I.J. Image, Graphics and Signal Processing, **2016, 6, 1-12**
Published Online June 2016 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijigsp.2016.06.01

Feature Tracking and Synchronous Scene Generation with a Single Camera

Zheng Chai
Graduate School of Information, Production and Systems, WASEDA University, Japan
Email: abner.sunny@moegi.waseda.jp

Takafumi Matsumaru
Graduate School of Information, Production and Systems, WASEDA University, Japan
Email: matsumaru@waseda.jp

Abstract—This paper shows a method of tracking feature points to update camera pose and generating a synchronous map for AR (Augmented Reality) system. Firstly we select the ORB (Oriented FAST and Rotated BRIEF) [1] detection algorithm to detect the feature points which have depth information to be markers, and we use the LK (Lucas-Kanade) optical flow [2] algorithm to track four of them. Then we compute the rotation and translation of the moving camera by relationship matrix between 2D image coordinate and 3D world coordinate, and then we update the camera pose. Last we generate the map, and we draw some AR objects on it. If the feature points are missing, we can compute the same world coordinate as the one before missing to recover tracking by using new corresponding 2D/3D feature points and camera poses at that time. There are three novelties of this study: an improved ORB detection, which can obtain depth information, a rapid update of camera pose, and tracking recovery. Referring to the PTAM (Parallel Tracking and Mapping) [3], we also divide the process into two parallel sub-processes: Detecting and Tracking (including recovery when necessary) the feature points and updating the camera pose is one thread. Generating the map and drawing some objects is another thread. This parallel method can save time for the AR system and make the process work in real-time.

Index Terms—Tracking, Synchronous map, Camera pose update, Parallel, Tracking recovery

I. Introduction

In this section we will introduce the general implementation steps of the AR system and some similar systems.

A. General Implementation Steps

Virtual reality (VR) and augmented reality (AR) are very hot topics nowadays, and the latter one has a brighter future because it is based on the real scene. There are two kinds of AR system classified by whether there are some markers or not. For marked AR, some signals or pictures will be the markers and detected by the system and then some virtual objects will be drawn on those markers. For unmarked AR, there are no signs defined at the beginning and what we should do is to find some feature points which can be markers, and then some virtual objects can be drawn on those markers.

As mentioned above, the detection of feature points which can be tracked accurately and robustly is very important. In this paper, we use the improved ORB corner detection algorithm to detect feature points (corners). This kind of corner works fast and has three kinds of invariances. More details will be shown in Section Ⅱ-A and Section Ⅲ-A. Then we use LK optical flow to track those corners and obtain the depth information by different distances of corners' movement. After selecting some corners (the way to select non-collinear but coplanar four feature points will be discussed in Section Ⅲ-B), we can use BF (Brute-Force) or FLANN (Fast Library for Approximate Nearest Neighbors) [4] match algorithm or optical flow algorithm to track them. By the position, size, direction and invariances of points, we can easily find out the same points in two frames. In this paper, LK optical flow algorithm is selected and we must consider several conditions which can make the process work well. More details will be shown in Section Ⅲ-B.

In AR system, the sense of reality is very important. For the virtual objects that we draw at some specified location, they must have rotation and translation along with the camera movement. It means they must look like that we see them with our own eyes. To realize this kind of effect, we should compute the camera pose by those tracked points. Firstly we obtain the internal parameters and distortions of the camera by Zhengyou Zhang calibration algorithm [5] [6]. Then we use corresponding 2D/3D points and the intrinsic matrix composed by internal parameters to calculate the rotation and translation matrix, as the camera pose. More details will be shown in Section Ⅱ-B and Section Ⅲ-C.

Once we calculate the camera pose, we can swap the camera pose data, control the camera in OpenGL by the model view matrix, and make the virtual objects more realistic. The map generation and objects rendering will in shown in Section Ⅲ-D. If the tracked feature points are missing, we compute the 3D feature points in the world coordinate that is the same as the previous one before

missing by using the new 2D feature points and the camera pose at that time. Then we can continue to track and render AR objects at the specified location. More details will be shown in Section Ⅲ-E.

B. Similar Systems

In robotic mapping, SLAM is the computational problem of constructing or updating the map of an unknown environment while simultaneously keeping track of an agent's location within it. Here we introduce two kinds of related system here: monoSLAM [7] (Monocular Simultaneous localization and mapping) and PTAM [3] (Parallel Tracking and Mapping), although there are also other tracking systems [8] [9] [10] [11] [12].

The former, monoSLAM selects an invariant visual feature like SIFT (Scale-Invariant Feature Transform) [13] corner to track and use EKF (Extended Kalman Filter) to update the covariance matrix and filter feature points. Then it updates the camera pose and does the graphical rendering.

The latter, PTAM selects FAST (Features from Accelerated Segment Test) [14][15] corner, which runs hundreds of times faster than SIFT corner to track, and uses the five-point algorithm to estimate initial camera pose. It obtains 3D points by triangulation and adds more 3D points by epipolar search. Then PTAM uses 3D points to estimate a significantly plane decided by calculating the minimal reprojection error. At last PTAM draws the AR objects and it uses parallel method to save time.

II. BACKGROUND

In this section we will introduce some basic knowledge about three kinds of feature detection and coordinate transformation.

A. Feature Detection

In the field of computer vision and image processing, the feature detection is a basic but important issue. A feature is defined as an "interesting" part of an image and features are used as a starting point for many computer vision algorithms. Detecting good features is very important for tracking system [16] [17]. There are main three kinds of features: Edges, Corners and Blobs.

Edges are points which compose a boundary (or an edge) between two image regions. In general, an edge can be many kinds of shape and may include the junctions by two or more edges. In practice, edges are usually defined as sets of points in the image which have a strong gradient magnitude. *Corners* are usually defined as point-like features in an image and generally have a regional two-dimensional structure. We usually detect the corners by looking for high levels of curvature in the image gradient. *Blobs* provide a complementary description of image structures in terms of regions and the detector generally detects areas in an image which are too smooth.

The corners can be detected and tracked easily and accurately in AR system, so we introduce several corner detection algorithms here.

Table 1 shows the comparison of corner detection algorithms. The experiments based on these algorithms in Table 1 are coded by ourselves, and they are tested on some images with 640*480 resolution. Harris [18] and FAST algorithms cost less time. But considering the rotation invariance and the scale invariance, the latter three algorithms are better choices. SIFT and SURF (Speeded up Robust Features) [19] algorithms cost too more time than ORB algorithm to make the system work in real-time. So we think in general, ORB algorithm is the best choice because of its faster speed and good effects. More details about ORB detection algorithm will be shown in Section Ⅲ-A.

