Date: <19.02.2018> Name:< ZHONG YU >

Summerschool Project report

< 3D reconstruction techniques >

1. Task

In this project, we used two camara to connect with PC, then used Matlab to control this two camara to take the picture from the object in the same time. Finally, we used Matlab to reconstruction the object.

2. Theoretical pre-examinations

➢ Image obtaining

Stereo image acquisition is based on stereo vision. There are many ways to obtain images which mainly depends on the application of occasions and purposes, but also need to consider the influence of such factors including a viewpoint points difference, lighting conditions, camera performance, and landscape characteristic. Thus we can calculate the three-dimensional.

➢ Camera Calibrator

To establish imaging model, camera calibration is used to get the camera position and attribute parameters to determine the corresponding relationship between the point of object and image point in space coordinate system. In this system, both the camera need to be calibrated. If the camera is fixed, when from 2-d computer image coordinates 3-d information, only need a single calibration.

1) World Coordinate transform to Camera Coordinate

There are two different coordinates including world coordinate and camera coordinate. World coordinate (X_W,Y_W,Z_W) is 3D rectangular coordinate system. The original point is in the lens optical center, x and y axis is parallel to the two sides of the plane. And the z-axis is the lens axis, which is perpendicular to the plane.

The transform function is as followed:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \quad (1)$$

R is 3*3 Rotation Matrices; t is Translation Vectors; $(x_c, y_c, z_c, 1)^T$ is homogeneous in camera coordinate; $(x_w, y_w, z_w, 1)^T$ is homogeneous in camera coordinate.

2) Image Coordinate transform to Pixel Coordinate

Figure 1 is the pixel coordinate of u-o-v, which is a 2D rectangular coordinate. In the pixel coordinate, the unit is pixel.

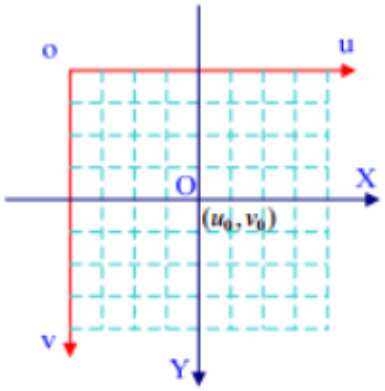

Figure 1 u-o-v coodinate

[http://blog.csdn.net/lql0716/article/details/71973318?locationNum=8&fps=1][17.02.2019]

The function of image coordinate transform to pixel coordinate is as follow:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} 1/dX & 0 & u_0 \\ 0 & 1/dY & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \tag{2}$$

Where dX、dY is the pixel size in the X, Y axis, and u_0, v_0 is Principal Point axis.

3) Pinhole camera Model

Pinhole camera is common camera calibration, which is known as imaging theory of holes. The core of the Pinhole camera model is involved in the coordinate system transformation. Figure 2 is the model of pinhole camera.

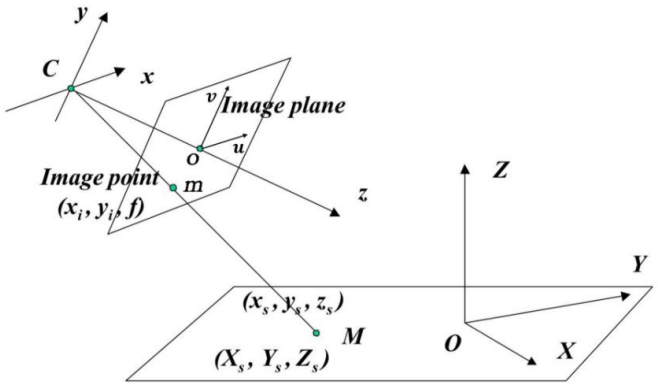

Figure 2 Pinhole camera Model

[http://blog.csdn.net/yhl_leo/article/details/49304265][17/02/2018]

Table 1 The meaning of symbol in the functions

Symbol	**Meaning**
C	Optical center of lens

o	Principal point
f	Focal length
M	the observed three-dimensional point
m	The image point of observed three-dimensional point
o-uv	Image coordinate
C-xyz	Image space coordinate
O-XYZ	Cartesian Coordinate System
(x_i, y_i, f)	Image point m in coordinates with respect to C, xyz in space.
(x_s, y_s, f_s)	The point M is the coordinate in space coordinate C- xyz.
(X_S, Y_S, Z_S)	The point M is in the coordinates of the object's coordinate system O-XYZ.

