Tracking Library for the Web (tracking.js)

by

Eduardo A. Lundgren Melo

Submitted to the Center for Informatics
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

FEDERAL UNIVERSITY OF PERNAMBUCO

February 2013

Author .....
Center for Informatics
Mar 20, 2013

Certified by .....
Silvio de Barros Melo
Associate Professor
Dissertation Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Master Theses

Tracking Library for the Web (tracking.js)

by

Eduardo A. Lundgren Melo

Submitted to the Center for Informatics
on Mar 20, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

In this dissertation, I designed and implemented a tracking library for the web aiming to provide a common infrastructure to develop applications and to accelerate the use of those techniques on the web in commercial products. It runs on native web browsers without requiring third-party plugins installation. This involves the use of several modern browser specifications as well as implementation of different computer vision algorithms and techniques into the browser environment. Between the several techniques available there are algorithms that can be used for different applications, such as, detect faces, identify objects and colors and track moving objects. The source language of the library is JavaScript that is the language interpreted by all modern browsers. The majority of interpreted languages have limited computational power when compared to compiled languages, such as C. The computational complexity involved in visual tracking requires highly optimized implementations. Some optimizations are discussed and implemented on this work in order to achieve good results when compared with similar implementations in compiled languages. A series of evaluation tests were made, to determine how effective these techniques were on the web.

Dissertation Supervisor: Silvio de Barros Melo
Title: Associate Professor

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements [21] foo [21].

This master dissertation has been examined by a Committee as follows:

Professor Silvio de Barros Melo .....
Dissertation Supervisor
Associate Professor

Professor Veronica Teichrieb .....
Chairman, Dissertation Committee
Associate Professor

Professor Alvo Dumbledore .....
Member, Dissertation Committee
Hogwarts School of Witchcraft and Wizardry

Contents

List of Figures

List of Tables

List of Acronyms

AJAX	Asynchronous JavaScript and XML
BAST	Bug Report Analysis and Search Tool
BTT	Bug Report Tracker Tool
BRN	Bug Report Network
CCB	Change Control Board

Listings

Chapter 1

Introduction

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

1.1 Motivation

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

1.2 Problem Definition

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

1.3 Objectives

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

1.4 Dissertation Structure

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

Chapter 2

Basic Concepts

2.1 Web

2.1.1 Contextualization

Using concepts from existing hypertext systems, Tim Berners-Lee, computer scientist and at that time employee of CERN, wrote a proposal in March 1989 for what would eventually become the World Wide Web (WWW) [62].

The World Wide Web is a shared information system operating on top of the Internet [62]. Web browsers retrieve content and display from remote web servers using a stateless and anonymous protocol called HyperText Transfer Protocol (HTTP) [62]. Web pages are written using a simple language called HyperText Markup Language (HTML) [62]. They may be augmented with other technologies such as Cascading Style Sheets (CSS) [63], which adds additional layout and style information to the page, and JavaScript (JS) language [36], which allows client-side computation [62]. Client-side refers to operations that are performed by the client in a client-server relationship in a computer network. Typically, a client is a computer application, such as a web browser, that runs on a user's local computer or workstation and connects to a server when necessary. Browsers typically provide other useful features such as bookmarking, history, password management, and accessibility features to accommodate users with disabilities [21].

In the beginning of the web, plain text and images were the most advanced features available on the browsers [62]. In 1994, the World Wide Web Consortium (W3C) was founded to promote interoperability among web technologies. Companies behind web browser development together with the web community, were able to contribute to the W3C specifications [62]. Today's web is a result of the ongoing efforts of an open web community that helps define these technologies and ensure that they're supported in all web browsers [21]. Those contributions transformed the web in a growing universe of interlinked pages and applications, with videos, photos, interactive content, 3D graphics processed by the Graphics Processing Unit (GPU) [29], and other varieties of features without requiring any third-party plugins installation [24]. The significant reuse of open source components among different browsers and the emergence of extensive web standards have caused the browsers to exhibit "convergent evolution" [21].

The browser main functionality is to present a web resource, by requesting it from the server and displaying it on the browser window [65]. There are four major browsers used today: Internet Explorer, Firefox, Safari and Chrome. Currently, the usage share of Firefox, Safari and Chrome together is nearly 60% [65].

2.1.2 Browser Architecture

For any result or information presented in this work related to the browser environment, three mature browser implementations were selected. For each browser, a conceptual architecture was described based on domain knowledge and available documentation. Firefox version 16.0, Safari version 6.0.4 and Chrome version 25.0.1364 were used to derive the reference architecture because they are mature systems, have reasonably large developer communities and user bases, provide good support for web standards, and are entirely open source [62, 21]. The reference architecture for web browsers is shown in Figure 2-1. It comprises eight major subsystems plus the dependencies between them [21]:

1. User interface: includes the address bar, back and forward buttons, bookmark-

ing menu and other user interface elements of the browser. Every part of the browser display except the main window where you see the requested resource [21].

2. Browser engine: an embeddable component that provides a high-level interface for querying and manipulating the Rendering engine [21, 32].

3. Rendering engine: performs parsing and layout for HTML documents [21, 32].

4. Networking subsystem: used for network calls, like HTTP requests. It has platform independent interface and underneath implementations for each platform [21, 32].

5. JavaScript parser: parses and executes the JavaScript [36] code [21].

6. XML Parser: parses the HTML markup into a parse tree [24], HTML is rather close to XML [32, 24].

7. UI backend: provides drawing and windowing primitives, user interface widgets, and fonts. Underneath it uses the operating system user interface methods [21].

8. Data persistence: stores various data associated with the browsing session on disk, including bookmarks, cookies, and cache [21, 32].

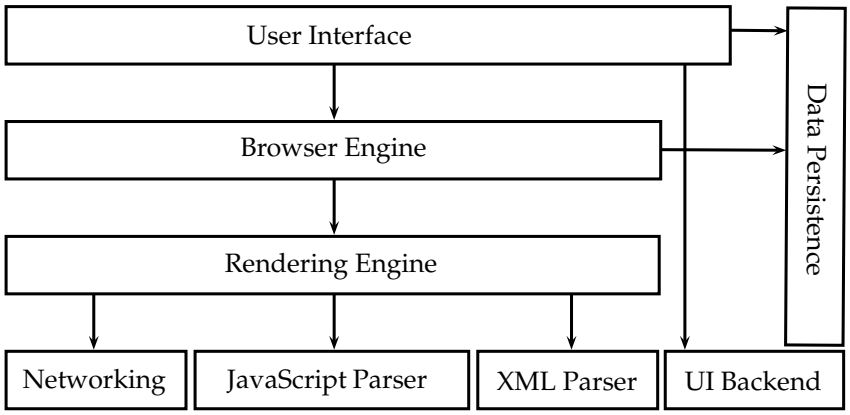

Figure 2-1: Reference architecture for web browsers.

Browser subsystems are swappable [21] and could vary for each browser vendor, platform or operational system. The browsers mostly differ between different vendors in subsystems (2) the Browser engine, (3) the Rendering engine, and (5) the JavaScript parser [52, 27, 28, 30]. In Firefox, subsystems (2) and (3) are known as Gecko [52, 50], Safari as WebKit [27, 28] and Chrome uses a fork of WebKit project called Blink [30, 31]. Those browsers subsystems, often called Browser engines, are shown on Figure 2-2.

Another common swappable subsystem is (5) the JavaScript parser. JavaScript [36] is a lightweight, interpreted, object-oriented language with first-class functions, most known as the scripting language for Web pages [50]. The JavaScript standard is ECMAScript [50, 36]. As of 2013, all modern browsers fully support ECMAScript 5.1. Older browsers support at least ECMAScript 3 [50, 36].

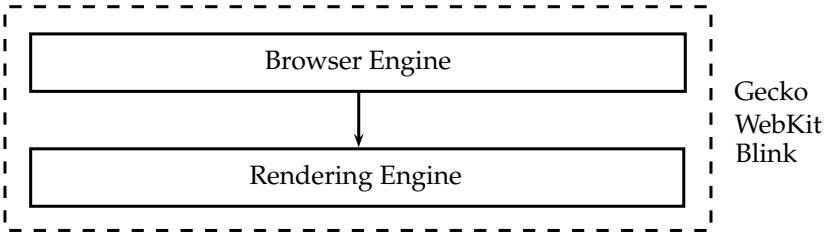

Figure 2-2: Reference architecture for browsers engines.

