Framework for an augmented reality application to add 3D objects to a scene

Braulio Chavez
Stanford University
braulioc@stanford.edu

Ritu Bajpai
Stanford University
rbajpai2@stanford.edu

March 22, 2018

Abstract

In this work we attempt to implement a small scale real time detection, mapping and tracking framework. We take real time video feed as input. On the first frame we do keypoint detection and evaluate descriptors for the keypoints. Using keypoint matching we track these points in the subsequent frames. New points are added as they are detected in the frame. Such tracking and mapping is useful for augmented reality applications. We also show basic image augmentation with a virtual object.

1. Introduction

Augmented reality applications require study and understanding of the real life scene. Using the real life scene parameters, we are able to build applications to add virtual objects and enhancements to the scene. Such applications can be used for fun as in the game Pokemon Go, for education such as to render multimedia in real time for students to participate and interactively gain knowledge in more natural way, for architecture in visualizing buildings, in commerce for print and video marketing and for various other applications.

A good understanding of simultaneous localization and mapping (SLAM) can be obtained in [10]. Many tutorials on the web can be followed to gain a good understanding of the workflow for real time tracking and mapping applications [5]

2. Background and Related Work

Developing realtime understanding of the scene and tracking the camera position is needed for augmented reality applications. In this work we attempt to develop a simple methodology to implement this task.

Figure 1 shows a general overview of the pipeline we are implementing to augment camera image frames. We'll make use of existing Android APIs to extract sensor data and camera intrinsic parameters. The Android's camera internal life-cycle will call our Image Processor module on each image frame. The Image Processor Module will be responsible for Feature Detection, Virtual Object Augmentation and reporting back the processed image frame. It has to be fast enough so that we don't slow down the frame rate. Finally the device's display screen will render the final produced frame to the user.

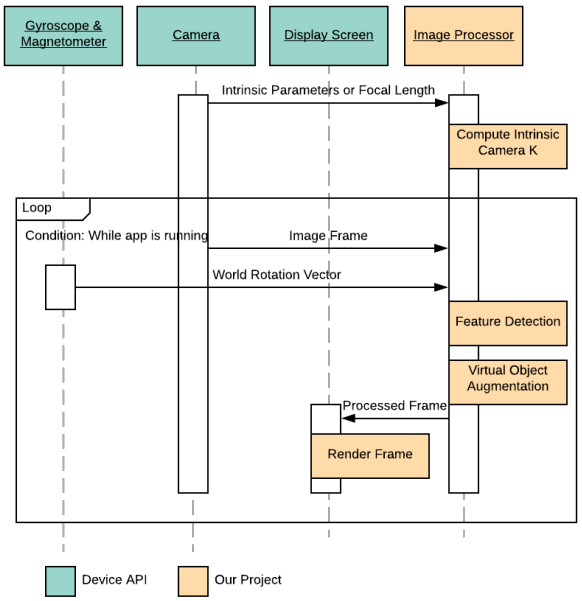


Figure 1. Processing Pipeline. We leverage the Android API to get sensor data, camera intrinsic parameters, and display screen rendering. Image Processor executes feature detection and image augmentation with 3D objects on each frame.

3. Approach

For most of the steps outlines in section 2 we have tried to use algorithms and approaches which have been widely used and also available as off the shelf open-source codes.

3.1. Keypoint/feature detection

Keypoint or feature detection is one of the first steps for establishing points of interest which are then tracked in the scene. First frame is used as a reference frame/image where we detect the keypoints of interest. Once a new/query frame is received, keypoint detection is also implemented on the query frame. Keypoints in the query frame are matched with the reference frame. If the matching is found for a keypoint in the query frame, it is marked as a point that we have already found. Any unmatched keypoints in the query frame are saved as a new keypoint. Keypoints for a given frame are highlighted with a marker in the frame.

Our first approach for detection and tracking was to detect a dominant plane in the image instead of the keypoint. For this implementation, we attempted image segmentation based on texture detection as described in [14]. Texture segment with largest area was to be assumed to be the ground plane. The algorithm was run on the dataset used in [14]. In our implementation, the descriptor was not strong enough to provide clear differences between various material textures. We did not find sufficient supporting information to troubleshoot our implementation. We next considered using HOG feature extractor and matching. However we concluded that HOG feature extraction and sliding window matching will be too slow for our application.