From the Figure 2, C, m, M is in the straight line which is collinear. According to Collinearity equation:

$$x - x_0 = -f \frac{\mathbf{R}_{11}(X - X_0) + \mathbf{R}_{21}(Y - Y_0) + \mathbf{R}_{31}(Z - Z_0)}{\mathbf{R}_{13}(X - X_0) + \mathbf{R}_{23}(Y - Y_0) + \mathbf{R}_{33}(Z - Z_0)}$$
$$y - y_0 = -f \frac{\mathbf{R}_{12}(X - X_0) + \mathbf{R}_{22}(Y - Y_0) + \mathbf{R}_{32}(Z - Z_0)}{\mathbf{R}_{13}(X - X_0) + \mathbf{R}_{23}(Y - Y_0) + \mathbf{R}_{33}(Z - Z_0)} \quad (3)$$

The matrix form is

$$\begin{bmatrix} X - X_0 \\ Y - Y_0 \\ Z - Z_0 \end{bmatrix} = \lambda \mathbf{R} \begin{bmatrix} x \\ y \\ -f \end{bmatrix} \quad (4)$$

Where the scale coefficient is λ.

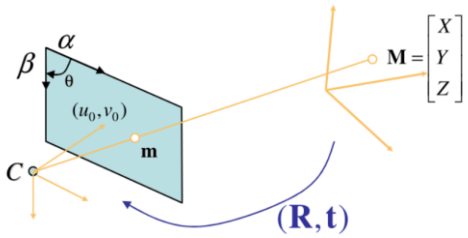

Figure 3 Transform Model

[http://blog.csdn.net/yhl_leo/article/details/49304265][17/02/2018]

In computer vision, points are represented by means of augmented vectors. $\tilde{m} = [u, v, 1]^T$ is represented m, $\tilde{M} = [X, Y, Z, 1]$ is represented M. The relationship between M and m can be described that:

$$s\tilde{m} = A[R \;\; t]\tilde{M} \equiv P\tilde{M} \quad (5)$$

$$\mathbf{A} = \begin{bmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6)$$

$$\mathbf{P} = \mathbf{A}[\mathbf{R}\ \mathbf{t}] \tag{7}$$

Where (R,t) is extrinsic parameters; A is intrinsic matrix; P is reprojection matrix. (u_0,v_0) is image principal point coordinates. α, β is pixel ratio of width and height. γ is projection matrix, including external parameters and internal parameters.

To sum up, all camera calibration methods are calculated the 11 parameters including the extraction of internal and external parameters.

- **Feature extraction**

For determined by the multi-view parallax of 3-d information, the key is to determine the same object point in the scene in the corresponding relationship of different images. Choosing appropriate image features and matching is one of the method to solve the problem. Feature is the pixel or the pixel set or their abstract expression, commonly used point feature, linear feature and regional features, etc. In general, large scale feature contains more informations, the number itself is less, easily to get fast matching, but extraction and description is relatively complex, and the accuracy of positioning is poor. And small scale features have high positioning accuracy, simple expression, but the number is more, the amount of information contained is less, and therefore requires a strong constraint criterion and the matching strategy.

So it needs to consider many kinds of factors and according to the different choices of landscape characteristics and application requirements. In general, as for the irregular shape and high mutation scenario, it suits to extraction point features, because take the characteristics such as line area is both difficult and will also produce error. As for the regular scene, line features and regionin features are more easily to realize the fast matching.

- **Stereo matching**

Stereo matching is based on calculation of the selected characteristics. It establishs the corresponding relationship between features, which is the same space point in the different image points.

Stereo matching method is divided into two categories, including gray correlation and feature matching. Gray correlation directly uses pixel gray to achieve the matching. The advantage is matching results is not affected by precision and density of feature detection, which can get very high positioning accuracy and dense in parallax surface. The disadvantage is dependent on the statistical properties of the image gray level, which has high sensitive reflection in the scenery surface structure and light. So in the the space scenery surface, it lacks of enough texture details and has large imaging distortion.

The matching method needs to solve the following problems: select the correct matching feature; look for the essential properties of the features; establish a stable algorithm that can correctly match the selected feature.

3. Equipment/ Materials

- PC
- two cameras(Logitech C920 HD Pro HD-Webcam)

- Matlab
- a hardboard

4. Experimental details/ procedure

➢ Camera select

In this project, we choose Logitech C920 HD Pro HD-Webcam. Figure This type camera is used in operating system of Windows®7/8/10. And the USB Video Class model is MacOS®10.6 or higher/Chrome OS™/ Android™5.0 or higher.

Figure 4 Logitech® HD Pro Webcam C920
[https://www.conrad.de/de/full-hd-webcam-1920-x-1080-pixel-logitech-hd-pro-webcam-c920-klemm-halterung-1386550.html]17/02/2019

The camera's parameters are as the Table 2 showed.

