FMAN35 - Project in Applied Mathematics

Augmented Reality

Robin Seibold* Hanna Björgvinsdóttir†

Lund University
Sweden

Abstract

Augmented reality is a term used for views of the real world being supplemented by computer generated data, such as sound or graphics. In this project, a 3D model is rendered on top of the live camera feed of an iPhone. The model is projected on top of a marker in the scene, and can be viewed from different angles, by moving the camera around.

1 Introduction

The task at hand involves relating objects of a constructed 3D world to objects of the real world, placing them in the same scene. The artificial camera used when exploring 3D graphics scenes is in a sense replaced by the physical camera of the iPhone, making the placement of the phone control the contents of the screen, as opposed to the mouse-movement of or buttons of a computer.

1.1 Utilities

Two libraries, except for the application programming interfaces (APIs) in Swift, were used during this project to successfully create augmented reality. The first one is OpenCV [Bradski 2000], which is a library of programming functions for computer vision and the like.

Figure 1: Example of an Aruco marker.

The second one is Aruco [Garrido-Jurado et al. 2016], which is a library for detecting square fiducial markers in images, based on OpenCV. The markers used in Aruco consist of $m \times m$ grids of bits, encoded in a somewhat modified version of the Hamming code, surrounded by a black border. An example of an Aruco marker can be seen in Figure 1.

For constructing, manipulating, and rendering the 3D object, Apple's SceneKit [Apple 2015] was used. SceneKit is a framework, with a high-level API which enables working with 3D graphics without having to implement any rendering algorithms. SceneKit also uses the data structure called scene graph, which is a tree like ordering of all the objects in a scene, and how they relate to each other.

2 Methodology

The concept of 3D graphics is to generate 2D images from 3D model descriptions. This is done by transforming, and projecting, objects between different coordinate systems. There are mainly five different coordinate systems often referred to.

The first one is object, or model, space. This is the coordinate system for which the object is defined. The second one is the world coordinate system. The third one is the view space, also referred to as camera or eye space. The fourth one is the so called clip space, and the last one is the screen, or window, coordinate system.

Given a 3D point, defined in homogeneous coordinates as

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \tag{1}$$

the following transformations are needed to convert it into a 2D point in screen coordinates. First a model transformation is applied to the point to transform it from model space into world space, then a view transformation is applied to the point to transform it from world space into view space. A projection transformation is then applied to transform it from view space into clip space.

The x, y, and z, are then divided by w to produce the normalised device coordinates, which are finally scaled and translated to at last create the window coordinates.

For this project the model, view, and projection transformations are of most concern. All these are constructed as 4×4 matrices. Transforming the 3D point from model space into clip space can then be represented by the following equation

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M} \cdot \mathbf{p}, \tag{2}$$

where $\mathbf{P}$ is the projection matrix, $\mathbf{V}$ is the view transformation matrix, and $\mathbf{M}$ is the model transformation matrix.

Usually when working with 3D graphics the camera is an abstraction created with intrinsic parameters to create a desired effect. When working with augmented reality however, both the projection matrix $\mathbf{P}$ and the view transformation matrix $\mathbf{V}$ have to be based on the real camera, in order to create a more realistic result. $\mathbf{P}$ will be created using the intrinsic parameters of the physical camera, while $\mathbf{V}$ will be found using the extrinsic parameters and the camera distortion coefficients.

The object to be rendered is intentionally placed in the center of the scene graph, and with no additional object specific transformation, like rotations or scaling. The $\mathbf{M}$ matrix is therefor equal to the identity matrix, and the model space for the object and the world space are equal. This simplifies Equation 2 to

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{p}, \tag{3}$$

which means that only $\mathbf{P}$ and $\mathbf{V}$ are left to be estimated.

*dat11rse@student.lu.se, robinbobseibold@gmail.com
†dat11hbj@student.lu.se, hannabe@live.com

2.1 Calculating the Projection Matrix

The real camera distortion coefficients and intrinsic parameters were calculated using OpenCV and images of a chessboard taken with the camera that was to be calibrated. This resulted in a 1×5 vector of distortion coefficients, and a camera matrix

$$\mathbf{F} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

where the intrinsic parameters (f_x, f_y) are camera focal lengths and (c_x, c_y) are the optical centers expressed in pixels.

With these parameters and the camera matrix $\mathbf{F}$, the projection matrix $\mathbf{P}$ can be created as

$$\mathbf{P} = \begin{bmatrix} \frac{2 \cdot f_x}{w} & 0 & \frac{w - 2 \cdot c_x}{w} & 0 \\ 0 & \frac{2 \cdot f_y}{h} & \frac{-h + 2 \cdot c_y}{h} & 0 \\ 0 & 0 & \frac{-z_f - z_n}{z_f - z_n} & \frac{-2 \cdot z_f \cdot z_n}{z_f - z_n} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \tag{5}$$

where w is half the image width, h is half the image height, z_n is the near clipping plane, and z_f is the far clipping plane. The near and far clipping planes are constants used in computer graphics to decide what should and should not be rendered. Only objects between the near and the far clipping planes will be rendered.