On further investigation about most popular descriptors used for AR applications, ORB (oriented FAST rotated BRIEF) based keypoint detection and matching [12] was chosen for our application. It has been shown [12] that ORB is well suited for applications such as panorama stitching and patch tracking, and to reduce the time for feature-based object detection on standard PCs. It performs as well as SIFT on these tasks (and better than SURF), while being almost two orders of magnitude faster.

ORB uses FAST (Features from Accelerated Segment Test) algorithm for keypoint detection as proposed in [11]. As shown in figure 2, 16 pixels are selected around a given pixel and several comparisons based on the intensity values of the pixels are made. These steps as outlined in the opencv documentation [2] are followed in this work:

1. Select a pixel p in the image which is to be identified as an interest point or not. Let its intensity be I_p.

2. Select appropriate threshold value t.

3. Consider a circle of 16 pixels around the pixel under test.

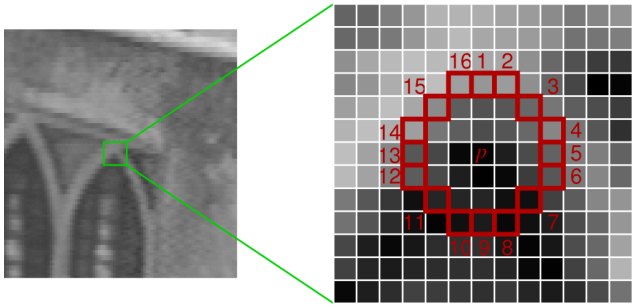


Figure 2. The highlighted squares are the pixels used in the corner detection. The pixel at p is the centre of a candidate corner. The arc is indicated by the dashed line passes through 12 contiguous pixels which are brighter than p by more than the threshold [11]. We have used 9 points for detection as compared to 12 [11].

4. Now the pixel p is a corner if there exists a set of n contiguous pixels in the circle of 16 pixels which are all brighter than $I_p + t$,or all darker than $I_p t$. We chose $n = 9$ for FAST-9.

5. A high-speed test was proposed to exclude a large number of non-corners. This test examines only the four pixels at 1, 9, 5 and 13 (First 1 and 9 are tested if they are too brighter or darker. If so, then checks 5 and 13). If p is a corner, then at least three of these must all be brighter than $I_p + t$ or darker than $I_p t$. If neither of these is the case, then p cannot be a corner.

6. The full segment test criterion can then be applied to the passed candidates by examining all pixels in the circle.

FAST keypoint detection algorithm is very senstive to edges and tends to find several keypoints along the edges in the image. Harris corner measure is used to filter out the points with highest corner-like properties.

ORB uses BRIEF (Binary Robust Independent Elementary Features) descriptor [7] to identify each keypoint detected using the FAST algorithm. Once again we referred to the opencv documentation for step by step implementation of BRIEF [1]. BRIEF provides a shortcut to find the binary strings directly without finding descriptors.It takes smoothened image patch and selects a set of $n_d(x, y)$ location pairs in an unique way (explained in paper). Then some pixel intensity comparisons are done on these location pairs. For eg, let first location pairs be p and q. If $I(p) < I(q)$, then its result is 1, else it is 0. This is applied for all the n_d location pairs to get a n_d-dimensional bitstring. This n_d can be 128, 256 or 512. We use 256 in this work.

BRIEF is a descriptor which does not provide any matching technique. We use Hamming Distance to match BRIEF descriptors as in the original paper. In simple terms for a given binary descriptor, hamming distance is the count of

the number of bits which are different across the two descriptors. It has been implemented using XOR operation in our work.

3.2. Estimating Camera Position

Important part of camera pose estimation is to find the intrinsic and extrinsic matrix of the camera. Our initial goal was to deploy the project into an Android powered device. We therefore leverage the Android API to avoid having to do our own calibration.

