Project INF573 : Augmented Reality

Maria Scarlleth Gomes De Castro, Capucine Leroux

December 2020

Contents

1 Introduction

The problematic of our project is to add 3D virtual objects into a real scene that has been filmed. It is part of a larger subject called Augmented Reality (AR), which is a very active field of research. It is widely used in the cinema industry for special effects. In many other fields, we can cite recent projects developing AR :
- for health, the OpenSight Augmented Reality System was created to project the inside of a body before surgery to save time [1]
- for culture, the city of Amiens developed an application to see the cathedral from your smartphone and computer in 3D from your bedroom [2]
- for entertainment, Google enables you to add a 3D animal of your choice in your room with your camera [3]

That is why we chose this project, since AR is a big part of computer vision future technologies, and that it finds a huge variety of applications.

2 The theory

Our project relies on several computer vision concepts. Because we tried different approaches to reach our goal, we will introduce some theories that might not be included in our final implementation.

2.1 Feature detection and matching

The first step to add a 3D object in a video is to track some key points in our video frames and match them from frame to frame. It can either use the help of a known pattern image that is placed in the video at the spot where we want to add the 3D objects, or without any marker, just tracking and making correspondences between the key points of each frame with one another.

In order to do that, there are three main steps :

- **Detection :** We first want to identify interest points, it can be a pattern, corners, edges or blobs.

- **Description :** Once the key points have been detected, we need to extract important features that can be compared from point to point.

- **Matching :** Last, we determine correspondence between descriptors in two views. After finding the 2 nearest neighbors of a descriptor, we can use Lowe's ratio to keep only the "good" matches, therefore, if the first neighbor of a descriptor is way closest to it than the second neighbor, it is considered as a "good" match. Then, the matching can get even more precise by calculating the homography H and the corresponding mask M. The mask will tell which matched points are outliers and which ones are inliers, and the homography gives the affine transformation that was used to get from the first view key points to the second view ones.

$$\begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix} = H \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}$$

2.2 Structure from motion

Once we have detected key points in each frame and tracked their correspondence either with a common known pattern or from frame to frame, we need to find the cameras parameters in order to be able to project new points (the ones from our 3D object) onto the images.

The pinhole camera model : (See figure 1)

Each view (or in our case each frame of the video) is represented by a camera. This camera has two types of parameters :
- Intrinsic parameters
- Extrinsic parameters

The intrinsic parameters are the same during the whole video, because it was taken from the same physical camera. They include the focal length f, and the principal point (c_x, c_y). Together they form the camera intrinsic matrix :

$$K = \begin{pmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

They can sometimes include distortion coefficients as well.

The extrinsic parameters change from view to view. Each frame is represented by a rotation R and a translation T.

To project a 3D point from the real world onto an image taken from a pinhole camera K, R and T, the transformation is :

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = P \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = K \left(\begin{array}{c|c} R & T \end{array} \right) \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = K \left(\begin{array}{ccc|c} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{array} \right) \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Calibration and bundle adjustment :

With our tracked points, we want to find the right projection matrix P for each frame, which will enable us to transform 3D objects in 2D points in the image. To find the projection matrix, we first need to know the camera intrinsic parameters K and the distortion coefficients. In order to do that, we do a camera calibration, taking pictures of an image with a known geometry such as a chessboard or a circle. With a few pictures from different angles and the 3D coordinates of the image, we can deduce the intrinsic parameters.

Once we know the intrinsic matrix K, we can now solve the projection matrix. If we have n cameras/images $(K, R, T) = (K, R_i, T_i)_{1 \leq i \leq n}$ and m points $(p_i)_{1 \leq i \leq m} = \begin{pmatrix} x_{ij} \\ y_{ij} \end{pmatrix}_{1 \leq i \leq m, 1 \leq j \leq n}$ tracked in each image, we will minimize the reprojection error :

$$g(R, T) = \sum_{i=1}^{m} \sum_{j=1}^{n} ||P(p_i, R_j, T_j) - \begin{pmatrix} x_{ij} \\ y_{ij} \end{pmatrix}||^2$$

Minimizing this reprojection error function to get the optimal set of projections is called the bundle adjustement.

3 Our implementation

In all of our implementation, we only used the opencv package and no other library.

