Camgaze.js: Browser-based Eye Tracking and Gaze Prediction using JavaScript

Alex Wallar *
Aleksejs Sazonovs †
University of St Andrews

Patrick Flynn ‡
Christian Poellabauer §
University of Notre Dame

Abstract

Eye tracking is a difficult problem that is usually solved using specialised hardware and therefore has limited availability due to cost and deployment difficulties. We describe Camgaze.js a client-side Javascript library that is able to measure the point of gaze using only commodity optical cameras without relying on external application installed besides a web-browser. In our work, we conduct experiments using Camgaze.js to show the usability of such a system. We also discuss the challenges and applications of using an in-browser eye tracking system. Since the described eye tracker works inside the browser without any additional installation setup, it provides a solution to a larger deployment of eye tracking systems.

1 Introduction

Eye tracking is a challenging problem, that has been attempted to be solved since the 19th century (Ahrens, A., 1891). Die Bewegung der Augen beim Schreiben. Rostock: University of Rostock). Currently, it is mostly viewed as a problem in Computer Vision.

The majority of eye tracking solutions available on the market today are a combination of software and specialised hardware. The principles behind hardware varies greatly: it ranges from head-mounted cameras to lenses with integrated coils. It is believed that the state of the art solutions can allow accuracy up to NN% link. Despite that, carrying out eye tracking experiments remains an issue it is expensive, requires complicated deployment and calibration and, in most cases, has to be carried out in a controlled environment.

In the recent years, it has been shown (Agustin, 2009; Sewell and Komogortsev, 2010) that it is possible to use commodity cameras, often built into modern computers to perform eye tracking with promising quality. Deployment of such systems is relatively simple, but in the described cases it is tied to specific computer platforms (Oleg - Texas iPad App).

Web applications are rich websites that are able to run without external plugins inside the browser. Recently, the web-browsers have evolved to adapt to the markets requirements new technologies have risen, allowing web application to be increasingly interactive. For example, WebRTC (Web Real-Time Communication) is an API that aims to enable in-browser audio and video communication. As of September 2013, WebRTC is supported in the stable versions of Google Chrome and Mozilla Firefox. A recent report (http://disruptive-analysis.com/webrtc.htm), claims that by 2016 there will be 3 billion capable devices and 1 billion individual users of WebRTC-enabled devices.

We describe Camgaze.js a Javascript library that uses WebRTC to obtain the video from built-in or USB cameras and measures the point of gaze. We believe that it can be deployed in a wide range of applications such as website interface analytics and concussion testing as well as interactive input.

*email: aw204@st-andrews.ac.uk
†email: as245@st-andrews.ac.uk
‡email: flynn@nd.edu
§email: cpoellab@nd.edu

2 Challenges

Running eye tracking on a wide range of hardware creates additional challenges.

The cameras that are built-in (or supplied with) consumer devices vary greatly: there is no standard resolution, color profile, brightness level or physical position. Currently, WebRTC provides very limited options for changing any of the parameters of the camera that is providing a stream. This problem can be partially addressed by defining individual profiles for a variety of popular devices. The identification of the device is not a trivial task by itself, but services like DeviceAtlas (https://deviceatlas.com/) solve it by looking at a variety of factors, such as browsers User Agent and screen resolution. Unfortunately, this approach would be mostly applicable for handheld devices and not commodity PCs.

The client-side Javascript environment in which Camgaze.js runs in sets some constraints. As of September 2013, we believe that there is lack of comprehensive Computer Vision library implemented in Javascript. There is currently no simple way to port native C/C++ code, which are languages in which popular libraries such as OpenCV are written in. Projects like Emscripten are making early attempts to allow translation of LLVM bitcode code to Javascript, potentially allowing to port some of the well-established Computer Vision libraries to Javascript in the future. For now, we had to create a custom implementation of ALEX to use in Camgaze.js.

Despite some of the described difficulties, the authors of this paper believe that a combination of Javascript and WebRTC is a reasonable technological stack in order to implement an eye tracking solution, suitable for large scale deployment.

3 Implementation

`Camgaze.js` goes through two steps in order to predict the gaze direction. Firstly, `Camgaze.js` detects each pupil. It then uses the pupils deviation from a unique point on the face to determine the gaze metric, $\mathcal{G}$. This metric needs to be calibrated in order for there to be a mapping from $\mathcal{G}$ to a point on the screen. Once this gaze metric has been calibrated, `Camgaze.js` is able to interpolate area of the screen the user is looking at. A high level description of the algortihm is shown below.

