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

Alexander Wallar *
Aleksejs Sazonovs †
University of St Andrews

Christian Poellabauer ‡
Patrick Flynn §
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 any external application installed besides a web-browser. 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.

CR Categories: I.4.8 [Image Processing and Computer Vision]: Scene Analysis—Tracking I.4.9 [Image Processing and Computer Vision]: Applications;

Keywords: gaze prediction, eye detection, pupil detection, low-cost eye tracking, linear calibration, JavaScript eye tracker, in-browser eye tracking

1 Introduction

Eye tracking is a challenging problem, that has been attempted to be solved since the 19th century [Ahrens 1891]. 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 [San Agustin et al. 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 [Holland and Komogortsev 2012].

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

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 [Disruptive Analysis 2013], 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.

2 Challenges

The deployment of eye tracking systems on a wide range of hardware creates number of challenges 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 [DeviceAtlas 2013] 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 connected component detection and image moment calculation 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

Algorithm 1 shows a high-level overview of `Camgaze.js`. The algorithm goes first through a calibration phase. Once the system is calibrated, it is able to project the gaze position of the user.

`Camgaze.js` goes through three steps in order to predict the position on the screen that the user is looking at. Firstly, the pupils are detected. Then the positions of these pupils with reference to the

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($\mathcal{F}(\mathcal{G})$)

eye are used to determine a *gaze metric*. This gaze metric is then calibrated and mapped to a position on the screen.

3.1 Pupil Detection

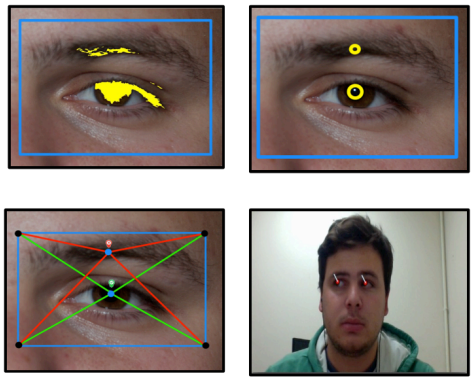

Figure 1: *Pupil Detection Process*

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 shades. Connected components are then detected on these binary images. 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 center of the connected component is from the center of the Haar bounding rectangle. We use angle deviation instead of pixel distance for this metric because we assume that the pupil would not always reside immediately in the center of the boundary rectangle. A direct pixel distance might yield other connected components more suitable. The angle deviation acts a weak error function in order to be more lenient without the use of constants. Once the connected component with the minimum error is extracted, its center 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 connected components detected with minimal ERR_{α}, the one 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.

Figure 1 depicts the process of pupil detection. The photo on the top left shows the connected component segmentation. In the top right image, the centers of the connected components are extracted. The image on the bottom left shows the deviation of the ERR_{α} for the two connected components. Since the has a smaller overall error and is returned as the pupil.

3.2 Determining the Gaze Metric

The gaze metric is a quantifiable measurement that describes the gaze direction for an eye. To determine the gaze direction, we must be able to capture the movement of the pupil relative to the eye position. This is done by computing the horizontal and vertical displacements from the center of the Haar bounding rectangle surrounding the eye to the pupil center. Since the center of the Haar bounding rectangle stays in a constant position with reference to the pupil, it can be used to discern the relative movment of the pupil.

Using a point that remains in a relatively constant position with reference to the pupil is vital for determining the gaze direction because the determined gaze will not be affected by the movment of the head or the jitter of the camera.

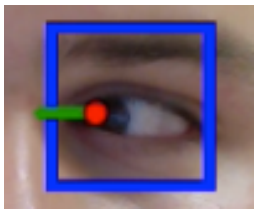

Figure 2: *Determining the Gaze Metric*

Since the gaze metric represents the pupil's displacement from the center, it can be represented as a 2D vector. In **Figure 2**, this vector is being drawn from the pupil center. This gives us a visual representation of the gaze direction. This metric is also used in calibration so that we can interpolate the position on the screen that user is looking.

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 [Holland and Komogortsev 2012] 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 between the uncalibrated point on the

top left and the uncalibrated point on the top right to the width of the screen. We also map the distance between the uncalibrated top left point the uncalibrated bottom right point to the height of the screen. This mapping determines parametric equations that be used to interpolate the gaze point on the screen.

The user essentially 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 then has 7 seconds to look at the point before the next point is shown.

3.4 Obtaining Video Data

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

4.1 Apparatus

For the test we used an Apple MacBook Pro 13' laptop (mid-2013 model: Intel Core i7 processor running on 2.9 GHz clock speed, 8 GB of RAM, 1280x800 screen resolution). It is equipped with a built-in FaceTime HD Camera, capable of recording video with 720p resolution. The computer was running Apple OS X (10.8.2) and Google Chrome web browser (build 30.0.1599.66) at the time of the experiment. The screen of the laptop was 90° with the base.

The room, where the experiments were held was equipped with mildly-bright flourescent lamps and had no natural light access.

