### recovering 3d information with a single camera

Haoran (Simon) Liang

Rose-Hulman Institute of Technology, Multidisciplinary Minor in Imaging, Final Project  
Email: [liangh@rose-hulman.edu](mailto:liangh@rose-hulman.edu)

#### Abstract

Most applications require stereo cameras in order to recover 3D information of a scene. However, using the camera’s intrinsic properties and the real-world coordinates of an object, one can recover the 3D information from an image of the given object taken by a single camera. In this paper, a Python-OpenCV application for recovering 3D information with a single camera has been created. The application takes input from a single webcam and tracks a square in the scene. After detecting the corners of the square, the recovery algorithm is applied to recover the 3D information.

**Keywords—** 3D information, mono-camera, Python, OpenCV

**1. Introduction**

The purpose of this study is to develop a way to recover the position and rotation of the camera with respect to a known-sized object in the real world by using a single camera. In order to recover such information from just a single camera, the intrinsic parameters of the camera must be given along with the size of the tracked object.

Every single point on the image in 2D space can be mapped to a 3D point in the real world coordinates. If we consider that the plane of the image is located on the Z-plane of the coordinate system (Z = 0), it can be written as

                               (        )
 (  x )          [           ]     X
λ(  y ) =  [K |0 ]  R   - RC    ||   Y    || ,
               3   0T3    1     (  Z = 0 )
    1                               1
[1]

where is a scale factor, are the coordinates in the image space, are the coordinates in the real world, is the intrinsic parameter of the camera, and the matrix is the extrinsic parameter of the camera, with being the rotation of the camera and being the position of the camera in 3D space.

If we take out the third dimension (since ), this equation can be rewritten as

  ( x )                       (  X  )
λ ( y )  =     K [ r r  t ]   (  Y  ) ,
               ◟---◝1◜-2--◞
    1      homography transform H   1
[1]

where are the first two columns of the 3x3 matrix , and . Matrix represents the planar homography transform [1] between the real world coordinates and the image coordinates.

In this case, in order to find and , needs to be obtained by camera calibration, the coordinates need to be obtained from the image capture, and the coordinates need to be set up properly to represent the real size of the tracked object.

**2. Process**

Before the program is run, a camera calibration routine needs to be executed to obtain . There are 4 main steps in the program to recover the 3D information. First is object segmentation, next is corner detection, then homography transform, and last is 3D recovery.

A 5 in. by 5 in. red square is chosen as the tracked object for the system because it is easy to set up the coordinates of a square in the real world, treating the center of the square as the origin, and at the same time, some rotation information is preserved. Red color is more saturated and easier to distinguish certain color spaces such as HSV.

**2.1. Camera Calibration**

The camera calibration is done within OpenCV, Open Source Computer Vision Library [2]. It provides a sample script that can recover from a set of multiple images of a known dimension black-and-white checkerboard pattern. By giving different orientations of the checkerboard pattern, the script can locate the projective vanishing points of the image, and recover the focal length and principal center of the camera. Figure 1 shows a sample image of a checkerboard pattern.

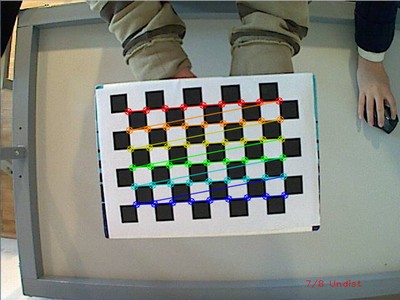


Figure 1. Sample image of the checkerboard pattern being processed by OpenCV camera calibration script [2].

**2.2. Object Segmentation**

The camera takes raw images from RGB space. RGB space does not well preserve hue information because the RGB values can be contaminated by lighting, which makes it harder to set thresholds. Hence, after an image is captured, the program converts the RGB space into HSV (Hue, Saturation, Value) space. Figure 2 shows the RGB space and HSV space of a captured image.

|  |  |
| --- | --- |
|  |  |
| a) | b) |

Figure 2. a) The raw image of the square from the camera b) The resulting image after converting RGB to HSV.

After HSV conversion, a binary mask is created based on the thresholds of the Hue, Saturation and Value of the red color on the square. There could be a lot of noise in the image, but the connected component representing the square should be very large compared to the noise. Thus, a size threshold is applied for the connected components on the mask and the noise is cleaned up. Figure 3 shows the process of noise removal of the binary mask.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| a) | b) | c) |

Figure 3. a) Original image b) Raw binary mask of the image based on the red color of the square c) Resulting mask after noise removal.

**2.3. Corner Detection**

In order to locate the corners of the image, a corner detection algorithm is applied to the image. The Harris corner detector is chosen because there is a fast implementation in OpenCV. Harris corner detector determines which windows produce large variations when moved in any direction [3]. In order to increase the chance of recognizing the corners of the red square, the blue channel is used instead of the grayscale image because the blue channel has very low values for red colors, making the contrast higher compared to the grayscale image.

To directly locate the corners on the red square, the binary mask created in the previous step is used in the corner detection procedure. Only the corner candidates which land on the binary mask are accepted. The mask is blurred so that chance of candidates landing on the mask increases. Figure 4 demonstrates the result of the blurred binary mask and the final result.

|  |  |  |
| --- | --- | --- |
| C:\Users\liangh\Documents\GitHub\MonoCameraRecovery\raw\result.png | C:\Users\liangh\Documents\GitHub\MonoCameraRecovery\raw\square.png | C:\Users\liangh\Documents\GitHub\MonoCameraRecovery\raw\result.png |
| a) | b) | c) |

Figure 4. a) Original image with corners marked (green circles being true candidates) b) Binary mask with blurred edges c) Masking the blurred binary mask onto original image to show the increased chance.

