Edge-Assisted Trackerless Augmented Reality

Sofiya Semenova, Vaibhav Chincholkar, Utkarsh
CSE 622 - Advanced Computer Systems

I. Introduction

There is a simultaneous push and pull for the development of production-ready, state of the art augmented reality (AR) applications; the production of more and more AR devices *pulls* the development of AR applications, while the desire for AR applications from the travel, entertainment, manufacturing/construction, and advertising industries (to name a few) *push* their development. As application developers create more production-ready AR applications, they will have to consider the performance and security requirements of these applications.

Security AR applications that display 2D or 3D objects on the user's AR device need some spatial representation and understanding to be able to render the object correctly — to make an object appear on the 2D camera plane as if it were "in real life". This spatial representation can be as simple as detecting the user's 3D pose in relation to a *tracker* that exists in real life, or as complex as a dense point cloud that effectively functions as a map for place detection.

Trackers can be any 2D object with a known size and shape, but frequently QR codes or apriltags [1] are used. They work well with AR applications, require little overhead for setup, and are "secure" because they never create a visual map of the environment. However, they are clunky for production applications because they must remain in the physical environment for the application to continue working. In some cases, keeping a printed QR code on all physical surfaces that can be augmented might be visually distracting — imagine walking down a busy street without an AR device and seeing QR codes on every surface. In others, it's completely unacceptable — imagine wanting to display yesterday's notes on a whiteboard with an AR device, but having to write around a QR code any time you want to use the whiteboard as an actual whiteboard.

For a spatial representation that *does* create a visual map of the environment, many applications may not want to allow the user to access the map in its entirety. Buildings that contain restricted areas as well as public-facing areas, for example, might want to allow a user that is already granted *physical* permission to a restricted area to be able to seamlessly view AR objects around them, but not want to allow a user who is not physically in the restricted areas to have access to the full map. Simply put, users should only be able to access only map information that is around their physical location.

While approaches to securing information from one user to another within a device exist, edge computing offers a viable alternative to the problem of map security. If only relevant, vetted information is sent to an AR device, that device can never retrieve map information internally.

Performance AR applications that are meant to be production-ready and deployed in real environments have a real-time requirement for frame processing. Because AR applications are nearly always visual and small drops in frame rate and rendering lags are easily perceivable to users, maintaining reasonable performance is critical to the success and adoption of any AR application. Further, the devices that AR applications run on are inherently resource constrained - they are, at best, new mobile phones, and at worst, AR headsets. With the predicted end of Moore's law, application developers cannot rely on better hardware to fix any performance problems.

Previous approaches to latency optimization in AR environments focus on the rendering side [2] or on splitting computation between client and servers [3]. A core limitation to latency optimization that [2] identified is that proprietary headsets don't al-

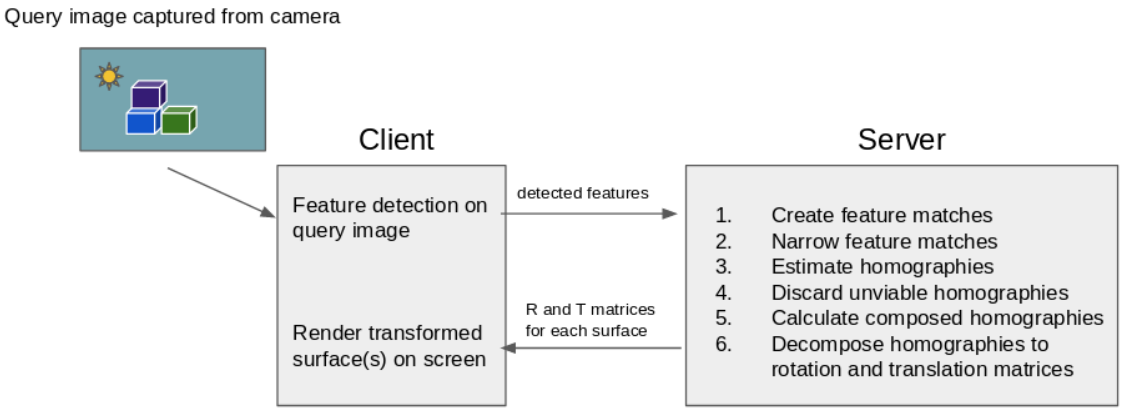

Fig. 1. Our system architecture

low applications to see or modify any internals, so any optimizations must be done at the application layer.