2.1.3 Audio and Video

Audio and video elements were introduced into the browsers by HTML5 specification [24]. Audio and video are HTML5 [24] features that attract a lot of attention. Often presented as an alternative to Flash [26] in the media, the video element has advantages due to its natural integration with the other layers of the web development stack such as CSS [63] and JavaScript [36] as well as the other HTML elements [62]. The three video formats supported by the three well known browsers cited in this dissertation [62, 32], webm (VP8 Vorbis) [17], mp4 (H.264 AAC) [37] and ogv (Theora Vorbis) [67]. The audio formats available are ogg (Theora Vorbis) [67] and mp4 (H.264 AAC) [37].

Audio and video addition to the browser environment was a good first step for visual tracking applications, on the other hand, the browsers were still not capable of capturing audio and video from the user's microphone and camera, respectively. Audio and video capture has been a limitation of web browsers for a long time [24]. For many years the authors had to rely on browser plugins, such as Flash [26] or Silverlight [48, 32]. With HTML5 [24], you can now add media to a web page with just a line or two of code [28]. Browser audio and video elements [24] are shown in Figure 2-3.

Figure 2-3: Video and audio HTML5 elements [28].

Missing media capturing is no longer a problem for the modern browsers cited in this dissertation [34, 24]. HTML5 specification [24] has brought a surge of access to device hardware, including Real-time Communication Between Browsers specification (WebRTC) [34] and with Media Capture and Streams specification [15]. Together, they provide a set of HTML5 [24] and JavaScript [36] APIs [34] that allows local media, including audio and video, to be requested from the user platform [62]. With Media Capture and Streams specification [15], the browser can access the microphone and camera input without requiring third-party plugin installation. It's available

directly into the browser.

The microphone and camera hardware access can be used in combination with the HTML5 [24] audio and video elements. An example of microphone and camera capturing using a video element is shown on Listing 2.1.

```
1 <video autoplay></video>
2 <script>
3   var video = document.querySelector('video');
4   navigator.getUserMedia({video: true, audio: true}, function(localMediaStream) {
5       video.src = window.URL.createObjectURL(localMediaStream);
6       video.onloadedmetadata = function(e) { alert('Ready to go.') };
7   }, onFail);
8 </script>
```

Listing 2.1: Capture microphone and camera and display using a HTML5 video element.

In Section 2.1.4, a new HTML5 [24] called canvas [12] is introduced. Trough the canvas element [12] video frames can be read, thus providing access to the raw binary data information of each frame [12, 24]. This raw binary data, also known as array of pixels [20], is extremely useful for visual tracking applications [60]. Since the area of study performed on this dissertation is based on visual tracking, from now on, it will focus on camera and video. There are several integration steps required from capturing video to reading the array of pixels [20]. In order to enable the browser to capture the user camera [15], stream the information into a video element [24], connect the video to a canvas element [12], to finally access the array of pixels of each video frame is a long run. The access flow of raw binary data captured from videos on modern browsers is shown on Figure 2-4 and comprises four major steps [34, 32]:

1. Hardware access: using HTML5 [24] new media capture and streams specification [15], the browser microphone and camera hardware is accessed [34].

2. Streaming: hardware streams audio and video to the browser UI elements [34].

3. UI elements: displays the video data stream into the browser viewport trough a HMTL5 video element [62].

4. Raw binary data: reads the video frames providing access to the array of pixels trough a HTML5 canvas element [62].

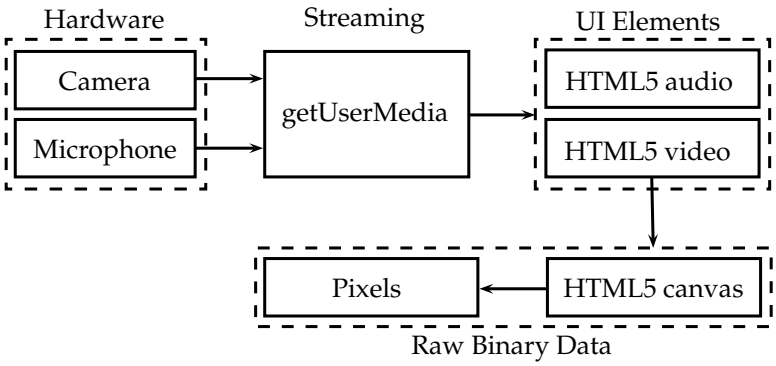

Figure 2-4: Access flow of raw binary data captured from videos on modern browsers.

2.1.4 Canvas Element

The canvas [12] is an HTML5 [24] element that provides scripts with a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, art, or other visual images on the fly [12].

Authors should not use the canvas element in a document when a more suitable element is available, e.g. it is inappropriate to use a canvas element to render a page heading. The usage of canvas conveys essentially the same function or purpose as the canvas bitmap. Listing 2.2 shows an example of a basic canvas element HTML markup given a width and height in pixels.

```
1 <canvas width="200" height="200"></canvas>
```

Listing 2.2: The HTML canvas element markup

The canvas [12] is a two-dimensional grid that could be described as a simple computer graphics coordinate system [22]. Normally one unit in the grid corresponds to one pixel on the canvas. The origin of this grid is positioned in the top left corner

coordinate $(0, 0)$. All elements are placed relative to this origin. So the position of the top left corner of the blue square shown on Figure 2-5 becomes x pixels from the left and y pixels from the top coordinate (x, y). The canvas coordinate space is shown on Figure 2-5 [51].

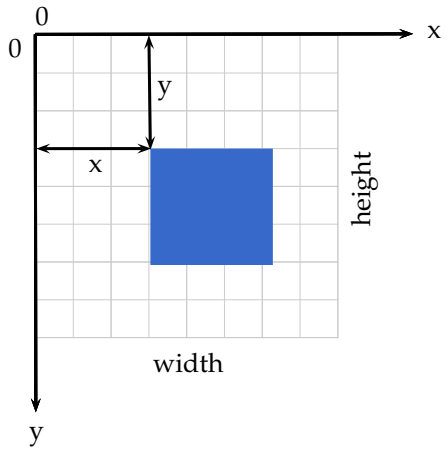

Figure 2-5: The canvas coordinate space.

For each canvas element a "context" is available. The canvas context provides the drawing context that can be accessed and JavaScript [36] commands can be invoked to draw or read data [12]. Browsers can implement multiple canvas contexts and the different APIs provide the drawing functionality. Most of the major browsers include the 2D canvas context capabilities. Individual vendors have experimented with their own three-dimensional canvas APIs, but none of them have been standardized. The HTML5 [24] specification notes, "A future version of this specification will probably define a 3D context". Even though 3D context is not available in most part of the major browsers, three-dimensional applications are already being developed based on the 2D canvas context.

Is mandatory the use of the canvas element [12] to develop visual tracking applications on the web, since it's the only way to read video frames array of pixels without any plugin in the browser environment. For more information about HTML5 [24] video element see Section 2.1.3. Canvas provides APIs to: draw basic shapes, images, videos frames, Bézier [55] and quadratic curves [55, 22]; apply transformations,

translate, rotate, scale and read the array of pixels information.

2.1.5 JavaScript Typed Arrays

The JavaScript language [36] is intended to be used within some larger environment, be it a browser, server-side scripts, or similar [21]. JavaScript [36] core language features comprises few major features:

1. Functions and function scope: function is a subprogram that can be called by external code, functions have a scope they reference for execution.

2. Global objects: refer to objects in the global scope, such as general-purpose constructors (*Array, Boolean, Date* etc.) and typed array constructors (*Float32Array, Int32Array, Uint32Array* etc.).

3. Statements: consist of keywords used with the appropriate syntax (*function, if...else, block, break, const, continue, debugger etc.*).

4. Operators and keywords: arithmetic operators, bitwise operators, assignment operators, comparison operators, logical operators, string operators, member operators and conditional operator [51].

