```
// point_cloud_HMI; Wyatt Newman, March 6, 2013
// demo program to show how to interact with point-cloud data
// Theory of operation:
//  *a callback function listens for pointclouds on topic: /multisense_sl/camera/points2, which is an
ordered pointcloud
//    (i.e. can be accessed from a matrix of points) from emulated stereo vision
//   This pointcloud is (realistically) imperfect, with noise and lots of NaN entries (failure to find
corresponding features
//   in left/right cameras)
//  *the pointcloud callback, "cloud_callback", checks if there has been a new mouse-click, and if so,
it
//    uses the mouse-click pixel coordinates to index into the point cloud, extracting 3-D coordinates
//    These 3-D coordinates are used to populate a geometry_msgs::PointStamped object, which gets a
copy of the pointCloud header
//    This stamped point is suitable for use with tf to cast results into alternative frames;
//    Each stamped point (if valid data) is published on topic "userPickedPoint"
//  *2-D images created from 3-D pointclouds are displayed and processed within a mouse callback
function, "onMouse()"
//    This function gets the display-window's pixel coordinates of the mouse-click location, and saves
these as global vars
//    for use by the cloud_callback function.  The 2-D image coordinates allow indexing into the 3-D
point cloud to infer 3-D
//    coordinates from a 2-D mouse click
//
//   Note: When the user enters a mouse click, the corresponding 3-D datapoint is published,
inheriting the header of the saved
//    (corresponding) point cloud.  However, if the user waits more than about 10sec between mouse
clicks, the resulting published
//    point will be fairly old, and a corresponding transform from this time is not likely to be in
cache.  Thus, the "consumer"
//    of this data may get a fault from tf_listener.transformPoint() in attempting to transform this
data to another frame.
//    The consumer should try/catch to allow for this possibility.  The user can click a point twice
in succession to make sure
//    the second transmission corresponds to a "fresh" timeStamp.

// To run the program, first create a live publisher for the topic: /multisense_sl/camera/points2
(i.e. start up drcsim, or playback
//    some recorded bagfile data).  Then start up this node.  Click on the display window and observe
the ROS_INFO stating
//    the image x,y, the corresponding 3-D point, x,y,z, and the RGB colors of both the 2-D image and
the corresponding 3-D point
// Also, in another terminal, run: rostopic echo /userPickedPoint.  At each mouse click, the
corresponding 3-D data (and header)
//    will be echoed.  This much demonstrates that the HMI code is functioning.  Note that the
reference frame for the data
//    is "frame_id: /left_camera_optical_frame", and the consumer of this data should transform it
into an appropriate frame

// Standard ROS Includes
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <sensor_msgs/Image.h>

// PCL includes
#include <pcl/ros/conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>

// OpenCV includes
#include <image_transport/image_transport.h> // don't need the transport headers for this routine;
#include <image_transport/subscriber_filter.h>

#include <cv_bridge/cv_bridge.h>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

//geometry messages
#include <geometry_msgs/PointStamped.h>

#include <ros/ros.h>
#include <sensor_msgs/image_encodings.h>
```

```
using namespace cv;
namespace enc = sensor_msgs::image_encodings;

// Function prototypes here
sensor_msgs::PointCloud2 filter_rgb(const sensor_msgs::PointCloud2ConstPtr& input_cloud,
                                                                                int l_r,
                                                                                int u_r,
                                                                                int l_g,
                                                                                int u_g,
                                                                                int l_b,