4.2 Procedure

A simple test was created to measure the precision of gaze prediction. At first, linear calibration is performed in order to ensure that the gaze will be projected on the screen correctly. During the calibration phase, users are asked to look at the circles that appear on the screen. The calibration points are sequentially shown at the corners of the screen, each for 7 seconds (as mentioned in the *Calibration* section of the *Implementation*). After the calibration phase, the precision test begins. A monochromatic circle appears on the screen in a random location, 20 pixels from each border, being displayed for 7 seconds. This is repeated 10 times. For the precision testing, the average gaze point is being recorded. There was no indication of where the tracker predicted the gaze to the user in order to limit the bias imposed by this information.

We then calculate the error rate – the is the distance between average gaze point and the center of the associated test circle being shown on the screen. We also provide supplementary error rates – the horizontal and vertical displacements from the average gaze point to the test circle. The 10 error rates are then averaged to determine the accuracy and the standard deviation is calculated to determine the uncertainty in measurement.

Table 1: *Average Accuracies of Gaze Prediction*

Average Accuracy	**Value** (in)	**Uncertainty** (in)
Distance	2.1	0.1
Horizontal Displacement	1.4	0.3
Vertical Displacement	1.3	0.3

5 Results

We have conducted 5 tests. The results of these tests are shown **Table 1**. The distance accuracy refers to the average radius around the test point that tracker predicted the gaze to be. The horizontal and vertical displacments refer to the average component displacement from the test point and the predicted gaze point. These results are comparable to that of Holland and Komogorstev which implemented gaze prediction on a unmodified common tablet [Holland and Komogortsev 2012]. Since the constraints that exists when programming on a tablet and programming in JavaScript (JavaScript can run on a tablet through a web browser), the accuracy in the results can be compared. Their work performed with better accuracy by 0.7 inches during their first phase and by 0.6 inches in the second phase. The key factor in this difference is the existence of a computer vision library in Objective-C but not in JavaScript. Increasing the performance of the implemented computer vision algorithms will lead to better results.

In general, we believe that the received results provide a positive indication of the viability of conducting eye tracking in a web-based environment. This work concentrates on creating a proof of concept implementation of an in-browser, client side gaze prediction application. The conducted tests were meant to show the feasibility of the proposed concept. The results are preliminary due to limited testing and we believe that further testing and a usability study should be done to determine the absolute accuracy that is possible to achieve using `Camgaze.js`.

6 Discussion

6.1 Privacy Concerns

`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.

6.2 Limitations

Currently, Camgaze.js lacks any spatial awareness between the camera and the user. In some cases, due to specific change in align-

ment of the user the result from the eye tracker will be imprecise. Likewise, a change in head pose potentially disrupts the precision of `Camgaze.js` (e.g. if the user is looking at the center of the screen but starts to tilt his / her head down, the gaze metric will predict the user looking up due to the displacement from the center of the eye to the pupil).

The lack of well trained Haar classifiers also limits the ability of this work. The current classifiers available in JavaScript are not as highly trained and thus do not detect the eyes as frequently in different lighting conditions. This causes problems with `Camgaze.js` because a gaze metric can not be determined and a ponit on the screen is not mapped.

7 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 `Camgaze.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.

8 Conclusion

In this paper, we have described the design and implementation of a in-browser, client side eye tracker written in JavaScript. We have also discussed the privacy benefits, limitations, and impact such a system would have on current eye tracking applications. Through our evaluation, we have found our implementation to be accurate to a radius of 2.1 inches ($\pm$ 0.1).

The performance of the proposed system is influenced by many mutable limitations and therefore has the potential to impove. This work represents an eye tracking system that will have neglible deployment costs and higher accessibility. However there exists a price in accuracy that the authors feel is worth the benefits.

This material is based upon work supported by the National Science Foundation under Grant No. CNS- 1062743. The source code is made public on GitHub site and is available from http://github.com/wallarelvo/camgaze.js.

References

AHRENS, A. 1891. *Untersuchungen über die Bewegung der Augen beim Schreiben ...* C. Boldt.

DEVICEATLAS, 2013. Homepage. `https://deviceatlas.com/`.

DISRUPTIVE ANALYSIS, 2013. WebRTC Market Status & Forecasts: The hype is justified: it will change telecoms. `http://disruptive-analysis.com/webrtc.htm`.

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.

HOLLAND, C., AND KOMOGORTSEV, O. V. 2012. *Eye Tracking on Unmodified Common Tablets: Challenges and Solutions, In Proceedings of ACM Eye Tracking Research & Applications Symposium, Santa Barbara, CA*.

SAN AGUSTIN, J., SKOVSGAARD, H., HANSEN, J. P., AND HANSEN, D. W. 2009. Low-cost gaze interaction. In *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems - CHI EA '09*, ACM Press, New York, New York, USA, 4453.

SEWELL, W., AND KOMOGORTSEV, O. 2010. Real-time eye gaze tracking with an unmodified commodity webcam employing a neural network. In *Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems - CHI EA '10*, ACM Press, New York, New York, USA, 3739.

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