3.1 First markerless model

The first idea we had was :

- Find keypoints in all the frames. We used the FAST detector because of its speed compared to other methods, which is important when processing a video with many frames.

- Match the keypoints from consecutive pairs of frames. After using a SURF detector, we matched the point with a flannbased knn-matcher, then proceeded to a Lowe's ratio test and refined the matching with an homography mask.

- Track the points that were in all of them building a correspondence graph. We did this manually, implementing a function able to use all the precedent matchings and building a vector of all the keypoints that were matched in every frame, and all of their 2D coordinates.

- Proceed to a camera calibration if needed to get the intrinsic matrix K. We used the chessboard corner detection and the camera calibration already implemented in opencv. The next step function "reconstruct" can auto-calibrate the camera but it gave better results to do a first calibration rather than randomly initialize K.

- Solve the camera extrinsic parameters of each frame camera to get the projection P. The function "reconstruct" of the cv::sfm package solves this with a set of 2D tracked points and an initialized camera matrix K. We decided later on to use another function instead called "solvepnp". It also calculates R and T but does not reconstruct the 3D coordinates of the tracked points, which we did not need.

- Initialize the position of our 3D shape (a simple 3D cube) on the first frame to get its real-world 3D coordinates by doing a reverse projection

- Project the 3D model on the other frames, knowing its 3D coordinates and the projection matrix. We easily get the projection matrices P out of the K, R and T parameters calculated by "reconstruct", there is even a opencv function doing it automatically.

The problem we encountered with this method is that we had trouble finding a good balance to get conclusive results. We could not find points that were detected in every single frame of the video, so operating a correspondence graph always ended by an empty vector in about 5 to 8 frames. We then decided to match the frames 4 by 4 to have a sufficient number of tracked points across 4 consecutive frames, then operated the bundle adjustment 4 by 4 to get all the projection matrices. We had enough tracked points to solve P but because we only did it on a small number of very similar frames, the projections we found were still a bad approximation. After this first try, we decided to simplify our goal, and use markers on our video to help us get conclusive results.

Note: Initially, we also thought about using OpenGL integrated with OpenCV so that we could render more complex 3D shapes in our augmented reality program. The idea seemed simple since there are even native methods of OpenCV dedicated specifically to this integration. But in practice, we had a lot of problems about library installations and OpenGL support in our project. We even tried to rebuild OpenCV with the proper support to OpenGL and all the extensions that seemed necessary, but in the end we didn't

manage to make it work. So we decided to use a simpler 3D shape which we could draw ourselves using only OpenCV methods, like a cube.

3.2 ArUco markers model

So in this second model we decided to make use of an ArUco marker (see figure 2 for exemples). An ArUco marker is a synthetic square marker composed by a wide black border and an inner binary matrix which determines its identifier. For this we made large use of the aruco OpenCV module which has a lot of methods specialized for this kind of marker. The aruco module of OpenCV is based on the ArUco library, which was developed by Rafael Muñoz and Sergio Garrido [4].

The great advantages of this kind of marker is that: first, it is easier to identify it in the image due to its black borders that are in general very distinguishable; also, the inner part of the marker that is composed of black and white pixels makes it full of good key points, that is features that are easily identified in the image and that caracterize each marker uniquely. This kind of marker is then robust to different conditions of color, light and even to partial occlusion. They can also have different sizes, being able to adapt to smaller or larger scenes.

Our algorithm then consisted in the following steps:

- First we choose an ArUco marker to use in our model and we pose this marker in the scene.

- We define the 3D coordinates of our marker's corners in the world coordinate system, for instance (0,1,0), (1,1,0), (1,0,0), (0,0,0). We take advantage of the fact that that the marker is plane and squared to put it in some place that it's easier for us to imagine, this makes bugs more recognizable.

- We also define the 3D coordinates of the cube, using the marker position as the base for the cube.

- Then, for each one of the first 10 frames of the video where the program is able to identify the marker in the scene, we save the image coordinates of the marker's corners. So, for instance, if the first frame when the model can localize the marker is the 50th, it will wait until then to start. Using these first 10 correspondences we can make a quick calibration of the camera and use this calibration for the following frames.

After this calibration, the rest of the steps are realized iteratively for each new frame:

- We identify again the image coordinates of the marker's corners, then we use the correspondence between the 3D coordinates and the image coordinates and the camera matrix that we obtained with the calibration to calculate the rotation and the translation of the camera.