Table 1. Comparison of Corner Detection with 640*480 Resolution

	Time cost (ms)	Feature points	Brightness invariance	Rotation invariance	Scale invariance
Harris	23	230	yes	yes	no
FAST	2	419	yes	no	no
SIFT	812	691	yes	yes	yes
SURF	160	1446	yes	yes	yes
ORB	19	502	yes	yes	yes

B. Coordinate Transformation

As we know, 3D points in the real world will be projected to 2D points on the image by a camera. To estimate and update the camera pose, we must know the relationship between the world coordinate and the image coordinate. Then we can use corresponding point sets and the intrinsic matrix of the camera to calculate the camera poses like rotation and translation. So the coordinate transformation is introduced firstly

The first coordinate introduced here is the image coordinate, which is shown in Fig. 1. The point O_1 (u_0, v_0) is the principal point that is usually at the image center, and the point (u, v) in image coordinate can be expressed by following equations:

$$u = {}^{x}/_{dx} + u_0 \tag{1}$$

$$v = {}^{y}/_{dy} + v_0 \tag{2}$$

In (1) and (2), u and v are pixel coordinate and dx and dy are units of x-axis and y-axis. Equation (1) and (2) can be expressed by following matrix expression:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{dx} & 0 & u_0 \\ 0 & \frac{1}{dy} & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{3}$$

The second coordinate introduced is the camera coordinate, which is shown in Fig. 2. The distance of O_CO is the focal length that will be presented by f in following two equations deduced by similar triangles rule:

$$x = f * {}^{X_c}/_{Z_c} \tag{4}$$

$$y = f * {}^{Y_c}/_{Z_c} \tag{5}$$

Expression (4) and (5) can be expressed by the following matrix expression (6) which shows the relationship between the image coordinate and the camera coordinate:

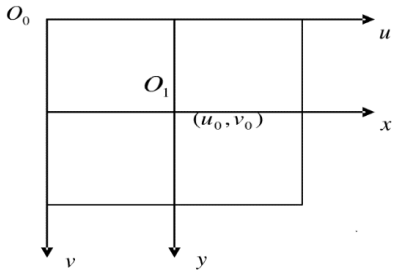


Fig.1. Image Coordinate

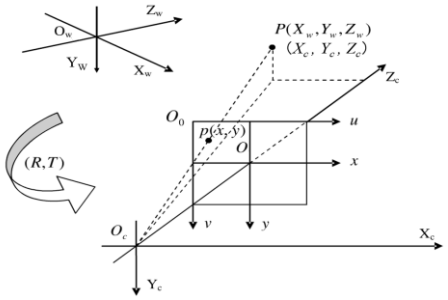


Fig.2. Image/camera/world Coordinate

$$Z_c \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} \tag{6}$$

We also can see from Fig. 2 that the world coordinate can be easily rotated and translated to the image coordinate like the following relationship matrix equation:

$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} = (\boldsymbol{R} \quad \boldsymbol{T}) \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} \tag{7}$$

In (7), $\boldsymbol{R}$ is the rotation matrix (3*3) and $\boldsymbol{T}$ is the translation matrix (3*1). Once we know both the relationship between the image coordinate and the camera coordinate by (3) and (6) and the relationship between the camera coordinate and the world coordinate by (7), we can deduce the relationship between the image coordinate and the world coordinate:

$$Z_c \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{dx} & 0 & u_0 \\ 0 & \frac{1}{dy} & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} (\boldsymbol{R} \quad \boldsymbol{T}) \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{pmatrix} (\boldsymbol{R} \quad \boldsymbol{T}) \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} = \boldsymbol{M_1 M_2} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} \tag{8}$$

In the relationship matrix (8), *fx* and *fy* are the focal lengths expressed in pixel units. $\boldsymbol{M_1}$ is called intrinsic matrix (3*3) including four internal parameters: *fx*, *fy*, u_0 and v_0. $\boldsymbol{M_2}$ is called extrinsic matrix (3*4) included two external parameters: rotation matrix $\boldsymbol{R}$ and translation matrix $\boldsymbol{T}$. This relationship (8) can help us to compute camera pose rapidly and recover tracking. More details will be presented in Section Ⅲ-C and Section Ⅲ-E.

III. Proposed Algorithm

In this section we will detail the proposed algorithm by five parts: corner detection, corner tracking, camera pose update, map generation and tracking recovery.

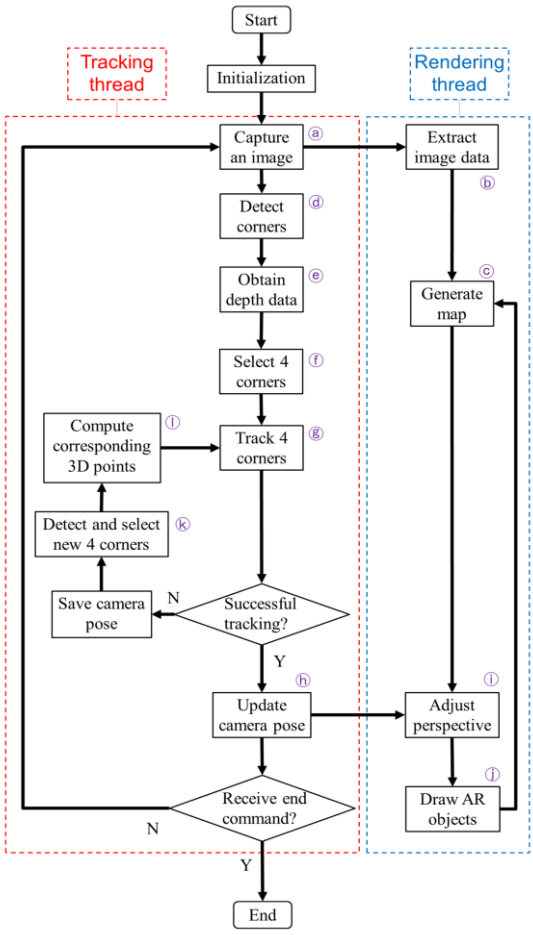


Fig.3. Architecture of Proposed Algorithm

Fig. 3 has shown the architecture of the proposed algorithm. Firstly, images are captured by moving camera (ⓐ in Fig. 3) and the data of images are extracted (ⓑ) to generate a synchronous map (ⓒ) in the whole process. The ORB corners are detected (ⓓ) and tracked by LK optical flow to obtain the depth information (ⓔ). Then

we can select four feature points (ⓕ) which are on the same plane (we select one point to be the center and the system selects other three points which have same depth automatically). Those four points are tracked again (ⓖ), and if the tracking is successful we use these 2D corners and corresponding 3D points initialized at the beginning to compute the camera pose including rotation and translation (ⓗ). The pose data is converted to the model view matrix in OpenGL to adjust the perspective (ⓘ), and then some AR objects are rendered (ⓙ) in a real scene. If the tracking is failed, we save the camera pose at that time and detect the ORB corners again (ⓚ). Select four of them to compute new corresponding 3D points (①), and then we use these pairwise points to compute the new camera pose to continue tracking and draw AR objects again, which is tracking recovery process.

A. Corner Detection (ⓐⓓⓔ in Fig. 3)

We have introduced several corner detection algorithms in Section Ⅱ-A and selected the ORB [1] corner detection algorithm. From the name of ORB (Oriented FAST and Rotated BRIEF) we can see that it is a combination and improvement of FAST detection and BRIEF (Binary Robust Independent Elementary Features) [20] description. It works faster than SIFT and SURF and it is free for business.

Firstly ORB uses FAST detector to detect feature points and then it uses a response function of Harris to select *N* feature points which have maximal response. Then it builds a Gaussian pyramid to solve the scale-invariance problem. For the rotation-invariance problem, we have following calculation to compute direction of feature points [21]:

$$\theta = \arctan(m_{01}, m_{10}) \tag{9}$$

The *m* is defined like following:

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y) \tag{10}$$

This $I(x, y)$ is the scale value of point (x, y).