3.2.1 Intrinsic Matrix

We used Android Camera 2 API to aid in the computation of the Intrinsic Matrix K. This API supports getting Camera Characteristics [4] that reflect parameters particular of the camera sensor. One of such characteristics is $LENS\_INTRINSIC\_CALIBRATION$, this property returns a list containing the internal camera parameters for focal length, center point coordinates, and skewness represented with array:

$$[f_x, f_y, c_x, c_y, s]$$

With these parameters we can easily compute the intrinsic camera matrix K:

$$\begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Unfortunately there are some Android devices which don't support this Camera Characteristic. As a fallback method we can do an approximation of K based of the $LENS\_INFO\_AVAILABLE\_FOCAL\_LENGTHS$ characteristic which is guaranteed to always have a value present. Once we have focal length f, we assume that f_x equals f_y. We also assume that the center point coordinates are precisely in the center of the image:

$$c_x = imageWidth/2$$

$$c_y = imageHeight/2$$

And finally we assume that we have no other distortions. This would give us approximated intrinsic camera matrix K:

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

3.2.2 Extrinsic Matrix

For the initial simplified version of our project we assume that there's no Translation at all, to accomplish this we plan to mount our camera to a tripod.

We will take care of Rotation of the camera by again leveraging the Android hardware sensors API. We will query to get the current rotation with $Sensor.TYPE\_ROTATION\_VECTOR$ [3], which internally uses a gyroscope and a magnetometer. This vector is described as follows:

- X is defined as the vector product Y.Z (It is tangential to the ground at the device's current location and roughly points East).
- Y is tangential to the ground at the device's current location and points towards magnetic north.
- Z points towards the sky and is perpendicular to the ground.

Alternatively, extrinsic matrix can be estimated using the intrinsic matrix and the detected key points. We start by taking points in the reference plane. We can assume these points to be on ground plane at a reasonable distance from the camera. We assume there is no rotation between the camera and the ground plane in the reference frame and all the points used for extrinsic matrix calculation are planar. Therefore from the reference frame, we know points on the image and corresponding points in 3D world reference system. We know the camera intrinsic matrix and can also detect key points in the query frame. Using the aforementioned information, we can now estimate the translation and rotation vectors that determine the pose corresponding to the query frame.

3.3. Establishing the Homography

Once enough keypoints are detected and matched in the reference frame and the query frame, we can compute the homography matrix which relates the two frames. We use RANSAC [8] to further refine the keypoint matching between the reference frame and the query frame.

3.4. Tracking and mapping keypoints across frames

In order to track the points in a video, we need to extract keypoints from different frames. First frame is used as a reference frame. For the subsequent frame (query frame), we match the keypoints with the reference frame. Matched points are updated with the point correspondence information. Unmatched points are added as new keypoints which can be matched with the future frames. Query frame then becomes the new reference frame for the subsequent incoming frame.

3.5. Augmentation with virtual objects

Once we have calculated the intrinsic and extrinsic camera parameters and established homography between the reference frame and the query frame, we want to ensure that the virtual object brought into the frame would also respond to the homography between the two frames.

We first extract keypoints from the first frame as an initial reference, then on the subsequent frames we compute keypoint matches and keep those around across frames. To augment the image we take one of the keypoints and follow it through frames, this keypoint will provide the coordinates to where we render the 3D Object.

4. Experiment

This section shows implementation details and results of different modules of the pipeline.

4.1. Code Implementation

All the related code for this project can be found at https://github.com/ElHacker/PTAM

4.2. Keypoint detector and descriptor

We start the implementation of the keypoint detector and descriptor algorithm assuming still images as the input. Most of the algorithm is based on the ORB algorithm. Results of FAST keypoint detector as obtained using our implementation are shown in figure 3. As we see from the detected keypoints, the FAST detector is very sensitive to the edges. To overcome the problem of detecting multiple points along the edges and improving the 'cornerness' measure, we use Harris corner measure to filter the top keypoints. Therefore, we use low enough threshold for the FAST detector to generate enough keypoints and then filter top N points using the Harris corner measure. Results from our implementation of harris corner measure based filtering of FAST9 keypoints are shown in figure 4.