Table 2 Camera's parameters

Diagonal Field of View (FOV)	78°
Focal Length	3.67 mm
Image Capture (4:3 SD)	N/A
Image Capture (16:9 W)	2.0 MP, 3 MP*, 6 MP*, 15 MP*
Video Capture (4:3 SD)	N/A
Video Capture (16:9 W)	360p, 480p, 720p, 1080p
Frame Rate (max)	1080p@30fps
Right Light	RightLight 2
Video Effects (VFX)	N/A
Buttons	N/A
Indicator Lights (LED)	Yes
Privacy Shade	No
Tripod Mounting Option	Yes

Figure 5 is the tripod we selected. We choose the Rollei Monkey Pod, which can be used in digital SLR cameras, digital cameras, cameras and smartphones. It can be 360° rotation, which is easy to adjust the angle.

Figure 5 Logitech® HD Pro Webcam C920 [https://www.conrad.de/de/tripod-rollei-5020797-arbeitshoehe8-25-cm-schwarz-1338984.html][17/02/2018]

- **Camera Calibration**

Based steps are as followed:

- ✓ Printing out a checkerboard in the size of 25mm*25mm, then stick it on a hardboard flat surface, which act as calibration object. Figure 6 is the checherboard.

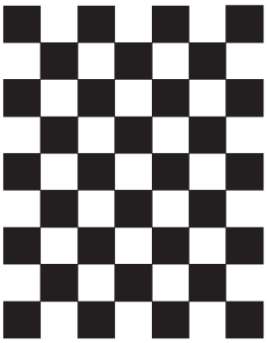

Figure 6 25mm*25mm Checherboard

- ✓ Taking different pictures throught adjusting the calibration object or the direction of the camera.
- ✓ extracting checkerboard angular point from pictures.
- ✓ In the case of an ideal without distortion, it has five internal and six external parameters.
- ✓ The distortion coefficient of the radial distortion is estimated by the least square method.

➢ **Matlab Camera Calibration**

- ✓ Open matlab, find the "**Camera Calibrator**" and open it.
- ✓ After add the image, it will be prompted to set the actual size of the checkerboard and click ok.
- ✓ Click "**Calibrate**" begin to calculate.
- ✓ Click "**Export Camera Parameters**" and output to the matlab command window. The export camera parameters are as Table 3 shown.

Table 3 Export Camera Parameters from Matlab

	FocalLength		Principal Point		Radial Destotion		Mean Reprojection Error
1	1684	1718.3	1104.7	865.3028	-0.1693	0.9131	0.8071
2	1677	1715.1	1123.8	874.4209	-0.0524	0.1904	0.6283
3	1612.6	1690.7	1147.7	858.4809	-0.1127	0.0811	1.1788
4	1622.8	1663.5	1145.4	838.2037	-0.0921	0.0402	1.0896
5	1652.6	1692.3	1123.7	847.3053	-0.0684	0.0444	0.9555
6	1685.3	1724.6	1114.3	845.2815	-0.0707	0.0672	0.8415
7	1684.9	1722.5	1116.3	861.2194	-0.0872	0.3941	0.7482
8	1629.4	1665.6	1135.5	846.8813	-0.0921	0.0753	1.0859
9	1636.3	1674.1	1120	854.7019	-0.0581	0.0241	0.7453

10	1650.4	1691.6	1132.7	846.7747	-0.089	0.0945	0.9705
11	1627.4	1664.2	1135	851.185	-0.009	0.0723	0.9275
12	1651	1692.8	1131.1	848.6661	0.0782	0.0704	0.9295
13	1651.6	1691.3	1127.9	846.4117	-0.0816	0.0747	1.0081
14	1651.6	1691.9	1127.3	848.5823	-0.0762	0.0631	0.959

- **Used two camera to take pictures in the same time**

Used Matlab to control the camera to realize automatic camera function at the same time. The Matlab code is shown as followed:

```
function twocamerapic()
clc;
clearvars;
webcamlist
cam1=webcam(1);
cam2=webcam(2);
cam3=webcam(3);
cam1.AvailableResolutions;
cam2.AvailableResolutions;
cam3.AvailableResolutions;
cam1.Resolution='640x480';
cam2.Resolution='640x480';
cam3.Resolution='640x480';
% preview(cam);
% preview(cam2);
n=10;
for i=1:n
disp(i);
%img1=snapshot(cam1);
img2=snapshot(cam2);
img3=snapshot(cam3);
%imshow(img1);
imshow(img2);
saveas(1,['l-',num2str(i),'.jpg']);
imshow(img3);
saveas(1,['r-',num2str(i),'.jpg']);
pause(1);
end
end
```

5. Results

By taking some picture from different angles and directions for calibration is in need. The pictures are shown in Figure7. The reconstruction and rectification of Checkerboard are

shown in Figure 8 and 9.