                                                                                int u_b);

// Global variables here: use these to communicate among main and callbacks
ros::Publisher cloud_pub; // will publish the cloud
ros::Publisher image_pub; // publish cloud converted to image
ros::Publisher pickedPointPub; //published picked point

sensor_msgs::Image ROS_image_of_input_cloud;

int mouse_click_x=0;
int mouse_click_y=0;

bool getPtFlag=false;
bool firstCloud=true;
bool pickedTarget=false;
pcl::PointCloud<pcl::PointXYZRGB> pcl_remembered;  //persistent pointCloud


//utility to copy transient point cloud into persistent memory
void copyCldPtrToCloud(pcl::PointCloud<pcl::PointXYZRGB>::Ptr
pcl_input_cloud,pcl::PointCloud<pcl::PointXYZRGB> &pcl_remembered) {
       for(unsigned int i = 0; i < pcl_input_cloud->size(); ++i)
        {

                // copy each point from input cloud pointer to output cloud (which is global)
                pcl_remembered.points[i].x = pcl_input_cloud->points[i].x;
                pcl_remembered.points[i].y = pcl_input_cloud->points[i].y;
                pcl_remembered.points[i].z = pcl_input_cloud->points[i].z;
                pcl_remembered.points[i].r = pcl_input_cloud->points[i].r;
                pcl_remembered.points[i].g = pcl_input_cloud->points[i].g;
                pcl_remembered.points[i].b = pcl_input_cloud->points[i].b;
        }
        pcl_remembered.header = pcl_input_cloud->header;
}



void cloud_callback(const sensor_msgs::PointCloud2ConstPtr& ros_input_cloud)
{
        int pcl_index;
        sensor_msgs::PointCloud2 ros_output_cloud;
        geometry_msgs::PointStamped userPickedPoint; // fill this from pointCloud data and publish
        float cloudPtX;
        float cloudPtY;
        float cloudPtZ;
        int  cloudPtR;
        int  cloudPtG;
        int  cloudPtB;

        if (firstCloud) { // do initializations here...
          // convert ROS input cloud to PCL format
          pcl::PointCloud<pcl::PointXYZRGB>::Ptr pcl_input_cloud(new
pcl::PointCloud<pcl::PointXYZRGB>);
          // convert Input cloud from ROS format to PCL format
          pcl::fromROSMsg(*ros_input_cloud, *pcl_input_cloud);
          ROS_INFO("received first pointcloud");
          ROS_INFO("width of cloud: %d ",pcl_input_cloud->width);
          ROS_INFO("height of cloud: %d",pcl_input_cloud->height);

          // convert input cloud to ROS image; this image will be suitable for 2-D openCV processing,
```

```
          // and its i,j matrix coordinates will correspond to i,j matrix coordinates of the point
cloud
          // thus, if one selects a pixel in 2-D, can look up corresponding 3-D coords...assuming the
pixel is valid
          pcl::toROSMsg (*ros_input_cloud, ROS_image_of_input_cloud); //convert the cloud
          // publish the ROS image message: ... not really necessary for this demo
          image_pub.publish(ROS_image_of_input_cloud);

          // first call: set up width, height and number of points for the global point-cloud object
"pcl_remembered"
          // this is needed since 2-D mouse selections relate to a prior pointCloud.  Thus, need to
save the prior
          // point Cloud in order to associate the 2-D mouse clicks with the corresponding 3-D points
          pcl_remembered.width = pcl_input_cloud->width;
          pcl_remembered.height = pcl_input_cloud->height;
          pcl_remembered.points.resize( pcl_remembered.width *  pcl_remembered.height);

          copyCldPtrToCloud(pcl_input_cloud,pcl_remembered); //stores current pointCloud for later
recall
          firstCloud=false;  //flag (global) to recognize that this initialization is done;
                             // would be cleaner with classes and a constructor
        }

        if(getPtFlag) {
          getPtFlag=false; // here if flag says a new mouse click selected a point; reset the trigger

          //if here, then two things happened: got a new pointcloud message and a new mouse click
occurred
          // mouse click refers to the OLD point cloud, so access data from pcl_remembered

          //pcl::PointXYZRGB pcl_pt; // could use this to help comb through pointCloud...