As web applications become more and more powerful, adding features such as audio and video manipulation, access to raw data using canvas (Section 2.1.3), and so forth, it has become clear that there are times when it would be helpful for JavaScript [36] code to be able to quickly and easily manipulate raw binary data [12, 23]. In the past, this had to be simulated by treating the raw data as a string and using the *charCodeAt()* method to read the bytes from the data buffer [51, 23]. However, this is slow and error-prone, due to the need for multiple conversions, especially if the binary data is not actually byte-format data, but, for example, 32-bit integers or floats. Superior, and typed data structures were added to JavaScript specification, such as JavaScript typed arrays [51, 36].

JavaScript typed arrays provide a mechanism for accessing raw binary data much more efficiently [51, 23]. This dissertation takes advantage of typed arrays in order

to achieve acceptable performance and robustness on the web of complex algorithms implementations.

Typed Arrays Performance Benchmark

A performance benchmark comparing regular *vs* typed arrays were executed on the three well known open-source browsers, Firefox, Safari and Chrome. The comparison was executed on Mac OS X 10.8.3, 2.6 GHz Intel Core i7 16 GB 1600 MHz RAM. The array types selected were the not strongly typed *Array*; *Float32Array*, represents an array of 32-bit floating point numbers; *Uint8Array*, represents an array of 8-bit unsigned integers [51].

In the benchmark, for each array type a read and a write operation were executed $100,000$ times. In order to not compromise benchmark results caused by run-time type conversion [36], the write value used for each array type were proper selected, e.g. *Number* 1.0 was used for regular arrays *Array*, *Number* 1.0 was used for *Float32Array*, and unsigned *Number* 1 for *Uint8Array*. Regular *vs* typed arrays performance benchmark is shown in Figure 2-6 [1].

As conclusion, typed arrays provides faster read and write operations than regular arrays in JavaScript, i.e. 7872 *ops/sec* for unsigned array *vs* 4437 *ops/sec* for regular arrays in Firefox browser, similar behavior is noticeable on Safari and Chrome, thereby float and unsigned arrays are vastly used on complex algorithms implementations on the web.

What's the relation between Typed Arrays and Canvas?

Reading and writing raw binary data [12, 23] using typed arrays only solves part of the problem of manipulating video and images data. The other missing feature was solved by HTML5 [24] canvas element, which one important feature is to provide access to the array of pixels of those medias [12, 24]. The raw binary data is used by visual tracking algorithms. For more information see Section 2.1.3.

Videos and images pixels can be draw on a canvas bitmap [12]. Canvas raw binary data can be accessed from the canvas JavaScript [36] API as an object of

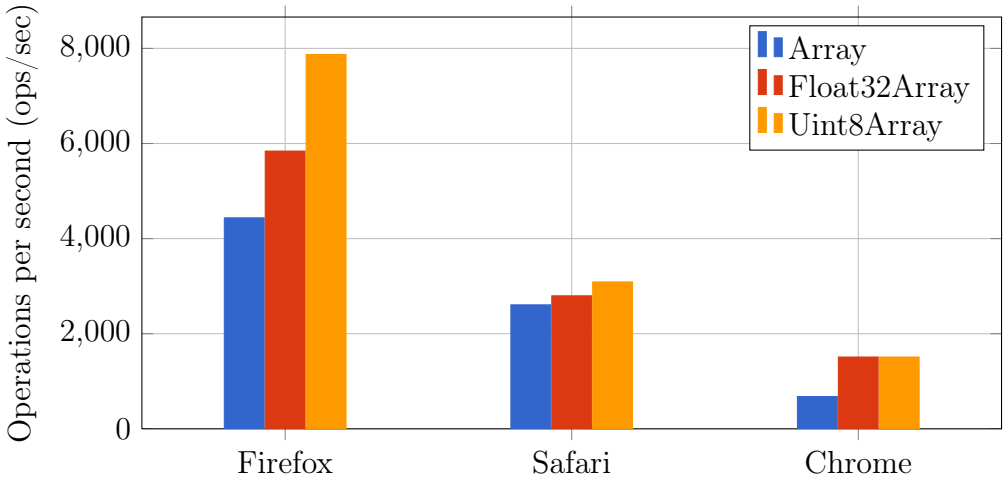

Figure 2-6: Regular *vs* typed arrays performance benchmark [1].

type *ImageData*. Each object has three properties: width, height and data [12, 51]. The data property is of type *Uint8ClampedArray* [23] that is a one-dimensional array containing the data in RGBA [20] order, as integers in the range 0 to 255. The *Uint8ClampedArray* interface type is specifically used in the definition of the canvas element's 2D API and its structure is similar to the previous shown typed array *Uint8Array*.

The *ImageData* data property, or array of pixels, is in row-major order [12, 51], a multidimensional array in linear memory. For example, consider the 2×3 array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, in row-major order it is laid out contiguously in linear memory as $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$. Each array value is represented as integers between 0 and 255 [12, 51], where each four-integer group represents the four color channels of one pixel: red, green, blue and alpha (RGBA). While RGBA [20] is sometimes described as a color space, it is actually simply a use of the RGB color model. This array linear typed structure improves read and write performance, since JavaScript [36] is as an interpreted language [51], previous knowledge of the type results in faster language execution [23]. An example of the canvas image data array of pixels is shown on Figure 2-7.

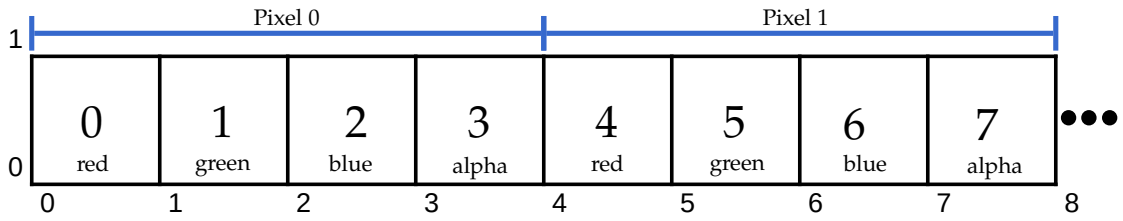

Figure 2-7: The canvas image data array of pixels.

2.2 Visual Tracking

2.2.1 Contextualization

Tracking an object in a video sequence means continuously identifying its location when either the object or the camera are moving. There are a variety of approaches, depending on the type of object, the degrees of freedom of the object and the camera, and the target application [44, 60]. Figure 2-8 shows a person walking on the street being tracked despite its level of occlusion by another object that is not of interest for the tracking.

Figure 2-8: Example of an accurate object tracking robust to occlusion [39].

When 2D tracking is used, the goal is to retrieve a 2D transformation from the object projected on the captured image that best represents the motion that occurred. 2D tracking happens on the image space, ignoring the deepness of 3D world [45]. Many models can be applied in order to handle appearance changes due to perspective effects or deformations. The homography is one of the most used transformations regarding 2D tracking of planar objects, since it is generic enough to deal with all possible perspective effects [47]. This dissertation focuses on tracking of planar objects since it serves as basis for more complex tracking approaches (for example, 3D tracking).

In computer vision, general video analysis can be broken down into three different steps: detection of interesting moving objects, frame to frame tracking of those objects and analysis of their tracks to recognize possible behaviors [69]. Because of such use in computer vision, object tracking is important in the following major areas:

1. augmented reality [3];
2. 3D reconstruction [70];
3. motion-based recognition [10];
4. automated surveillance [38];
5. video indexing [38];
6. human-computer interaction [56];
7. traffic monitoring [18];
8. vehicle navigation [40].

Examples of computer vision applications in each of the previously described areas is shown in Figure 2-9. Besides these areas, object tracking algorithms can also be used in AR systems that require real-time registration of the object to be augmented.

Vision has the potential to yield non-invasive, accurate and low-cost solutions for tracking objects. This dissertation introduce a tracking library for the web, that provides visual tracking techniques. Currently, the library provides three main modules, they are: markerless tracking (Section 3.2), rapid object detection (Section 3.3) and color tracking (Section 3.4).