Given the above considerations, our project focuses on implementing a trackerless AR application that splits computation between a mobile phone and edge device. We also provide an overview of the different approaches to place recognition and pose estimation as they pertain to AR applications. The rest of the report is structured as follows: Section II provides an explanation of our application's architecture, with an overview of our vision pipeline. Section III provides an in-depth discussion of the problems of place recognition, pose estimation, and rendering as they apply to AR applications, with an overview of different approaches for each. Section IV-V provides discussion of two techniques we looked into but did not end up working with — ARToolkitX for tracker detection, pose estimation, and rendering, and OpenGL for rendering.

II. Application Architecture

Our application splits the vision pipeline implementation between a client and a server. The code for the server can be found in [6] and the code for the client can be found in [7]. An overview of the architecture can be seen in Figure 1. Offline, we calibrate the camera of our test device, using OpenCV's `cameraCalibration` function and a printout of a chessboard, to get the camera calibration matrix (see section III-A for a more detailed explanation).

A. Place Database

We also provide a *place database* for localization — it is structured as a series of *places* with known *surfaces* for projection. Both places and surfaces are image files, where each place is a photo of a location, each surface is a photo of a 2D plane (we used textbooks and whiteboards, but anything could be used), and each surface corresponds to a plane to project AR objects onto. We also provide a *surface occurrences* file, which lists every surface that appears in every place and the 2D image coordinates of the surface. Previously, we attempted to forgo the surface occurrences step and perform template matching to automatically detect surfaces, but we found that template matching algorithms did not perform well when the template to match (a surface) was smaller than ~1/4 of the scene (a place).

During setup, we link places and surfaces together in memory, calculate the homography for each surface in each place (see section III-C for a detailed explanation on the math behind homographies), and perform feature detection on each place image. Because these steps will need to be taken during runtime for every query, we can save some computation time by front-loading them and saving them to a database.

B. Client

The client is an Android application that retrieves camera frames from the Android device's camera, uses OpenCV to perform feature detection on a frame, and sends the feature descriptors and

keypoints to the server. Upon a response from the server, the client uses OpenCV to render the surface on the camera frame.

C. Server

On every query, the client sends a set of feature descriptors and keypoints to the server. Upon receiving these, the server:

1) Create feature matches from I_q to all images in the database. We found that using ORB descriptors gave the best visual results out of all the descriptors we tried (SURF/SIFT, ORB, and BRIEF).
2) Narrows feature matches to only good matches, using the threshold test with a ratio threshold of .6.
3) Estimate a homography using good matches between I_q and all images in the database. If there are fewer than 10 matches or we cannot find a homography given the matches, that image is discarded.
4) Discard image matches with "unviable" homographies. A homography is unviable if fewer than 50% of the original feature matches still correctly match when applying the homography (ie. there are fewer than 50% inliers).
5) For the image match with the largest amount of inliers, I_{match}, calculate the composed $H_{\text{surface}\rightarrow\text{image match}} * H_{\text{image match}\rightarrow\text{query image}}$ for all surfaces that appear in I_{match}. Remember that $H_{\text{surface}\rightarrow\text{image match}}$ has been precomputed during the offline step. This composed homography describes the series of transformations required to transform the unmodified surface, to the image space of I_{match}, to the image space of I_q.
6) If any image matches with viable homographies (that are not I_{match}) contain surfaces that were not found in I_{match}, repeat step 5 for them. This ensures that all possible surfaces that appear around I_q are accounted for, just in case they don't appear in I_{match}.
7) Decomposes each composed homography into rotation and translation matrices. See Appendix III-E for an example of homography decomposition.
8) Sends the rotation and translation matrices for all surfaces to the client.

D. Network Serialization

The image captured by the phone/client camera ranges from 40KB-60KB on the storage but transferring the image over the network can lead to unnecessary network traffic and hinders the main purpose of a real time application.

However, as the client just needs to transfer the feature descriptors and key-points to the server the same is decomposed on client side using the API docs of OpenCV.

The feature descriptors can be extracted into an array of integers, float-points, double or byte which is based on the type of Mat object returned by type() API call on the Mat object. The target type is classified using the CvType enum defined in the class.