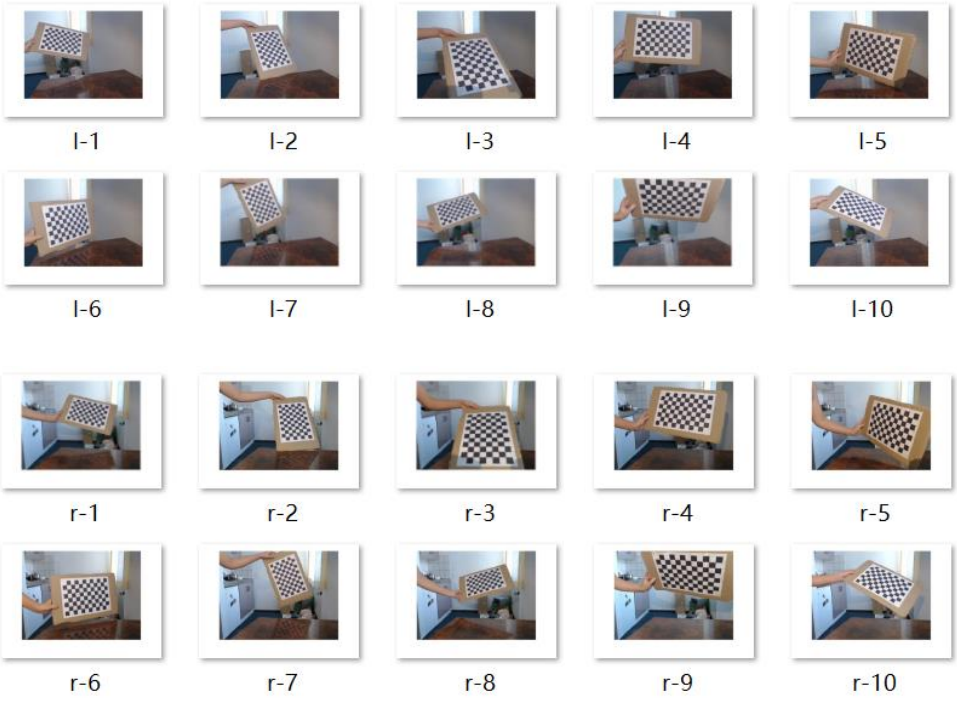

Figure 7 Pictures from different angles and directions

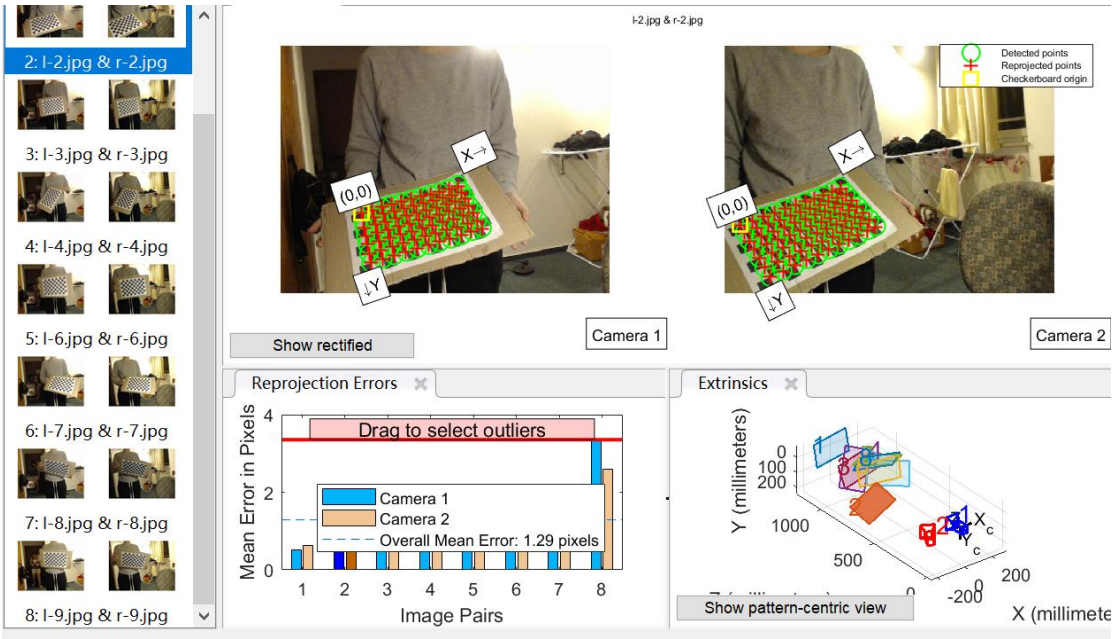

Figure 8 Reprojection of Checkerboard

Figure 9 Rectification and Disparity map of Checkerboard

The cloud of points of object is shown in Figure 10. And Figure 11 shown the reconstruction of an object and its disparity map.