2.2 Finding the View Transformation Matrix

To find the transformation matrix $\mathbf{V}$ the extrinsic parameters of the physical camera have to be found. This is done by finding the relation between four 3D points in SceneKit, and four 2D points found in an image. The corners of Aruco markers were used as the 2D points in the image, and since they are square, the chosen 3D points are the four points at the bottom of a unit square, that is

$$\mathbf{p}_0 = \begin{bmatrix} 0.5 \\ -0.5 \\ -0.5 \end{bmatrix}, \ \mathbf{p}_1 = \begin{bmatrix} 0.5 \\ -0.5 \\ 0.5 \end{bmatrix}, \ \mathbf{p}_2 = \begin{bmatrix} -0.5 \\ -0.5 \\ 0.5 \end{bmatrix}, \ \mathbf{p}_3 = \begin{bmatrix} -0.5 \\ -0.5 \\ -0.5 \end{bmatrix}.$$

In order to get a hold of the 2D points, Apple's framework AV Foundation was used to start the back camera of the iPhone, from where images were fetched. The images received from the functions of AV Foundation are in the `CMSampleBuffer` format, used when transferring different media types. In order to use the images, and display them on screen, the images had to be converted to the `UIImage` format.

Furthermore, the images captured by the iPhone camera do not share proportions with the physical iPhone screen. For this reason, the images also had to be resized. After an image has been resized and converted, it is passed to Aruco, which returns the corner coordinates and the ID of the marker, should one be found in the image.

The `solvePnP` function was used to find 3D-2D point correspondences, given the 3D object points, the 2D points found in the image, the camera matrix $\mathbf{F}$, and the distortion coefficients. The function is based on the Levenberg-Marquardt optimization, and returns a transformation matrix

$$\mathbf{T} = \begin{bmatrix} & \mathbf{R} & & \mathbf{t} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6}$$

with minimized reprojection error, where $\mathbf{R}$ is a 3×3 rotation matrix, and $\mathbf{t}$ is a 3×1 translation vector. This matrix was used to calculate the position of the camera in the 3D scene, in relation to the object. To do this the matrix was first inverted. Since matrix inversion is a costly function, the properties of the matrix $\mathbf{R}$ and vector $\mathbf{t}$ were utilised as

$$\mathbf{T}^{-1} = \begin{bmatrix} & \mathbf{R}^T & & -\mathbf{R}^T \cdot \mathbf{t} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{7}$$

to invert the matrix $\mathbf{T}$.

This is possible since the transformation matrix transforms a point $\mathbf{x}$ as

$$\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t} \tag{8}$$

which can be rewritten as

$$\mathbf{R}^{-1}(\mathbf{x}' - \mathbf{t}) = \mathbf{x}, \tag{9}$$

and

$$\mathbf{R}^{-1} = \mathbf{R}^T \tag{10}$$

since rotation matrices are orthogonal.

Before the resulting transformation matrix $\mathbf{T}$ can be used to transform the position and rotation of the camera in world space, two things have to be done. First, the coordinate space in OpenCV and SceneKit differ. SceneKit uses a right-handed coordinate system as can be seen in Figure 2a, while OpenCV uses the coordinate space seen in Figure 2b.

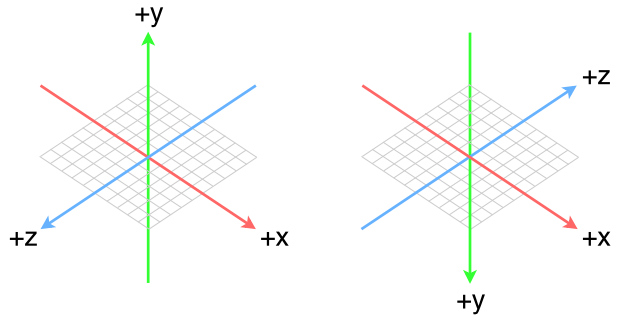

Figure 2: The difference of the coordinate system orientations between SceneKit (a) and OpenCV (b).

To make up for this, a rotation matrix which rotates $180°$ around the x-axis is applied to the matrix as

$$\mathbf{T}_x^{-1} = \mathbf{T}^{-1} \cdot \mathbf{R}_x = \\ = \mathbf{T}^{-1} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(180°) & -\sin(180°) & 0 \\ 0 & \sin(180°) & \cos(180°) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{11}$$

The second thing is that OpenCV stores matrices in row major order, while SceneKit stores them in column major order. So before passing the matrix to SceneKit, it has to be transposed. The final transformation matrix used to transform the camera node in the scene graph is then equal to

$$\mathbf{T}_{sk} = (\mathbf{T}_x^{-1})^T. \tag{12}$$

SceneKit will create the matrix $\mathbf{V}$ from the new position and orientation of the camera when rendering the scene.

