Augmented Reality with Optical Flow and a Homography Transformation

Walinton Cambronero
College of Computing, Georgia Institute of Technology
wcambronero3 {at} gatech.edu

ABSTRACT
Computer Vision continues to be an important topic of research. The CV research is extensively used in the Augmented Reality (AR) field. A common use-case of AR consists of displaying images or information about real-world's objects on a mobile phone's screen as the camera moves around pointing to different objects. A common technique is to map the camera's internal coordinates to the real world's coordinates where the virtual images are projected, and these coordinates are tracked as the camera moves, to dynamically adjust the projection to match the environment (applying the corresponding rotate, scale and translate transformations).

In this paper, I show how can an arbitrary image be projected onto a camera's plane, and how to track the position of the projected image in the real-world coordinates as the camera rotates around its axis, using only well-known Computer Vision techniques and a Homography for the spatial transformations. The method shown in this paper does not require specially placed markers in the physical world or additional sensors such as GPS or an accelerometer.

1 INTRODUCTION
Computer Vision (CV) is a mature field of study. It has been around for several decades, however research is still very active. For example, the Canny Edge Detector [1] was introduced in 1986, and a paper [2] that studies the applicability of Canny to high-resolution video-streams using modern software platforms such as Hadoop, has been published this year.

In this paper, a method that uses CV and other image processing techniques is developed to show a common AR use-case. Using open-source tools such as OpenCV and NumPy for the image processing, I write a program in Python that takes any pre-recorded video of an open room and projects an arbitrary image onto some carefully chosen coordinates in the real-world's plane. It uses a Homography transformation and various feature matching techniques to perform the spatial transformations (rotate, scale and translate) that allows to project the image as it would appear in the real world as the camera moves around its axis. It also tracks the virtual image coordinates even when those coordinates are no longer within the camera's field of view.

This paper is organized as follows. Section 2 covers the hardware and software specifications of the machine where this program is developed. Section 3 makes a small introduction to feature selection and matching. Section 4 proposes an algorithm to choose the virtual image coordinates. Section 5 analyzes methods used to track the virtual image coordinates as the camera's field of view changes. Section 6 discusses the program's performance. Finally, Section 7 makes a brief comparison against state-of-the-art Augmented Reality.

2 SYSTEM SPECS
The machine and software used is:

- **Software**: Virtual Box (v5.2.12) Ubuntu (v18.04) VM, Python (v2.7.13), OpenCV (v2.4.13) and NumPy (v1.15.1). Docker (v18.06) container.

- **Hardware**: 8GB RAM, 2 vCPU (Intel® Core™ i7-6600U CPU @ 2.60GHz).

- **Camera:** Samsung Galaxy A7, 16 MP, f/1.9, 27mm (wide), AF. Recording at VGA (640x480) and HD (1280x720)

3 MECHANISM TO FIND FEATURES
A feature (a.k.a corner) is a pixel for which significant change in intensity (or brightness) is observed in both dimensions. Finding good features is one of the most important steps for this program as the features serve two key purposes:

1. Help identify an area to project the image
2. Matching pixels from one frame to another

The Computer Vision community offers a variety of methods which can be used to find features on an image. In [3], several methods are discussed, each with its own set of cons/pros. For this paper, the method must be invariant to both scale and rotation as the scenes used are subject to those transforms. All features discussed in [3] meet this requirement, however some perform better than others. Oriented FAST and Rotated BRIEF (ORB) [4] is not the most scale or rotation invariant algorithm, however (a) it does perform well on those transformations and (b) is "the most efficient feature-detector-descriptor with least computational cost" [3]. This paper is developed on a VM that offers strong but virtualized processing power, hence, the method with the least computational cost is desired. For reasons given in (a) and (b), ORB has been selected for feature finding.

4 HOW TO DECIDE WHERE TO PROJECT
In a real-world application, the location in the physical plane where the Augmented Reality image is to be projected would be chosen by human-interaction or some algorithm. For this paper, the latter has been chosen. An algorithm that automatically chooses a location from the recorded frames is developed. The algorithm tries to find a suitable space to project the image based on the features extracted from the recorded frames. Let us define "good space" as a rectangular area that fits inside a group of features (a.k.a corners).