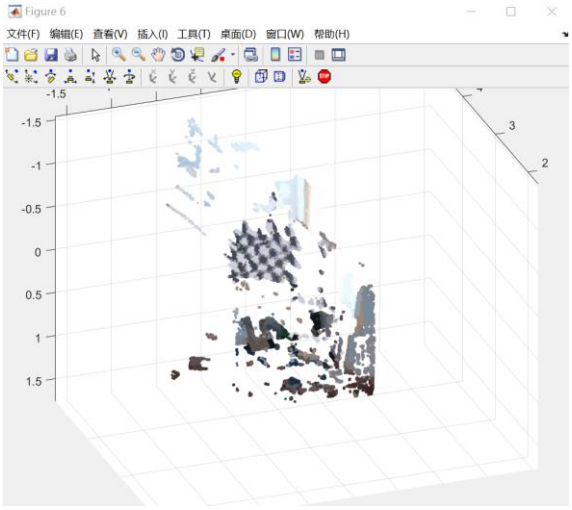

Figure 10 Reconstruction of Checkerboard

Figure 11 Reconstruction of Checkerboard

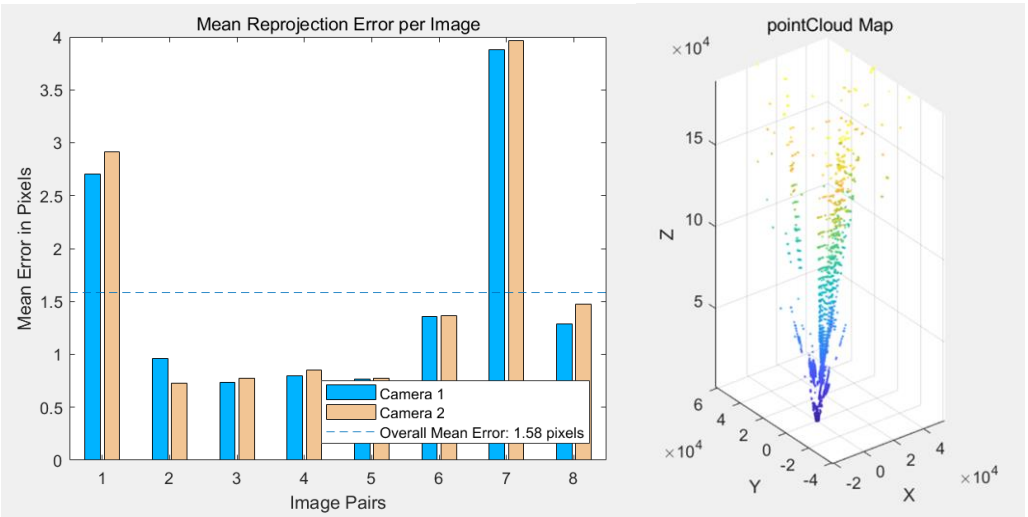

Figure 11 The mean reprojection error and pointcould map of an Object

6. Conclusion

After complete all the tasks, I have better understanding of the 3D reconstruction and also learn some theories and principles of 3D reconstruction as well as the operation aspect.

Appendix

Scripts

Twocamerapic

```
function twocamerapic()
clc;
clearvars;
webcamlist
cam1=webcam(1);
cam2=webcam(2);
cam3=webcam(3);
cam1.AvailableResolutions;
cam2.AvailableResolutions;
cam3.AvailableResolutions;
cam1.Resolution='640x480';
cam2.Resolution='640x480';
cam3.Resolution='640x480';
% preview(cam);
% preview(cam2);


n=10;

for i=1:n
disp(i);
%img1=snapshot(cam1);
img2=snapshot(cam2);
img3=snapshot(cam3);
%imshow(img1);
imshow(img2);
saveas(1,['l-',num2str(i),'.jpg']);
imshow(img3);
saveas(1,['r-',num2str(i),'.jpg']);
pause(1);
end
end
```