2.2.2 Tracking in Augmented Reality Applications

Imagine a technology in which you could see more than others see, hear more than others hear, and even touch things that others cannot [43]. A technology able to perceive computational elements and objects within our real world experience that

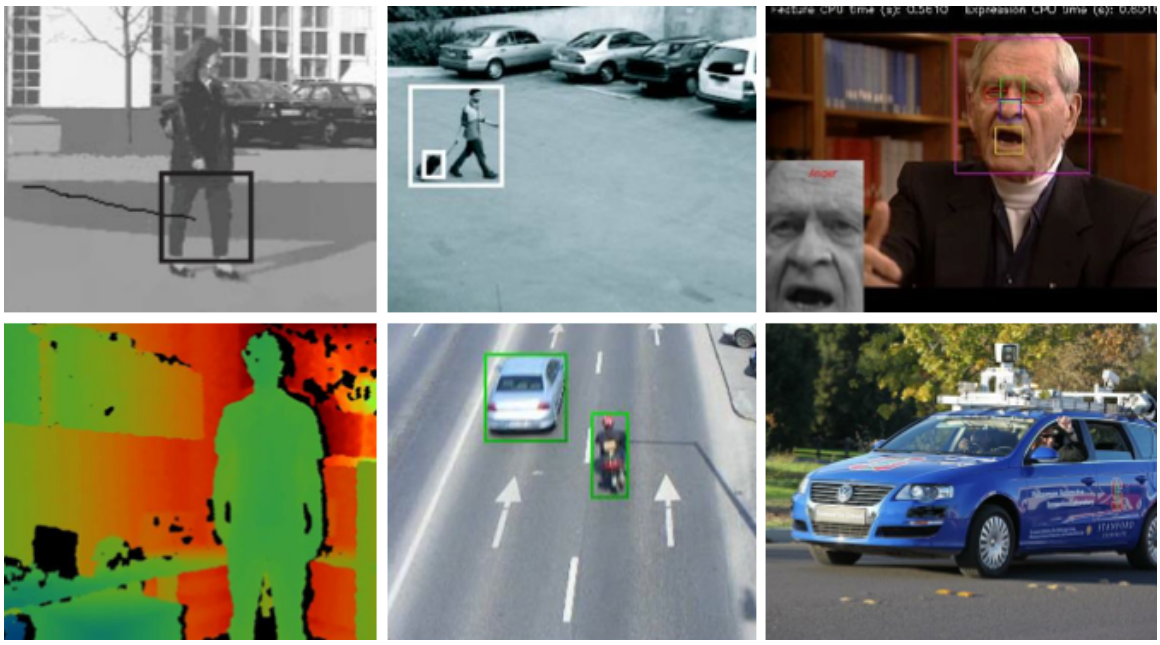

Figure 2-9: Computer vision applications: motion-based recognition (top left) [10]; automated surveillance (top center) [38]; video indexing (top right) [38]; human-computer interaction (bottom left) [56]; traffic monitoring (bottom center) [18]; vehicle navigation (bottom right) [40].

help us in our daily activities, while interacting almost unconsciously through mere gestures [43, 60]. With such technology, mechanics could see instructions what to do next when repairing an unknown piece of equipment, surgeons could take advantage of Augmented Reality (AR) while performing surgery on them and we could read reviews for each restaurant in the street we're walking in on the way to work [43]. On the reality-virtuality *continuum* by Milgram and Ki [49], AR is one part of the general area of mixed reality. Both virtual environments (or virtual reality) and augmented virtuality, in which real objects are added to virtual ones, replace the surrounding environment by a virtual one. In contrast, AR provides local virtuality. The reality-virtuality *continuum* is shown on Figure 2-10.

AR applications involve superimposing computer-generated content on real scenes in real-time [3]. Tracking is a critical component of most AR applications, since the objects in both real and virtual worlds must be properly aligned with respect to each other in order to preserve the idea of the two worlds coexisting [70].

Traditionally used in AR, vision-based tracking is described as two main steps:

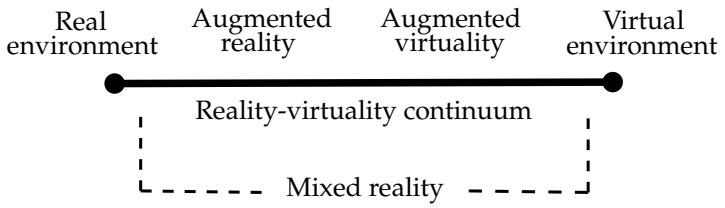

Figure 2-10: Reality-virtuality continuum [4].

extract information from the input video using image processing algorithms and perform the pose estimation itself, or, in other words, find the transformation that best maps the object model from one frame to the next one [66].

The information that comes from the input image is basically composed of image features that simplifies extraction from the scene. Both Marker-based AR and Markerless AR [46] are based on this principle, with the difference that the first one adds artificial templates, such as special markers that do not originally belong in the scene in order to enable the tracking [66]. These templates are often called fiducials and they constitute image features easy to extract as well as reliable measurements for the pose estimation [11, 44]. It is therefore much more desirable to rely on naturally present features, such as edges, corners, or texture [44], those are known as Markerless AR. In this case, the template to be used is the object to be tracked itself, using its natural features.

This dissertation will provide more detail regarding the template-based approaches developed and optimized aiming web environment. The template-based tracking techniques do not necessarily rely on local features such as edges or other features, but rather on global region tracking through the use of the whole pattern of the object to be tracked. These methods can be useful in handling more complex objects that are difficult to model using local features due to their repeatability, for example. Such scenarios can be computationally expensive, but in some cases effectively formulated [69].

2.2.3 Which devices could use tracking.js?

Different devices such as mobile phones, notebooks, and even head-worn [4] (Google Project Glass [33]), provides an embedded web browser capable to run JavaScript [36] and HTML5 [24]. There are several displays devices that could be used on visual tracking or AR applications, Benford [4], describes that they could be divided as follows [4]:

1. Visual display [4]
    (a) Video see-through [4];
    (b) Projective [4];
    (c) Monitor [4].
2. Display positioning [4]
    (a) Head-worn [4];
    (b) Hand-held [4];
    (c) Spatial [4].

The possibility to use this work as a cross-platform tracking library is a reality. The browser environment is evolving fast and due to the cross-platform ability, JavaScript [36] is becoming a popular solution for multiple devices and platforms. In a near future, other devices and visual displays could, potentially, have embedded browser versions as well and they could all benefit from *tracking.js* library.

2.2.4 Discussion

In this dissertation, it was designed and implemented tracking library for the web aiming to provide a common infrastructure to develop applications and to accelerate the use of those techniques on the web in commercial products. Thus, the techniques implemented in this work were chosen aiming to cover common use-cases, such as: facilitate user interaction with the computer trough color tracking (Section 3.4); Tracking complex objects in a scene trough markerless tracking (Section 3.2); and track

humans body parts, e.g. faces and eyes, trough rapid object detection (Section 3.3). It runs on native web browsers without requiring third-party plugins installation, therefore any browser ready device can eventually use the proposed cross-platform code base to develop AR applications. In order to develop for different devices different skills are required, since APIs and programming languages may differ between them [51, 36]. In the current day, the most popular devices, such as smart phones, tablets, computers, notebooks and HDM (i.e. Google Glass [33]) [4] are browser ready [24]. They all could benefit from a reusable, cross-platform library for AR applications.

Chapter 3

Tracking Library for the Web (tracking.js)

3.1 Contextualization

The desktop platform is the target environment most commonly addressed when developing AR systems. However, depending on the requirements of an AR application, the use of different execution platforms may be necessary. If the system has to be published to several users, the web platform shows to be more adequate, where the application is executed through the Internet in a web browser [54]. The use of markerless tracking, which is based on natural features of the scene, has also been gaining more space on web targeted AR applications for advertising. The media used in this kind of application needs to be as appealing as possible in order to catch consumers' attention. Markerless tracking satisfies this requirement, since the idea of having a real scene augmented with virtual objects without any artificial elements such as markers added to the environment is very attractive. In addition, the product being advertised can be tracked and augmented with virtual elements. Browsers are evolving very fast when compared to the the previous decade [24]. JavaScript language [36, 51] wasn't prepared to handle typed data structures [23] able to manipulate raw binary data safely [12], all the computational complexity required by AR algorithms was too much for that growing environment. Browsers weren't able to capture audio

and video [15, 34] natively, without plugin installation [26], an essential feature for AR applications. This reality has changed, this involves the use of several modern browser specifications as well as implementation of different computer vision algorithms and techniques into the browser environment taking advantage of all those modern APIs [24, 62].