After obtaining the FAST feature points, we need to describe them by BRIEF descriptor. BRIEF generate a binary code string whose length is 256 from 31*31 pieces around feature points. For each 31*31 piece, we use an average scale of a 5*5 block instead of one pixel's scale to remove interference of noise. There are totally (31-5+1)*(31-5+1) sub-blocks, and we use some methods to decide the way to select 256 pairs of sub-blocks to generate a binary code string. Here we introduce ORB detection algorithm simply because we just use it with a little change. More details are in the paper [1]. In our system, we remove the close feature points by a loop algorithm which make the distance between two feature points at least 10 pixels. It is useful to select other three feature points automatically after we select one center, which will be introduced in Section Ⅲ-B.

After the detection of feature points, we obtain many ORB corners and we have to select four points which are on the same plane to compute the camera pose. Here we use LK optical flow algorithm to track those corners and compare the motion of corners after the camera moved a little distance. (The LK optical flow is introduced in the next Section Ⅲ-B.) Firstly, we select one frame as the initial frame including many corners. Then we move the camera a little in a straight line at a suitable speed while keeping the frame after the movement. We count the distance for each feature points from the initial frame to the end frame to find the minimal and maximal distance, and then we save the depth data by the following method:

$$depth = \left(d - min/_{max - min}\right) * \mathrm{e} \tag{11}$$

In (11), *d* is the distance of one point from the initial frame to the end frame. And *e* is expanding multiples for drawing the feature points on the image with a suitable size. If the depth is less than 1, we set it to 1. Fig. 4 shows the corner detection in some practical scenes.

The corner detection process costs 24 milliseconds including ORB corner detection (19 milliseconds), LK optical flow tracking (4 milliseconds) and depth information calculation (1 milliseconds). Although it is a little time-consuming, we need not worry about that, because the corner detection process only occur at the beginning and the tracking recovery process. We can see it from Fig. 3 (the architecture of proposed the algorithm).

B. Corner Tracking (ⓕⓖ in Fig. 3)

In the previous part we use LK optical flow [2] [22] to track all corners which are detected. The approximate distance between corners and the camera is calculated by the corners' movement. LK optical flow is also used to track four corners which are selected by manual operation. In this part, we introduce this tracking algorithm.

LK optical flow algorithm is an improvement for optical flow algorithm [23] [24]. At the beginning, we build the Gaussian pyramids for two frames (image *I* and image *J*) which need to be tracked and the initialize the guess estimation of each pyramid:

$$\boldsymbol{g}^{Lm} = [g_x^{Lm}\ g_x^{Lm}]^T = [0\ 0]^T \tag{12}$$

Then we define the location of point $\boldsymbol{u}$ on image *I*:

$$\boldsymbol{u}^L = [p_x\ p_x]^T = \boldsymbol{u}/2^L \tag{13}$$

We calculate the partial derivative $I(x, y)$ for coordinate *x* and *y* respectively:

$$I_x(x,y) = \frac{I^L(x+1,y) - I^L(x-1,y)}{2} \tag{14}$$

$$I_y(x,y) = \frac{I^L(x,y+1) - I^L(x,y-1)}{2} \tag{15}$$

And we build spatial gradient matrix *G*:

$$G = \sum_x^\alpha \sum_y^\beta \begin{bmatrix} I_x^2(x,y) & I_x(x,y)I_y(x,y) \\ I_x(x,y)I_y(x,y) & I_y^2(x,y) \end{bmatrix} \tag{16}$$

Then we initialize the iterative for the first level and calculate the image difference δI and the image mismatch vector $\bar{b}$ to obtain the optical flow and guess for each level:

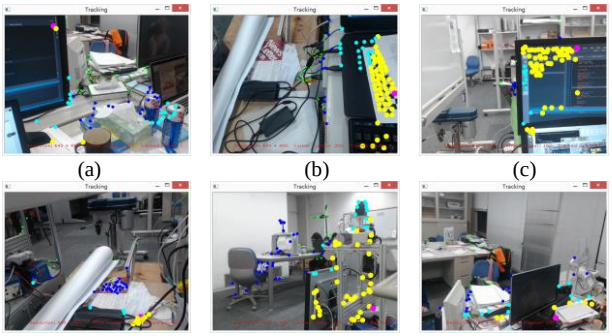
$$\delta I_k(x,y) = I^L(x,y) - J^L(x + g_x^L + v_x^{k-1}, y + g_y^L + v_y^{k-1}) \quad (17)$$

$$\bar{b}_k = \sum_x^{\alpha}\sum_y^{\beta}[\begin{matrix}\delta I_k(x,y) I_x(x,y) \\ \delta I_k(x,y) I_y(x,y)\end{matrix}] \quad (18)$$

Last we add the $\boldsymbol{d_0}$ and $\boldsymbol{g_0}$ to be the final optical flow vector $\boldsymbol{d}$. So the location of corresponding point $\boldsymbol{v}$ on image J is like following:

$$\boldsymbol{v} = \boldsymbol{u} + \boldsymbol{d} \quad (19)$$

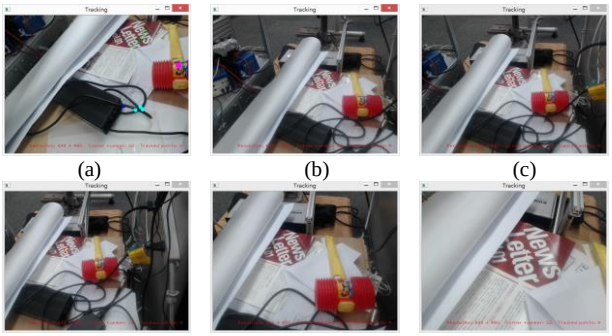
Here we introduce LK optical flow tracking algorithm simply because we just use it without any change. More details are in the paper [2]. After a certain number of tests, we obtain the time cost of tracking process using LK optical flow algorithm, which costs 4 milliseconds. Fig. 5 shows some results of tests.



(a) (b) (c)

(d) (e) (f)

Fig.4. ORB Corners with Depth Information (depth is sorted by color: pink>yellow>cyan>blue>green>red) (a) scene 1, (b) scene 2, (c) scene 3, (d) scene 4, (e) scene 5, and (f) scene 6.



(a) (b) (c)

(d) (e) (f)

Fig.5. Four Tracking Feature Points (a) the frame obtained ORB corners with depth information, (b) one center detected by ourselves and other three corners detected by the system automatically, (c) moving the camera to the right a little, (d) moving right again, (e) approaching to corners a little, and (f) approaching closer again.

Next we talk about how to obtain these four feature points. Firstly, we can select one point from those points like Fig. 4 after we get the depth information. Then the system sort the depth which is computed by (11) with ascending order for all points. We find the point which we select before in the sequence, and we extract another three points around that point. We make sure that these four points are different by comparing their x-coordinate and y-coordinate. In the next part about the camera pose calculation process, any three points in tracked four points must be non-collinear. So here we must make sure it. There are three kinds of combination: *(p1, p2, p3)*, *(p1, p2, p4)*, *(p2, p3, p4)* for four points and we inspect each combination. As we know, a line will be expressed by *p1(x1, y1)* and *p2(x2, y2)* like following:

$$(x_2 - x_1) * y = (y_2 - y_1) * x + y_1 * x_2 - y_2 * x_1 \quad (20)$$

If the point *p3(x3, y3)* is on the line (20), it must satisfy the following equation:

$$(x_2 - x_1) * y_3 = (y_2 - y_1) * x_3 + y_1 * x_2 - y_2 * x_1 \quad (21)$$

By using (21) we can judge whether three points are collinear or not and we must make sure that above three kinds of combination are all non-collinear, in other words, any three points in tracked four points is non-collinear.