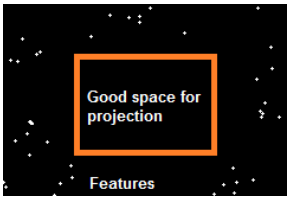


***Figure 1**. A good space is within a group of features.*

A feature is used as a delimiter because objects found in a scene will typically have a feature associated. A feature-less region of the scene is an indication of a, possibly, planar and free of objects space where an image can be projected.

However, if only features (corners) and not edges were to be considered, areas that cross edge boundaries might be selected, as seen in **Figure 2**.

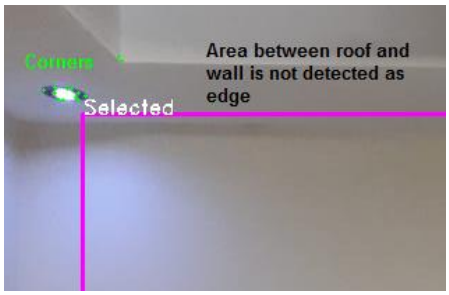


***Figure 2**. Area is selected over an undetected edge.*

A "good space" is then redefined as a rectangular area that fits inside a group of features and edges. Edges are found using the Canny [1] method. The resulting selected area is now in a space where no edges or corners exist.

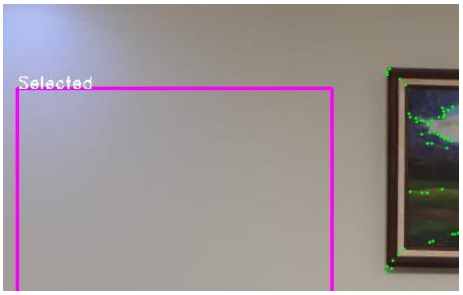


***Figure 3.** A space that considers edges and corners.*

Selecting a good space

The algorithm is designed to avoid selecting the scene corners (physical corners in image). For this purpose, a circular window is slided across the scene. The circle's perimeter is always within the image perimeter, i.e. the sliding circle starts and ends at the image edges. This causes the pixels at each image corner to be ignored. In **Figure 4**, the image corners are the yellowish pixels.

***Figure 4.** Sliding circle ignores corners.*

Only pixels inside the circle's perimeter (blue pixels in **Figure 4**) are considered. The algorithm is looking for the pixel that is furthest away from any corner or edge that is also found inside the circle. An Euclidean distance-transform function is used for this purpose. This function calculates the distance to the closest zero-pixel for all non-zero pixels. A simple method is used:

1. Initialize a binary mask with all 0s
2. Every pixel inside the circle is set to 1
3. Every feature or edge pixel is set to 0
4. Apply the distance transform to the mask

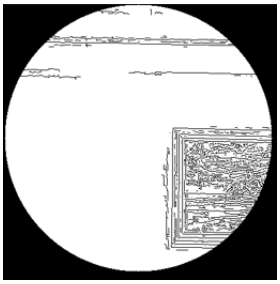


***Figure 5.** Binary mask. Features and edges are zero-pixels.*

After applying the transform to the binary mask, every non-zero pixel has its distance calculated to the nearest feature. The pixel with the maximum value is the most distant to all features contained within that region. This is represented as the brightest pixel in **Figure 6.**

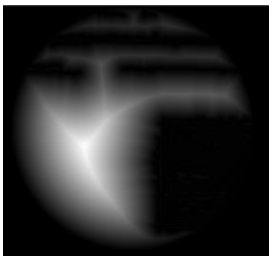


***Figure 6**. Distance transform function applied to the mask.*

The circle's radius also controls the maximum size of the space that can be found, as the maximun possible distance from the circle's center to a zero-pixel is the circle's perimeter. The algorithm used to find the best space for the projection is:

Initialization

1. Find edges and features
2. Set circle position = radius, radius
3. Set max distance = 0
4. Set selected point = circle position

While circle inside image

1. Create binary mask
2. Apply distance transform
3. Update **max distance** and **selected point**
4. Slide circle (with a fixed step)

At the end of the loop, the area where the image is to be projected is represented as the rectangle that fits inside a circle with its center equals to the **selected point** and radius equal to the **max distance**. The rectangle's base and height are calculated using a fixed ratio which works well for the projected image. **Figure 7** shows this graphically. The colored bounding box is the current best space, the ghostly circle is the distance transform, the inner circle has radius equal to the distance from the brightest pixel to the closest feature/edge. The bounding box inside that circle is a potential space.

