Googly Eyes Cam

Augmented Reality Eye Tracking App – Project Documentation

Building Web and Mobile Apps
Hochschule Fulda, WS2016/2017

Jonas Kleinkauf (Mat.Nr.: 143871)

1. Project Idea

This project is based on the Paper "Remote visual tracking for the mobile web" by Olbrich et. al. [1] They did research on how a server application, which does heavy computation work for computer vision algorithms could be used for augmented reality applications on the web and on mobile devices. The idea is to stream a video feed (or single frames of if) from the web or mobile device to a server, whereas the server calculates characteristic data (for example positions of key objects in the image) and sends them back as vectors. The clients then show 3D objects on top of the recorded video.

The Googly Eyes Cam app itself is also inspired by the popular photo chat app Snapchat, where several AR filters are available to modify selfie photos. The purpose of the Googly Eyes Cam app is to recognize the eyes of a user and place funny 3D googly eyes on top of them. The user can then take a photo of himself with the funny eyes for a comical effect.

To do this, the app sends video frames to a server. The server calculates the position of faces and eyes in the frame and sends this information back to the client app. The client app places the 3D eyes on top of the video feed, where a user can save a snapshot as selfie photo.

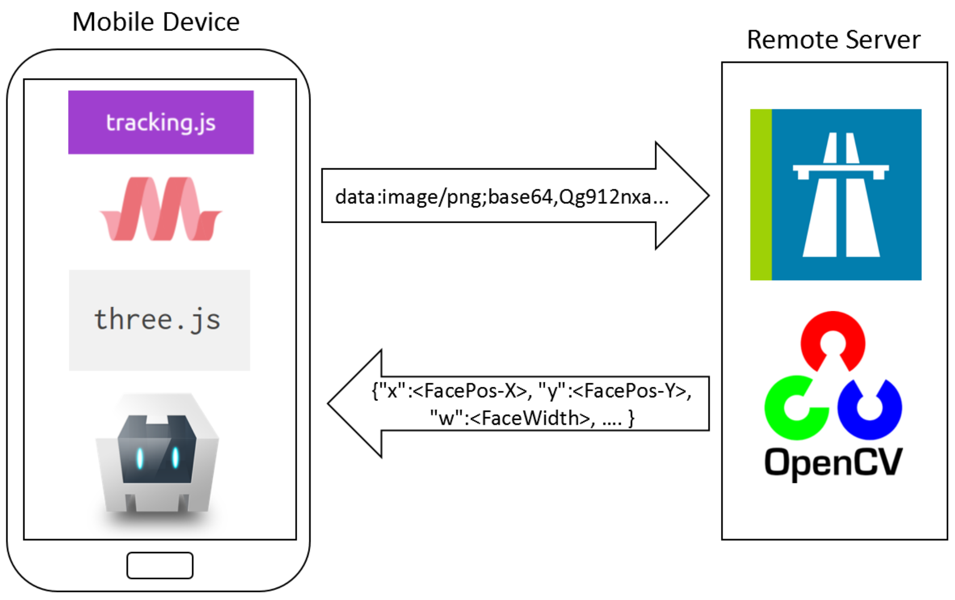
The app also features a settings screen where a user can set the IP-address of the remote server. The settings screen also allows local eye tracking without the remote server (with severely less performance).

[1] Olbrich M., Franke T., Rojtberg P. 2014. Remote Visual Tracking for the (Mobile) Web, Fraunhofer IGD, TU Darmstadt

2. System Architecture

The project consists of two main parts: A client app and a server software. The client app was developed with web technologies, using the cross-platform app engine Apache Cordova to deploy it as Android app. Since no personal Apple devices where available, it has only been deployed to Android as of now, but could be ported to iOS with little effort. The client app connects to the server software using WebSocket technology. WebSockets allow to create an efficient, full duplex communication between a client and a server with very little overhead. Thus, they are ideal for the app's purpose.

The server itself, written in the Python programming language, creates a WebSocket endpoint to listen to. When a client sends a video frame, the server analyzes it using the computer vision framework OpenCV2 for Python. The results from OpenCV are then formatted and sent back to the client.


Mobile Device
tracking.js
three.js
Remote Server
data:image/png;base64,Qg912nxa...
{"x":<FacePos-X>, "y":<FacePos-Y>, "w":<FaceWidth>, ... }
OpenCV

3. Server Software

Autobahn.ws for Python is used on the server side to create a WebSocket endpoint. It accepts WebSocket messages in form of Base64 encoded JPEG images.

Then it uses the **`cv2.CascadeClassifier`** function to analyze the images' content. This OpenCV function is based on the HAAR-Cascade classifier. It uses a pre-generated classifier knowledge base (called the cascade) in XML format. The default Python distribution of OpenCV already features HAAR-cascades for facial and eye recognition. The server uses the face HAAR-cascade first to filter the positions and sizes of faces in an image as sub-images. Afterwards these sub-images are searched for eye positions and sizes. The first step is necessary to filter out some hits that the eye classifier may find, but that are not inside any face.