This part selects four non-collinear feature points which have similar or the same depth, and we uses LK optical flow to track them. Next part we talk about how to find the corresponding 3D points and calculate the rotation and translation of moving camera.

C. Camera Pose Update (ⓗ *in Fig. 3*)

In this section we talk about the update of camera pose by the intrinsic matrix of the camera, the 2D points which are tracked and the corresponding 3D points. Firstly, we talk about the intrinsic matrix and the corresponding 3D points. Then we introduce how to compute the camera pose.

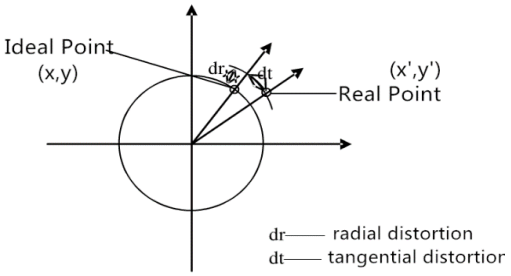


Fig.6. Distortion of Lenses

In the Section Ⅱ-B, we introduced the relationship between the image coordinate and the world coordinate. $\boldsymbol{M_1}$ in (8) is called intrinsic matrix including four internal parameters: f_x, f_y, u_0 and v_0. These four parameters are changed by the resolution of the image captured by the camera. In this paper, we calibrate the camera by using Zhengyou Zhang calibration algorithm [5] to obtain the internal parameters and distortions of lenses. Real lenses usually have distortion, mostly radial distortion and slight tangential distortion like *dr* and *dt* in Fig. 6. Real point location *x'* and *y'* can be expressed like following:

$$x' = x\frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + 2p_1x'y' + p_2(r^2+2x'^2) \quad (22)$$

$$y' = x\frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + 2p_2x'y' + p_1(r^2+2y'^2) \quad (23)$$

The distortion parameters *p1, p2, k1 to k6* can be also calibrated by Zhengyou Zhang calibration algorithm.

Next, we introduce how to obtain the corresponding 3D points from the 2D points which are tracked by LK optical flow algorithm. The four 2D points which we select in the previous part is as in Fig. 7 and we make the first selected point as the center of the world coordinate.

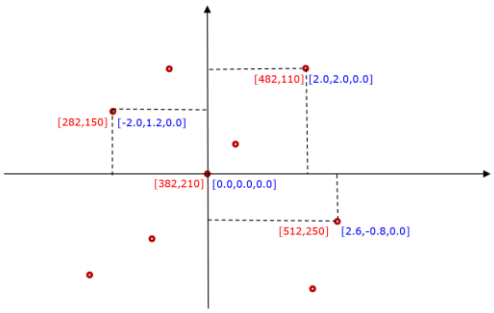


Fig.7. Corresponding 3D Points

According to the difference of x-coordinate and y-coordinate between the rest of the three 2D points and the center, we calculate x-coordinate and y-coordinate of the rest of the three 3D points. As we know, these four 3D points are on the same plane because they have similar or the same depth so that we set the z-coordinate to zero. We just compute the coordinate of the 3D points only once at the beginning of tracking.

Last we talk about how to compute the camera pose by the 2D points which are tracked and the corresponding 3D points. We give a transformation of (8) like following:

$$Z_c\begin{pmatrix}x\\y\\1\end{pmatrix} = \begin{pmatrix}f_x & 0 & u_0\\0 & f_y & v_0\\0 & 0 & 1\end{pmatrix}\begin{pmatrix}R_{11} & R_{12} & R_{13} & T_1\\R_{21} & R_{22} & R_{23} & T_2\\R_{31} & R_{32} & R_{33} & T_3\end{pmatrix}\begin{pmatrix}X_w\\Y_w\\Z_w\\1\end{pmatrix} \quad (24)$$

We use x and y instead of u and v to express the pixel coordinate. $\boldsymbol{R}$ is a 3*3 rotation matrix and $\boldsymbol{T}$ is a 3*1 translation matrix. They are expressed by 9 elements R_{11} *to* R_{33} and 3 elements T_1 *to* T_3. After expanding (24), we get following three equations:

$$Z_c * x = X_w * (f_x * R_{11} + u_0 * R_{31}) + Y_w * (f_x * R_{12} + u_0 * R_{32}) + Z_w * (f_x * R_{13} + u_0 * R_{33}) + f_x * T_1 + u_0 * T_3 \quad (25)$$

$$Z_c * y = X_w * (f_y * R_{21} + v_0 * R_{31}) + Y_w * (f_y * R_{22} + v_0 * R_{32}) + Z_w * (y * R_{23} + v_0 * R_{33}) + f_y * T_2 + v_0 * T_3 \quad (26)$$

$$Z_c = X_w * R_{31} + Y_w * R_{32} + Z_w * R_{33} + T_3 \quad (27)$$

We set Z_w to zero and bring (27) into (25) and (26), then we get the following equations after extracting X_w and Y_w:

$$\begin{cases} X_w * (f_x * R_{11} + u_0 * R_{31} - x * R_{31}) \\ +Y_w * (f_x * R_{12} + u_0 * R_{32} - x * R_{32}) \\ \quad = T_3 * x - f_x * T_1 - u_0 * T_3 \\ X_w * (f_y * R_{21} + v_0 * R_{31} - y * R_{31}) \\ +Y_w * (f_y * R_{22} + v_0 * R_{32} - y * R_{32}) \\ \quad = T_3 * y - f_y * T_2 - v_0 * T_3 \end{cases} \quad (28)$$

In (28), f_x, f_y, u_0 and v_0 are internal parameters and (X_w, Y_w, Z_w) and (x, y) are 3D and 2D points' coordinates. There are only 9 unknown variables: R_{11}, R_{12}, R_{13}, R_{21}, R_{22}, R_{23}, T_1, T_2 and T_3. For rotation matrix $\boldsymbol{R}$, there is a rule that value of sum of squares in each row is 1:

$$\sum_{i=1}^{3} R_{1i}^2 = 1 \quad (29)$$

So R_{13} and R_{23} can be expressed by R_{11}, R_{12}, R_{21} and R_{22}. After that, we have only 7 unknown variables and they can be solved by at least 7 equations. From (28) we find that one group of corresponding 3D (X_w, Y_w, Z_w) and 2D (x, y) points provide 2 equations so that 4 groups can provide 8 equations to solve those 7 unknown variables. Then we use (29) to calculate R_{13} and R_{23}. Because Z_w is zero, R_{31}, R_{32} and R_{33} cannot be solved by (29). But for rotation matrix $\boldsymbol{R}$, there is another rule that value of sum of squares in each column is also 1:

$$\sum_{i=1}^{3} R_{i1}^2 = 1 \quad (30)$$

So R_{31}, R_{32} and R_{33} can be expressed by R_{11}, R_{21}, R_{12}, R_{22} R_{13} and R_{23}.

