Matthew Nease

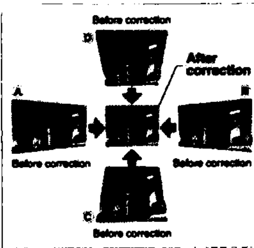
Projection Correction

Abstract

The goal of this project was to develop a system using the Microsoft Kinect to track the varying depths and 3D coordinates of whatever scene it's pointing at. A connected projector can then project an undistorted image onto one of the three largest visible planes in the scene (selected by the user), free from warping such as the Keystone effect.

Body

The primary goal of this project is to develop a system using the Microsoft Kinect to track the varying depths and 3D locations of the surface it's pointing at, and project an undistorted image onto



the surface (see image at left). At the beginning of the quarter I outlined a few milestones so that at each stage there would be a working design that accomplishes at least some part of the overall goal, in order to accommodate both best and worst case scenarios. These milestones included: gathering/displaying data from the Kinect, determining the visible planes in the scene, calculating the best-fit plane based on the average normal, projecting all points onto this plane, finding the best-fit rectangle around the points on the plane, and lastly computing the homography from the projection to the surface on which we wish to display.

First, interaction with the Kinect through my PC required the downloading of drivers. Installing Point Cloud Library and its dependencies, boost, Eigen, FLANN, Vtk, Qt, QHull, and OpenNi (for the Kinect connection) then provided some built-in functionality for accessing the Kinect's data and displaying it. I spent some time looking through the various sample code and demos, and decided to use the library's openni_planar_segmentation.cpp as my starter code. This code connected to the Kinect to read the 3D data for the surfaces at which it is pointing, and return a PointCloud containing the inliers of the largest plane visible. I then added code to display this PointCloud using PCLVisualizer. From there, I began adding code to do the processing I wanted, pulling together some of it from various other sample PCL files, and storing the results at each major step in a new PointCloud for display/analysis later.

The first step was to separate out different planes from the Kinect's input, which was accomplished with help from pcl's SACSegmentation and ExtractIndices classes. First, the cloud was segmented into inliers and outliers of the largest plane present. The inliers were then stored in a cloud, and the outliers were processed as the new input cloud in the same manner as before to find the next largest plane. When all planes had been found, the plane requested by the user (0, 1, or 2) was copied into cloud_plane for further processing.

From this cloud, the normal vector for each point in the cloud was computed using the pcl::Normal class. Averaging these out gave the normal for the plane as a whole, with which the best fit plane for the points was able to be defined. With this knowledge, all the inliers in the PointCloud could be projected onto the plane based on the following equations:

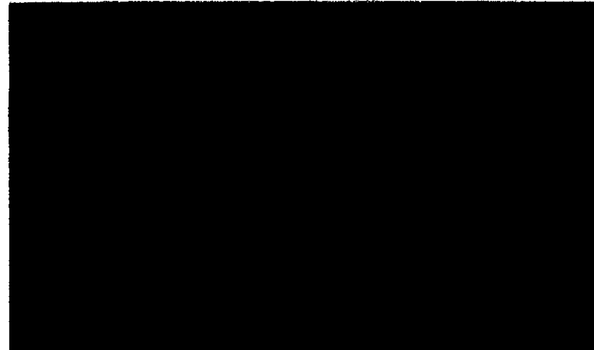
$$t_0 = -\frac{au+bv+cw}{a^2+b^2+c^2}$$

$$x_0 = u + at_0\,, \qquad y_0 = v + bt_0\,, \qquad z_0 = w + ct_0$$

where a, b, and c are the x,y,z values of the plane's normal vector, and u, v, and w correspond to the x, y, z coordinates of the point to be projected onto the plane. The centroid (chosen randomly, and later called p0) must first be subtracted from the points before the projection can be applied. The points x_0,

y_0, and z_0 were then stored as the new coordinates for the point in the flattened cloud, after adding the centroid back in.

In the image at right, which is displaying only the largest plane (as opposed to the 3 largest as the program currently does), the blue dots represent the 3D points of the input plane. Projecting the points on a flat plane as described above results in the red points.



Using these new points, I defined a new 2d PointCloud with a coordinate system that lies on the plane, and calculated each point's coordinates in relation to that coordinate system (with each point's z coordinate being zero). The first step for this was to define a point to use as the origin and then choose two other points from the cloud, defining two vectors from the origin to these two points. Let p0 be the centroid of the points and p1, p2 be two randomly selected points which have been projected onto the plane. We compute vectors v1 = p1-p0 and v2=p2-p0 which lie in the plane.

Finding the magnitude (length) of v1 and dividing by that magnitude turned it into a unit vector, which could then be used to find v3, the orthonormal basis with v1. This was done by projecting v2 onto v1_hat and finding the component of v2 orthogonal to v1_hat. This result was then also turned into a unit vector, as in the equations below:

v1_hat = v1 / |v1|
*v3 = v2 - (v2'*v1)*v1_hat*
v3_hat = v3 / |v3|

This new orthonormal basis is then checked for errors, and if it passes, then all the points on the plane are converted to the new coordinate system such that

$$s = (x_i - u)^T * v1_hat \qquad \& \qquad s = (x_i - u)^T * v3_hat$$

Using the OpenCV library and its cv::minAreaRect(points) function, the corners of the best-fit rectangle of the flat z-aligned plane were found. These four corner points were then translated back to original 3D coordinates using

$$newPoint = corner_x * v1_{hat} + corner_y * v3_{hat} + p0$$

for each corner point.