          // look up 3-D point info corresponding to mouse-click coords--compute 1-D index:
          pcl_index =  mouse_click_y * (pcl_remembered.width) + mouse_click_x; //computed 1-D index

          //alt: could use pcl_pt = pcl_input_cloud->at(row,col);  then access as pcl_pt.x, etc
          // get the data;

          cloudPtX = pcl_remembered.points[pcl_index].x;
          cloudPtY = pcl_remembered.points[pcl_index].y;
          cloudPtZ = pcl_remembered.points[pcl_index].z;
          ROS_INFO("selected point x,y,z = %f %f %f",cloudPtX,cloudPtY,cloudPtZ);
          cloudPtR = pcl_remembered.points[pcl_index].r;
          cloudPtG = pcl_remembered.points[pcl_index].g;
          cloudPtB = pcl_remembered.points[pcl_index].b;
          ROS_INFO("color of pt, RGB = %d %d %d",cloudPtR,cloudPtG,cloudPtB);

          // should publish target..will need to convert from left-camera optical frame

          // recognize new target only if data is valid:
          // the following tests if any of the values are NaN
          if (cloudPtX!=cloudPtX || cloudPtY!=cloudPtY || cloudPtZ!=cloudPtZ)  {
              pickedTarget=false;
              ROS_INFO("bad point: NaN data");
          }
          else  {
             pickedTarget=true; //this global flag indicates that a valid new point has been selected
             // publish this identified point as a PointStamped, suitable for use with frame transforms
             userPickedPoint.point.x=cloudPtX;
             userPickedPoint.point.y=cloudPtY;
             userPickedPoint.point.z=cloudPtZ;
             userPickedPoint.header = pcl_remembered.header; // this will keep the same time stamp and
reference frame
                                                             // as the original point cloud

             // and publish the result
             pickedPointPub.publish(userPickedPoint);
          }


          // make a new 2-D image from point cloud available for more mouse clicks
          // convert input cloud to ROS image and publish it:
```

```
          // Input cloud in PCL format
          pcl::PointCloud<pcl::PointXYZRGB>::Ptr pcl_input_cloud(new
pcl::PointCloud<pcl::PointXYZRGB>);

          // convert Input cloud from ROS format to PCL format
          pcl::fromROSMsg(*ros_input_cloud, *pcl_input_cloud);


          // here is an example of color filtering of a point cloud:
          // its use is suppressed here, but it works;
          // currently only allowing near black colors through--hard-coded filter bounds
          /*
          ros_output_cloud = filter_rgb(ros_input_cloud,
                                                                0, 47,  // red
                                                                0, 42,  // green
                                                                0, 52); // blue

          cloud_pub.publish(ros_output_cloud);
          */

          pcl::toROSMsg (*ros_input_cloud, ROS_image_of_input_cloud); //convert the cloud
          image_pub.publish(ROS_image_of_input_cloud); //publish this 2-D image--not necessary for
this demo, since
                                                       // ROS_image_of_input_cloud is a global var

          copyCldPtrToCloud(pcl_input_cloud,pcl_remembered); //save the new cloud to refer to after
            //receiving the next mouse click
        }

}



// callback function for mouse events
// save the coords of a mouse click on scene of interest--save these as global vars and use in
pointCloud callback
// also, make note that a new click occurred by setting the global flag "getPtFlag"
void onMouse(int event, int x, int y, int flags, void* param)
{
    Mat img,img3;
    cv_bridge::CvImagePtr cv_ptr;  //use cv_ptr->image in place of "image" in OpenCV manual example

    cv_ptr = cv_bridge::toCvCopy(ROS_image_of_input_cloud, enc::BGR8); //convert ROS message to openCV
format

    img = cv_ptr->image; // synonym...slightly simpler

    //img2 = img.clone();  //example of how to copy an image

    // respond only to left button clicks, in this example code:
    if (event == CV_EVENT_LBUTTONDOWN)
    {
        mouse_click_x = x;  //pass mouse coords as globals
        mouse_click_y = y;

        // some debug and info display:
        Vec3b p = img.at<Vec3b>(y,x);
        ROS_INFO("image RGB = %d %d %d",p[2],p[1],p[0]);
        ROS_INFO("image x,y = %d %d",mouse_click_x,mouse_click_y);
        getPtFlag=true; // flag to indicate new mouse click needs processing
    }

    imshow("image_window", img); //update the image display
}



// example of color filtering directly in pointCloud space