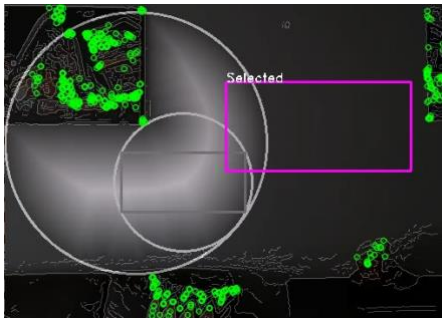


Figure 7. *Sliding circle with distance transform.*

5 TRACKING THE IMAGE

5.1 Objective

The projected image must remain in a fixed position in real-world coordinates as the camera moves. A few approaches were taken before finding one that works well. A description is given for each.

5.2 Image origin coordinates

The "image origin coordinates" are the image coordinates when placing its (0, 0) pixel at the origin of our coordinate system. This term is used in the rest of the paper. The coordinates are: $(0,0), (0, width), (height, 0), (width, height)$

5.3 Feature matching

In order to compute the Homography between two frames, a list of corresponding features must be found in both frames (matching features). The ORB [4] detector is used to find features on each frame, separately. For each feature, a scale and rotation invariant descriptor is computed. One method to match features is the Brute Force matcher from OpenCV. It returns the list of corresponding features that it determined to be present in both frames as well as an error metric (a.k.a distance) assigned to each one. The matches with the lowest distance are the most accurate ones. Matching features are sorted by distance (ascending) and then the first N features are used. The number of features (N) is not fixed and can be estimated by experimentation. N=50 produced the best results for the scenes used in this paper. This method was tested in sections **§5.5**, **§5.6** and **§5.8**.

5.4 Computing the Homography

A list of matching features is passed to the OpenCV findHomography function. The function supports several methods. A method is considered robust if it can exclude outliers from the sample. RANSAC [5] is an example of a robust method. This is the method used on this paper.

5.5 First approach: naïve warp

The first approach uses a naïve technique in which the image is inserted into the first frame, and then warps the entire first frame over the subsequent frames. The image is projected in the correct space (along with the rest of the first frame).

Initialization:

1. Insert the image image into frame **0**
2. Find and remember features from frame **0**

Enter loop: For each frame, start **t=1**

3. Find features in frame **t**
4. Match features between frame **t** and **0**
5. Compute the Homography
6. Warp the frame **0** into frame **t**

Results:

Since the entire first frame is continuously warped from frame to frame, and the lighting between frames is not the same, the warped portion of the frame shows the brightness difference between the first and subsequent frames. This manifests in **Figure 8** as a shadow-like plane.

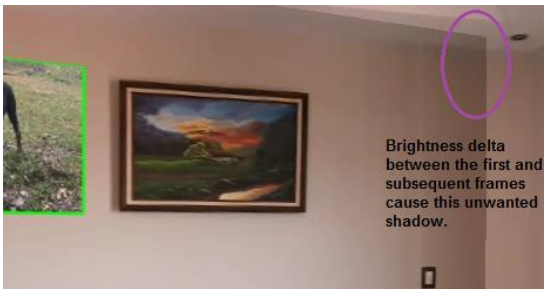


Figure 8. *Unwanted shadow due to brightness delta between first and subsequent frames.*

5.6 Second approach: forward warping

Consists in keeping track of the image coordinates. By keeping track of where the image would be in subsequent frames, it is possible to warp the image from the first frame to the current frame as follows. At frame **t**, the image coordinates $\boldsymbol{A^t}$ are known. Find the Homography $\boldsymbol{H_t^{t+1}}$ between frames **t** and **t+1**, and use $\boldsymbol{H_t^{t+1}}$ to forward warp $\boldsymbol{A^t}$ to $\boldsymbol{A^{t+1}}$. With the new image coordinates $\boldsymbol{A^{t+1}}$, it is possible to find the Homography $\boldsymbol{H_0^{t+1}}$ between the image $\boldsymbol{A^0}$ origin coordinates and $\boldsymbol{A^{t+1}}$. Finally, using $\boldsymbol{H_0^{t+1}}$, warp the image onto the current frame.

