Jesse Weisberg - Daniel de Cordoba Gil - Tushar Mathew
CSE 559a
Furukawa
Final Project (Part 1/2)

Real-time Marker-based 3D Augmented Reality

(Part 1/2)

Brief Overview:

For our project, we decided to create a real-time marker-based 3D augmented reality application. To do this, we explored the topics of camera calibration, pose estimation, 3D reconstruction, marker detection, and facial recognition. We used Python and OpenCV to implement a wide selection computer vision algorithms that we have learned over the course of the semester, and OpenGL as a tool to render 3D objects as the final step of our augmented reality application. We were able to successfully implement this real-time system and will explain the steps we took below.

Camera Calibration

The first step of this process was calibrating our camera; that is, calculating intrinsic properties of our camera. To do this, we used a technique outlined by Jean-Yves Bouguet, where we used pictures of a chessboard pattern at various angles to calculate intrinsic parameters from known 3D object to 2D image point correspondences [1]. The main equations, which are the motivation for this procedure (as well as 3D reconstruction), are shown below.

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Eq. 1: Camera Calibration

This equation relates the homogeneous 2D image coordinates with the camera's intrinsic parameters, extrinsic parameters and 3D homogeneous object coordinates (which are known on the chessboard which is composed of identical unit squares). Fx and fy represent the focal lengths of the camera in the x and y directions, respectively, where cx and cy represent the coordinates of the camera's optical center. The 3x4 matrix is an augmented matrix which represents the extrinsic parameters (used for 3D reconstruction), where the 3x3 r terms represent the rotation matrix and the t terms compose a translation vector. This equation above can be mathematically manipulated to the one below, where terms are explained below the equation to take into account radial and tangential distortion of the camera as well.

[1] The technique we used can be found at http://www.vision.caltech.edu/bouguetj/calib_doc/index.html

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

Eq. 2: Camera Calibration with Rotation matrix and reorganized

$$x' = x/z$$
$$y' = y/z$$
$$x'' = x' \frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + 2p_1x'y' + p_2(r^2+2x'^2)$$
$$y'' = y' \frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + p_1(r^2+2y'^2) + 2p_2x'y'$$
$$\text{where} \quad r^2 = x'^2 + y'^2$$
$$u = f_x * x'' + c_x$$
$$v = f_y * y'' + c_y$$

Eq. 2 continued: variables explained

Here are some pictures of us going through the camera calibration process:

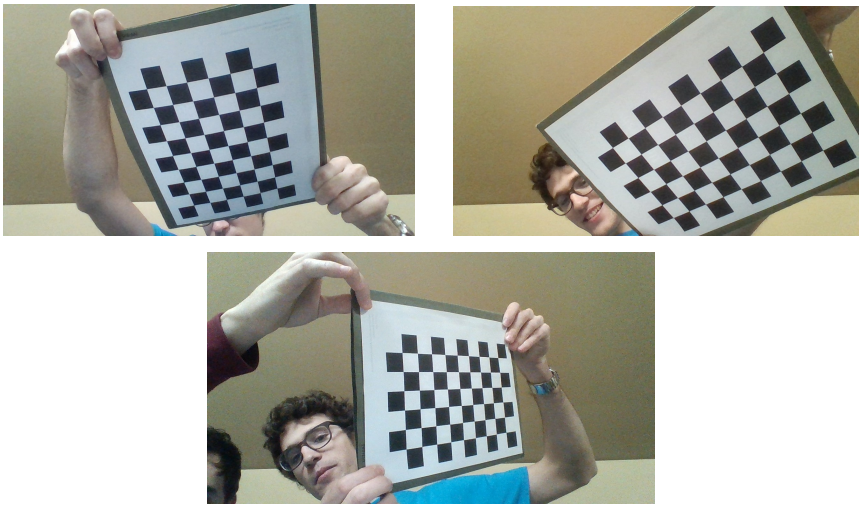

Fig 1: various angles of calibration images

We took 12 pictures in total at various angles to capture the intrinsic parameters and distortion parameters for the cameras we were working with. Here is another photo showing the calibration script running.