reconstruction

```
function reconstruction ()
numImagePairs = 9;
imageFiles1 = cell(numImagePairs, 1);
imageFiles2 = cell(numImagePairs, 1);
imageDir = fullfile('D:',  'matlabProjets');
for i = 1:numImagePairs
    imageFiles1{i} = fullfile(imageDir, sprintf('l-%d.jpg', i));
```

```
    imageFiles2{i} = fullfile(imageDir, sprintf('r-%d.jpg', i));
end

[imagePoints, boardSize,pairsUsed] =
detectCheckerboardPoints(imageFiles1,imageFiles2);%¼ì²âÆåÅÌ½ÇµãÌ

squareSize = 25; % millimeters
worldPoints = generateCheckerboardPoints(boardSize, squareSize);

imageFiles1 = imageFiles1(pairsUsed);
figure;
for i = 1:numel(imageFiles1)
     I = imread(imageFiles1{i});
     subplot(3,3,i);
     imshow(I);
     hold on;
     plot(imagePoints(:,1,i,1),imagePoints(:,2,i,1),'ro');
end
annotation('textbox',[0 0.9 1 0.1],'String','Camera 1',...
    'EdgeColor','none','HorizontalAlignment','center')

% Compute the stereo camera parameters.
stereoParams = estimateCameraParameters(imagePoints, worldPoints);
figure;
showReprojectionErrors(stereoParams);
% Read in the stereo pair of images.
I1 = imread('left01.jpg');
I2 = imread('right01.jpg');

% Rectify the images.
[J1, J2] = rectifyStereoImages(I1, I2, stereoParams);

% Display the images before rectification.
figure; imshow(cat(3, I1(:,:,1), I2(:,:,2:3)), 'InitialMagnification', 50);
title('Before Rectification');

% Display the images after rectification.
figure; imshow(cat(3, J1(:,:,1), J2(:,:,2:3)), 'InitialMagnification', 50);
title('After Rectification');

disparityMap = disparity(rgb2gray(J1), rgb2gray(J2));
figure; imshow(disparityMap, [0, 64], 'InitialMagnification', 50);
colormap('jet');
colorbar;
```

```
title('Disparity Map');

pointCloud = reconstructScene(disparityMap, stereoParams);
figure;
pcshow(pointCloud);
title('pointCloud Map');
xlabel('X');
ylabel('Y');
zlabel('Z');
% Convert from millimeters to meters.
pointCloud = pointCloud / 1000;


% Reduce the number of colors in the image to 128.
[reducedColorImage, reducedColorMap] = rgb2ind(J1, 128);

% Plot the 3D points of each color.
hFig = figure; hold on;
set(hFig, 'Position', [1 1 840   630]);
hAxes = gca;

X = pointCloud(:, :, 1);
Y = pointCloud(:, :, 2);
Z = pointCloud(:, :, 3);

for i = 1:size(reducedColorMap, 1)
    % Find the pixels of the current color.
    x = X(reducedColorImage == i-1);
    y = Y(reducedColorImage == i-1);
    z = Z(reducedColorImage == i-1);

    if isempty(x)
        continue;
    end

    % Eliminate invalid values.
    idx = isfinite(x);
    x = x(idx);
    y = y(idx);
    z = z(idx);

    % Plot points between 3 and 7 meters away from the camera.
    maxZ =7;
    minZ = -5;
```

```
    x = x(z > minZ & z < maxZ);
    y = y(z > minZ & z < maxZ);
    z = z(z > minZ & z < maxZ);

    plot3(hAxes, x, y, z, '.', 'MarkerEdgeColor', reducedColorMap(i, :));
    hold on;
end

% Set up the view.
grid on;
cameratoolbar show;
axis vis3d
axis equal;
set(hAxes,'YDir','reverse', 'ZDir', 'reverse');
camorbit(-20, 25, 'camera', [0 1 0]);
end
```