Initialization:

1. Remember the image origin coordinates $\boldsymbol{A^0}$

Enter loop: For each frame

1. Remember the image coordinates at time **t**: $\boldsymbol{A^t}$
2. Find matching features at frames **t** and **t+1**.
3. Compute the Homography, $\boldsymbol{H_t^{t+1}}$
4. Forward warp the image coordinates $\boldsymbol{A^t}$ with $\boldsymbol{H_t^{t+1}}$. Gives new coordinates $\boldsymbol{A^{t+1}}$
5. Compute the Homography, $\boldsymbol{H_0^{t+1}}$, between the image origin coordinates $\boldsymbol{A^0}$ and $\boldsymbol{A^{t+1}}$
6. Project the image into the frame with $\boldsymbol{H_0^{t+1}}$.

Results:

Worked well for the first few frames. However, the projected image quickly becomes significantly miss-aligned. With forward warping it is expected that the estimated location of the coordinates is slightly wrong, as the projected pixel may land in between pixels in the destination image. This manifests in **Figure 9** as a progressive miss-alignment of the projected image.

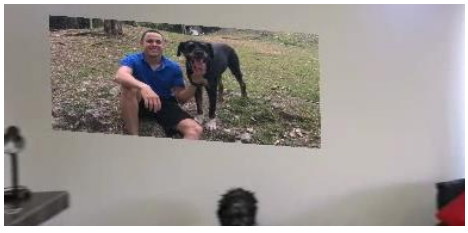


Figure 9. *Forward warping progressively deforms the projected image.*

5.7 Third approach: optical flow

The approach is similar to the Second approach, but instead of estimating the image coordinates with forward-warping,

Optical Flow with the Lukas-Kanade Pyramidal method is used. The coordinates at time **t+1** are estimated as the coordinates at time **t** plus the average displacement of the found features.

Initialization:

1. Remember the image origin coordinates $\boldsymbol{A^0}$

Enter loop: For each new frame

1. Remember the image coordinates at time t $\boldsymbol{A^t}$
2. Find features coordinates $\boldsymbol{F^t}$
3. Compute Optical Flow for $\boldsymbol{F^t}$. This gives the features coordinates at time **t+1**: $\boldsymbol{F^{t+1}}$
4. Calculate the displacement vector $\boldsymbol{D^t}$ by subtracting the $\boldsymbol{F^{t+1}}$ from $\boldsymbol{F^t}$
7. Calculate the average displacements from all features in $\boldsymbol{D^t}$ by taking the average on the X and Y dimensions: $\overline{\boldsymbol{D^t}}$
8. Estimate the image location $\boldsymbol{A^{t+1}}$ by adding $\overline{\boldsymbol{D^t}}$ to $\boldsymbol{A^t}$
9. Compute the Homography $\boldsymbol{H_0^{t+1}}$ between the image origin coordinates $\boldsymbol{A^0}$ and $\boldsymbol{A^{t+1}}$
10. Project the image into the frame with $\boldsymbol{H_0^{t+1}}$

Results:

Worked well for horizontal and vertical camera movement. The projected image maintains its position as the camera moves, as seen in **Figure 9**. However, all coordinates must be moving in the same direction because the motion is estimated by taking average of the displacement vector.

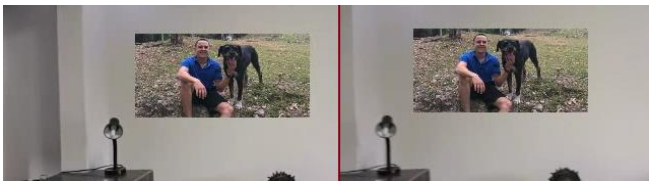


Figure 9. *Image location at different frames.*

When the camera is rotating, some features move in opposite directions (e.g. top left corner features move to the right, while bottom right features move to the left). Because of this, when taking the average, some of the displacements would cancel each other. The projected image does not rotate properly as seen in **Figure 10**.