We use BRIEF descriptor for the keypoints with a descriptor length of 256. Hamming distance is used for matching keypoints across the frames. Based on the results in [7], we use hamming distance of 64 or less to establish a keypoint match. Keypoint matching results between two images using BRIEF and hamming distance are shown in figure 5. As we see from figure 5(a), we get good match using hamming distance for the scaled image. For the scaled and rotated image as shown in 5(b), we get a good match between some points but there are also a lot of spurious matches. We use RANSAC to refine the matches which produces good results with no spurious points as shown in 5(c).

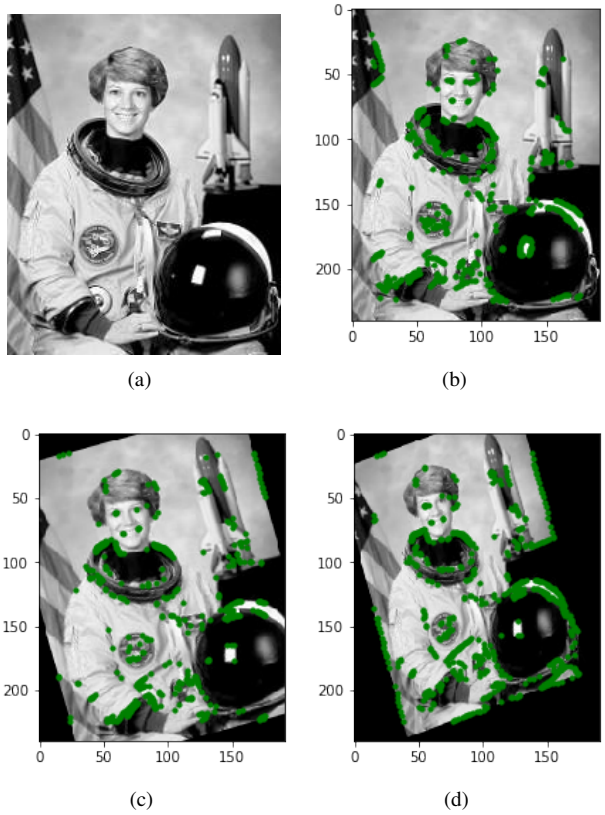


Figure 3. (a) Original input image (b) Keypoints generated on the input image using FAST9 algorithm. Keypoints marked with green dots on the image (c) Keypoints generated on the rotated image (d) Keypoints generated on the rotated and scaled image. This image was downloaded from the NASA Great Images database https://flic.kr/p/r9qvLn. Photograph of Eileen Collins, an American astronaut.

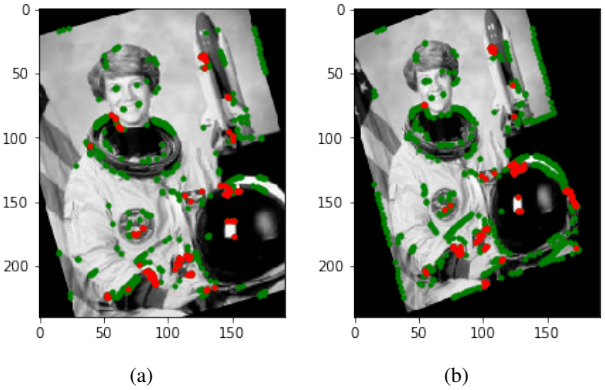
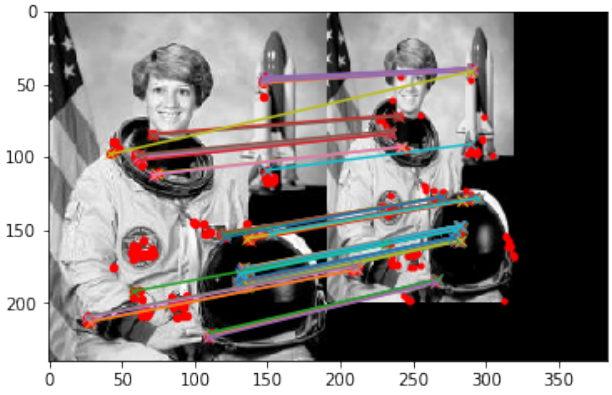
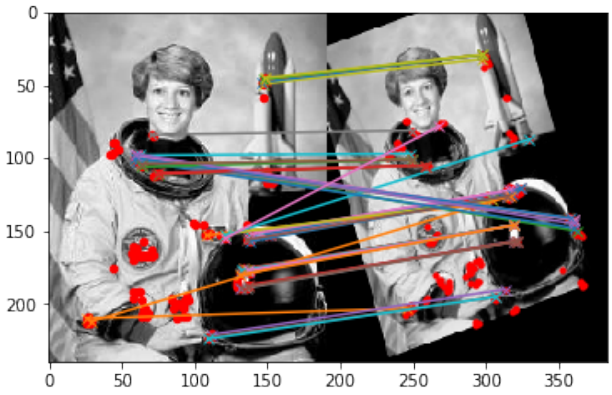