- Once we have the camera matrix, the rotation and the translation of the camera, we can reconstruct the projection matrix and apply it to the 3D coordinates of the cube that we defined before.

- Finally we use the image coordinates of the 8 points of the cube to draw the cube edges in the final image.

With this approach, we were able to get quite accurate results, but with some disadvantages as we will better explain in the section 4. For the generation and detection of the ArUco marker, we took inspiration of the code referenced in [5].

3.3 Simplified final markerless model

After this successful model, we tried to adapt this model to achieve our first goal : initialize the position of the cube at the beginning of the video without any marker, and being able to automatically pose the cube across the complete video after that. Adapting the marker model gave us a more simple implementation than the first try and way better results.

The idea is simple : since we know how to pose the cube once we have the 2D coordinates of the base, we only need to find the 4 points of the cube base across the video, then we will be able to project the cube on every frame. Hence this new steps :

- Initialize 4 points at the beginning of the video where we want to put the base of the cube. We just show the first frame and recollect the position of the 4 clicks of the user.

- For each new frame, match key points with the precedent frame with the same method used in the first model (Section 3.1), get the homography H between the two views, and use this transformation on our 4 anchor points.

- Once we have the position of the 4 anchor points, we project the cube with the same method described in the ArUco markers model (Section 3.2). To get better results, we proceeded to the same initial camera calibration as described in Section 3.1.

With this method, we get conclusive results, even though it is less precise than with markers.

4 Results

ArUco markers :

For the ArUco markers model, the first thing we could observe is that the pipeline is really fast for each frame. This is a good advantage because it allows us to use it with high quality videos and not only with prerecorded videos but also in live mode, using the webcam of the computer for instance.

Another good point of this model is that by doing the calibration of the camera with the first frames of the video itself containing the marker, we can apply this model to any video despite of not knowing which kind of camera was used to make it and its intrinsic parameters. There is though two little problems with this approach.

The first one is that we create a hyper parameter that we have to try to optimise to improve our model, that is the number of frames to use in the calibration. In our case, we observed that 10 was the number that gave the best results. Initially we thought that the bigger the number of frames the better, but for some reason with this calibration it got actually worse, so we maintained this number as 10.

The second problem is that as the calibration is done with these 10 first frames, the quality of this calibration is not always guaranteed, because it is highly dependent of how the marker appears in these 10 frames. Because of this, sometimes we had to rerun the program a few times until we obtain a good calibration. This problem becomes more important in the case of a prerecorded video, because in this context make a new try means to make a new record, which is not always possible depending on the situation. And in our case, calibration is really important since a bad camera matrix can lead to errors such as distortion in the dimensions of the cube and inverted rotations.

The main advantage of using a marker is that they are made to the purpose of being easily identified, so this makes its detection really precise and this precision influences of course in the final result. Also, this detection is made independently for each frame, so the errors made in the detection of the marker's corners in one frame will not compromise the quality of the detection for the other frames.

But the main disadvantage of using markers is that it makes the model less general exactly because it is completely dependent of a marker's presence. So we are constrained to have one marker somehow present in the scene where we want to produce the augmented reality. We will need for instance to print the marker or, as we've done, to show the marker on a smartphone screen, or some similar idea.

To improve this model, we think we could try to make some change in the camera calibration method in order to get a better guarantee with relation to its quality, so that we would be sure that the program will always work well in its first try.

The markerless model :

The markerless option is a lot slower. It takes between 0.1 and 0.3 seconds to do the matching and the projection for each frame. That is why we tried to use only 1 frame out of 4, or 1 frame out of 2, to save some time by processing less frames. Because we only filmed videos with our phone cameras, the original resolution was already low (30fps, around 300 frames for a short video of 10 seconds). So we could not process less than half the frames to get a correct final result.

The power and the weakness of this method is that it does not use any marker. It can be very useful, but it also produces a final result that is less accurate than with markers. Indeed, because we only set the anchor points once at the beginning of the video, the errors sum up through time. If the homography transformation is badly approximated from one frame to another, the anchor points will be moved a little from their spot, and the error will be dragged along the rest of the video. Therefore, it is another reason to choose only one frame out of two, since we observed that the errors were less visible when we processed less frames.