Figure 10*. Image location at different frames after rotating camera. Projected image does not rotate.*

5.8 Fourth approach: continuous Homography update

The method described in [6] is a marker-less tracking system that works on scenes that contain one or more planes. It tracks camera pose by calculating the Homography between consecutive frames. While no camera calibration is done on this paper, an important insight is extracted from the **equation 3.3** in [6].

$$H_0^i = H_{i-1}^i \, H_{i-2}^{i-1} \ldots H_0^1$$

Equation 1. *Homography from the first to the the ith frame*

In **Equation 1,** $\mathbf{H_0^i}$ is the Homography between an initial frame **0** and a subsequent frame **i**. In sections **§5.6** and **§5.7**, effort is put into tracking the image coordinates as they move in order to estimate $\mathbf{H_0^i}$. **Equation 1** shows that this Homography can be calculated sequentially as frames are processed, without the need to track the image coordinates. This approach consists in sequentially calculating $\mathbf{H_0^i}$ by finding the Homography, $\boldsymbol{H_{t-1}^t}$, between consecutive frames and updating $\mathbf{H_0^i}$ as per **Equation 1**. The image is projected onto the every frame with $\mathbf{H_0^i}$.

Initialization:

1. Compute the Homography, $\boldsymbol{H}$, between the image origin coordinates and the space where the image is to be initially projected.
2. Set $\boldsymbol{H_0^t}$= $\boldsymbol{H}$, where $\boldsymbol{H_0^t}$ *is* $\boldsymbol{H_0^i}$ in **Equation 1.**

Enter loop: For every frame, start **t=1**,

1. Project the image onto frame **t-1** using $\boldsymbol{H_0^t}$
2. Find matching features at frames **t** and **t-1**.
3. Compute the Homography $\boldsymbol{H_{t-1}^t}$
4. Update $\boldsymbol{H_0^t}$ per **Equation 1**, $\boldsymbol{H_0^t = H_{t-1}^t H_0^t}$

Results:

The projected image follows the camera as it moves, including rotation. This solves the rotation problem found in section **§5.7**. See **Figure 11**.

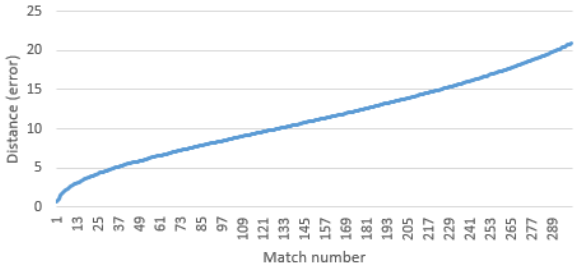
Figure 11. *Rotation at different degrees*

However, $\boldsymbol{H_0^t}$ accumulates drift after each frame, causing the projected image to get miss-aligned after a while, as seen in **Figure 12**.

Figure 12. *Incorrect alignment due to accumulated drift*

One reason why $\boldsymbol{H_0^t}$ accumulates drift is that the frame-to-frame Homography $\boldsymbol{H_{t-1}^t}$ is not accurate. $\boldsymbol{H_{t-1}^t}$ is computed by giving the best N features that were matched between frame **t** and **t-1**, as described in **§5.3**. This limits the number of features that are used for the Homography. In general, more features produce more accurate results. An attempt was made by increasing the number of matched features, from N=50 to larger numbers, but the results did not improve. One reason why this can happen is that the features itself aren't accurate enough. **Graph 1** shows the relation between the error calculated for the feature (a.k.a distance) and the feature number (after sorting ascending by distance to extract the first N matches). N=300, and the data is averaged over 250

samples. It can be observed that the error increases almost-linearly as more ORB features are matched.



***Graph 1**. Error per match number with ORB and BFMatcher*

As per [7], several methods exist for feature matching **1. Descriptor matching**, that allows for large displacements but have limited accuracy and **2. Energy minimization**, that provides accurate results but fail as displacements get too large (e.g. Dense Optical Flow). The method used on this approach is a descriptor based one. A method that provides higher accuracy is explored in section **§5.9**.

