This example shows how to calibrate a stereo camera system, and reconstruct a 3-D scene from a stereo pair of images.

Open this Example

On this page…

Overview

Stereo vision is the process of recovering depth from camera images by comparing two or more views of the same scene. The output of this computation is a 3-D point cloud, where each 3-D point corresponds to a pixel in one of the images. Binocular stereo uses only two images, taken with cameras that were separated by a horizontal distance known as the "baseline. Calibrating the stereo camera system allows us to compute the 3-D world points in actual units, such as millimeters, relative to the cameras.

This example shows how to calibrate a camera system consisting of two web-cams. It then uses the parameters estimated by calibration to rectify a stereo pair of images and reconstruct the corresponding 3-D scene. Applications of this process include measurement, aerial photogrammetry, and robot navigation.

The process consists of the following steps

- Calibrate the stereo camera system
- Rectify a pair of stereo images
- Compute disparity
- Reconstruct the 3-D point cloud

Calibrate A Stereo Camera System

Calibration of a stereo camera system involves the estimation of the intrinsic parameters of each camera. It also includes the estimation of the translation and rotation of the second camera relative to the first one. These parameters can be used to rectify a stereo pair of images to make them appear as though the two image planes are parallel. The rectified images are then used to compute the disparity map required to reconstruct the 3-D scene.

To calibrate the stereo system, you must take multiple pairs of images of a calibration pattern from different angles. The entire pattern must be visible in each image. You should store the images in a format that does not use lossy compression, such as PNG. Formats that do use lossy compression, such as JPEG, cause image artifacts, which reduce calibration accuracy.

A typical calibration pattern is an asymmetric checkerboard. One side should contain an even number of squares, and the other side should contain an odd number of squares. This property allows the `detectCheckerboardPoints` function to determine the orientation of the checkerboard unambiguously.

The calibration pattern must be affixed to a flat surface. You should place the pattern at approximately the same distance from the camera as the objects you want to measure. You must measure the size of a square in world units, for example

Please see Find Camera Parameters with the Camera Calibrator for more information on how to take calibration images.

Specify Calibration Images

Create two cell arrays containing file names of calibration images taken with each camera.

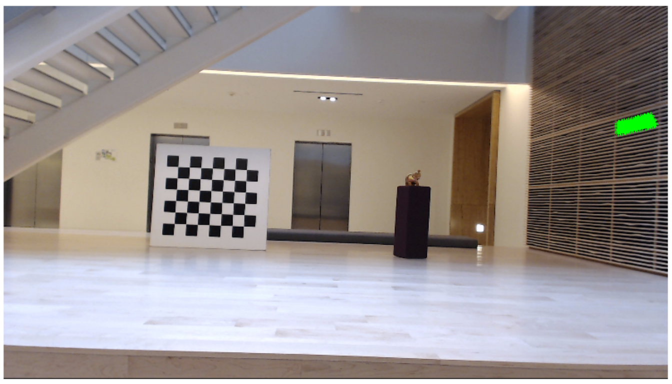
```
numImagePairs = 10;
imageFiles1 = cell(numImagePairs, 1);
imageFiles2 = cell(numImagePairs, 1);
imageDir = fullfile(matlabroot, 'toolbox', 'vision', 'visiondemos', ...
    'calibration', 'stereoWebcams');
for i = 1:numImagePairs
    imageFiles1{i} = fullfile(imageDir, sprintf('left%02d.png', i));
    imageFiles2{i} = fullfile(imageDir, sprintf('right%02d.png', i));
end
```

Try to Detect Checkerboards

The next step is to detect the checkerboard in the images. Unfortunately, in these images the texture of the wall on the right causes the checkerboard detection to fail.

```
% Try to detect the checkerboard
im = imread(imageFiles1{1});
imagePoints = detectCheckerboardPoints(im);

% Display the image with the incorrectly detected checkerboard
figure; imshow(im, 'InitialMagnification', 50);
hold on;
plot(imagePoints(:, 1), imagePoints(:, 2), '*-g');
title('Failed Checkerboard Detection');
```

Failed Checkerboard Detection



What if Checkerboard Detection Fails?

Mask out the wall on the right. You can determine the coordinates of the region containing the problematic texture using `imtool` or `imrect`.