The 8 enums are classified as `CV_32S`, `CV_32SC2` and `CV_32SC3` correspond to an array of integers, `CV_32F`, and `CV_32FC2` correspond to an array of float type, the set `CV_64F` and `CV_64FC2` represent a double array and the type `CV_8U` utilizes a byte array to store the information related to the feature descriptors.

Once extracted the corresponding data type along with the details of size of the array and type is transferred over the network socket. Then the same information is used to reconstruct the Mat descriptor object back on the server side.

The key-points in OpenCV is represented by an array of KeyPoint object. Now the API docs of class KeyPoint in `org.opencv.core.KeyPoint` lead us to decompose key-points into the following listed parameters:

1) angle (a float parameter representing computed orientation of the key-point, -1 if not applicable)
2) class_id (an integer parameter representing object ID, that is used to cluster key-points by an object they belong)
3) octave (an integer representing the pyramid layer, from which the key-point has been extracted)
4) Point (an object representing the x and y float coordinates of the key-point)
5) response (a float parameter which denotes the response by which the strongest key-points have been selected)

6) size (a float parameter representing the diameter of the useful adjacent area).

Once for each image these parameters are decomposed, they are transferred as an ArrayList of their respective types over the network socket. The collected information by the server is used to reconstruct the each KeyPoint object and get back the old key-points from the original image.

III. AR Vision Pipelines

Whether an AR application relies on trackers or builds a map of the environment, projects 2D or 3D objects, the general vision pipeline remains the same. The application must first find a surface on which it should project an object, find the correct way to transform the object into the 2D plane, and render the object.

A. Camera Calibration

For many of the following techniques, getting the correct result relies on having a calibrated camera. Based off of the pinhole model, the camera calibration matrix (the intrinsic matrix), K, defines the way in which coordinates in the camera's coordinate system map to coordinates in the image plane.

$$K = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

The camera calibration matrix remains static for the same camera, for the same focal length. Thus, it can be calculated offline and reused for the same camera. In production applications, a camera calibration matrix database can be created for common AR devices and the correct matrix can be retrieved for each user's device. If no database entry exists for the user's camera, an application can try to construct a matrix from the user's video feed, but it would need:

1) At least 15 frames to get good results
2) A 2D object with known dimensions (a tracker can double as a calibration target)
3) Widely varying positions of the 2D object in each of the provided images

Many applications for camera calibration exist, and all follow a similar process — a user provides images (whether through a directory of images or a realtime video feed) taken with the target camera of a 2D object with known dimensions in various configurations. OpenCV's camera calibration function uses a printout of a chessboard; AprilTag's application uses AprilTags. For our application, we used the OpenCV function with a printout of a chessboard, pre-computed the camera calibration matrix offline, and used the raw matrix in our code.

On top of the calibration matrix K, the camera calibration process can also glean camera distortion coefficients, which can further be used to modify the points in the image plane. However, our application does not use distortion coefficients to modify our results.

B. Converting Between Coordinate Spaces

Given 3D, real world coordinates $\binom{X}{Y}{Z}{1}$ of an object we would like to project, we need to convert them to pixel coordinates $\binom{u}{v}{1}$ in the image plane.

The camera calibration matrix, K, describes a way to convert 3D coordinates in the camera's coordinate space to 2D coordinates on the image plane.

The rotation and translation matrices (the *extrinsic matrix*), describe a way to convert 3D coordinates in real life to the camera's 3D coordinate space:

$$\left[\begin{array}{c|c} R & T \end{array}\right] = \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{2}$$

Thus, one can find $\binom{u}{v}{1}$ with the following formula:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \left[\begin{array}{c|c} R & T \end{array}\right] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3}$$

For projecting a surface onto known points in a scene, the pixel values of the surface are subbed for X and Y, and all Z values are 0. The pixel values of its known 2d, projected location are u and v.

If we want to project either a surface onto known points in a scene (as we do in the offline database

setup portion) or if we want to project a scene into another scene (as we do when we respond to a query image), we also need to find rotation and translation matrices.

C. Template Matching With Trackers

If an AR application is using trackers, finding the surface to project an object onto becomes a problem of *template matching*, where the template is an image of the tracker.