In this context, this dissertation aims to present the implementation and evaluation of a solution regarding tracking techniques for web targeted AR. The available algorithms and techniques can be used for different applications, such as, detect faces, identify objects and colors and track moving objects. The solution is called *tracking.js*. Some optimizations are discussed and implemented in this work in order to achieve good results when compared with similar implementations in compiled languages.

3.1.1 Related Work

There are not many web based RA solutions available and registered in the literature. The ones available are mainly focused on fiducial markers [11], such as FLARToolKit [68] and JSARToolkit [41], they both are ports of ARToolKit [25]. ARToolKit is a desktop library which is useful to make vision-based AR applications. The Metaio company developed Unifeye Viewer [35], a proprietary plug-in for Flash [26] that allows the utilization of markerless AR applications on the web. In order to run Flash based applications, the installation of its plugin is required. Third-party plugins, such as Flash, are in decadency on modern and mobile web browsers, instead JavaScript [36] based solutions are preferred, since they can run in any modern browser without requiring any user effort of installing external software. Some smart-phones do not even support Flash plugin into their browsers, e.g. Safari for mobile [27] is one example of a mobile browser that has banned Flash [26].

1. FLARToolKit: is a port of the well-known ARToolKit [25] marker tracking library to ActionScript [26], which is the language utilized in the development of Flash applications for the web. This was the first initiative towards AR solutions for the web [54]. Using FLARToolKit [68], is possible to develop AR

applications that run on client's browser. A marker based AR example for the web, developed for a marketing campaign of General Electric's company using FLARToolKit, is shown on Figure 3-1.

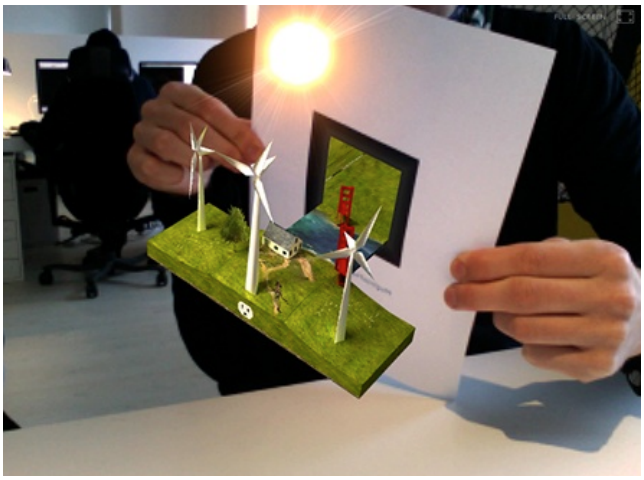

Figure 3-1: Marker based AR for the web using FLARToolKit.

2. JSARToolkit: is a JavaScript [36] port of FLARToolKit [68], operating on canvas images [12] and video element [24] contents, provides another marker tracking library. This was the first, open-source, JavaScript based, AR solution available for the web. A marker based AR example for the web using JSARToolKit is shown on Figure 3-2.

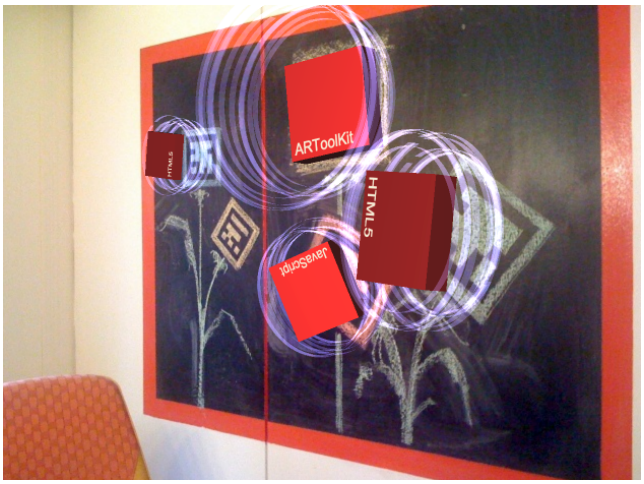

Figure 3-2: Marker based AR for the web using JSARToolKit.

3. Unifeye Viewer: from Metaio company, offers a robust markerless tracking solution for the web. Unifeye [35] also depends on Flash [26] plugin in order to run on web browsers. A similar example of General Electric's marker based solution, this time markerless based, is shown on Figure 3-3. Note that the 3D image is projected over a magazine cover instead of a fiducial marker [11].

Figure 3-3: Markerless example of image projected over a magazine cover using Unifeye Viewer solution.

There is a disadvantage of using marker based AR. Depending on an artificial marker in order to augment the scene with virtual elements is counterintuitive. Commonly, web applications are utilized by novice users that do not have sufficient technical knowledge to perform manual setup, such as print fiducial markers or perform manual initialization for the tracking. FLARToolKit [68] and JSARToolkit [41] are both marker based techniques, using Flash [26] and JavaScript [36], respectively. FLARToolKit has one more issue which is dependency on Flash plugin installation. Unifeye Viewer by Metaio, was the only existing solution that provided markerless tracking for the web, although it uses Flash, excluding it from a potential competitor of *tracking.js*. Markerless tracking techniques aim to not depend on any artificial marker or advanced user initialization. The space on web targeted AR applications for advertising is gaining more space and the media used in this kind of application needs to be as appealing as possible [54]. Making markerless tracking a suitable technique to such applications.

The solution proposed in this dissertation, *tracking.js*, provides the first known, open-source, markerless tracking solution for the web that runs entirely in JavaScript [36] and HTML5 [24].

3.1.2 Library Modules

The proposed library is divided in modules in order to allow extension and addition of new features, such as new RA techniques or math utilities. For a better understanding of the library architecture, the current implementation is divided in two packages separating base from visual tracking classes. Base classes modules are shown in Figure 3-4 and visual tracking classes in Figure 3-5.

To develop AR applications using only raw JavaScript [36] APIs [51] could be too verbose and complex, e.g. capturing users' camera and reading its array of pixels. The big amount of steps required for a simple task makes web developers life hard when the goal is to achieve complex implementations. Some level of encapsulation is needed in order to simplify development. The proposed library provides encapsulation for common tasks on the web platform.

The two main available packages splits base from visual tracking classes. Furthermore, each class of those packages are described. Let's start with the base classes:

1. Math: provides common math utilities optimized for the web, such as geometry, linear algebra [22] and hamming operations. Typed arrays [23] are used in order to optimize performance, see subsection 2.1.5 for more information about typed arrays.

2. Attribute: allows developers to add attributes to any class through an Attribute interface. The interface adds get and set methods to your class to retrieve and store attribute values, as well as support for change events that can be used to listen for changes in attribute values.

3. DOMElement: provides a way to create, and manipulate HTML [24] DOM nodes [62]. Each DOMElement instance represents an underlying DOM node.

In addition to wrapping the basic DOM API and handling cross browser issues, Nodes provide convenient methods for managing styles and subscribing to events.

4. Canvas: provides an utility class to create, and manipulate HTML5 [24] canvas element [12]. Each Canvas instance represents an underlying canvas DOM node. In addition to wrapping the basic DOM API [62], also provides methods to extract via *getImageData* method, to loop via *forEach* method, and to set the canvas array of pixels via *setImageData* method.

5. Video: provides an utility class to create, and manipulate HTML5 [24] video element. Each Video instance represents an underlying video DOM node [12]. In addition to wrapping the basic DOM API [62], also provides methods to *play*, *pause* and register tracker algorithms via *track* method. See subsection 2.1.3 for more information about video element.

6. VideoCamera: extends all functionalities from Video class with the addition of capturing the user camera via *capture* method. The underlying implementation uses WebRTC [34] and Media Capture and Streams [15] specifications.

Visual tracking classes includes several computer vision algorithms, such as FAST [57], BRIEF [9] implementations, homography estimation and others. As the library grows, many other computer vision algorithms are going to be added to the library, such as 3D pose calculation.