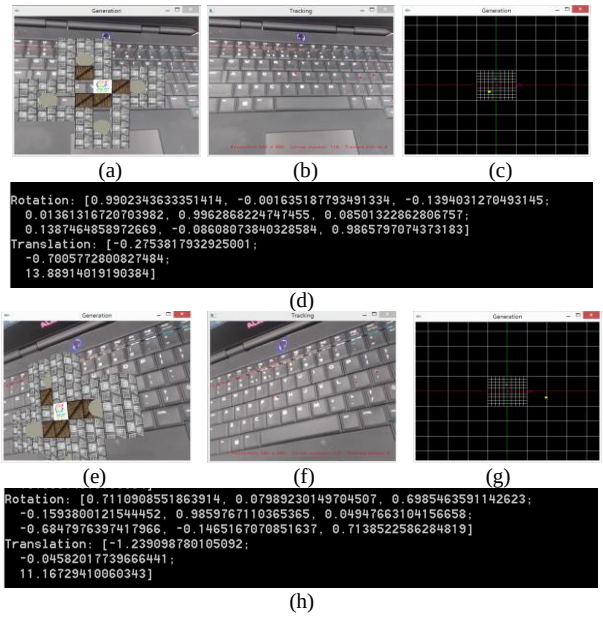


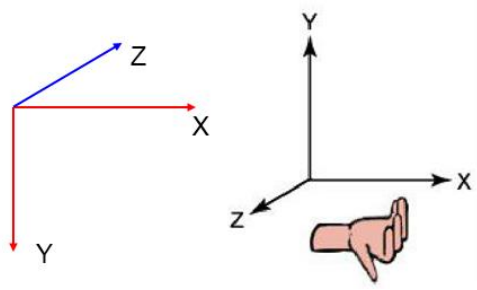
Fig.8. Camera Pose Update cene 1: (a) AR object rendering, (b) four tracked corners, (c) world coordinate (red points are corners and yellow point is camera), and (d) camera pose under scene 1. cene 2: (e) AR object rendering, (f) four tracked corners, (g) world coordinate (red points are corners and yellow point is camera), and (h) camera pose under scene 2

The above shows the calculation process about camera pose including 9 parameters of rotation and 3 parameters of translation, which costs only 1 ms. The four 3D points will not be changed until they are missing and the four 2D points will always be changed in the update process because of the camera movement. Fig. 8 shows the camera pose in two different scenes.

D. Map Generation (ⓑⒸⓘⓙ*in Fig. 3*)

In the previous part, we have calculated the camera pose of moving camera by a group of the 2D feature points in the image coordinate and the corresponding 3D points in the world coordinate. In this part, we load the textures of images captured by camera firstly and then adjust the model view matrix in every frame by the vector of rotation and the vector of translation. Finally some AR objects are rendered on this map.

In this paper, we use OpenGL to render the map and AR objects. We must firstly transform the coordinate space from OpenCV to OpenGL. As we know, it's right-handed Cartesian coordinate in OpenGL. But in OpenCV, the X-axis turns towards right, Y-axis turns towards down, and Z-axis turns towards the inside of screen like Fig. 9. So we must rotate it by 180 degrees around X-axis. Then we make a scaling in OpenGL according to the size of the image captured by the camera. We extract the data to make it into a texture and load the texture. Last we adjust the translation transformation and render the texture at a specified location.



(a) (b)

Fig.9. Different Coordinates in OpenCV and OpenGL a) OpenCV, (b) OpenGL(Right-handed Cartesian Coordinates)

For rendering of AR objects, we can draw it by OpenGL's functions or just load some 3D models. Here we develop a box pushing game Sokoban by OpenGL's functions and we render it around the center of the world coordinate. In the rendering process, we need to adjust the perspective by loading the model view matrix which can control the OpenGL's camera. Notice that the matrix elements are stored in a column-major order in OpenGL like following matrix:

$$\begin{bmatrix} R_{11} & R_{21} & R_{31} & 0 \\ R_{12} & R_{22} & R_{32} & 0 \\ R_{13} & R_{23} & R_{33} & 0 \\ T_1 & T_2 & T_3 & 1 \end{bmatrix} \quad (31)$$

After adjustment in OpenGL about the camera pose, we can render the AR objects with a specific perspective which is the same as the camera in real world. Fig. 10 shows the result of the map generation and the AR objects rendering in three different scenes.

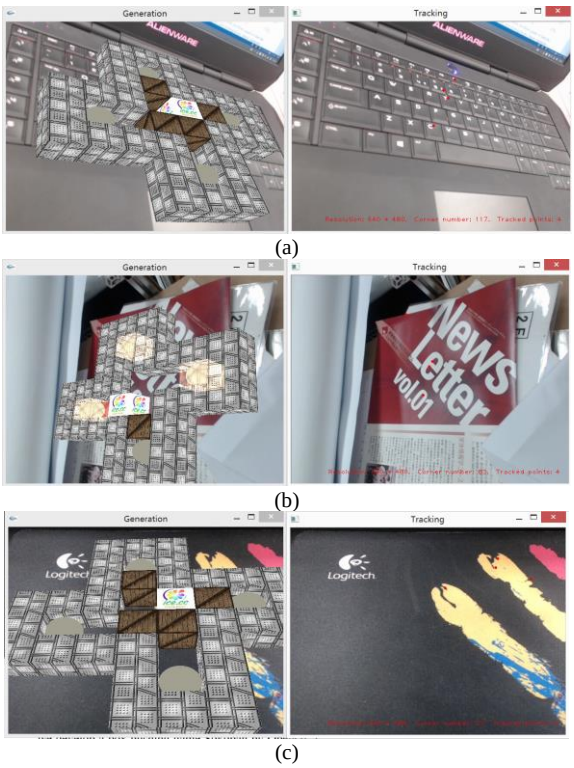


Fig.10. Map Generating and AR Object Rendering (right – original image with four corners. left – same scene with AR objects rendered) (a) keyboard of laptop, (b) newspaper in sundries, and (c) mouse pad.

This process has two sub-processes: map generating and objects rendering. Extracting texture data from the image captured by the camera and rendering the map are the map generation sub-process which cost 1.5 milliseconds. Adjusting the perspective and rendering the AR objects are the objects rendering sub-process which cost 0.5 milliseconds. So the whole process costs 2 milliseconds. Considering about loading 3D model files costs more time (here we just draw the AR objects by OprnGL and sometimes the 3D model will improve the effect), so we divide the system into two parallel threads: tracking and rendering. It is easy and convenient for us to do more things like loading some 3D model files in rendering thread, and this way could save time for the whole system. Each thread will wait for another thread and start a new loop together, which is as shown in Fig. 3 (the architecture of the proposed algorithm).

E. Tracking Recovery (ⓚⒾ *in Fig. 3*)

In the past four parts, we have introduced the main process of AR tracking system in detail. Next we talk about the case that the tracked feature points are missing. Because the camera is always moved, those four feature points will be out of image frequently. One idea about tracking recovery is the corner matching, which means we can detect new corners on the new images and use some matching algorithm to find out those four corners before missing. But this method need accurate and fast

matching algorithm. BF algorithm and FLANN [4] algorithm have been tried but the results are not good in terms of speed and accuracy. We must give up this way and try another idea which uses the relationship between the image coordinate and the world coordinate like (24). It means we can detect new corners on new images, select four feature points, and compute their corresponding 3D points. Then we can use the new corresponding 2D and 3D points to compute the camera pose again, and the new world coordinate is the same as the old world coordinate before missing because we use the camera pose from the old world coordinate to compute the 3D points in the new world coordinate by new 2D feature points.