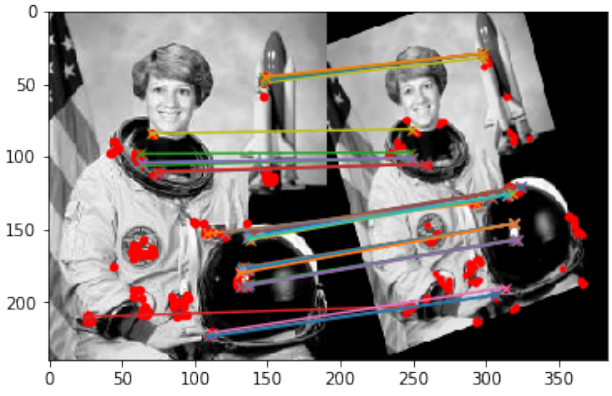
Figure 4. (a) Top keypoints in Fig. 3(c) filtered by Harris corner measure marked in red (b) Top keypoints in Fig. 3(d) filtered by Harris corner measure marked in red



(a)



(b)



(c)

Figure 5. (a) Kepoint matching in scaled image (b) Keypoint matching in scaled and rotated image (d) Keypoint matching in scaled and rotated image, further refined with RANSAC.

4.3. Tracking keypoints across frames

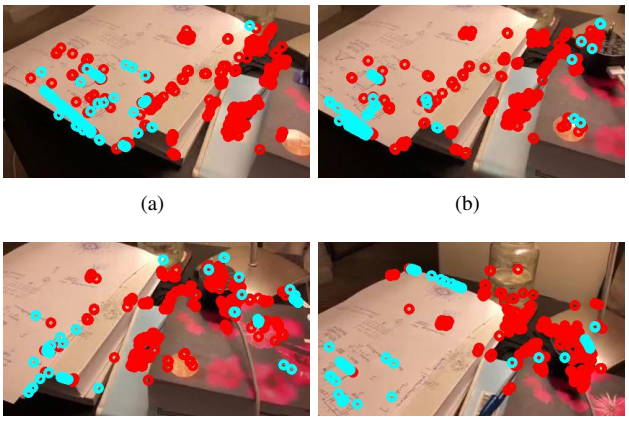
Once the keypoint detection and matching algorithm was implemented and finalized, we used it on the video for tracking and mapping across the frames. In order to compare our results we also used ORB keypoint detector and matching module from the scikit library [13].

We read the first frame from the video. This frame is the reference frame. Keypoints and keypoint descriptors are calculated for the reference frame. We call the next frame as the query frame. Keypoints and descriptors are evaluated from the query frame. Keypoint matching is performed between the reference frame and the query frame. For every match found there are two possibilities i) the matching is new, meaning we haven't tracked that keypoint before. In this case the keypoint is added as a new keypoint to the set of tracked keypoints. ii) the match is a continuation which means we have tracked that keypoint before. In this case, the keypoint is updated to have the same identifier as in the reference frame. Results of keypoint tracking across the frames are shown in figure 6. We used a scene with a desk showing usual items such as hand written notes, a book, phone charger, water jar. Figure 6 shows frame 1, frame 100, frame 200 and frame 300 from the video. Red keypoints correspond to the ORB detection and descriptor module used from opencv. Blue points correspond to the keypoints detected using our implementation of FAST keypoint detector further filtered by Harris corner measure. BRIEF descriptor has been used with hamming distance and RANSAC for matching. We see that despite filtering using harris corner measure, our detector is still very sensitive to edges. Also note that the displayed number of blue points is less than the red points. This is because our detector and descriptor implementation is not as fast as opencv implementation. Therefore, in order to maintain relatively fast speed, specially in real time operation as on a phone application or webcam video, we reduced the number of keypoints for our implementation.