1. FAST: provides an implementation of Features from Accelerated Segment Test (FAST) [58] for features detection via *findCorners(data, threshold)* method, where *data* is the *ImageData* of the canvas [12] frame. It also depends on a *threshold* argument. The pixel at p is the center of a candidate corner classified if the concentric contiguous arcs around p are different by more than the *threshold*, see Figure 3-7.

2. BRIEF: provides an implementation of Binary Robust Independent Elementary Features (BRIEF) [9] for feature extraction via *getDescriptors(data, corners)*

method and matching via *match(c1, d1, c2, d2)*, where *data* is an *ImageData*, and $c1$ and $c2$ are the found corners array return by *FAST.findCorners* method and $d1$ and $d2$ are feature descriptors array return by *BRIEF.getDescriptors* method.

3. RANSAC: provides an interface used to achieve robust estimation method for homographies and camera pose. There are two available estimation methods implemented that inherits from RANSAC [22], Homography and Pose.

4. Homography: provides an API to estimate a homography matrix H between images by finding feature correspondences in those images.

5. Pose: TODO.

6. ViolaJones: TODO.

7. Color: TODO.

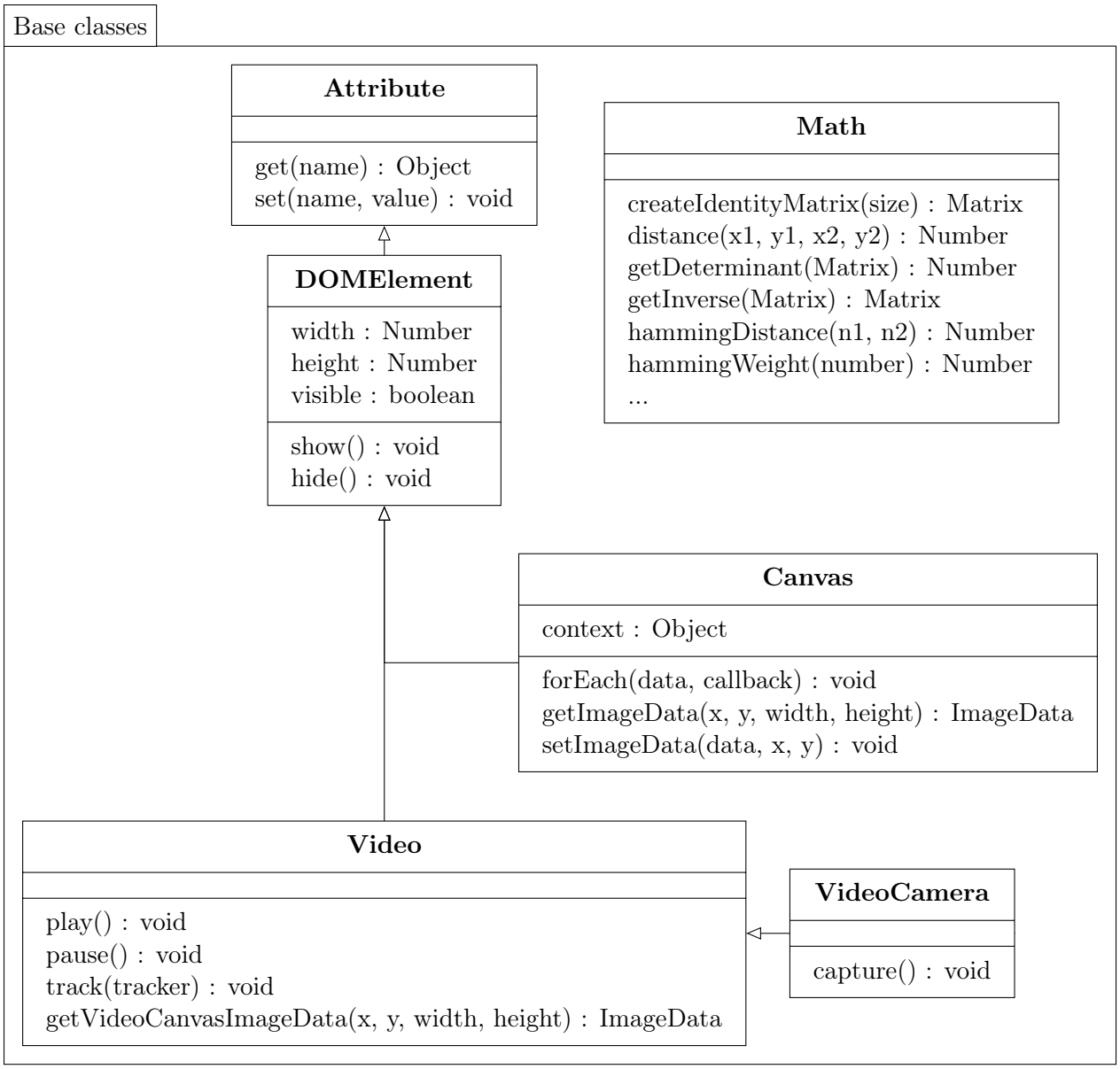

Figure 3-4: Base classes of tracking.js library.

3.2 Markerless Tracking Algorithm

3.2.1 Contextualization

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

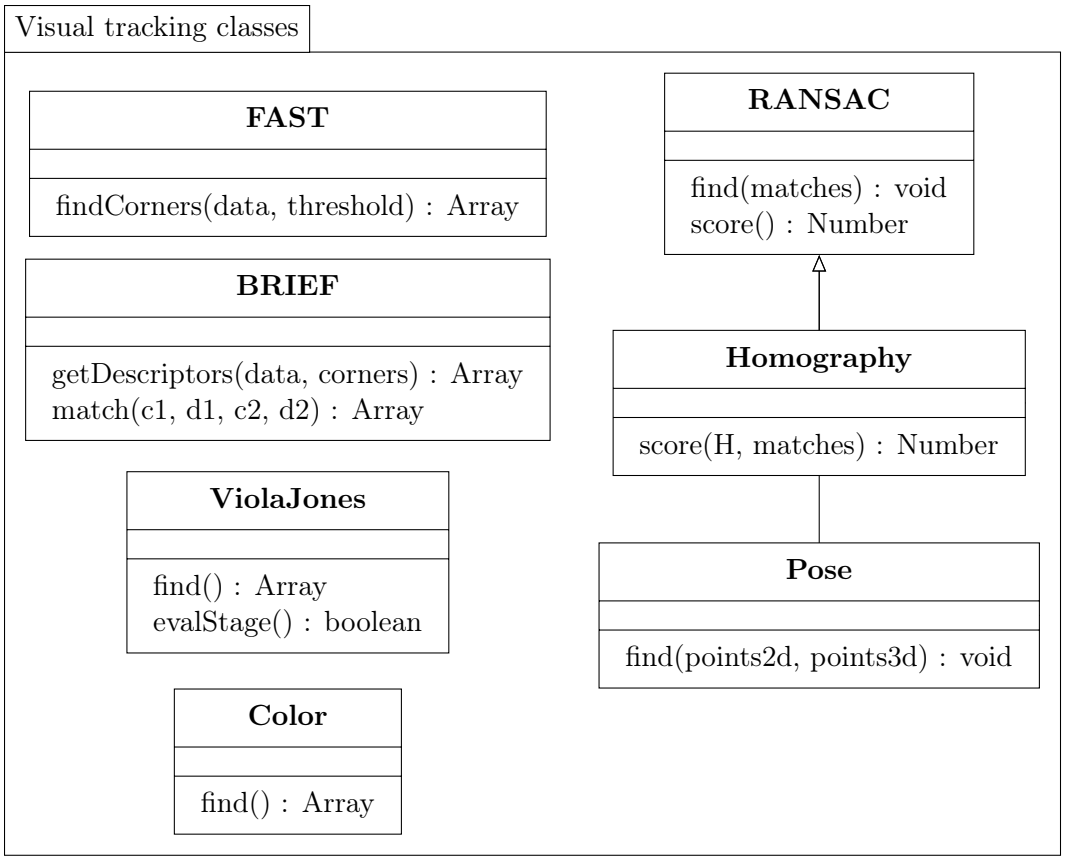

Figure 3-5: Visual tracking classes of tracking.js library.