Algorithm 1 Pseudocode for `Camgaze.js`

$\mathcal{F} \leftarrow$ InitGazeMapping()
while StillCalibrating() == `true` **do**
  $P_{list} \leftarrow$ DetectPupils()
  $\mathcal{G} \leftarrow$ DetermineGazeMetric(P_{list})
  $\mathcal{F} \leftarrow$ Calibrate($\mathcal{G}, \mathcal{F}$)
while SessionFinished() == `false` **do**
  $P_{list} \leftarrow$ DetectPupils()
  $\mathcal{G} \leftarrow$ DetermineGazeMetric(P_{list})
  ProjectGazeOntoScreen($\mathcal{F}(\mathcal{G})$)

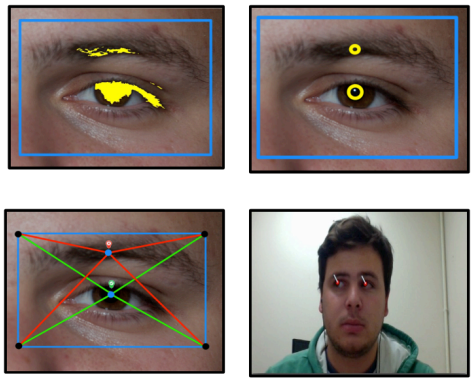

Figure 1: *Pupil Detection Process*

3.1 Pupil Detection

Detecting the pupils enables `Camgaze.js` to determine the gaze direction. Pupil detection in this approach is aimed to be fast in order to be deployable onto web frameworks.. Firstly, the frame is converted to grayscale and the eye is detected using the Viola-Jones Object Detection Framework [Viola and Jones 2001]. The region of interest (ROI) is then thresholded for an array of different colors and blob detection takes place. All of the detected connected components are stored as possible pupils. Out of these possible pupils, the one with the minimum overall error is designated as the pupil. Below are the expressions to be minimized.

$$\text{ERR}_{\alpha}(p) = \frac{\sum_{c \in Corners} |\frac{\pi}{4} - \text{ARCTAN}(|\frac{p_y - c_y}{p_x - c_x}|)|}{\pi} \quad (1)$$

$$\text{ERR}_{size}(p) = \frac{|avgPupilSize - \text{SIZE}(p)|}{2} \quad (2)$$

ERR_{α} refers to how far the blob center is from the center of the Haar bounding rectangle. We use angle deviation instead of pixel distance from the center of the Haar bounding rectangle for this metric because we assume that the pupil would not always reside too close to the center and a direct pixel distance might yield other blobs more suitable. The angle deviation acts a weak error function in order to be more lenient without the use of constants. Once the blob with the minimum error is extracted, the center of the blob is returned.

ERR_{size} refers to the difference between the scaled area of the detected blob and the average scaled area of a general pupil. This means that during the thresholding process, out of the array of blobs detected with minimal ERR_{α}, the blob that has the area closest to that of an average pupil will be returned.

Using a combination of these two error functions, we are able to detect the pupils accurately and derive a gaze metric.

3.2 Determining the Gaze Metric

The gaze metric is determined by establishing a reference point that will remain in a constant position with reference to the pupil center. Using this point, we are able to capture the motion of the pupil without the influence of head or camera movement.

For this work, we use the center of the Haar rectangle that is detected around the eye. This point will remain in a constant position with reference to the movement of the pupil and therefore we are able to determine the gaze of the user.

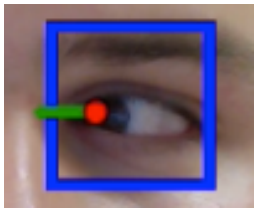

Figure 2: *Determining the Gaze Metric*

3.3 Calibration

Calibration in an eye tracking system is necessary to ensure that the predicted point of gaze resembles the expected point of gaze. The calibration creates a mapping from what the eye tracker determines the gaze metric is to a point on the screen. For visible light based eye tracking using commodity eye trackers, neural network [**?**] and linear based calibration techniques have been deployed. In our work, we use a linear based approach due to ease of implementation and computing power restrictions imposed by doing the computation inside the browser in real-time.