4.4. Image augmentation

We first calibrated the camera we were going to use for the augmentation. We used out of the box OpenCV calibration functions with a chessboard grid of $9x7$. This helped us get the camera's internal parameters. You can see one of the 14 calibration images we used on figure 7

To augment the image we made use of Python OpenGL and Pygame. We first draw the camera image with an orthogonal projection onto a OpenGL texture and use this texture as the background of the Window's Surface. Before overlying a 3D Model onto our image we must first compute a Perspective Matrix so that our Projection matches the background image that was taken with our Camera internal parameters. To do this we compute a Perspective Matrix using our calibrated camera internal parameters, near clipping and far clipping distances. Here's how our final Perspective



(a) (b) (c) (d)

Figure 6. (a) Frame1 (b) Frame 100 (c) Frame 200 (d) Frame 300 from a video. Red points are generated using the opencv ORB detector and descriptor. Blue points are generated using our custom keypoint detector and descriptor which is also following majority of the ORB algorithm.

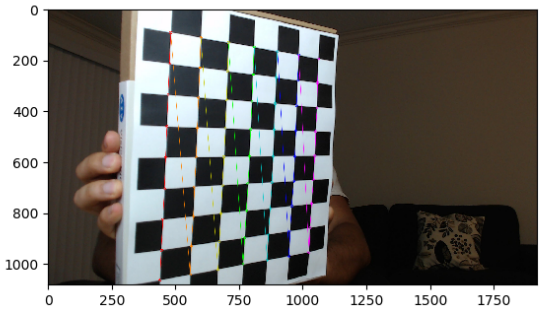


Figure 7. The 9x7 chessboard with the drawn corners used for calibration.

Matrix looks like:

$$M_{persp} = \begin{bmatrix} \alpha/c_x & 0 & 0 & 0 \\ 0 & \beta/c_y & 0 & 0 \\ 0 & 0 & \frac{-(far+near)}{(far-near)} & \frac{-2far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

α is camera focus length on x, β is camera focus length on y. far and $near$ are OpenGL Projection Matrix Clipping planes. To apply this matrix to OpenGL's Projection Matrix we do $glMultMatrixf(M_{persp})$. Read more on [9].

Now we're ready to render our chosen 3D Object extracted from a *.obj* file. We still need to map the camera image coordinates to OpenGL coordinates since OpenGL coordinate origin is on the center of the image. After doing this simple operation we render the model on top of our background texture. You can see an augmented image on Figure 8 You can also see a video demo of this augmentation pipeline on Youtube [6]

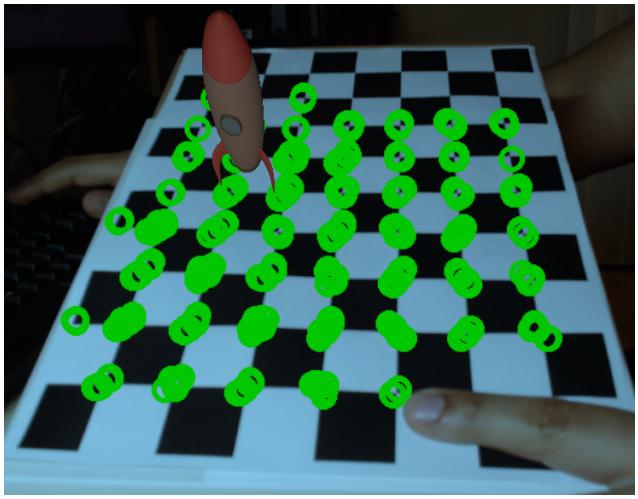


Figure 8. We can see an augmented image, the rocket model was downloaded from Poly. The model is rendered by following around the closest keypoint.