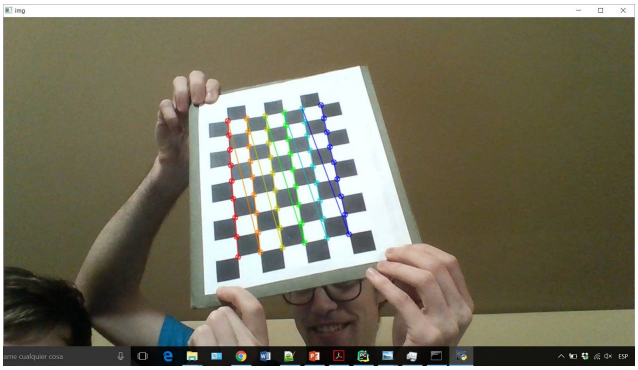

Fig 2: program calculating intrinsic parameters

The intrinsic parameters and distortion coefficients were crucial for accurately rendering 3D objects in the 3D reconstruction phase of the project.

Glyph Detection Algorithm, Pose Estimation

We used 2D markers as platforms on which to reconstruct 3D objects. Our one main specification was that the markers should be rotationally invariant, so that we could determine unique alignment for reconstruction. After researching the topic, we came to the idea of using glyphs as markers, which are essentially square grids of white and black cells. Rotationally invariant formations of these grids can be used as simple marker representations. Here are some examples of glyphs below:

Fig 3: glyph examples that we used in our project

Below, we outline the steps we took to detect a glyph marker and calculate the rotation matrix and translation vector to reconstruct an object with the marker as the base.

Step 1: Edge Detection

Fig 4: screenshot of edge detection program running

The first step to detect the glyph is edge detection. We used a Canny Edge detector, which involves the following steps: noise reduction, calculate intensity gradient of image, non-maximum suppression, and hysteresis thresholding. We used the OpenCV function, cv2.Canny(), which follows the aforementioned steps. The function uses a 5x5 Gaussian filter to reduce noise, or smooth, the image. Then, convolution across the smoothed image is applied with a Sobel kernel in both the horizontal and vertical directions. The edge gradient and direction for each pixel can then be computed as we have done in the feature detection lab. We then used non-maximum suppression, which essentially thins the edge by checking if every pixel is a local maximum in its neighborhood in the direction of the gradient (which is normal to the edge). The final technique we applied was hysteresis thresholding, which ensures the connectivity of edges by checking against maximum and minimum intensity gradient values. Those above the maximum value are kept as edges, those below the minimum value are discarded, and those between the maximum and minimum values are only kept if the preceding and following values are edges (above maximum value).

Step 2: Detect Quadrilaterals

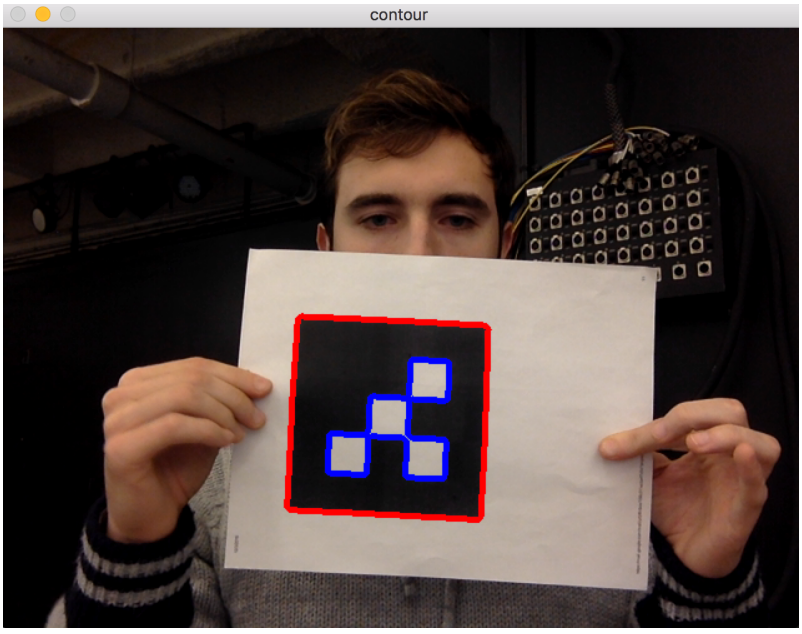

Fig 5: quadrilateral detection

Next, we found contours of polygons using a border-following algorithm based on the paper, “Topological Structural Analysis of Digitized Binary Images by Border Following” (Suzuki, 1985). We then constrained this algorithm to only find quadrilaterals by using the Ramer-Douglas-Peucker algorithm, which recursively cuts line segments until we end up with the least amount of line segments necessary to form a polygon. This essentially gave us the vertices of each polygon, so we could determine a polygon to be a quadrilateral if there were four vertices in that polygon. We then further constrained the algorithm to find only *n*

quadrilaterals, where *n* is around the number of quadrilaterals in the glyph (above, there are 5 shown). Then, we sorted these quadrilaterals by area and started checking for a glyph from the largest polygon.