detect checker board

```
function [points, boardSize] = detectCheckerboard(I, sigma, peakThreshold)
%#codegen
[cxy, c45, Ix, Iy, Ixy, I_45_45] = ...
    vision.internal.calibration.checkerboard.secondDerivCornerMetric(I,
sigma);
[Ix2, Iy2, IxIy] = computeJacobianEntries(Ix, Iy);

points0 = vision.internal.calibration.checkerboard.find_peaks(cxy,
peakThreshold);
scores0 = cxy(sub2ind(size(cxy), points0(:, 2), points0(:, 1)));
board0 = growCheckerboard(points0, scores0, Ix2, Iy2, IxIy, 0);

points45 = vision.internal.calibration.checkerboard.find_peaks(c45,
peakThreshold);
scores45 = c45(sub2ind(size(c45), points45(:, 2), points45(:, 1)));
board45 = growCheckerboard(points45, scores45, Ix2, Iy2, IxIy, pi/4);

points = [];
boardSize = [0 0];
if board0.isValid && board0.Energy < board45.Energy
    board0 = orient(board0, I);
    [points, boardSize] = toPoints(board0);
    points = vision.internal.calibration.checkerboard.subPixelLocation(Ixy,
points);
elseif board45.isValid
```

```
    board45 = orient(board45, I);
    [points, boardSize] = toPoints(board45);
    points = vision.internal.calibration.checkerboard.subPixelLocation(I_45_45,
points);
end
end
%--------------------------------------------------------------------------
function [Ix2, Iy2, Ixy] = computeJacobianEntries(Ix, Iy)
Ix2 = Ix .^ 2;
Iy2 = Iy .^ 2;
Ixy = Ix .* Iy;
G = fspecial('gaussian', 7, 1.5);
Ix2 = imfilter(Ix2, G);
Iy2 = imfilter(Iy2, G);
Ixy = imfilter(Ixy, G);

end
%--------------------------------------------------------------------------
function board = growCheckerboard(points, scores, Ix2, Iy2, Ixy, theta)

% Exit immediately if no corner points were found
if isempty(scores)
    if isempty(coder.target)
        board = struct('BoardIdx', zeros(3), 'BoardCoords', zeros(3,3,3), ...
            'Energy', Inf, 'isValid', 0);
    else
        board = vision.internal.calibration.checkerboard.Checkerboard;
    end
    return;
end
% only use corners with high scores as seeds to reduce computation
seedIdx = 1:size(points, 1);
[~, sortedIdx] = sort(scores(seedIdx), 'descend');
seedIdx = seedIdx(sortedIdx);
if numel(sortedIdx) > 2000
    seedIdx = seedIdx(1:min(2000, round(numel(seedIdx)/2)));
end
angleThreshold = 3 * pi / 16;
if isempty(coder.target)
    v1_matrix = [];
    v2_matrix = [];
    seedIdx_matrix = [];

    for i = seedIdx
```

```
        [v1, v2] = cornerOrientations(Ix2, Iy2, Ixy, round(points(i, :)));
        alpha1 = abs(atan2(v1(2), v1(1)));
        alpha2 = abs(atan2(v2(2), v2(1)));
        if abs(abs(alpha1 - pi) - theta) > angleThreshold && ...
                abs(abs(alpha2 - pi) - theta) > angleThreshold
            continue;
        else
            v1_matrix = [v1_matrix;v1]; %#ok<AGROW>
            v2_matrix = [v2_matrix;v2]; %#ok<AGROW>
            seedIdx_matrix = [seedIdx_matrix;i]; %#ok<AGROW>
        end
    end

    board =
visionInitializeAndExpandCheckerboard(seedIdx_matrix,single(points),v1_matri
x,v2_matrix);
else
    previousBoard = vision.internal.calibration.checkerboard.Checkerboard;
    currentBoard = vision.internal.calibration.checkerboard.Checkerboard;
    for i = 1:numel(seedIdx)
        [v1, v2] = cornerOrientations(Ix2, Iy2, Ixy,
round(points(seedIdx(i), :)));
        alpha1 = abs(atan2(v1(2), v1(1)));
        alpha2 = abs(atan2(v2(2), v2(1)));
        if abs(abs(alpha1 - pi) - theta) > angleThreshold && ...
                abs(abs(alpha2 - pi) - theta) > angleThreshold
            continue;
        end

        currentBoard.initialize(seedIdx(i), points, v1, v2);
        expandBoardFully(currentBoard);
        if currentBoard.Energy < previousBoard.Energy
            tmpBoard = previousBoard;
            previousBoard = currentBoard;
            currentBoard = tmpBoard;
        end
    end
    board = previousBoard;
end
end

%-------------------------------------------------------------------------
function [v1, v2] = cornerOrientations(Ix2, Iy2, Ixy, p)
% The orientation vectors are the eigen vectors of the
```

```
% structure tensor:
% [Ix^2  Ixy ]
% [Ixy   Iy^2]

a = Ix2(p(2), p(1));
b = Ixy(p(2), p(1));
c = Iy2(p(2), p(1));

% % Computing eigenvectors "by hand", because the eig() function behaves
% % differently in codegen.
% % Since the matrix is positive-semidefinite, its eigenvectors are
