Marker-based Augmented Reality 12

Augmented reality applications rely on the estimation of the pose of the viewing device with respect to some environmental feature or fiducial marker. The most common method is to use a camera that captures the images upon which the virtual objects are rendered. The process implies a first step of detecting some relevant information on the input image that once matched against its model enables the estimation of its pose in the camera frame or vice-versa.

In this work we will use ArUco markers whose images are captured by the computer camera and then detected and used for the estimation of their pose with respect to the camera frame.

12.1 OpenGL/C++

For the C++ implementation we will rely on OpenCV, which is an open source library that implements a large amount of computer vision algorithms. You can install it on ubuntu with

— C++ —

```
1 apt install libopencv-dev
```

As an example, we can use it for do real-time contour extraction by using the code listed hereafter.

— C++ —

```
1 #include "opencv2/opencv.hpp"
2 using namespace cv;
3
4 int main(int, char**){
5     VideoCapture cap(0); // open the default camera
```

```
6       if(!cap.isOpened())  // check if we succeeded
7           return -1;
8
9       Mat edges;
10      namedWindow("edges",1);
11      for(;;)
12      {
13          Mat frame;
14          cap >> frame; // get a new frame from camera
15          cvtColor(frame, edges, COLOR_BGR2GRAY);
16          GaussianBlur(edges, edges, Size(7,7), 1.5, 1.5);
17          Canny(edges, edges, 0, 30, 3);
18          imshow("edges", edges);
19          if(waitKey(30) == 27) break;
20      }
21      return 0;
22  }
```

As we can see in this code, the main loop consists in image acquisition, processing and display.

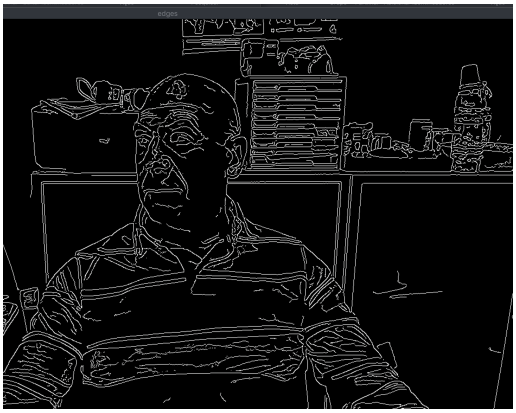

Figure 12.1: On line contour extraction.

By changing the number on the constructor of the object VideoCapture you may select the second or third camera connected to your computer. You even use an IP camera of your smartphone acting as one as long as you have it connected to an accessible IP network and with the appropriate software. For Android devices you may use "IP Webcam" and on IOS "IP Cam". On the latter case and assuming your device has IP 192.168.1.2 you may access it with

— C++ —

```
1   VideoCapture cam;
2   cam.open("http://192.168.1.2:8020/videView");
```

Exercise 12.1

Acquire the image from the camera or video file and display it as a texture of a rectangle, and the contour image in another rectangle.

12.2 Marker pattern detection

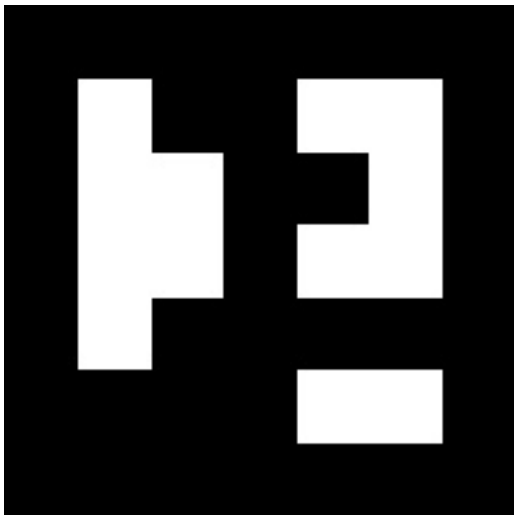

Figure 12.2: An ArUco marker

There are several possibilities of using markers for building AR. The first condition is that the markers' detector should be as much robust as possible, to avoid detection misses but also avoid false positives. For the current case we are using the ArUco markers, which are simple to detect and may encode useful information to build AR applications. We may think of them like barcodes, or QR-codes, but easier to detect.

The support for detection of these markers is available under OpenCV via

— C++ —

```
1 cv::aruco::detectMarkers(frame, dictionary, markerCorners,
2                           markerIds, detectorParams,
3                           rejectedCandidates);
```

where *frame* is the input image, *dictionary* is a pointer to one of the available dictionaries of markers, *markerCorners* is the output vector containing the coordinate pairs of the corners of the detected markers, *markerIds* is the encoded identifiers of the detected markers, *detectorParams* are some customisable parameters of the detector, and *rejectedCandidates* are the corners of rejected candidate regions.