The scene finally created, with the 3D model in the middle of the scene, and a camera position and orientation calculated from the 2D

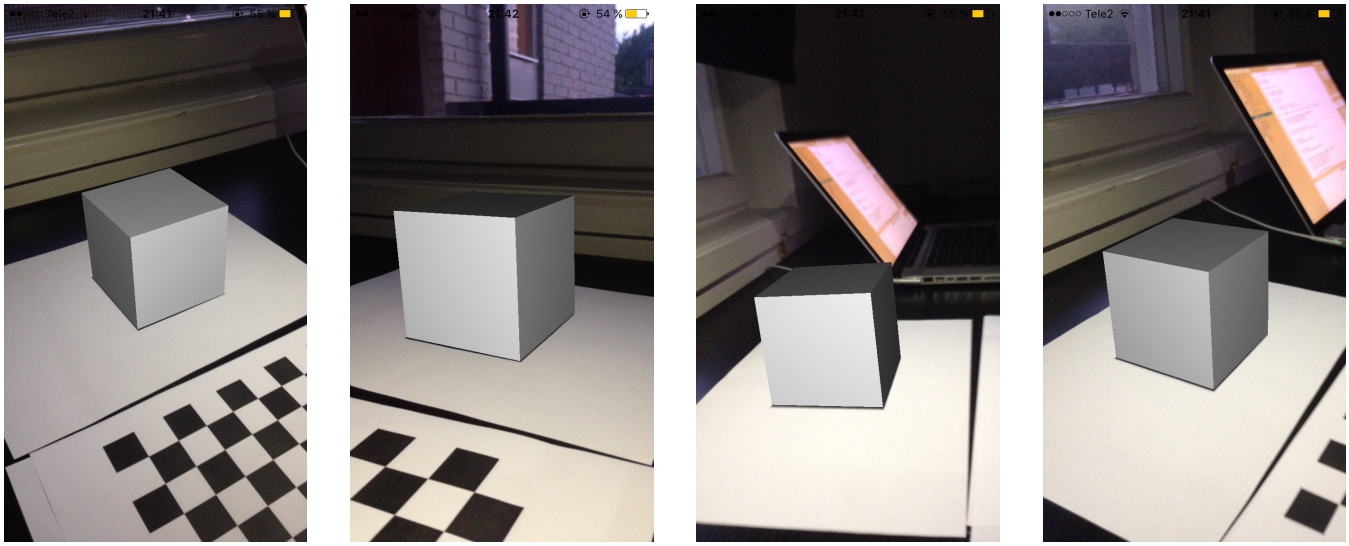

Figure 3: Screenshots from the augmented reality application.

Listing 1: Pseudo code for the augmented reality application.

```
Start application
    Load F and distortion coefficients
    Create projection matrix P from F
    Create scenegraph
    Create model and place it in the center of the scenegraph
    Start iPhone camera
    While application is running
        For each frame
            If marker is found
                Calculate T with solvePnP()
                Calculate T_sk
                Translate camera using T_sk
                Render model on top of frame
            Else
                Hide model
```

points, is rendered with a transparent background. The camera image which was processed with OpenCV to find, and calculate, $\mathbf{T}_{sk}$ is then added as the background, making the rendered 3D object blend into the real world captured by the camera.

Listing 1 describes the different stages of the final application, giving an overview of its lifetime.

3 Result

Figure 3 shows screenshots from the application running on an iPhone 5, taken at different angles. The 3D cube follows the marker well, though its size can vary slightly in comparison to the marker.

The detection of the marker works best when the lighting conditions are good. For this reason, it could be useful to have the option to turn on the light next to the camera on the iPhone when using the application.

When using AVFoundation to capture video, the autofocus is turned on by default. In this mode, the model started to flicker in response to the focusing. For this reason, the autofocus was turned off.

The frame rate is slightly below expectations, largely due to the slow conversion between the `CMSampleBuffer` image format provided by the functions of AV Foundation, and the `UIImage` format needed for display.

Except for finding a way of speeding up the image conversion, one might consider using lower resolution images for the sake of speed improvement, perhaps at the cost of declining detection.

As of now, only the corner coordinates of the Aruco markers are being exploited. As a further development of this application, the ID's of the markers could be used as well, so that each unique marker would result in the display of a particular 3D model.

4 Conclusion

All in all, the project was successful. The application does a good job at locating the marker, and the cube follows it well. In further development of the application, a higher frame rate would be desirable, and different markers could produce projections of different 3D models.

References

APPLE, 2015. SceneKit framework reference.

BRADSKI, G. 2000. *Dr. Dobb's Journal of Software Tools*.

GARRIDO-JURADO, S., MUÑOZ-SALINAS, R., MADRID-CUEVAS, F., AND MEDINA-CARNICER, R. 2016. Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recognition 51*, 481 – 491.