If the end goal of the application is to project a 2D image, then the application merely needs to calculate a *homography* (an affine transformation between two image planes) that will transform the 2D object into the image plane of the query image. In this scenario, the camera calibration matrix is not needed — a homography gives us a way to convert 2D image coordinates to *different* 2D image coordinates, so no conversion from a 3D camera plane is needed.

This process is essentially that which occurs when we set up the place database offline. The template, an image of a surface, is matched to a scene, an image of a place. However, for the final application which does not use trackers, we need a way to get rotation and translation matrices between the matched scene and the query scene.

D. Place Detection Without Trackers

If the application is not using trackers, it must use other information in the environment to find the user's location. This problem is much closer to the problem of *place detection*. Many state-of-the-art place detection approaches involve building a 3D, virtual, visual representation of the real world. SfM (structure from motion) is a commonly-used technique to build this visual representation using only camera frames taken from a moving camera. However, we ultimately only care about our camera's relation to a "virtual" surface, so the overhead of SfM algorithms might not be necessary.

An alternative approach to SfM is to feature match a query image with a database of images with known locations. Our application's approach takes this a step further — the database of images doesn't technically *need* known locations, it merely needs known rotation and translation vectors between an unmodified object visible in the frame and the frame. We calculate these rotation and translation vectors by fitting a homography between a surface image and a place image, and decomposing the homography into rotation and translation vectors.

E. Homography Decomposition

Regardless of the technique used, a camera's intrinsic parameters need to be provided to the homography decomposition function so they can be decoupled from the homography itself.

The OpenCV function `decomposeHomography` can be used to decompose a homography into rotation and translation matrices that satisfy the equation. Because the result of decomposing the homography is not unique, it returns a list of viable options.

We also tried decomposing a homography by hand, and that gave similar results to the openCV function — there don't seem to be any benefits to doing it by hand.

IV. ARToolkitX

ARToolkitX is an application that provides an easy-to-use API for AR applications. It handles tracker detection, pose estimation, and openGL rendering, so all the internals are hidden to Android application developers. Because ARToolkitX implements a vision pipeline very similar to the one for our application, it appears to be a good candidate for building our application off of. However, a thorough review of the ARToolkitX code (open-sourced on GitHub) [5] reveals that none of the modules can be meaningfully offloaded — that is, some modules *can* be offloaded, but it would require sending too many bytes over the network. Further, ARToolkitX relies on tracker detection for object placement, and most of the API cannot be easily decoupled from this. Though we did not end up using ARToolkitX to render objects or detect trackers, we provide the following overview of the modules found in Figure 2.

A. Module Overview

ARVideoSource Provides video frames to the tracking module, and contains information about video (size, pixel format, camera params, raw video data). Because this module relays camera