The linear approach works by placing sample points in the corners of the viewing screen and asking the user to look at these points, one at a time. Using the data gathered from the eye tracker and the knowledge of what point the data corresponds to on the screen, we can construct a linear mapping from the gaze metric onto the screen. We map the distance from the *preceived* point on the top left to the preceived point on the top right to the width of the screen. We also map the distance from the preceived top left point to the preceived bottom right point to the height of the screen. This mapping build parametric equations that be used to interpolate the looking point on the screen.

The user basically creates a box in which the eye tracker thinks its looking in and maps the height and width of this preceived box to the dimensions of the screen.

The application we created for calibration presents a point in each corner of the screen one at a time. The user is told to look at the given point click on the screen when they are done. The click on the screen will show each calibration point until all of the corners have been mapped.

3.4 Using JavaScript

Due to constraints imposed by JavaScript as a language while writing computer vision applications, there are certain steps that need to be taken when implementing the algorithms described above. Firstly, a video tag needs to be already present in the HTML or created by the JavaScript application in order for a video feed to be retrieved from the camera. Likewise, a canvas element needs to be present in the HTML in order to retrieve the RGBA pixel values for the video. This is because the video element needs to be drawn onto the canvas in order to retrieve the `ImageData` object for it. Once the `ImageData` has been obtained, processing mentioned above can take place.

4 Methodology

5 Results

6 Discussion

6.1 Ethics

Camgaze.js has an ambition of bringing eye tracking to larger-scale deployment. This intensifies the need to address the privacy concerns some users might have. The library we describe attempts to do this in several ways.

When the user accesses a web page that uses Camgaze.js, they are prompted a notice that the website is attempting to use the video from their computer. This is the default behaviour of both Chrome and Firefox and this behaviour can not be overridden by any website. Camgaze.js displays a notice, disclosing for what specific purposes this data will be used. Thus, we prevent capturing the data from unaware user.

As previously mentioned, Camgaze.js is implemented in Javascript. Since the absolute majority of modern web browsers have an built-in Javascript interpreter, it is possible to do the eye tracking on the client-side. This allows the proposed solution to avoid sending and storing the users video stream to an external server.

In general, we believe that sensible measures have been taken to mitigate the impact on the users the privacy.

7 Limitations

8 Future Research & Applications

Preliminary experiments show that Camgaze.js is working on a tablet computer (Google Nexus 7 – 2013 version, Chrome Beta 30.0.1599.81, V8 3.20.17.13 Javascript engine), which brings a potentiall to bring eye tracking to a variety of hanheld devices. Further research could concentrate of feasibility of eye tracking on even smaller devices – smartphones and phablets (phones with a screen wider than 5 inches).

We hope to make progress with porting one of the popular Computer Vision libraries to Javascript, thus allowing to apply the latest developments in the field for the task that Camgaze.js tries to solve.

Additional research on the normalization of the video stream could be done. Porting some of the algorithms to Javascript would be a novel task.

We believe that Camgage.js should be used in cases, where the simplicity and scalability of deployment overweights the need for perfect precision of point of gaze prediction. Ability to open a webpage, click ok on a pop-up bar, and start eye tracking opens up possibilities for use of eye tracking for various new applications, which were previously solvable by eye tracking, but never were able to get applied outside the controlled lab environment.

For example, Heitger et al. has shown that impaired eye movements in post-concussion syndrome patients indicate trauma on the brain that surpasses the influence intellectual ability [Heitger et al. 2009]. The use of current generation eye tracking hardware / software combinations makes a deployable solution for testing concussions based eye movements difficult. `Camgaze.js` has the ability to be used in these deployable senarios such as Emergency Rooms, abulances, and sports centers.

References

HEITGER, M. H., JONES, R. D., MACLEOD, A. D., SNELL, D. L., FRAMPTON, C. M., AND ANDERSON, T. J. 2009. Impaired eye movements in post-concussion syndrome indicate suboptimal brain function beyond the influence of depression, malingering or intellectual ability. *Brain : a journal of neurology 132*, Pt 10 (Oct.), 2850–70.

VIOLA, P., AND JONES, M. 2001. Robust real-time object detection. In *International Journal of Computer Vision*.