Moreover, we noticed that the model worked better when we put the cube onto a plane with distinctive geometrical shapes such as perpendicular lines rather than a plain uni-color table (see examples with figure 3).That can be explained by the fact that if we have corners and good key points where the anchors points will be, the matching will more likely recognise those points from one frame to another, which means the homography H will transform accurately these points and make less mistakes when moving the anchor points of the 3D object.

The initialisation has some limits, it requires that the user defines a base close to a square and that the anchor points are clicked in the right order (the same order as the one in our code when we defined the 3D coordinates of the base anchor points).
But one strong advantage of this method compared to the marker ones is that the cube can disappear and reappear without any problem, since we do not need to match the anchor points. As long as the homography is correct, the points of the will be drawn at the right place, even if it is out of the video frame.

To have better results with this model, it requires finding a good balance between processing-time, video resolution and homography accuracy, by finding the right number of frames to process. Choosing a good traceable spot to put the 3D object can also improve results, and a good initialisation is essential.

5 Conclusion

Both models (marker and markerless) are actually complementary in terms of application. The first model requires to generate ArUco markers, but can be very easily used by someone in live, who could just download the markers on its phone and add any type of 3D shape on its screen with its webcam. The second model cannot be used live because it requires matching operations that are too slow. But it can be more helpful when someone wants to edit a video and add 3D objects afterward.

Our model could go even further by adding an automatic recognition of the ground, or horizontal planes of the room to pose the 3D objects without any initialization or marker.

Moreover, if we managed to use OpenGL package to print meshes over the opencv images, we could add pretty much anything with this model. Indeed, our model is able to project a 3D cube onto a plane. If we put any 3D mesh in this cube, then the cube represents the 3D coordinate system of the 3D complex mesh. The vertices will only need to be projected the same way as the cube, which will be the volume envelop of the 3D object (See figure 4).Therefore, our model can be used very easily to add more complex shapes than a cube.