From (25), (26) and (27), we can obtain following equations after sorting:

$$\begin{cases} X_w * (f_x * R_{11} + u_0 * R_{31} - x * R_{31}) \\ +Y_w * (f_x * R_{12} + u_0 * R_{32} - x * R_{32}) \\ +Z_w * (f_x * R_{13} + u_0 * R_{33} - x * R_{33}) \\ \quad = T_3 * x - f_x * T_1 - u_0 * T_3 \\ X_w * (f_y * R_{21} + v_0 * R_{31} - y * R_{31}) \\ +Y_w * (f_y * R_{22} + v_0 * R_{32} - y * R_{32}) \\ +Z_w * (f_y * R_{23} + v_0 * R_{33} - y * R_{33}) \\ \quad = T_3 * y - f_y * T_2 - v_0 * T_3 \end{cases} \tag{32}$$

Fig.11. Tracking Recovery (right – original image with corners, left – same scene with AR objects rendered) (a) corners are going to miss (blue one is the center), (b) having missed and detecting feature points again, and (c) selecting four feature points and recovering tracking (the blue center appears again), (d) the recovered corners are going to miss, (e) having missed and detecting feature points again, and (f) selecting four corners and recover tracking again (the blue center appears at the same location).

As we can see from (32), computing the 2D point coordinate is very easy if we know the 3D point coordinate, but the reverse is not true. Fortunately, those 3D point are on the same plane whose Z_w is zero. So we can easily calculate the 3D point coordinate if we know the 2D point coordinate by following equations which is the matrix transformation of (32) when Z_w is zero:

$$\begin{bmatrix} F_x * R_{11} + u_0 * R_{31} - x * R_{31} & f_x * R_{12} + u_0 * R_{32} - x * R_{32} \\ f_x * R_{12} + u_0 * R_{32} - x * R_{32} & f_y * R_{22} + v_0 * R_{32} - y * R_{32} \end{bmatrix} * \begin{bmatrix} X_w \\ Y_w \end{bmatrix} = \begin{bmatrix} f_y * R_{22} + v_0 * R_{32} - y * R_{32} \\ T_3 * y - f_y * T_2 - v_0 * T_3 \end{bmatrix} \tag{33}$$

Solving the 3D point coordinate problem equals to find the unique solution of non-homogeneous linear equations. Like the following equation:

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix} * \begin{bmatrix} X_w \\ Y_w \end{bmatrix} = \begin{bmatrix} e \\ f \end{bmatrix} \tag{34}$$

We define D is determinant of coefficient like following:

$$D = \begin{vmatrix} a & c \\ b & d \end{vmatrix} = ad - bc \tag{35}$$

We also define d_1 and d_2 like following:

$$d_1 = \begin{vmatrix} a & e \\ b & f \end{vmatrix} = af - be \tag{36}$$

$$d_2 = \begin{vmatrix} e & c \\ f & d \end{vmatrix} = ed - fc \tag{37}$$

According to Cramer's rule [25], if D is not zero and not both e and f are zero, non-homogeneous linear equations like (34) has the unique solution like:

$$\begin{cases} X_w = {}^{d_1}/_{D} \\ Y_w = {}^{d_2}/_{D} \end{cases} \tag{38}$$

So when the tracked feature points are missing, we lock the frame and detect the new ORB corners and select four corners which on the same plane by ourselves. We calculate the camera pose from the previous frame and use (33) to (37) to calculate the 3D point coordinates (X_w, Y_w, 0) by the camera pose and the internal parameters of camera for each of those four 2D points.

Once we obtain the corresponding 3D feature points, we can track 2D corners again and use (25) to (30) in Section Ⅲ-C to calculate the new camera pose. This way can find the center of the world coordinate before missing and render the AR objects again, which is the tracking recovery. Fig. 11 shows the experimental results of two consecutive tracking recovery.

The results shows the tracking recovery works well in limited number of times. We find there is a little error after repeated calculations because we use the camera pose at the previous frame before missing and 2D points at the next frame. More about the data analisis is shown in Section Ⅳ-C. The tracking recovery process includes ORB corners detection which costs 19 milliseconds and 3D points calculation which costs 0.5 milliseconds. So the whole process costs 19.5 milliseconds but it only occurs when feature points are missing.

IV. Simulation Experiment And Result Analysis

We have given simulation experiments and analyzed the result for each part of the previous section. Here we give a demonstration of the whole process of our system.

Table 2. Depth Distribution of Corners

Moving distance (depth: pixel)	Number (total: 194)
14.835 (minimal) – 20	29
20 – 25	20
25 – 30	21
30 – 35	94
35 – 38.778 (maximal)	30

Table 3. Processed Depth Distribution of Corners

Depth distribution	size / color	Number (total: 194)
1 (0 – 19.624)	1 / red	28
1 – 2 (19.624 – 24.412)	3 / green	18
2 – 3 (24.412 – 29.201)	5 / blue	21
3 – 4 (29.201 – 33.989)	7 / cyan	93
4 – 5 (33.989 – 38.778)	9 / yellow	33
5 (38.778)	10 / pink	1

A. Corner detection and selection

Firstly we detect ORB corners like Fig. 12, and then we get the different number of depth corners by the moving distance like Table 2. Then we set the expanding multiples to 5 and use (11) to obtain processed depth data, which are shown in Table 3. We also set different sizes and colors for points to show the depth information of points clearly, which are shown in Fig .12 and Table 3. Last we select four corners whose depth data are shown in Fig. 13_(c) and render the initial AR object around the center of the world cooedinate which is decided by those four corners.

(a) (b) (c)

Fig.12. Depth ORB Corners Detection (a) ORB corners detection, (b) LK optical flow tracking, and (c) obtain the depth information after calculation.

(a) (b) (c)

Fig.13. Corner Selection and Object Rendering (a) select four corners to track (blue one is the center), (b) render AR objects around the blue center, and (c) the depth data of four corners.

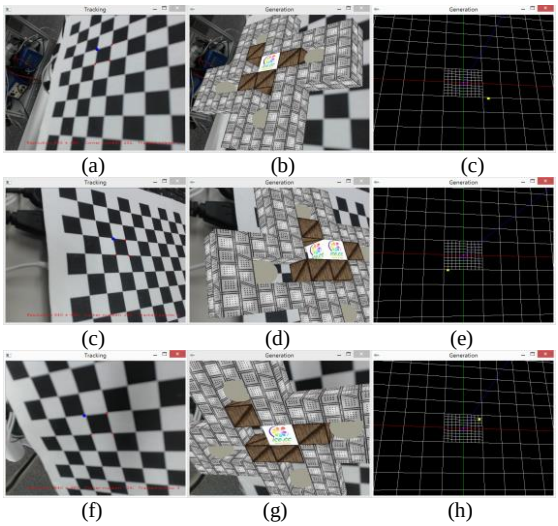
Table 4. Camera Pose of Specific Perspective

Perspective	Camera Pose			
	Rotation			Translation
1 in Fig. 14	0.7920751	-0.149264	0.5918927	0.4616123
	-0.193632	0.8581277	0.4755242	-1.709866
	-0.578898	-0.491260	0.6507992	9.7796952
2 in Fig. 14	0.8432300	0.1511578	-0.515862	0.8800349
	0.0256208	0.9472583	0.3194450	-1.627832
	0.5369419	-0.282582	0.7948839	10.19600
3 in Fig. 14	0.8659636	0.1697859	0.4704037	-1.024721
	-0.000671	0.9410001	-0.338405	-1.306448
	-0.500106	0.2927310	0.8149858	7.5661901

This experiment demonstrates that we can obtain ORB corners with approximate depth information successfully. The selection about four non-collinear but coplanar feature points also works well, and it is necessary for later experiments.

B. Corner tracking & Camera pose updating

After obtaining feature points, we can move the camere and use LK optical flow to track them. Then we use (24) to (30) to compute the camera pose consecutively. We control the OpenGL's camera to adjust the perspective by swapping the camera pose data.