sensor_msgs::PointCloud2 filter_rgb(const sensor_msgs::PointCloud2ConstPtr& ros_input_cloud,
                                                                        int l_r,
                                                                        int u_r,
                                                                        int l_g,
```

```
                                                                int u_g,
                                                                int l_b,
                                                                int u_b)
{
        // Input cloud in PCL format
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr pcl_input_cloud(new pcl::PointCloud<pcl::PointXYZRGB>);
        // Filtered cloud in PCL format
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr pcl_filtered_cloud(new
pcl::PointCloud<pcl::PointXYZRGB>);

        sensor_msgs::PointCloud2 ros_filtered_cloud; // Filtered Cloud in ROS Format

        // convert Input cloud from ROS format to PCL format
        pcl::fromROSMsg(*ros_input_cloud, *pcl_input_cloud);

        pcl_filtered_cloud->width = pcl_input_cloud->width;
        pcl_filtered_cloud->height = pcl_input_cloud->height;
        pcl_filtered_cloud->points.resize(pcl_filtered_cloud->width * pcl_filtered_cloud->height);

        for(unsigned int i = 0; i < pcl_input_cloud->size(); ++i)
        {
                if(pcl_input_cloud->points[i].r < l_r || pcl_input_cloud->points[i].r > u_r)
                        continue;
                if(pcl_input_cloud->points[i].g < l_g || pcl_input_cloud->points[i].g > u_g)
                        continue;
                if(pcl_input_cloud->points[i].b < l_b || pcl_input_cloud->points[i].b > u_b)
                        continue;

                // if made it to here then the point is within all 3 color ranges
                // so copy the point to the filtered cloud
                pcl_filtered_cloud->points[i].x = pcl_input_cloud->points[i].x;
                pcl_filtered_cloud->points[i].y = pcl_input_cloud->points[i].y;
                pcl_filtered_cloud->points[i].z = pcl_input_cloud->points[i].z;
                pcl_filtered_cloud->points[i].r = pcl_input_cloud->points[i].r;
                pcl_filtered_cloud->points[i].g = pcl_input_cloud->points[i].g;
                pcl_filtered_cloud->points[i].b = pcl_input_cloud->points[i].b;
        }

        // convert Filtered cloud from PCL format to ROS format
        pcl::toROSMsg(*pcl_filtered_cloud, ros_filtered_cloud);

        ros_filtered_cloud.header = ros_input_cloud->header;

        return ros_filtered_cloud;
}



int main(int argc, char** argv)
{
        ros::init(argc, argv, "point_cloud_HMI");
        ros::NodeHandle nh_processed,nh_orig_image,nh_handGoal;
        //image_transport::ImageTransport it(nh_orig_image); // fancy class used for dealing with
image topics

        ros::Rate looprate(10); //10Hz max loop rate
        ROS_INFO("subscribing to multisense points2...");

        // subscribe to this pointcloud2 topic
        ros::Subscriber sub = nh_orig_image.subscribe("/multisense_sl/camera/points2", 1,
cloud_callback);

        // convert the incoming, original pointcloud into an image and publish it to this topic:
        image_pub = nh_orig_image.advertise<sensor_msgs::Image>(ros::this_node::getName() + "/
cloud2image", 1);
        // the above is not really needed, since the mouse-interaction is integrated in this node

        // publish the processed point-cloud topic to this topic...
        // uncomment to restore
        //cloud_pub = nh_processed.advertise<sensor_msgs::PointCloud2>(ros::this_node::getName() + "/
cloud", 1);
```

```
        // publisher for selected points:
        pickedPointPub = nh_handGoal.advertise<geometry_msgs::PointStamped>("userPickedPoint",1);


        ROS_INFO("waiting for pointcloud from Atlas..."); // need to receive an image from callback
before can display it;
        // first iteration of callback fnc sets "firstCloud" flag to "false"

        while(firstCloud) {
           ros::spinOnce(); // let the callback get an image;
           ros::Duration(0.1).sleep();
        }

        ROS_INFO("setting up named window");
        namedWindow("image_window"); //window for openCV displays

        // assign "onMouse()" as the callback fnc for mouse events
        setMouseCallback("image_window", onMouse, 0);

        // done with setups; enter main loop;  callbacks do all the work now

        while(ros::ok()) {
          ros::spinOnce(); // allow getting fresh pointclouds received;
          waitKey(100); // images update only after mouse events;
          looprate.sleep();
        }
}
```