5.9 Fifth approach: Fourth approach with optical flow

In [10], Optical Flow is used for feature matching when a number of good matches is known. This idea is explored here. Let us use the same algorithm as section **§5.8**, except replacing ORB descriptor matching with Optical Flow. OpenCV's **calcOpticalFlowPyrLK** function is provided with the list of features at frame **t-1**, and it returns the list of corresponding matching points if the Optical Flow was found in frame **t**. This list of corresponding points is used to compute $\boldsymbol{H_{t-1}^{t}}$. ORB is still used on each frame to detect the set of features (at frame **t-1**), and Optical Flow provides the matches at **t**.

Same algorithm in section **§5.8** except step 2 is redefined:

2.1 Find ORB features at frame **t-1:** $\boldsymbol{F^{t-1}}$
2.2 Compute Optical Flow for $\boldsymbol{F^{t-1}}$using frame **t-1** and frame **t**, this gives $\boldsymbol{F^{t}}$
2.3 Discard any feature in $\boldsymbol{F^{t-1}}$ for which the Optical Flow was not found in frame **t**
2.4 The matching features are $\boldsymbol{F^{t-1}}$, $\boldsymbol{F^{t}}$

Results:

430 matching features were obtained in average using this technique (ORB set to 500 max). Only very accurate results are expected from the Optical Flow; hence a large number of good features are used to compute $\boldsymbol{H_{t-1}^{t}}$. The $\boldsymbol{H_{0}^{t}}$ accumulating drift problem observed in **§5.8** is resolved by this. The image is properly projected onto every frame **t** with $\boldsymbol{H_{0}^{t}}$. It can handle rotation and scale variance as well. **Figure 13** compares the results in **§5.8** where the accumulating drift problem was observed (right) to the results in this section (left).

***Figure 13.** Optical Flow vs Descriptor Matching*

This is the final implementation that was selected for this paper. A demostration video can be seen in [11] along with other results in [12] through [19].

6 PERFORMANCE ANALYSIS

This program was developed using virtualized hardware and using Python as its programming language. The program currently reads the entire video recording from disk into memory and then writes the processed images back to disk. With these limitations, the Frames per second (FPS) achieved when processing an HD (1280x720) and VGA (640x480) input video is:

- **20 FPS** at HD
- **40 FPS** at VGA

The disk I/O operations consumed ~20% of the processing time. If in-memory processing only (a live video feed) is used in the same environment, at most a 20% performance increase is expected.

Optical Flow and ORB, responsible for feature matching, together consumed over 30% CPU time. The Optical Flow configuration parameters have been tuned to produce good results without consuming excessive CPU. Further tuning these values produced unwanted behavior. Another way to reduce Optical Flow processing time is to pass it less features. Since there is no extreme scale variance in the scenes presented on this paper, ORB can be optimized by changing its scale factor and number of pyramid levels. The scale factor defines the rate at which each next level is reduced. It has been found that by setting the scale factor to 2 (i.e., each next level has 4x less pixels than the previous) and reducing the number of levels to 4 (from 8), there was a 4 FPS improvement.

7 STATE OF ART COMPARISON

AR has become "commonly available to the general public, due to technological advances in mobile computing and sensor integration" [9]. Two potential areas of improvement for this paper are mentioned on this simple statement: Sensor Integration and Mobile Computing.

While no testing has been done on a mobile device, it is expected that this code as-is will perform poorly, given the high demand of resources required (See section **§2**), and that lower processing power is typically found on mobile devices. In [10], a marker-less AR system is capable of achieving near 30 FPS on a mobile device. Additionally in the mobile world, state of the art AR incorporates "sensor fusion techniques, such as using GPS or IMU" , "IMU sensor based tracking uses magnetometer, gyroscope, and accelerometer" [8], especially in smartphones that "contain an increasingly sophisticated array of sensors", "enabling AR to become more personally meaningful and situated" [9]. These kinds of sensors were available for this paper as part of the recording device used, but this was left out of scope. "A complete AR system should include three main elements, i.e., tracking, registration, and visualization" where tracking is defined as "dynamic sensing and measuring of the spatial properties", one way this is accomplished is "vision-based tracking [that] uses image processing to calculate the camera pose" [8]. The tracking problem as described above has not been solved for this paper. The algorithm does not attempt to calculate camera pose. It depends on the camera rotating around its own axis in order to properly compute the Homography. A demo video of this code applied to a camera not moving around its own axis is provided in Demo [14].