**2.4. Homography Transform**

After recognizing the corners, the center of the red square is set to the origin of the real world (0, 0, 0), and the whole image plane is mapped onto the Z-plane. The real world size of the square is set to 1 inch = 1 pixel so that the conversion process is simplified (for example, if , the camera is 13.3 inches away from the center of the red square). Once everything is set up, the homography transform matrix can be obtained by calculating the null space of matrix

[1]

where are the red square’s corners on the image, are the red square corners in the real world, and .

**2.5. 3D Recovery**

Once is computed, we can do a reverse process to recover :

1. Multiply inverse of with to obtain ;
2. Obtain (cross product);
3. Multiply the inverse of by to obtain .

After obtaining and , VPython [4] is used to reconstruct the scene in 3D space. For the camera, 2 arrows (<0,0,-1> and <0,1,0>) is used to demonstrate the full rotation of the camera. Multiplying with the arrow vectors will rotate the arrows to the desired rotation.

**3. experiments**

Before the experiment starts, one must calibrate the camera by capturing several images of a checkerboard pattern with different orientations. In this experiment, 14 images were taken and the calculated intrinsic parameter of the camera . Figure 5 shows the calibration images.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Figure 5. Some of the checkerboard images used to calibrate camera

The experiment is set up by taping a 5 in. by 5 in. red square on the wall, and a Logitech C110 webcam is connected to a computer running the OpenCV Python script.

Before the script is run, the auto white-balance, auto exposure and auto low-light compensation features must be turned off for the camera in order to better control the lighting consistency on the camera. Once the script starts, one must select the red square region with the mouse on the preview screen to adaptively set up the mask threshold for the script. The script will start tracking the red square and reconstruct the scene on another 3D preview screen. The location of the camera is averaged uniformly over a 5-frame window, while the rotation parameters do not get averaged. By moving, panning, and spinning the camera, the two green arrows will accurately represent the orientation of the camera. Figure 6 shows the screen cast of the experiment, and the video demonstration of the experiment can be found at [5].

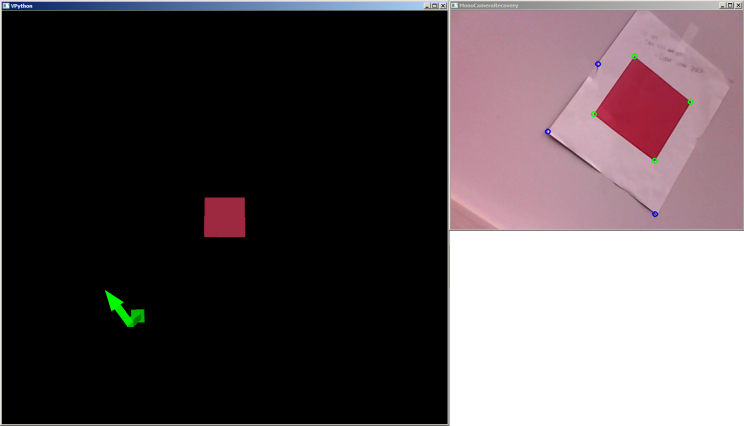


Figure 6. Experiment screen cast.

**4. Discussion**

For the experiment, the camera is first located around 15 inches away from the camera, the magnitude of in the result shows that the script returns very accurate recovery results on the position of the camera. At the same time, the rotation of the camera is also accurately represented.

The overall performance of the script is very promising. On a dual-core CPU laptop (Intel Core 2 Duo T9600 @ 2.8GHz x 2), the whole script is running at about 23-25 fps (frames per second), and the script has not utilized any GPU computing power.

The y and z coordinates of were negated from the actual position shown on the screen, so before the position is set on the arrows, those coordinates were negated back to original. It is the same with the rotation matrix. In order to return the accurate rotation, the Eulerian angles need to be extracted from the rotation matrix by

then to be reconstructed by flipping the sign of , as follows

**5. Future Work**

In the future, the resulting can be studied more to see if there is a way to recover the true information without flipping partial results.

Since every frame is currently being analyzed independently, there could be more extensibility by analyzing the video stream continuously. The corners can be tracked and the full rotational information will be able to be recovered (the current script loses z-axis rotation result at about 45 degrees).

Finally, the script can have potential use with embedded applications such as mobile robotics, because using one camera is relatively cheaper than using two.

**6. References**

[1] Yannick Morvan. “Acquisition, Compression and Rendering of Depth and Texture for Multi-View Video.” June 9, 2009. <http://www.epixea.com/research/multi-view-coding-thesisse9.html>

[2] OpenCV. “Camera Calibration with OpenCV.” April 5, 2013.  
<http://docs.opencv.org/doc/tutorials/calib3d/camera\_calibration/camera\_calibration.html>

[3] Utkarsh Sinha. “Harris Corner Detector.” April 26, 2010.   
<http://www.aishack.in/2010/04/harris-corner-detector/>

[4] Bruce Sherwood. “VPython.” March 2, 2013.   
<http://vpython.org/>

[5] Haoran (Simon) Liang. “Monocamera Recovery – Video Demo.” May 6, 2013.   
<http://www.youtube.com/watch?v=wXrEu-Yz1ko>