6 Appendix : Images

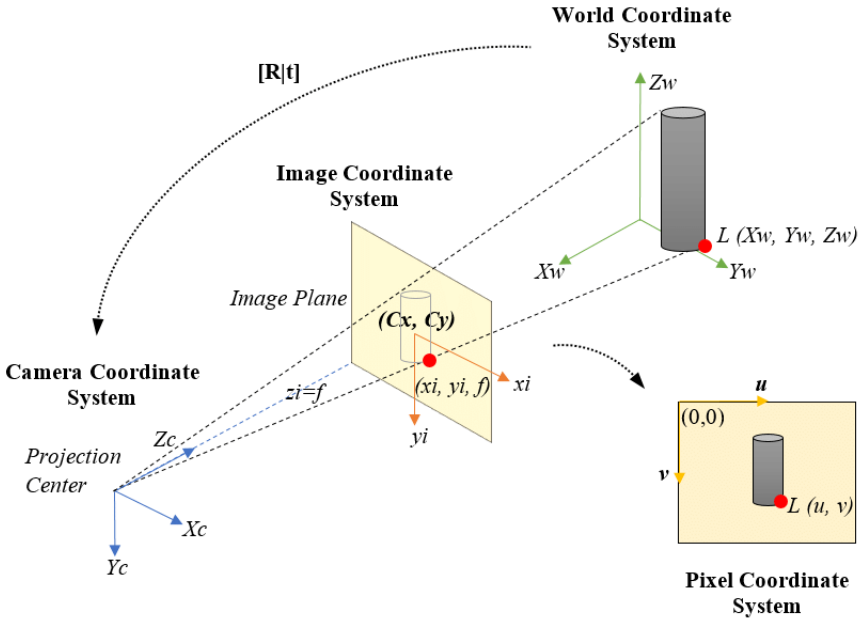

Figure 1: The pinhole camera model

Figure 2: Exemples of ArUco markers

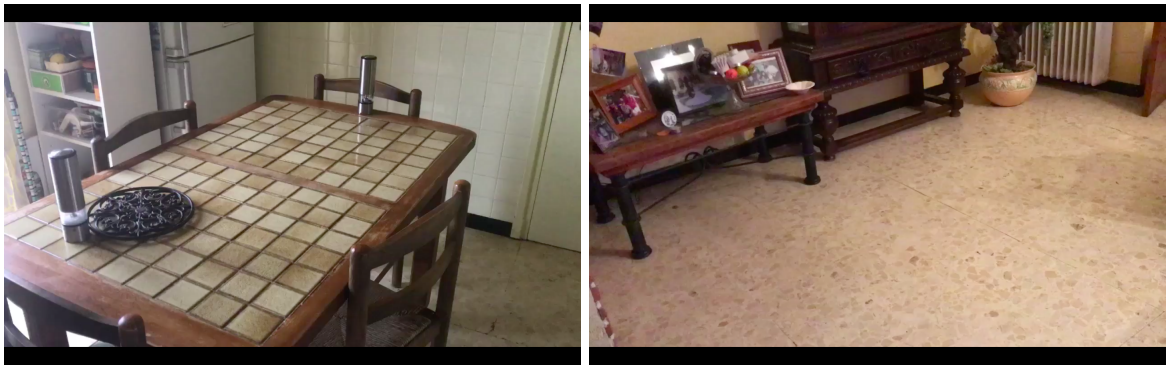

Figure 3: Examples of a good video setting (left) and a bad one (right)

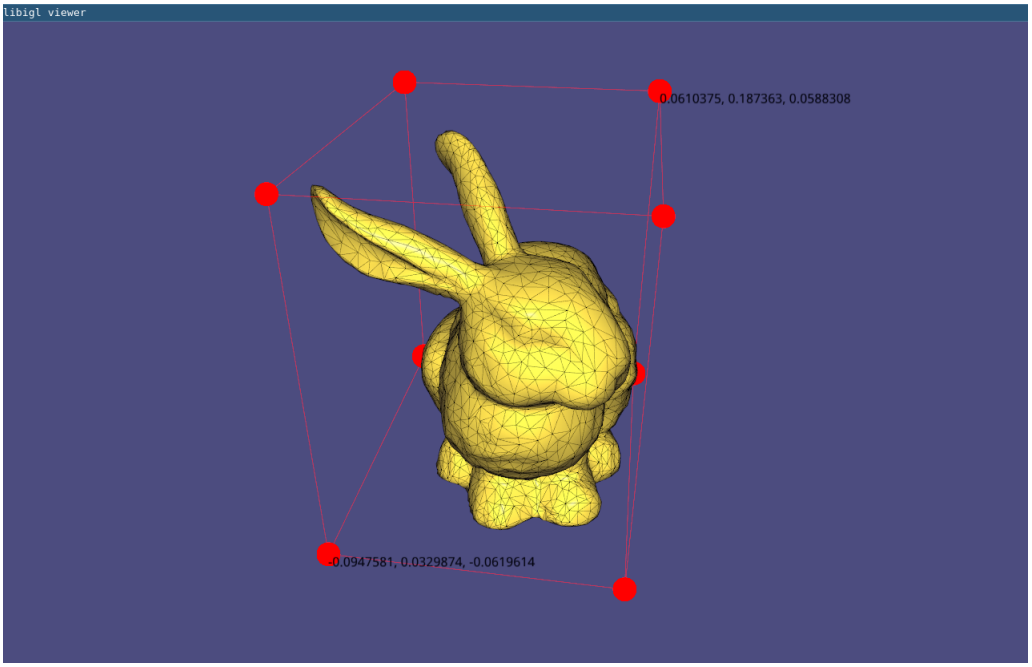

Figure 4: 3D complex mesh inside a 3D cube

Bibliography

[1] *OpenSight Augmented Reality System for Microsoft HoloLens Cleared by FDA*
https://www.hospimedica.com/business/articles/294775651/opensight-augmented-reality-system-for-microsoft-hololens-cleared-by-fda.html
HospiMedica, 2018

[2] *Amiens Cathedral*
http://www.amiens-tourisme.com/cathedrale/visiter_la_cathedrale/la_realite_augmentee
Amiens Tourisme, 2010

[3] *Google 3D animals and objects*
https://9to5google.com/2020/11/17/google-3d-animals-list/
5to9Google 2020

[4] *OpenCV Documentation*
https://docs.opencv.org/master/d5/dae/tutorial_aruco_detection.html

[5] *ArUco markers in OpenCV tutorial*
https://www.learnopencv.com/augmented-reality-using-aruco-markers-in-opencv-c-python/
Learn OpenCV, 2020