Check if Quadrilateral is a Glyph

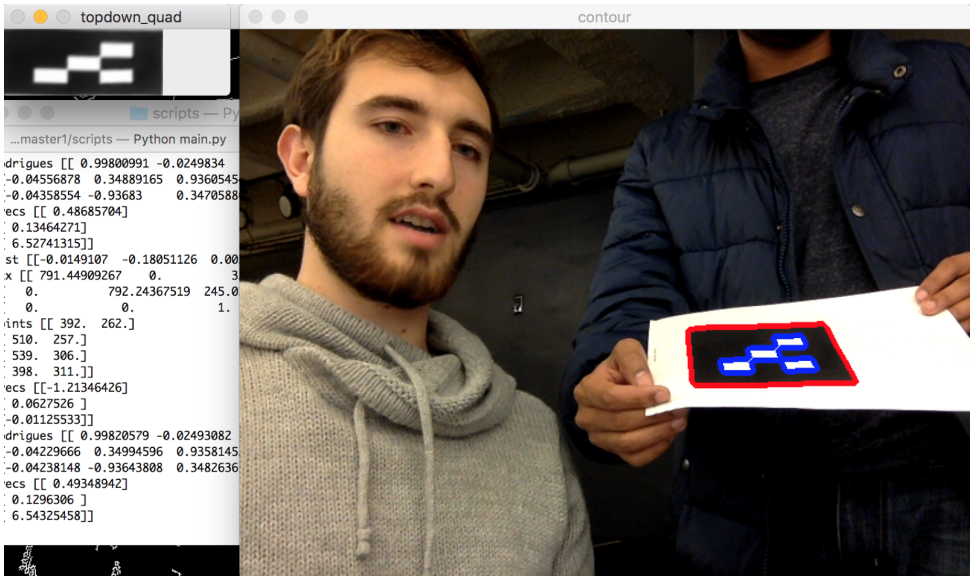

Fig 6: perspective projection of quadrilateral

For each quadrilateral in our list, we checked a pixel near its top left corner. However, the marker was often warped or angled, so to accurately perform this check we had to carry out a 2D perspective projection of the quadrilateral to make it upright and facing forward. We stored the projection of the glyph on a new image, which you can see in the top left image of the window above, called "topdown_quad." We then were able to accurately check the corner pixel intensity of the quadrilateral. If that pixel was black (below our fixed black threshold), then the quadrilateral was deemed to be a glyph, since all glyphs contain a black border. This check eliminated all white quadrilaterals within the glyph as potential candidates and provided us with the final (red-bordered) glyph that we wanted to further analyze.

Match Observed Marker with Database Marker

After detecting the glyph, we determined which glyph it was by converting it to a binary array and matching it which our marker database, which contained 4 binarized rotational variants of the glyph. The glyphs are binarized in the following fashion, where the outer border is ignored, and the inner 3x3 cell grid is represented as a binary array (where black is 1, white is 0). Examples of the four binarized rotational variants for the glyph we have used throughout this paper is shown below.

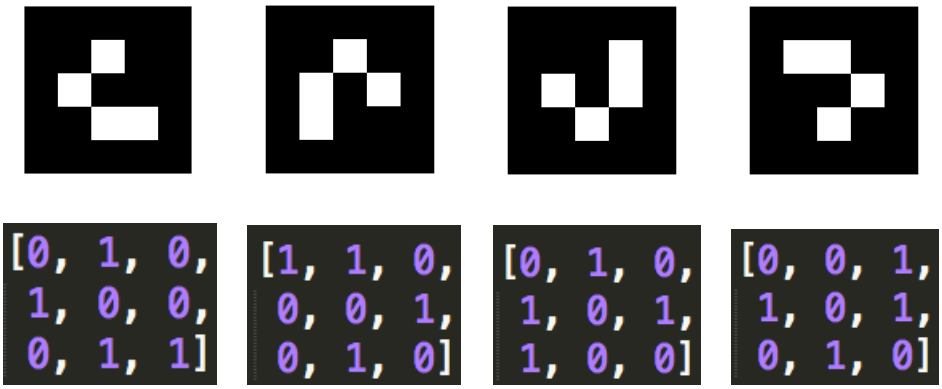

Fig 7: all 4 rotational variants of glyphs, and their respective binary representations