For Javascript the detection is made by using the following class

JavaScript

```
this.detector = new AR.Detector();
```

and for each received image the markers present can be detected using

JavaScript

```
   // imagedata contains the acquired image data
   var markers = this.detector.detect(imagedata);

    console.log("Markers detected = " + markers.length);
     if (markers.length > 0){
        for (var j = 0; j !== markers.length; ++ j){
                        console.log("Marker "+j + " ID= " + markers[j].id);
                var corners = markers[j].corners;
                        // we may print the corner coordinates
                for (var i=0; i !== corners.length; ++ i){
                        var corner = corners[(i + 1)
                        console.log( "Corner " + i + " " + corner.x + " " + corner.y);
                   }
               }
\end(lstlisting)

Once the markers have been detected and as we know their dimensions, and therefore their cor
algorithm for estimating the relative camera-marker pose.

\begin{lstlisting}[language=C++]
 aruco::estimatePoseSingleMarkers(markerCorners, markerLength,
                                  camMatrix, distCoeffs,
                                  rvecs, tvecs);
```

Exercise 12.2
Explain every parameter of the above function.

12.3 Calibrating the camera

Do not forget that you should calibrate the camera and use the camera matrix (and eventually the distortion coefficients) on the marker pose estimation function (not on the javascript version). Also note that if you use different cameras you must have the appropriate calibration matrices for each of them, store these on files that you load instead of recalibrating the camera every time. The camera calibration matrix can be made obtained from the website `http://calibdb.net` using the Chrome web browser.

As we are willing to superimpose on a view of the environment, as captured by a webcam, the projection of a virtual object attached to some marker (or object) on the scene, the OpenGL projection must model as close as possible to the physical phenomenon.

As we known the camera intrinsic parameters matrix is a 3x3 matrix of the form

$$\mathbf{K} = \begin{bmatrix} \alpha & s & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

but the projection matrix used in OpenGL (WebGL) is 3×4, to perform the prior conversion to NDC, instead of directly projecting on a plane. In fact the projection matrix may be obtained directly from the camera intrinsics matrix using

$$\mathbf{P} = \begin{bmatrix} \frac{2*K_{0,0}}{width} & \frac{-2*K_{0,1}}{width} & \frac{(width-2*K_{0,2}+2*x_0)}{width} & 0 \\ 0 & \frac{-2*K_{1,1}}{height} & \frac{(height-2*K_{1,2}+2*y_0)}{height} & 0 \\ 0 & 0 & \frac{(-zfar-znear)}{(zfar-znear)} & \frac{-2*zfar*znear}{(zfar-znear)} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The above is valid considering window coordinates "y-up", for the case "y-down"

$$\mathbf{P} = \begin{bmatrix} \frac{2*K_{0,0}}{width} & \frac{-2*K_{0,1}}{width} & \frac{(width-2*K_{0,2}+2*x_0)}{width} & 0 \\ 0 & \frac{2*K_{1,1}}{height} & -\frac{(height-2*K_{1,2}+2*y_0)}{height} & 0 \\ 0 & 0 & \frac{(-zfar-znear)}{(zfar-znear)} & \frac{-2*zfar*znear}{(zfar-znear)} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

On the above $K_{i,j}$ refers to the elements of the $\mathbf{K}$ matrix, *width* and *height* refer to the dimensions of the camera images, and (x_0, y_0) to the camera image origin which is normally $(0, 0)$. Note that the camera image dimensions must be the same for which the camera was calibrated, otherwise a new calibration must be performed.

Being an AR application you must display the camera image on the background using the method described on 10.4.

12.4 Javascript examples

Exercise 12.3
Run and study the code of the given Jupyter notebooks and using an Aruco marker generated from `https://chev.me/arucogen/` by selecting the Original Aruco Dictionary.

Exercise 12.4
Do the camera calibration and store the results in a text file for later use.

Exercise 12.5
Build a projection matrix from the camera calibration matrix and check the results with any scene you have built before.

12.5 C++ exercises

Exercise 12.6
Compile and run ar.cpp code and study how to get the extracted pose for in an OpenGL application superimpose a cube or other object on the marker image.

Exercise 12.7
Integrate the ar.cpp code in your own code to implement your first AR application. Use the extracted pose to assign it to a virtual object. Do not forget to use the camera image as the background of your application as explained in last week's sheet.

Explain any problems that may arise.