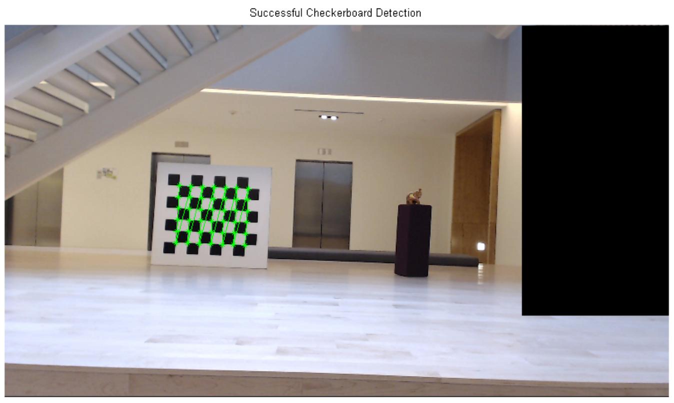
```
images1 = cast([], 'uint8');
images2 = cast([], 'uint8');
for i = 1:numel(imageFiles1)
    im = imread(imageFiles1{i});
    im(3:700, 1247:end, :) = 0;
    images1(:, :, :, i) = im;

    im = imread(imageFiles2{i});
    im(1:700, 1198:end, :) = 0;
```

```
end
```

Detect Checkerboards in the Modified Images

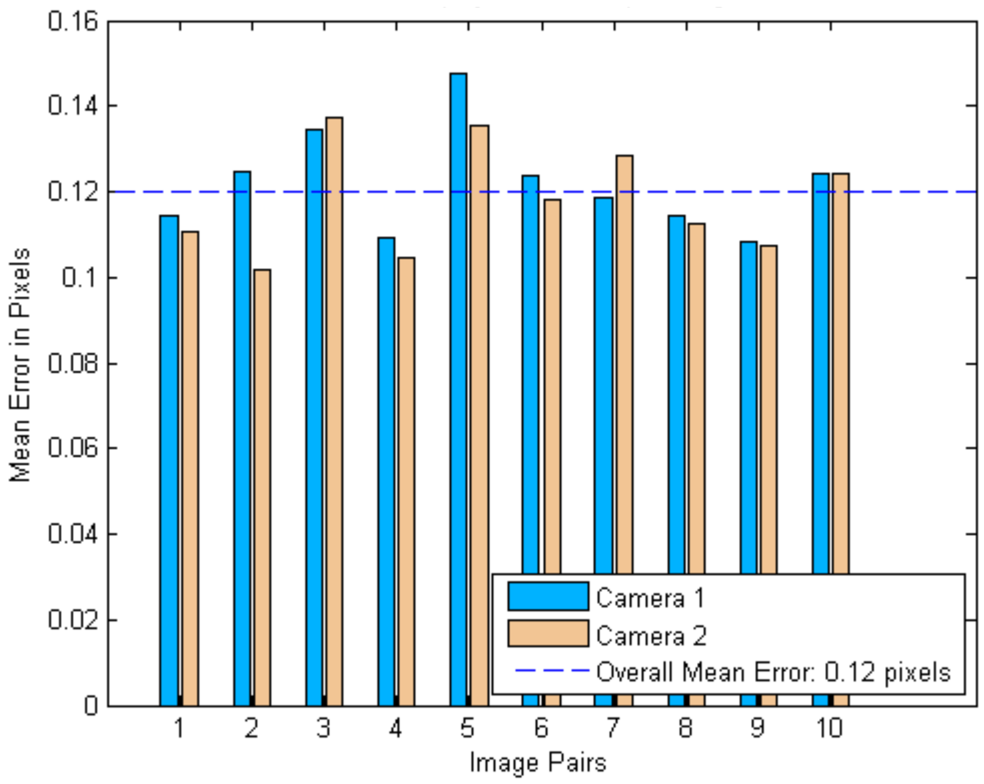
Detect the checkerboard in all stereo pairs of images. You must pass in the images as two sets, one from each camera. The images in both sets must be in the same order to form stereo pairs. The `detectCheckerboardPoints` function returns imagePoints, a 4-D array, such that `imagePoints(:,:,:,1)` are points from camera 1, and `imagePoints(:,:,:,2)` are points from camera 2.

```
[imagePoints, boardSize] = detectCheckerboardPoints(images1, images2);

% Display one masked image with the correctly detected checkerboard
figure; imshow(images1(:,:,:,1), 'InitialMagnification', 50);
hold on;
plot(imagePoints(:, 1, 1, 1), imagePoints(:, 2, 1, 1), '*-g');
title('Successful Checkerboard Detection');
```



Calibrate the Stereo Camera System

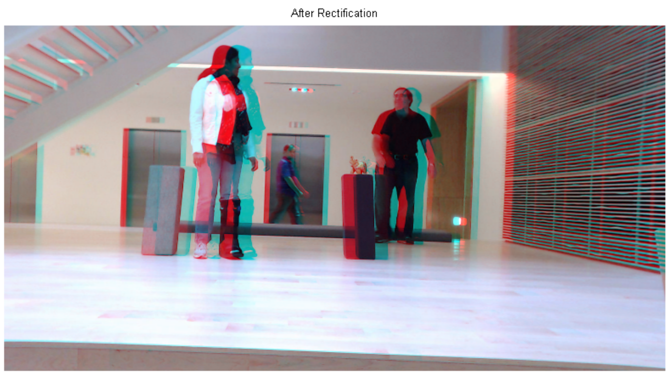
```
% Generate world coordinates of the checkerboard points.
squareSize = 108; % millimeters
worldPoints = generateCheckerboardPoints(boardSize, squareSize);

% Compute the stereo camera parameters.
stereoParams = estimateCameraParameters(imagePoints, worldPoints);

% Evaluate calibration accuracy.
figure; showReprojectionErrors(stereoParams);
```