3.2.2 Feature Detector

This technique relies on detecting individual features across images and are therefore easy to increase robustness against partial occlusions or matching errors. Illumination invariance is also simple to achieve. Feature points detection is used as the first step of many vision tasks such as tracking, localization, image matching and recognition [44]. In this work we call "feature" or "keypoint" to refer to a point of interest in two dimensions (Figure 3-6).

For each frame, the object features are matched by localizing feature templates in search windows around hypothesized locations [44]. The method to extract feature points proposed is Features from Accelerated Segment Test (FAST) [58]. FAST [57] hypothesizes the matches using corner detection. A large number of corner detectors exists in the literature. However, in order to allow AR solutions to be developed on top of this work, we have a strong interest in real time frame rate applications which

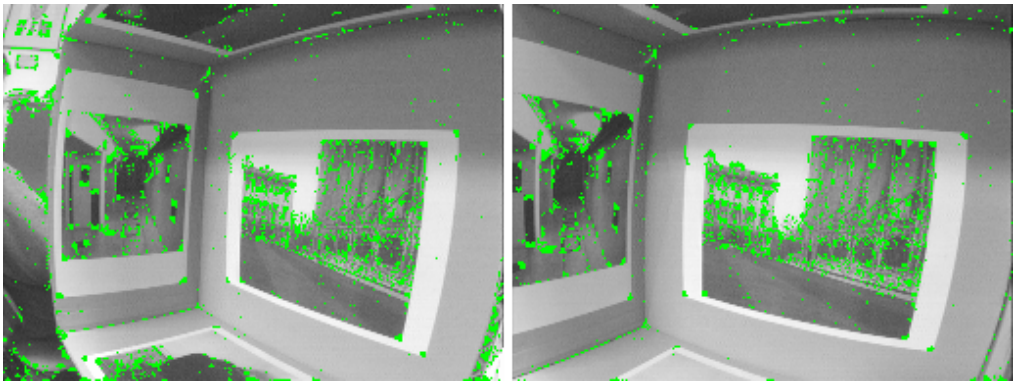

Figure 3-6: Image features detected on two different frames, green pixels represents found keypoints.

computational resources are required requisites. The approach proposed by FAST allows the detector to produce a suite of high-speed detectors which we currently use for real-time tracking and AR label placement [9]. In particular, it is still true that when processing live video streams at full frame rate, existing feature detectors leave little if any time for further processing, even despite the consequences of Moore's Law [58].

A number of the detectors described below computes a corner response: (1) Edge-based corner detectors, correspond to the boundary between two regions; (2) Gray-level-derivative based detectors, the assumption that corners exist along edges is an inadequate model for patches of texture and point-like features, and is difficult to use at junctions. Therefore a large number of detectors operate directly on gray level images without requiring edge detection; and (3) Direct gray level detectors, another major class of corner detectors works by examining a small patch of an image to see if it "looks" like a corner [58].

The dissertation choice was (3) Direct gray level detectors, since its robustness against partial occlusions or matching errors, illumination invariance is simple to achieve and the computational complexity of the technique is low. Despite the design of FAST for speed, this detector has excellent repeatability, in addition, a variation of FAST technique that uses machine learning (FAST-ER [57]) can be implemented in a future work, providing dramatic improvements in repeatability over FAST-9 (especially in noisy images).

It works by testing a small patch of an image to see if it could be a corner. The detector is evaluated using a circle surrounding the candidate pixel, the test is based on whether the concentric contiguous arcs around the pixel are significantly different from the central pixel p. To classify p as a corner should exist a set of n contiguous pixels in the circle which are all brighter than the intensity of the candidate pixel $I_p + t$ (threshold), or all darker than $I_p - t$ [58].

The number of contiguous tested pixels could vary accordingly [58], being more common to be FAST-12 or FAST-9. Empirically, FAST-9 showed to have a good repeatability and a better efficiency on the web. The repeatability of found feature points is also important because determines whether the technique is useful in a real-world application.

This detector itself exhibits high performance, but there are several weaknesses: (1) This high-speed test does not reject as many candidates; (2) The efficiency of the detector will depend on the ordering of the questions and the distribution of corner appearances; and (3) Multiple features are detected adjacent to one another [58].

On Figure 3-7, the highlighted squares are the pixels used in the corner detection. The pixel at p is the central pixel. The arc is indicating that the dashed line passes through FAST-n, let n be 9 or 12 contiguous pixels which are brighter or darker than p [58].

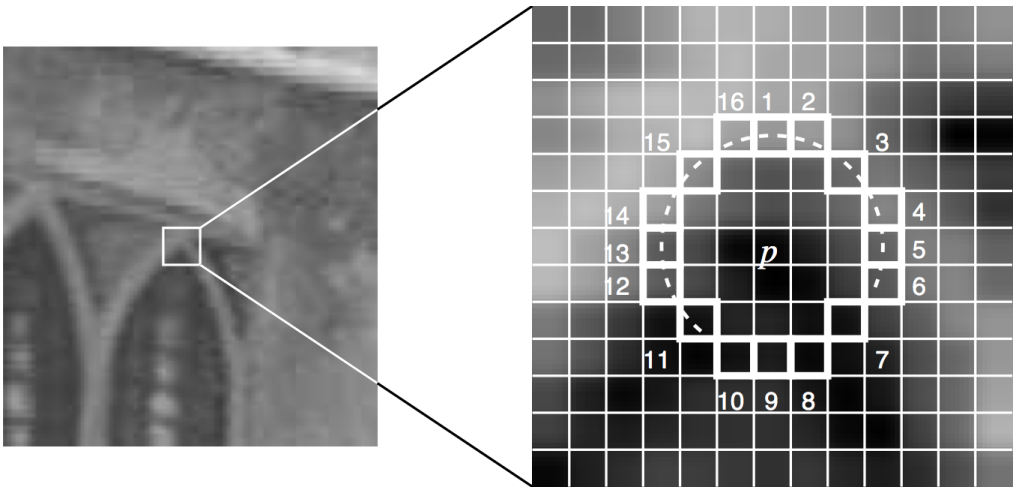

Figure 3-7: Point segment test corner detection in an image patch [33].

3.2.3 Feature Extractors

To estimate motion, one can then match sets of features $\{m_i\}$ and $\{m'_j\}$ extracted from two images taken from similar, and often successive, viewpoints. A classical procedure [9] runs as follows. For each point $\{m_i\}$ in the first image, search in a region of the second image around location $\{m_i\}$ for point $\{m'_j\}$. The search is based on the similarity of the local image windows, also knowns as kernel windows, centered on the points, which strongly characterizes the points when the images are sufficiently close [44].

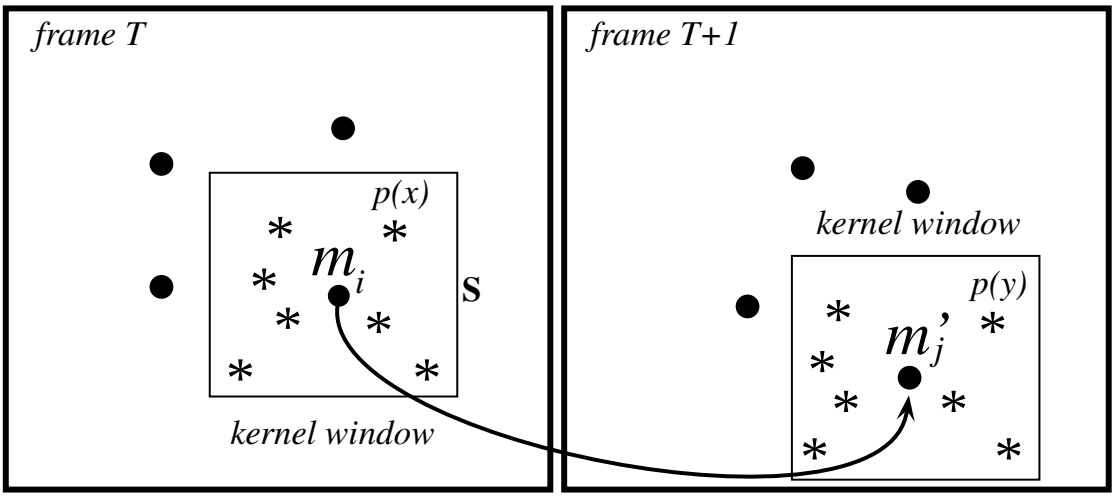

Figure 3-8: BRIEF [44] feature extractor.