The points in relation to the orthonormal basis are represented by the green plane in the image above, and the minimum area rectangle is shown as a grey box around the original plane. With the newfound minimum area rectangle for the original 3d plane of choice, the next step was to apply a homography between its corner points and those of the image to be projected. This was computed using calibrated values for the projector, as well as the projector's coordinates in relation to the Kinect.

First, the 3D coordinates of the plane to be projected on were flattened to 2D, by multiplying the matrices:

*Flattened point = (calibration * (rotation * translation)) * pointMatrix*

and then extracting the x and y coordinates of the resulting vector and dividing by the z coordinate. The calibration matrix held the x and y focal lengths as well as the center point of the projection (640 x 400). The rotation matrix was simply the identity matrix based on our relative locations with the projector above the Kinect. The translation matrix contained the 3D translation between the eye of the Kinect and the center of the projector, in our case -.05, -.075, -.075 . Lastly, pointMatrix was simply a vector containing each individual point to be flattened, as the method was run in a loop for each individual point. The end result in effect determined which pixels in the projector were part of the plane we wished to project onto.

With the new flattened points, a perspective transformation could then be computed between the four image [src] coordinates, (0, 0), (src.cols, 0), (src.cols, src.rows), and (0, src.rows), and the four newly flattened corner points of the plane. After computer the transformation, it was applied to every point in the image, so that it would directly overlap with the plane on which it was being projected. However, the image would not always appear upright in the projection, as the four corners of the plane would not consistently match up with the four corners of the image.

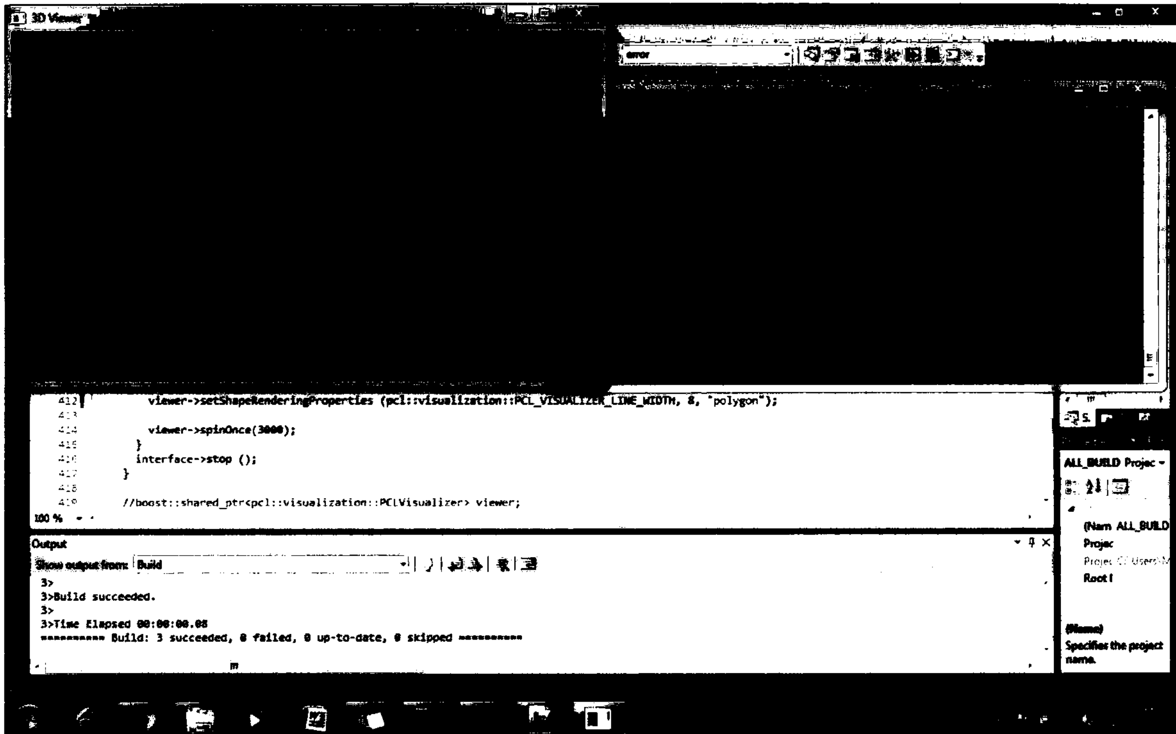
To remedy this, the previous ordering of the plane's four corners were kept track of, and a match was found with the starting corner to ensure the homography would consistently start at the same corner. Even with the same starting corner, however, the second point could be either one of the two adjacent points. With the start point solved, consistent ordering of the other points was solved by taking the cross product of the two vectors from the start point and looking at the z value. A simplified equation for the cross product could be used, since the z values of the vectors were both zero:

*z = vecta.x*vectb.y - vecta.y*vectb.x*

If the z value was negative (pointing towards the Kinect), then the ordering was correct, and the image would appear upright. If it was positive, then the ordering was reversed so that the image wouldn't flip to display upside-down.

With the correct start point and ordering established, the image could be displayed full screen on the computer connected to the projector, and when the image hit the surface, it would appear upright and un-warped.

The below image shows sample output from both the console (which prints cloud data and corner point coordinates) and graphical display (which shows various planes used as described above). The end result was a system that reads in the 3D data of what it's looking at and offers a corrected projection on the plane of choice, no matter what angle or surface the projector is pointing at.


3D Viewer
viewer->setShapeRenderingProperties (pcl::visualization::PCL_VISUALIZER_LINE_WIDTH, 8, "polygon");
viewer->spinOnce(3000);
}
interface->stop ();
}
//boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer;
100 %
Output
Show output from: Build
3>
3>Build succeeded.
3>
3>Time Elapsed 00:00:00.08
========== Build: 3 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
ALL_BUILD Projec
(Name)
Specifies the project name.