Rectify a Stereo Pair of Images Taken with the Calibrated Cameras

Rectification is an important pre-processing step for computing disparity, because it reduces the 2-D stereo correspondence problem to a 1-D problem. Rectified images appear as though the two image planes are parallel and row-aligned. This means that for each point in image 1, its corresponding point in image 2 lies along the same row.

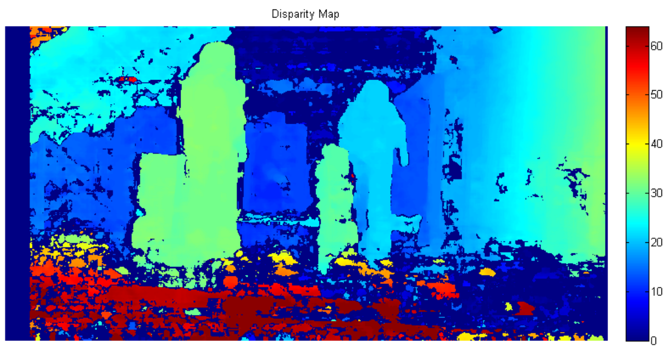
```
% Read in the stereo pair of images.
I1 = imread('sceneReconstructionLeft.jpg');
I2 = imread('sceneReconstructionRight.jpg');

% 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');
```



Compute Disparity for 3-D Reconstruction

The distance in pixels between corresponding points in the rectified images is called disparity. The disparity is used for 3-D reconstruction, because it is proportional to the distance between the cameras and the 3-D world point.

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



Reconstruct the 3-D Scene

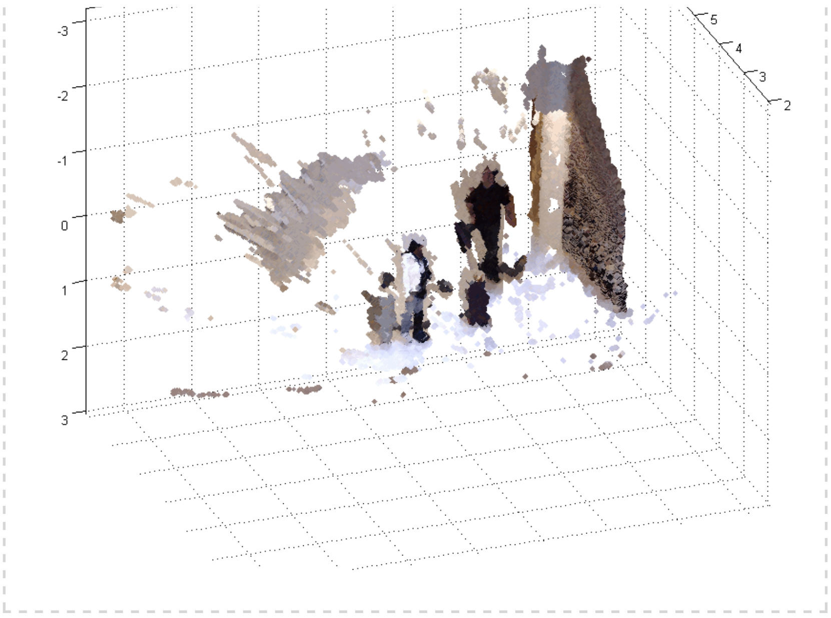
Reconstruct the 3-D world coordinates of points corresponding to each pixel from the disparity map.

```
% Convert from millimeters to meters.
pointCloud = pointCloud / 1000;
```

Visualize the 3-D Scene

Plot the 3-D points by calling the `plot3` function. We have to call the function in a loop for each color, because it can only plot points of a single color at a time. To minimize the number of iteration of the loop, reduce the number of colors in the image by calling `rgb2ind`.

```
% 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 = 3;
    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]);
```



Summary

This example showed how to calibrate a stereo camera system and then use it to reconstruct a 3-D scene.

References

[1] Bouguet, JY. "Camera Calibration Toolbox for Matlab." Computational Vision at the California Institute of Technology. http://www.vision.caltech.edu/bouguetj/calib_doc/

[2] G. Bradski and A. Kaehler, "Learning OpenCV : Computer Vision with the OpenCV Library," O'Reilly, Sebastopol, CA, 2008.