To binarize the observed glyph, we used simple math to estimate the coordinates of the center of each cell in the inner 3x3 grid of the glyph. The formula we used is shown below:

$$(x, y) = \left(\frac{3 \cdot n \cdot width}{widthCells \cdot 2}, \frac{3 \cdot n \cdot height}{heightCells \cdot 2} \right)$$

Eq 3: finding coordinates for marker's cell centers

Here, *(x,y)* represents the center of the desired cell, *n* is the index of the cell (within the 3x3 grid), *widthCells* and *heightCells* are the number of cells of the width and height of the 5x5 grid (they are both 5 in this case), and *width* and *height* are lengths in pixels for the width and height of the entire 5x5 glyph (from the projection we found earlier). We then were able to binarize the observed glyph and could match it against the marker database that we had stored. For this specific glyph, the database was composed of the four arrays above. Once the glyph was detected and matched, we could move onto the final stage of calculating the rotation matrix and translation vector for 3D reconstruction of the image.

Solve for Rotation Matrix and Translation Vector

This final step required solving the perspective-n-point problem, where given a set of object points and their corresponding image points we determined the pose of the image. We used a function (*cv2.solvePnPRansac*) that combines the RANSAC process with the solution of a PnP problem, which essentially uses principle component analysis to find the rotation matrix and translation vector that minimizes the sum of squared distances between the observed projection points and the theoretical projection points of the marker. This allowed us to bring the model coordinate system to the camera coordinate system using essentially the same motivating equations that we used for camera calibration, except this time we solved for the extrinsic parameters in real-time. That is, we solved for the rotation matrix, *R*, and translation vector, *t*, in the equation below.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

(Eq. 2 revisited)

This method, though simple, seemed to work well enough to get a working real-time system.

Future Work and Related Works:

There are things we definitely wanted to do but did not have enough time to implement. One was an improvement to our edge-detection algorithm, which works fairly well but is not completely light invariant due to the hardcoded hysteresis thresholding. We looked into some adaptive thresholding techniques such as Otsu thresholding for future implementation. We would also like to explore the art of marker creation. We understand that glyphs are the simplest form of markers; in fact, there are only 64 possible rotationally invariant glyphs at the size we were working with. Related works we have looked at explored the performance of different types of markers (i.e. Bergamasco, Albarelli, Torsello), and we found that an extremely interesting direction to move forward. We also found another work that used an RGB-D sensor to overlay a real 3D object with a virtual 3D point cloud (Ibe, Yamashita, Asama), which is another path we would like to explore in the future.

References:

1. "4 Point OpenCV GetPerspective Transform Example - PyImageSearch." *PyImageSearch*. N.p., 13 July 2015. Web. 09 Dec. 2016.
2. Bergamasco, Filippo, Andrea Albarelli, and A.Torsello."Image-Space Marker Detection and Recognition Using Projective Invariants." *2011 International Conference on 3D Imaging, Modeling, Processing, Visualization and Transmission* (2011): n.pag. Web.
3. Boguet, Jean Yves. "Camera Calibration Toolbox for Matlab." *Camera Calibration Toolbox for Matlab*. N.p., n.d. Web. 09 Dec. 2016.
4. "Camera Calibration and 3D Reconstruction¶." *Camera Calibration and 3D Reconstruction — OpenCV 2.4.13.1 Documentation*. N.p., n.d. Web. 09 Dec. 2016.
5. Damiand, Guillame, and Patrick Resch. "Split-and-merge Algorithms Defined on Topological Maps for 3D Image Segmentation." *Split-and-merge Algorithms Defined on Topological Maps for 3D Image Segmentation*. N.p., n.d. Web. 09 Dec. 2016.
6. Figures Courtesy Of Siegwart & Nourbakhs. *COS 495 - Lecture 11 Autonomous Robot Navigation* (n.d.): n. pag. Web.
7. "Geometric Image Transformations¶." *Geometric Image Transformations — OpenCV 2.4.13.1 Documentation*. N.p., n.d. Web. 09 Dec. 2016.
8. "Glyphs' Recognition." *AForge.NET::Glyphs' Recognition*. N.p., n.d. Web. 09 Dec. 2016.
9. Heikkila, Janne, and Olli Silven. "Automatic Calibration and Error Correction." *Digital Converters for Image Sensors* (n.d.): n. pag. Web.
10. Ibe, Naoki, Atsushi Yamashita, and Hajime Asama. "Flexible Marker-based Augmented Reality Based on Estimation of Object Pose With RGB-D Sensor." (n.d.): n. pag. Web.