The server sends back the vector data over the WebSocket using JSON encoded strings. The JSON list contains one entry for each Eye that has been found in the following result format (all values are integers as they are pixel coordinates):

```
[
{"x":<FacePos-X>, "y":<FacePos-Y>, "w":<FaceWidth>, "h":<FaceHeight>,
"ex": <EyeOffset-X>, "ey": <EyeOffset-Y>, "ew": <EyeWidth>, "eh": <EyeHeight>},
...
]
```

After OpenCV for Python has been installed, the following command starts the server software inside the application source folder:

```
pip install -r requirements.txt
python -u googly_server.py
```

4. Client App

In the development process, the client app was first built as pure web application and was later ported to Android using Apache Cordova. It uses the following third-party web libraries for the local tracking, the user interface look-and-feel and for general programming convenience:

- Tracking.js (for local face tracking)
- MaterializeCSS (for UI styling, buttons..)
- jQuery3 (for general convenience)
- Three.js (for 3D visualization)

It first just used the webcam functionality of HTML5 WebRTC. While this was working fine in the browser, it caused problems when it was used on the actual mobile phone with Cordova. As it turned out, Cordova does not natively support the HTML5 <video> tag with WebRTC to access a phone's camera. The default Cordova functionality for camera access always opens the native camera application on the phone to take a picture or video and then handles the taken picture or video in a callback function. As this is not working for the GooglyEyesCam's use case, Cordova plugins had to be used to achieve the wanted behavior.

The following Cordova plugins have been used:

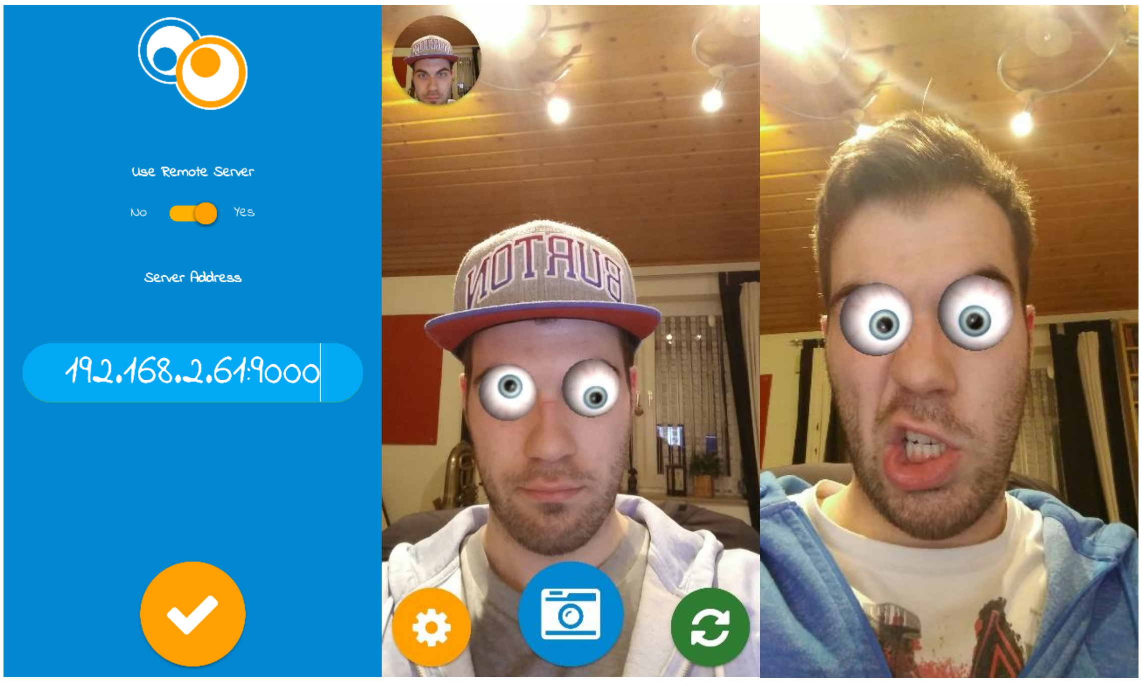
- cordova-plugin-compat (for backwards compatibility)
- cordova-plugin-whitelist (for Android app permission handling)
- cordova-plugin-camera-preview (to preview the actual live camera feed in an applications background)
- cordova-mm-canvas2image (to save an HTML5 canvas content to the phone's file system → snapshot functionality)

The cordova-plugin-camera-preview plugin was also customized to some extend to improve its real-time performance for the application. This was probably the most work-intensive part of the project.

To install the app on an Android phone, simply copy the GooglyEyesCam.apk file to the phone and run it. To build the app from source, install Apache Cordova and Android Studio (API Version 25) and run the following command inside the *app* folder:

```
cordova build android
```

5. Screenshots



From left to right: Settings screen, Main screen and taken picture

6. Summary

The result of this project is a funny little selfie app that shows a way to use a remote server for augmented reality calculations. Compared to local calculations, the application runs a lot smoother when the remote server is used, but the actual HAAR-cascade calculations of OpenCV still take around 0.2 seconds per frame, which make the application not as responsive as one could expect. HAAR-cascades are also probably not the very best example for real-time augmented reality, as there might be more efficient algorithms for this, but this sample application gives a nice little insight on what is possible when the computer vision calculations are outsourced.

Porting the application to the iOS platform was first planned, but postponed due to the lack of personal Apple devices.