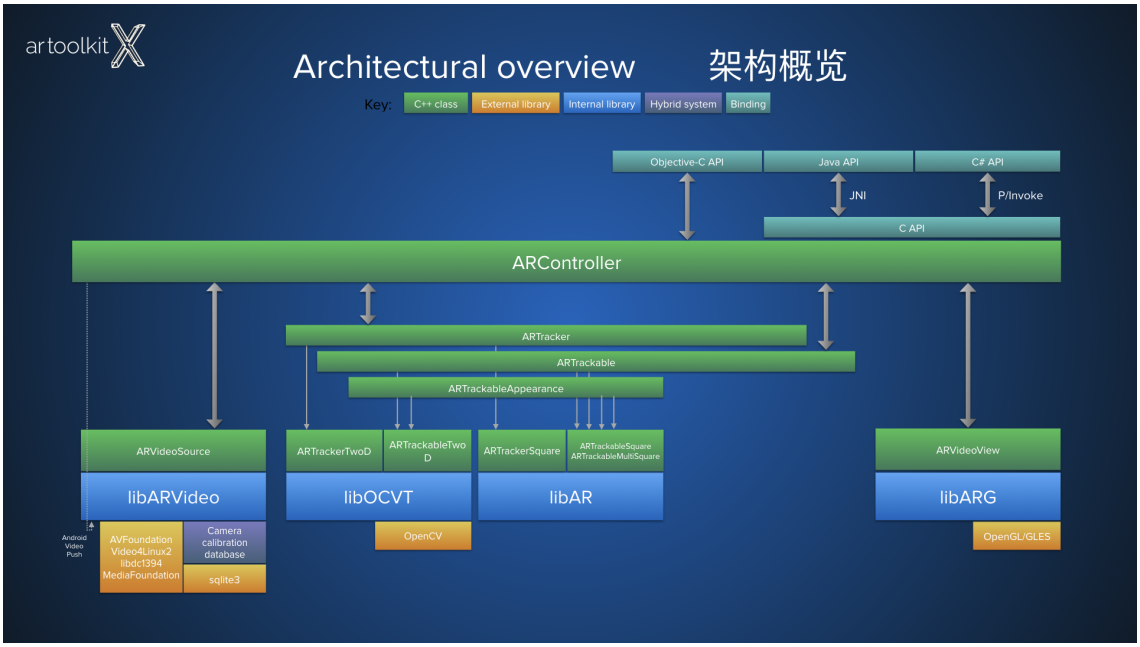

Fig. 2. ARToolkitX architecture

input, it can't be meaningfully offloaded — sending entire camera frames over the network at any reasonable frame rate would be too expensive.

ARTracker An extendable class that defines functions to process camera frame data, adds markers (trackers) to the environment, and loads the database of markers. Two subclasses extend this class: `ARTracker2D` and `ARTrackerSquare`. To offload this to an edge device, we would need to offload the entire video frame contents.

ARTrackable This is a base class for supported trackable types. ARToolkitX implements two trackable types by default: `ARTrackable2D`, `ARTrackableSquare`/ `ARTrackableMultiSquare`.

ARVideoView Draws the output of an `ARVideoSource` to a rendering context. The largest, and only non-trivial method, here is `draw()`, which also cannot be offloaded because this is the openGL rendering of an object.

V. OpenGL

OpenGL for Embedded Systems (OpenGL ES or GLES) is a subset of the OpenGL computer graphics rendering application programming interface for rendering 2D and 3D computer graphics. To display any graphic on the display it needs to go through the OpenGL rendering pipeline.

OpenGL graphics rendering pipeline Rendering Pipeline is the sequence of steps that OpenGL takes when rendering objects, see Figure 3

Vertex array It is a list of vertices that define the boundaries of the primitive. Along with this, one can define other vertex attributes like color, texture coordinates etc. Later this data is sent down and manipulated by the pipeline.

In the pipeline only two components can be modified which are:

Vertex Shader The vertex specification defined above now pass through Vertex Shader. Vertex Shader is a program written in GLSL that manipulate the vertex data. The ultimate goal of vertex shader is to calculate final vertex position of each vertex. Vertex shaders are executed once for every vertex(in case of a triangle it will execute 3-times) that the GPU processes. So if the scene consists of one million vertices, the vertex shader will execute one million times once for each vertex. The main job of a vertex shader is to calculate the final positions of the vertices in the scene. Also vertex shader can pass the data down the pipeline to the Fragment shader to process it further.

Fragment Shader This user-written program in GLSL calculates the color of each fragment that user sees on the screen. The fragment shader runs for each fragment in the geometry. The job of the

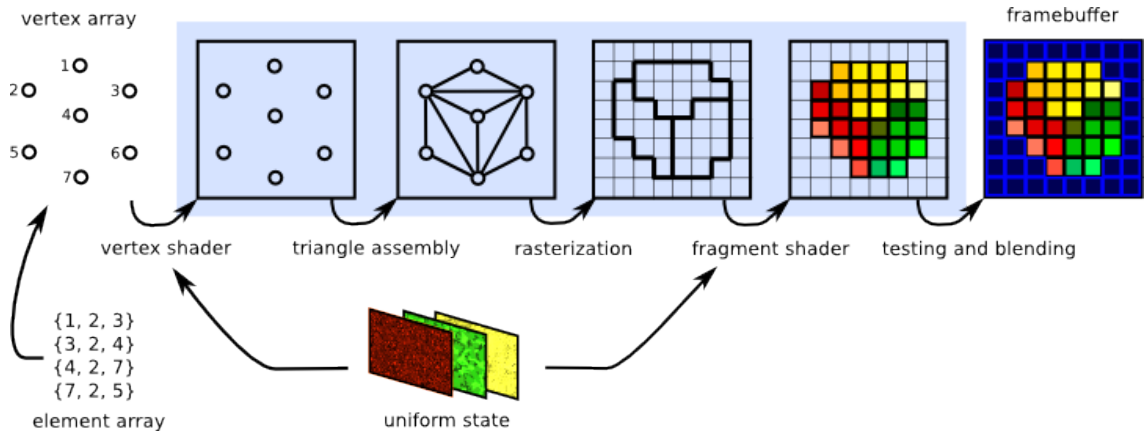

Fig. 3. OpenGL rendering pipeline

fragment shader is to determine the final color for each fragment. We used .mtl file of an object which contains color information for an object.