% % orthogonal. Compute the first eigenvector, then swap its elements and
% % negate the y-component to make the second one.
sm = a + c;
df = a - c;
adf = abs(df);
tb = b + b;
ab = abs(tb);

if adf > ab
    rt = adf * sqrt(1 + (ab/adf)^2);
elseif adf < ab
    rt = ab * sqrt(1 + (adf/ab)^2);
else
    rt = ab * sqrt(2);
end

if sm < 0
    sgn1 = -1;
else
    sgn1 = 1;
end

if df > 0
    cs = df + rt;
    sgn2 = 1;
else
    cs = df - rt;
    sgn2 = -1;
end

acs = abs(cs);
if acs > ab
    ct = -tb / cs;
```

```
    sn1 = 1 / sqrt(1 + ct * ct);
    cs1 = ct * sn1;
else
    if ab == single(0)
        cs1 = single(1);
        sn1 = single(0);
    else
        tn = -cs / tb;
        cs1 = 1 / sqrt(1 + tn * tn);
        sn1 = tn * cs1;
    end
end
if sgn1 == sgn2
    tn = cs1;
    cs1 = -sn1;
    sn1 = tn;
end

v1 = [-sn1, cs1];
v2 = [cs1, sn1];

% Rotate the vectors by 45 degrees to align with square edges.
R = [cos(pi/4) -sin(pi/4); sin(pi/4) cos(pi/4)];
v1 = v1 * R;
v2 = v2 * R;
end

%--------------------------------------------------------------------------

function [points, boardSize] = toPoints(this)
% returns the points as an Mx2 matrix of x,y coordinates, and
% the size of the board

if any(this.BoardIdx(:) == 0)
    points = [];
    boardSize = [0 0];
    return;
end

numPoints = size(this.BoardCoords, 1) * size(this.BoardCoords, 2);
points = zeros(numPoints, 2);
x = this.BoardCoords(:, :, 1)';
points(:, 1) = x(:);
y = this.BoardCoords(:, :, 2)';
```

```
points(:, 2) = y(:);
boardSize = [size(this.BoardCoords, 2)+1, size(this.BoardCoords, 1)+1];
end

%-------------------------------------------------------------------------
function board = orient(board, I)
    if ~isinf(board.Energy)
        % orient the board so that the long side is the X-axis
        if size(board.BoardCoords, 1) < size(board.BoardCoords, 2)
            board = rot90_checkerboard(board, 1);
        end
        % try to orient the board so that (0,0) is on a black square
        if ~isUpperLeftBlack(board, I)
            board = rot90_checkerboard(board, 2);
        end

        % if both sides are odd or both sides are even, make sure
        % that (0,0) is at the upper-left corner.
        if ~xor(mod(size(board.BoardCoords, 1), 2) == 0,...
            mod(size(board.BoardCoords, 2), 2) == 0)
            if any(board.BoardCoords(1,1,:) > board.BoardCoords(end, end, :))
                board = rot90_checkerboard(board, 2);
            end
        end
    end
end

%-------------------------------------------------------------------------
function board = rot90_checkerboard(board, k)
board.BoardIdx = rot90(board.BoardIdx, k);
newBoardCoords1 = rot90(board.BoardCoords(:,:,1), k);
newBoardCoords2 = rot90(board.BoardCoords(:,:,2), k);
board.BoardCoords = cat(3, newBoardCoords1, newBoardCoords2);
end

%-------------------------------------------------------------------------

function tf = isUpperLeftBlack(this, I)
% check if the upper-left square of the board is black

% create a mask for the upper-left square
upperLeftPolyX = [this.BoardCoords(1, 1, 1), ...
    this.BoardCoords(1, 2, 1), this.BoardCoords(2, 2, 1), ...
    this.BoardCoords(2, 1, 1)];
```

```
upperLeftPolyY = [this.BoardCoords(1, 1, 2), ...
    this.BoardCoords(1, 2, 2), this.BoardCoords(2, 2, 2), ...
    this.BoardCoords(2, 1, 2)];
upperLeftMask = poly2RectMask(upperLeftPolyX, upperLeftPolyY, ...
    size(I, 1), size(I, 2));
% create a mask for the square to the right of it
nextSquarePolyX = [this.BoardCoords(1, 2, 1), ...
    this.BoardCoords(1, 3, 1), this.BoardCoords(2, 3, 1), ...
    this.BoardCoords(2, 2, 1)];
nextSquarePolyY = [this.BoardCoords(1, 2, 2), ...
    this.BoardCoords(1, 3, 2), this.BoardCoords(2, 3, 2), ...
    this.BoardCoords(2, 2, 2)];
nextSquareMask = poly2RectMask(nextSquarePolyX, nextSquarePolyY,...
    size(I, 1), size(I, 2));
% check if the first square is darker than the second
tf = mean(mean(I(upperLeftMask))) < mean(mean(I(nextSquareMask)));
end

%--------------------------------------------------------------------------
function mask = poly2RectMask(X, Y, height, width)
X = sort(X);
Y = sort(Y);
x1 = X(2);
x2 = X(3);
y1 = Y(2);
y2 = Y(3);
mask = false(height, width);
mask(y1:y2, x1:x2) = true;
end
```