Other areas on which the state of the art is ahead, include but are not limited to real world plane detection, projection of 3D objects and, user input and control (e.g. changing the image size or position).

8 LIMITATIONS

The following are some known limitations:

- Does not detect real planes in physical world. For good results, the camera must be facing towards a physical plane like a wall or floor. See [12] for an example of how things go wrong when not facing a physical plane.

- No camera calibration. The camera must rotate around its own axis. See [14] for an example of a camera not rotating against its own axis.

- Optical Flow is sensible to large pixel displacements and/or sudden brightness changes. See [13] for an example of a fast-moving camera.

9 CONCLUSIONS

Using only open-source tools, a Homography for the spatial transformations and a couple of well-known Computer Vision (CV) techniques, the basic Augmented Reality (AR) use-case of displaying additional information on top of the real-world objects, has been implemented and discussed in this paper using input from fairly recent CV publications.
The Homography transformation that is used to transform the virtual image from its origin coordinates to any arbitrary camera's coordinates is updated after processing each subsequent camera frame using the method described in [6]. The basic intuition is that only the origin and final coordinates are needed to make the spatial transformations (rotate, scale, translate) that project the virtual image to the real-world's coordinates as the camera's field of view changes. The well-known CV technique known as "Optical Flow" discussed in [10] is used for the feature-matching between frames.

Even though Computer Vision is decades-old research, the main contributions used in this paper came from papers written fairly recently, e.g. [10] from 2013, [3] from 2018, and [7] from 2009.

ACKNOWLEDGEMENTS

Special thanks to Jorge Sauma (Hewlett Packard Enterprise) for reviewing this paper and providing invaluable feedback.

REFERENCES

[1] John Canny (1986). A Computational Approach to Edge Detection. IEEE Transactions On Pattern Analysis And Machine Intelligence.

[2] Iqbal, B., Iqbal, W., Khan, N. et al. Canny edge detection and Hough transform for high resolution video streams using Hadoop and Spark. Cluster Comput 23, 397–408 (2020). https://doi.org/10.1007/s10586-019-02929-x

[3] Shaharyar Ahmed, K. T. (2018). A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK. International Conference on Computing, Mathematics and Engineering Technologies (iCoMET).

[4] Rublee, Ethan & Rabaud, Vincent & Konolige, Kurt & Bradski, Gary. (2011). ORB: an efficient alternative to SIFT or SURF. Proceedings of the IEEE International Conference on Computer Vision.

[5] Martin A. Fischler & Robert C. Bolles (June 1981). "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography

[6] Gilles Simon, A. W. (2000). Markerless Tracking using Planar Structures in the Scene. Proceedings IEEE and ACM International Symposium on Augmented Reality.

[7] Thomas Brox, C. B. (2009). Large Displacement Optical Flow. Proc. IEEE International Conference on Computer Vision and Pattern Recognition (CVPR).

[8] Wenkai Li, A. Y. (2017). A State-of-the-Art Review of Augmented Reality in Engineering Analysis and Simulation. Multimodal Technologies and Interaction.

[9] Elizabeth FitzGerald, A. A. (2012). Augmented reality and mobile learning: the state of the art. 11th World Conference on Mobile and Contextual Learning.

[10] Ufkes, A., & Fiala, M. (2013). A markerless augmented reality system for mobile devices. Proceedings of the International Conference on Computer and Robot Vision.

DEMOS

[11] Full demo

dropbox.com/s/6wn56v9ydrkwcwc/final.mp4

[12] Known limitation 1

dropbox.com/s/ha532n4v0d0ydqg/plane.avi

[13] Known limitation 2

dropbox.com/s/0d4znh7nzpxj58l/fast.avi

[14] Known limitation 3

dropbox.com/s/c0q1lgsh6ktmjj7/axis.avi

[15] Sliding circle

dropbox.com/s/qgwarqgbh91geyr/sliding.avi

[16] Bedroom scene

dropbox.com/s/n3xhnye69k7n1px/bed.avi

[17] Ceiling scene

dropbox.com/s/90lkc89lfb5chgc/ceiling.avi

[18] Living room scene

dropbox.com/s/dpmtd4rummqcibj/living.avi

[19] Airport office scene

dropbox.com/s/nqb21oghze4jorv/office.avi