4.5. Android Implementation

We implemented an integration with Android Camera 2 API. This is designed to give us full control of each image frame that the camera sensor captures. Each frame is sent to our image processing method that computes the feature detection, this image is passed to from Java to a Python Numpy array, which ideally should do the frame processing and 3D object rendering. You can see all the Android work we did on our Github project page.

Note that each frame is processed by us before therefore we need this computation to be fast and realtime to don't affect the frame rate of the application and keep up with Device's camera movement in the world coordinates.

Our Android implementation only covers the Android Camera 2 API integration and reporting the image to a Numpy array in the Python environment. Although we used Python For Android build framework, we couldn't get all the Python packages we used in class into our Android Build. We were only able to get Numpy to be included. Not having our usual tools available made our work on the Android platform really hard as we're not familiar with class equivalent packages in Java. That's why we decided to keep developing our processing and augmentation pipeline on Desktop where we had access to all our familiar tools.

5. Challenges

1. Algorithms using looping over each pixel of an image are slow and computationally heavy. Therefore feature descriptors have to be carefully chosen and implemented to address this issue. This becomes specially important for real time applications.

2. Although our initial goal was to run the application on an Android hand held device, we were not successful in figuring out and executing the complete pipeline. We therefore changed the goal to executing the application on desktop. We were able to implement it using the opencv ORB module. Our implementation was relatively slow for real time application.

3. Even if we had figured out the pipeline on the android device, we are not sure if we would have enough computation speed on the phone.

6. Conclusion

We were able to implement the basic modules of augmented reality applications. We investigated the most suitable feature detectors and descriptors for this task and found ORB to be the best choice. It is rotation invariant and fast for real time applications. We were successfully able to track keypoints in a video and the webcam feed. We also established homography between the reference frame and the query frame. Further, scene augmentation by rendering a virtual object in the image was accomplished.

However, our implementation is not fast enough for real time applications. It can be further improved to match the performance of the ORB module from opencv. Mapping of virtual object across the frames was not demonstrated. This is due the lack of enough time for execution. Given that we have established the homography between the frames, we can easily map the virtual object across different frames. We were not successful in demonstrating the application on handheld android device as per the initial goal. Again, this was partly due to strict time constraint to finish this project. Overall, this work can be improved upon to show full execution of a 3D augmented reality application.

References

[1] http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_brief/py_brief.html.

[2] https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_fast/py_fast.html.

[3] Android. https://developer.android.com/reference/android/-hardware/sensor.html#type_rotation_vector.

[4] Android. https://developer.android.com/reference/android/hardware/camera2/cameracharacteristics.html.

[5] Blog. http://www.anandmuralidhar.com/blog/tag/camera-intrinsic-matrix/.

[6] R. B. Braulio Chavez. https://youtu.be/0qt57x_es4a.

[7] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. Brief: Binary robust independent elementary features. In *European conference on computer vision*, pages 778–792. Springer, 2010.

[8] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. In *Readings in computer vision*, pages 726–740. Elsevier, 1987.

[9] K. George. http://kgeorge.github.io/2014/03/08/calculating-opengl-perspective-matrix-from-opencv-intrinsic-matrix.

[10] S. Riisgaard and M. R. Blas. Slam for dummies. *A Tutorial Approach to Simultaneous Localization and Mapping*, 22(1-127):126, 2003.

[11] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *European conference on computer vision*, pages 430–443. Springer, 2006.

[12] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE international conference on*, pages 2564–2571. IEEE, 2011.

[13] Scikit. http://scikit-image.org/docs/dev/auto_examples/features_detection/plot_orb. 2018.

[14] A. T. Targhi, E. Hayman, J.-O. Eklundh, and M. Shahshahani. The eigen-transform and applications. In *Asian Conference on Computer Vision*, pages 70–79. Springer, 2006.