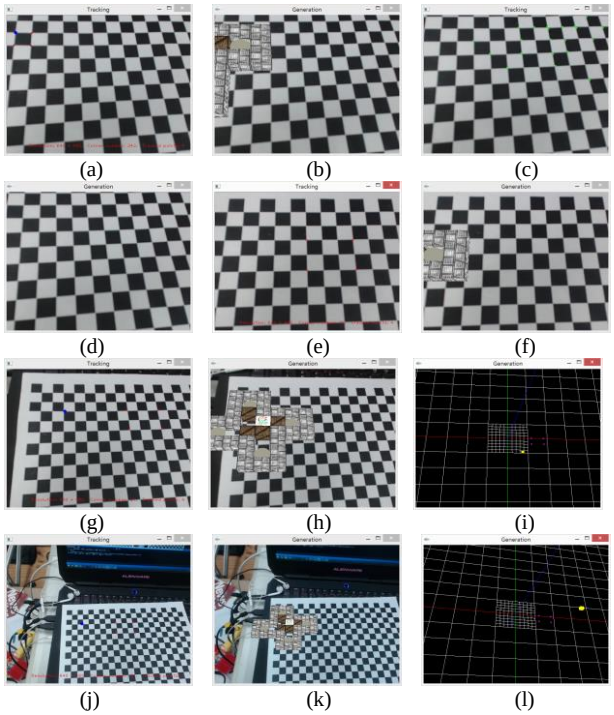


(a) (b) (c)

(c) (d) (e)

(f) (g) (h)

Fig.14. Camera Pose Update Perspective 1: (a) track four corners, (b) render AR objects, and (c) four corners (pink) and camera (yellow) in OpenGL coordinate.Perspective 2: (d) track four corners, (e) render AR objects, and (f) four corners (pink) and camera (yellow) in OpenGL coordinate.Perspective 3: (g) track four corners, (h) render AR objects, and (i) four corners (pink) and camera (yellow) in OpenGL coordinate.

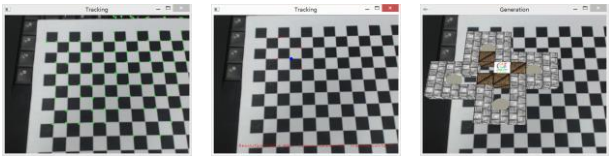


(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)



(m) (n) (o)

Fig.15. Tracking Recovery (a) tracked four corners are going to miss, (b) AR object is going to miss, (c) detect now feature points after missing, (d) no object is rendered, (e) select new four corners, (f) AR object is rendered again, (g) the blue center does not change, (h) AR object is rendered at the same location like before missing, (i) four new corners (pink) and camera (yellow) in OpenGL coordinate, (j) make the camera farther away, (k) render AR object, (l) show them in OpenGL coordinate (m) tracking missed and detect new corners again, (n) select four of them and the blue center appears again, and (o) AR object is rendered again

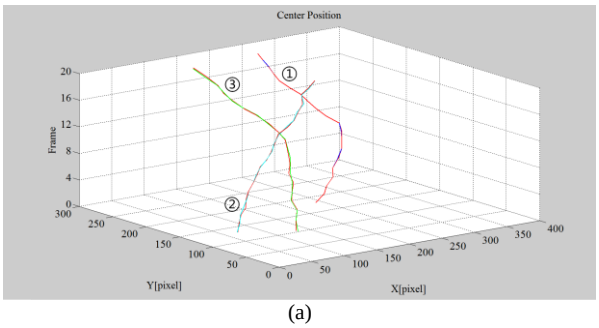
Fig. 14 shows the results of three consecutive perspectives when we move the camera. Table 4 shows the camera pose data of above three perspectives. Combining the data of translation matrix, we draw the camera in the world coordinate by OpenGL, which is shown in Fig. 14_(c)(f)(i).

The average time cost of the calculation for camera pose update is only 1 ms, which is less than it in monoSLAM (5 ms) and PTAM (3.7 ms). This experiment demonstrates that our algorithm about rapid update of camera pose works well.

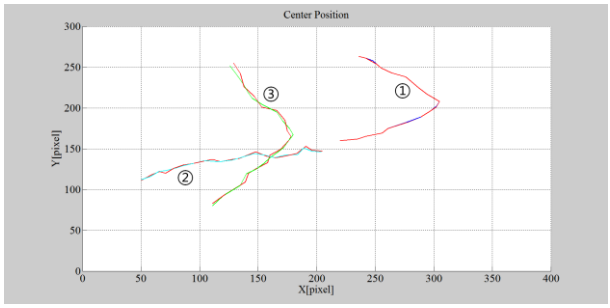
C. Tracking Recovery

Here we do the experiment about tracking recover, which is shown in Fig. 15. When the feature points are missing, we detect and select the corners again. Then we use the camera pose at the previous frame and the new 2D corners to compute the new 3D points. Last we use these new corresponding 2D and 3D points to calculate the camera pose again.

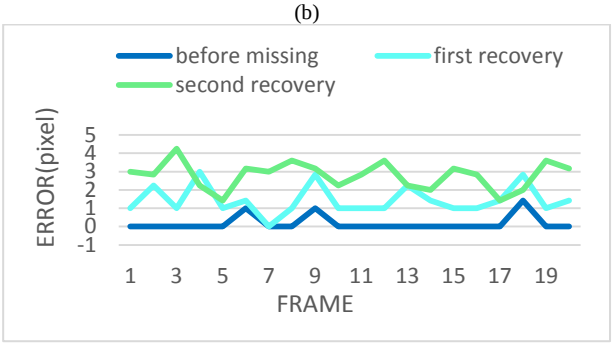
To test the performance of the tracking recovery, we calculate the new center by new camera pose and the 3D center coordinate (*0*, *0*, *0*) of the world coordinate. Then we draw it on the 2D image and calculate the error between the new center and the initial center by distance of pixels. The result is shown in Fig. 16. As we said before, the error is very small although it will be increased slowly.



(a)



(b)



(c)

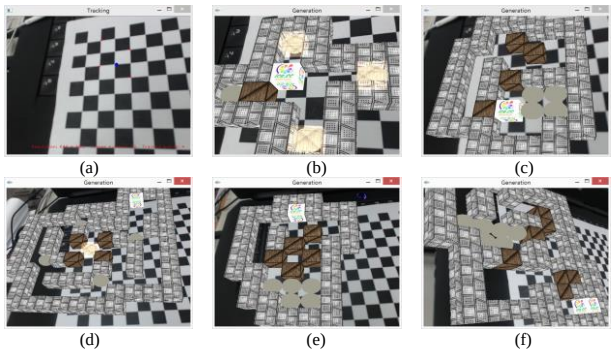
Fig.16. Error between Initial Center and New Center after Tracking Recovery(a), (b) error in sequential 20 frames (① before missing ② first recovery ③ second recovery), the red curve is initial center pose and the other colorful curves are center pose after calculation, and (c) error comparison between ①, ② and ③.

This experiment demonstrates that our tracking recover has a good performance in limited number of times.

D. Comparison with other System

We compare our system (Table 7) with monoSLAM (Table 5) and PTAM (Table 6) that are introduced in Section Ⅰ-B in terms of calculation speed. Here we give the cost of sub-process in each thread. We ignored the time cost of ORB corners detection with approximate depth information and the tracking recovery in time statistics, because the former only occurs at the beginning and the latter only occurs when the feature points are missing. We can see from above tables that our system saves much time meanwhile also has a good result.

