this document as links. The initial point of each of these two vectors is one of the endpoints of the considered edge, while its terminating point is the other endpoint of the edge. An example is shown in Fig. 2, where the links assigned to the edge $\overline{P_0P_2}$ representing a side of the grey triangle T_0 are depicted by the red and the blue arrow.

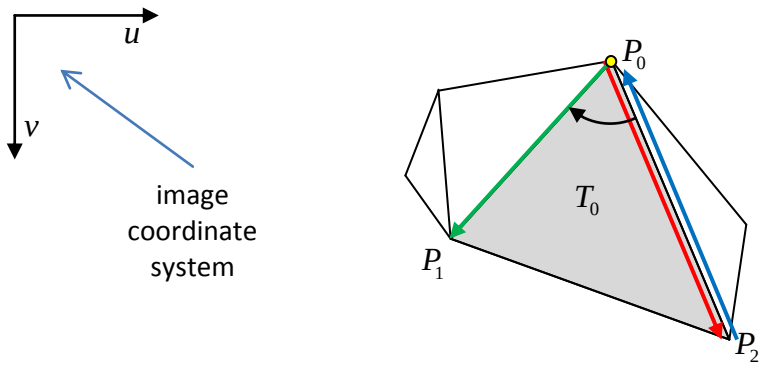

Fig. 2 Triangle T_0 (gray) defined by the vertices P_0, P_1 and P_2 and links denoted by the red, green and blue arrow

Links are represented by the data structures *RVLMESH_LINK* with the following fields

iPix0 - the index of the pixel representing initial point of the link;

du, *dv* - the components of the vector corresponding to the link;

pOpposite - the pointer to the *opposite link* corresponding to the same edge;

pNext - the pointer to the *next link* in the clockwise direction having the same initial point;

pPrev - the pointer to the *previous link* in the clockwise direction having the same initial point;

vp2DRegion - the pointer to the triangle on the right side of the link with respect to the link direction;

For example, the opposite link of the red link in Fig. 2 is the one represented by the blue arrow and the next link of the red link in the clockwise direction is represented by the green arrow. The red link is the previous link of the green link. If u and v are the coordinates of a pixel in the image coordinate system shown in Fig. 2, then the pixel index is computed as $u + v \cdot w$, where w is the image width. For each triangle in a mesh, there are three links related to it by their pointers *vp2DRegion*. One of them is the reference link of the triangle. The parameter *m_PtArray* of a triangle represents the pointer to the reference link of this triangle. The following code illustrates how all vertices of a triangle can be accessed given the pointer `pTriangle` to the considered triangle and the image width `w`.

```
RVLMESH_LINK *pLink = (RVLMESH_LINK *)(pTriangle->m_PtArray);

int u1 = pLink->iPix0 % w;

int v1 = pLink->iPix0 / w;

pLink = pLink->pNext->pOpposite;

int u2 = pLink->iPix0 % w;

int v2 = pLink->iPix0 / w;

pLink = pLink->pNext->pOpposite;

int u3 = pLink->iPix0 % w;

int v3 = pLink->iPix0 / w;
```

The image coordinates of the vertices of the considered triangle are (`u1`, `v1`), (`u2`, `v2`) and (`u3`, `v3`). The following code illustrates how the adjacent triangles of a given triangle can be accessed. Let `pTriangle` be a pointer to a triangle T_0. The following code provides pointers `pAdjacentTriangle1`, `pAdjacentTriangle2` and `pAdjacentTriangle3` to three triangles which share the same side with T_0.

```
RVLMESH_LINK *pLink = (RVLMESH_LINK *)(pTriangle->m_PtArray);

pLink = pLink->pNext;
```

```
CRVL2Dregion2 *pAdjacentTriangle1 = (CRVL2Dregion2 *)(pLink->vp2DRegion);
pLink = pLink->pOpposite->pNext;
CRVL2Dregion2 *pAdjacentTriangle2 = (CRVL2Dregion2 *)(pLink->vp2DRegion);
pLink = pLink->pOpposite->pNext;
CRVL2Dregion2 *pAdjacentTriangle3 = (CRVL2Dregion2 *)(pLink->vp2DRegion);
```

One of the links assigned to the sides of a triangle is the reference link of the triangle.