A. OpenCV to OpenGL

We tried converting rotation and translation vectors in OpenCV to a format that OpenGL can use to render 3D objects. OpenGL requires a model, view, and projection matrix that it uses to multiply with the points of the 3D object to get the projected 2D points of the 3D object. We used the tutorial in [11] to understand the meaning of these matrices. While the model matrix roughly works like the extrinsic matrix and the projection matrix roughly works like the intrinsic matrix, there are some discrepencies in the way OpenGL and OpenCV represent them. While we didn't get this method working, here is what we learned.

B. Differences

There are a few main differences between OpenGL and OpenCV coordinate system representations. They are:

1) The 3D coordinate system for OpenGL is arranged differently than OpenCV. We followed the explanation in [9].
2) The 2D coordinate system for OpenGL has the (0,0) pixel in the *bottom left* corner (as if the entire image plane were the top right quadrant in a graph), whereas the (0,0) pixel in OpenCV is in the *top left* corner and y values increase as you go down (as if it were a matrix with rows and columns).
3) The OpenGL 3D coordinate system normalizes real-world coordinates from [-1, 1]. While some examples online reference that coordinates for X range from -1 (on the far left) to 1 (on the far right), and coordinates for Y range from -1 (on the bottom) to 1 (on the top), we found that, in actuality, the coordinates for X range from 1 (on the far left) to -1 (on the far right).
4) OpenGL matrices are represented a little strangely. See the extrinsic matrix and model matrix in V-C for an example. Even without the sign changes, the order of the items in R and T change.

C. Model Matrix

The extrinsic matrix can be converted into a model matrix as follows:

$$Extrinsic = \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{array}\right] \quad (4)$$

$$Model = \begin{bmatrix} r_{11} & -r_{21} & -r_{31} & 0 \\ -r_{12} & r_{22} & r_{32} & 0 \\ -r_{13} & r_{23} & r_{33} & 0 \\ t_1 & t_2 & -t_3 & 1 \end{bmatrix} \quad (5)$$

The sign changes account for the different coordinate systems between OpenGL and OpenCV. Further, notice that r_{12} is in the (0, 1) spot in the extrinsic matrix, but the (1, 0) spot in the model matrix.

D. Projection Matrix

The intrinsic matrix can be converted into a projection matrix using the formula from [8]:

$$\begin{bmatrix} \frac{f_x}{c_x} & 0 & 0 & 0 \\ 0 & \frac{f_y}{c_y} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -2\frac{f+n}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{6}$$

where f_x, f_y, c_x, and c_y are from the intrinsic matrix, and f and n are the far and near clipping planes, respectively. These can be anything, but a default that many people use for OpenGL is .1 and 1000.

We set the projection matrix directly, but it's also possible to call the OpenGL function `glFrustumf` to compose the projection matrix given different variables (most notably, a frustum and offset). This is the method used in [10], which purportedly generated good results, but our implementation of their method didn't give us any reasonable results.

REFERENCES

[1] https://april.eecs.umich.edu/software/apriltag

[2] Jaewon Choi, HyeonJung Park, Jeongyeup Paek, Rajesh Krishna Balan, and JeongGil Ko. 2019. "LpGL: Low-power Graphics Library for Mobile AR Headsets.", 2019 International Conference on Mobile Systems, Applications, and Services (MobiSys 19).

[3] Q. Liu and T. Han, "DARE: Dynamic Adaptive Mobile Augmented Reality with Edge Computing," 2018 IEEE 26th International Conference on Network Protocols (ICNP),

[4] http://www.artoolkitx.org/

[5] https://github.com/artoolkitx/artoolkitx

[6] https://github.com/ssemenova/CSE622Project

[7] https://github.com/vaibhavchincholkar/CameraFrames

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

[9] https://amytabb.com/ts/2019_06_28/

[10] http://urbanar.blogspot.com/2011/04/from-homography-to-opengl-modelview.html

[11] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/