E. AR objects rendering & Application



(a) (b) (c)

(d) (e) (f)

Fig.17. AR Game(a) four points are tracked (sequential with the previous figure), (b) AR game is rendered, and (c), (d), (e), (f) different levels of Sokoban.

Here we design an AR game named Sokoban, a box pushing game as in Fig. 17, while using OpenGL to render the texture. The rendering process is implemented in an independent thread, and it runs with the main thread at the same time. Any kinds of objects can be rendered on this map, and the time cost of rendering is increased with the increasing of complexity of AR objects.

Table 5. The Cost of Sub-process in MonoSLAM [7]

Sub-process	Time cost
Image loading and administration	2 ms
Image correlation searches	3 ms
Kalman Filter update	5 ms
Feature initialization search	4 ms
Graphical rendering	5 ms
Total	19 ms

Table 6. The Cost of Camera Pose Update in PTAM [3]

Sub-process	Time cost
Keyframe preparation	2.2 ms
Feature projection	3.5 ms
Patch search	9.8 ms
Iterative pose update	3.7 ms
Total	19.2 ms

Table 7. The Cost of Sub-process in Our System

	Sub-process	Time cost
Tracking thread	(ORB corners detection with depth)	(24 ms)
	Feature tracking	4 ms
	Camera pose update	1 ms
	(Tracking recovery)	(19.5 ms)
Rendering thread	Map generation	1.5 ms
	AR objects rendering	0.5 ms
	Total	5 ms

V. Conclution

This work presents a method of tracking feature points with depth information to update the camera pose and generating a synchronous map for AR system with a certain tracking recovery ability.

The ORB corners with depth information works well and it is easily for us to select some feature points on the same plane. LK optical flow algorithm tracks feature points robustly, which is very important for calculating the camera pose later. Calculating rotation and translation of moving camera by four non-collinear but coplanar feature points also works well. Sometimes three of them are almost collinear, which make the pose unstable. To avoid this kind of cases, we can select those four points by ourselves instead of the system as what we do in tracking recovery. In rendering process, the map and AR objects are rendered well and objects can rotate and translate together with the camera accurately. The parallel method saves time successfully so that more complex AR objects can be rendered. For tracking recovery, it has a good result after calculation in limited number of times. When the number of missing times is increased, the calculation error will become larger because we use the

camera pose at the frame before missing. In the future work, the system will be improved more in stability and accuracy.

ACKNOWLEDGMENT

This research is supported partially by Waseda University (2014K-6191, 2014B-352, 2015B-346), Kayamori Foundation of Informational Science Advancement (K26kenXIX-453) and SUZUKI Foundation (26-Zyo-I29), to which we would like to express our sincere gratitude.

REFERENCES

[1] Rublee, Ethan, et al. "*ORB: an efficient alternative to SIFT or SURF*." Computer Vision (ICCV), 2011 IEEE International Conference on. IEEE, 2011.

[2] Bouguet, Jean-Yves. "*Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm.*" Intel Corporation 5 (2001): 1-10.

[3] Klein, Georg, and David Murray. "*Parallel tracking and mapping for small AR workspaces.*" Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on. IEEE, 2007.

[4] Marius Muja and David G. Lowe, "*Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration*", in International Conference on Computer Vision Theory and Applications (VISAPP'09), 2009.

[5] Zhang, Zhengyou. "Flexible camera calibration by viewing a plane from unknown orientations." Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on. Vol. 1. IEEE, 1999.

[6] Zhang, Zhengyou. "*A flexible new technique for camera calibration.*" Pattern Analysis and Machine Intelligence, IEEE Transactions on 22.11 (2000): 1330-1334.

[7] Davison, Andrew J., et al. "*MonoSLAM: Real-time single camera SLAM.*" Pattern Analysis and Machine Intelligence, IEEE Transactions on 29.6 (2007): 1052-1067.

[8] Davison, Andrew J., Walterio W. Mayol, and David W. Murray. "Real-time localization and mapping with wearable active vision." Mixed and Augmented Reality, 2003. Proceedings. The Second IEEE and ACM International Symposium on. IEEE, 2003.

[9] Bailey, Tim, et al. "Consistency of the EKF-SLAM algorithm." Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on. IEEE, 2006.

[10] Chekhlov, Denis, et al. "Real-time and robust monocular SLAM using predictive multi-resolution descriptors." Advances in Visual Computing. Springer Berlin Heidelberg, 2006. 276-285.

[11] Davison, Andrew J. "Real-time simultaneous localisation and mapping with a single camera." Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on. IEEE, 2003.

[12] Rao, G. Mallikarjuna, and Ch Satyanarayana. "Visual object target tracking using particle filter: a survey." International Journal of Image, Graphics and Signal Processing 5.6 (2013): 1250.

[13] D.G. Lowe, "*Object Recognition from Local Scale-Invariant Features*" Proc. Seventh Int'l Conf. Computer Vision, pp. 1150-1157, 1999.

[14] Rosten, Edward, and Tom Drummond. "*Fusing points and lines for high performance tracking.*" Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on. Vol. 2. IEEE, 2005.

[15] Rosten, Edward, and Tom Drummond. "*Machine learning for high-speed corner detection.*" Computer Vision–ECCV 2006. Springer Berlin Heidelberg, 2006. 430-443.

[16] Castle, Robert O., Georg Klein, and David W. Murray. "Wide-area augmented reality using camera tracking and mapping in multiple regions." Computer Vision and Image Understanding 115.6 (2011): 854-867.

[17] Castle, Robert O., and David W. Murray. "Keyframe-based recognition and localization during video-rate parallel tracking and mapping." Image and Vision Computing 29.8 (2011): 524-532.

[18] Harris C, Stephens M. A combined corner and edge detector[C]//Alvey vision conference. 1988, 15: 50.

[19] Bay H, Tuytelaars T, Van Gool L. Surf: Speeded up robust features[M]//Computer vision–ECCV 2006. Springer Berlin Heidelberg, 2006: 404-417.

[20] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. Brief: Binary robust independent elementary features. In In European Conference on Computer Vision, 2010.

[21] P. L. Rosin. Measuring corner properties. Computer Vision and Image Understanding, 73(2):291 – 307, 1999.

[22] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereovision," Int. Joint Conf. on Artificial Intelligentc, pp. 674-679, 1981.

[23] Horn B K, Schunck B G. Determining optical flow[C]//1981 Technical symposium east. International Society for Optics and Photonics, 1981: 319-331.

[24] Brandt J W. Improved accuracy in gradient-based optical flow estimation[J]. International Journal of Computer Vision, 1997, 25(1): 5-22.

[25] Cramer, Gabriel (1750). "Introduction à l'Analyse des lignes Courbes algébriques" (in French). Geneva: Europeana. pp. 656–659. Retrieved 2012-05-18.

Authors' Profiles

Zheng Chai, male, is currently working toward the Master degree at the Bio-Robotics & Human-Mechatronics Laboratory in the Graduate School of Information, Production and Systems Department, WASEDA University. His research interests include Computer Vision, Image Processing, Augmented Reality and Software Development.

Takafumi Matsumaru, male, is a Professor at the Bio-Robotics & Human-Mechatronics Laboratory in the Graduate School of Information, Production and Systems Department, WASEDA University. His research interests are mainly on Human-Robot Interaction both physically and informatively.