The image points inside a triangle can be determined using the member function *GetPtsInConvexPolygon* of the class *CRVLDelaunay*. The following code computes the coordinates of all points inside a triangle addressed by the pointer `pTriangle`.

```
int iFirstScanLine, iLastScanLine, iScanLine;
int u, v, iPix;
RVLMESH_LINK *pLink = (RVLMESH_LINK *)(pTriangle->m_PtArray);
VS.m_pDelaunay->GetPtsInConvexPolygon(pLink, VS.m_PSD.m_iScanLineStart,
      VS.m_PSD.m_iScanLineEnd, iFirstScanLine, iLastScanLine);
for(iScanLine = iFirstScanLine; iScanLine <= iLastScanLine; iScanLine++)
      for(iPix = VS.m_PSD.m_iScanLineStart[iScanLine];
            iPix <= VS.m_PSD.m_iScanLineEnd[iScanLine]; iPix++)
      {
            u = iPix % w;     // w is the image width
            v = iPix / w;
            // (u, v) are the coordinates of a point inside the triangle
            // addressed by the pointer pTriangle.
      }
```

Another way to get the points assigned to a triangle is using the member variable *m_2DRegionMap* of the *CRVLPlanarSurfaceDetector* class. The following code illustrates how to access the triangle containing a point with coordinates `u` and `v` given the image width `w`.

```
CRVL2Dregion2 *pTriangle = VS.m_PSD.m_2DRegionMap[u + v * w];
```

If no triangle contains the considered image point, `pTriangle` has value *NULL*.

Approximately Convex Segments

As a result of the segmentation process following the mesh building, each triangle is assigned a label corresponding to the approximately convex segment to which the triangle belongs. Each segment is assigned a unique label and all triangles belonging to this segment are assigned the same

label stored in the triangle parameter `m_Label`. The following code illustrates how to get the label of the segment containing a point with coordinates `u` and `v`.

```
CRVL2Dregion2 *pTriangle = VS.m_PSD.m_2DRegionMap[u + v * w];

DWORD Label = (pTriangle ? pTriangle->m_Label : -1);
```

If no segment contains the considered image point, the variable `Label` is set to -1.

3D Data

Each image point with a valid depth is assigned 3D coordinates relative to the Kinect reference frame in millimeters. The following code illustrates how to get the 3D coordinates `x`, `y`, and `z` of the image pixel with coordinates `u` and `v` in the image coordinate system.

```
double *P = VS.m_PSD.m_Point3Dmap[u + v * w]->XYZ;

double x = P[0];

double y = P[1];

double z = P[2];
```

It should be noted here that these coordinates in millimeters are computed using the intrinsic camera parameters: focal lengths f_u (horizontal) and f_v (vertical) as well as the coordinates of the principal point u_c and v_c. By default, the program uses the values which worked rather well with our Kinect sensor. Nevertheless, if you have some better values, obtained by some calibration procedure for your particular Kinect sensor, you can use your parameters by entering their values in the *RVLPCSdemo.cfg* file. The intrinsic camera parameters in the *RVLPCSdemo.cfg* file are denoted by *StereoVision.Kinect.fu*, *StereoVision.Kinect.fv*, *StereoVision.Kinect.uc* and *StereoVision.Kinect.vc*.

References:

[1] F. Schmitt and X. Chen, *Fast segmentation of range images into planar regions*, in Proceedings of the IEEE Computer Society Conf. on Computer Vision and Pattern Recognition (CVPR), 1991, pp. 710-711.

[2] R. Cupec, E. K. Nyarko, D. Filko, *Fast 2.5D Mesh Segmentation to Approximately Convex Surfaces*, Proceedings of the 5th European Conference on Mobile Robots, pp. 127-132, Örebro, Sweden, 2011 (PDF)

Appendix A: Input file format

The format of the input depth image file is described in the following. The first line starts with the word "width" followed by a space and the number representing the image width in pixels. The second line starts with the word "height" followed by a space and the number representing the image height. The third line represents the depth format. It can be either "1mm", denoting that the depth is represented in millimeters, or it can be empty, denoting the raw depth code, which is something analogous to disparity value in stereo vision systems. The other lines represent the depth data. Each line represents an image row consisting of integer depth values separated by a space.

Appendix B: Defining environment variables

The procedure for defining the environment variables on Windows 8 is described in the following.

1. On the desktop, move the mouse to the bottom-left corner of the screen.
2. Select "Settings" → "Control panel" → "System" → "Advance system settings" → "Environment variables".
3. In the list "User variables" enter the user variable name and the path of the considered library.
4. Confirm your entry by clicking on "OK".