The feature matching used in the case studies performed in this work searches for correspondent points in the current frame. Only points that are highly descriptive invariant features, called keypoints, are tested. Those keypoints were detected by using FAST [58] in Section 3.2.2. After the keypoints are detected they need to be described and the respective matching point should be found. Since web and handled devices have limited computational power, having local descriptors that are fast to compute, to match and being memory efficient are important aspects, and for that reason, it was used an efficient method called Binary Robust Independent Elementary Features (BRIEF) [9].

To generate the binary string for each keypoint found in the smoothed frame, the individual bits are obtained by comparing the intensities of pairs of points, $(\mathbf{p}; x, y)$, represented by $*$ symbol on Figure 3-8, along the kernel window centered on each keypoint without requiring a training phase. Empirically, this technique shows that 256 or even 128 bits [9], often suffice to obtain very good matching results. In order to have better recognition rates, the best spatial arrangement of the tested ($\mathbf{x}$, $\mathbf{y}$)-pairs of points are reached when selected based on an isotropic Gaussian distribution [9]. To compute the Gaussian distribution can be time consuming. As an optimization proposed by this work, the Gaussian distribution could be simply replaced by a uniform version of JavaScript *Math.random()* function. Function *Math.random()* should return an evenly-distributed Number X such that $0.0 \leq X < 1.0$ and X is commonly seeded from the current time [36]. The locations of ($\mathbf{x}$, $\mathbf{y}$)-pairs are evenly distributed over the patch and tests can lie close to the patch border, i.e. $(-\frac{S}{2}, \frac{S}{2})$.

To generate the binary strings it is defined test τ on patch $\mathbf{p}$ of size $\mathbf{S} \times \mathbf{S}$ as

$$\tau(\mathbf{p}; x, y) := \begin{cases} 1 & \text{if} \quad \mathbf{p(x)} < \mathbf{p(y)}, \\ 0 & \text{otherwise} \end{cases}$$

where $\mathbf{p(x)}$ is the pixel intensity. The set of binary tests is defined by the n_d ($\mathbf{x}$, $\mathbf{y}$)-location pairs uniquely chosen during the initialization. The n_d-dimensional bit-string is our BRIEF descriptor for each keypoint

$$f_{n_d}(\mathbf{p}) := \sum_{1 \leq i \leq n_d} 2^{i-1} \tau(\mathbf{p}; x, y).$$

In [9], $n_d = 128, 256, 512$ were used in the tests and any of those values yield good compromises between speed, storage efficiency, and recognition rate. In this work, $n_d = 128$ was used, since it presented good matching results and performance. The number of bytes required to store the descriptor can be calculated by $k = n_d/8$, proving that BRIEF is also a memory-efficient method. Detailed results can be found in Chapter 4.

Once each keypoint is described with its binary string [9], they need to be com-

pared with the closest matching point. Distance metric is critical to the performance of intrusion detection systems. Thus using binary strings reduces the size of the descriptor and provides an interesting data structure that is fast to operate whose similarity can be measured by the Hamming distance which, on desktop implementations, the computation time could be driven almost to zero by using the POPCNT instruction from SSE4.2 [13]. Only the latest Intel Core i7 CPUs support this instruction.

The Hamming distance is an important step on feature matching, it provides a fast and memory-efficient way to calculate distance between binary strings. Given two image patches x and y, denote their binary descriptors as $b(x) \in \{0,1\}^n$ and $b(y) \in \{0,1\}^n$ respectively. Then their Hamming distance is computed by:

$$Ham(x,y) = \sum_{i=1}^{n} b_i(x) \otimes b_i(y)$$

In which n is the dimension of binary descriptor and $\otimes$ stands for bitwise XOR operation. According to the definition of Hamming distance, all the elements of a binary descriptor contribute equally to the distance. From the hamming distance, the Hamming weight can be calculated. It is used to find the best feature point match. Here, is generalized the Hamming distance to the weighted Hamming:

$$WHam(x,y) = \sum_{i=1}^{n} w_i(b_i(x) \otimes b_i(y))$$

Where w_i is the weight of the ith element. The goal is to learn $w_i, i = 1, 2 \cdots, n$ for the binary descriptor (BRIEF) based on a set of feature points. By assigning different weights to binary codes, what we expect is to obtain a distance space in which the distances of matching patches are less than those of non-matching patches.

3.2.4 Homography Estimation

Typically, homographies are estimated between images by finding feature correspondences on them. A 2D point (x, y) in an image can be represented as a 3D vector

$\mathbf{x} = (x_1, x_2, x_3)$ where $x = \frac{x_1}{x_3}$ and $y = \frac{x_2}{x_3}$ [16]. This is called the homogeneous representation of a point and it lies on the projective plane P^2. An homography is an invertible mapping of points and lines on the projective plane P^2. Hartley and Zisserman [22] provide the specific definition that a homography is a mapping from $P^2 \rightarrow P^2$ is a projectivity if and only if there exists a non-singular 3×3 matrix H such that for any point in P^2 represented by vector $\mathbf{x}$ it is true that its mapped point equals $H\mathbf{x}$. It should be noted that H can be changed by multiplying by an arbitrary non-zero constant without altering the projective transformation. Thus H is considered a homogeneous matrix and only has 8 degrees of freedom even though it contains 9 elements.

The method chosen to solve the homography estimation was the Direct Linear Transformation (DLT) [19, 22] algorithm. The DLT algorithm is a simple algorithm used to solve for the homography matrix H given a sufficient set of point correspondences [16].

Since we are working in homogeneous coordinates, the relationship between two corresponding points **x** and **x'** can be re-written as [16]:

$$c \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = H \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad H = \begin{pmatrix} h1 & h2 & h3 \\ h4 & h5 & h6 \\ h7 & h8 & h9 \end{pmatrix},$$

where c is any non-zero constant, $(\ u\ v\ 1\ )^T$ represents **x'**, $(\ x\ y\ 1\ )^T$ represents **x**. Dividing the first row of equation (2.1) by the third row and the second row by the third row we get the following two equations [16]:

$$-h1x - h2y - h3 + (h7x + h8y + h9)u = 0 \tag{3.1}$$

$$-h4x - h5y - h6 + (h7x + h8y + h9)u = 0 \tag{3.2}$$

Equations (3.1) and (3.2) can be written in matrix form as $A_i\mathbf{h} = 0$. Where,

$$A_i = \begin{pmatrix} -x & -y & -1 & 0 & 0 & 0 & ux & uy & u \\ 0 & 0 & 0 & -x & -y & -1 & vx & vy & v \end{pmatrix}$$

and

$$\mathbf{h} = \begin{pmatrix} h1 & h2 & h3 & h4 & h5 & h6 & h7 & h8 & h9 \end{pmatrix}.$$

Since each point correspondence provides 2 equations, 4 correspondences are sufficient to solve for the 8 degrees of freedom of H. JavaScript typed arrays, defined in Section 2.1.5, were used in the homography estimation implementation for better performance results.

3.2.5 Random Sample Consensus (RANSAC)

RANSAC (Random Sample Consensus) [22] is an iterative method to estimate parameters of a mathematical model from a set of observed data which contains outliers. It is a non-deterministic algorithm in the sense that it produces a reasonable result only with a certain probability, with this probability increasing as more iterations are allowed. It is the most commonly used robust estimation method for homographies according to [16]. The idea of the algorithm is pretty simple; For a number of iterations, a random sample of 4 correspondences is selected and a homography H is computed from those four correspondences. Each other correspondence is then classified as an inlier or outlier depending on its concurrence with H. After all of the iterations are done, the iteration that contained the largest number of inliers is selected. H can then be recomputed from all of the correspondences that were consider as inliers in that iteration [16].

One important step when applying the RANSAC algorithm described above is to decide how to classify correspondences as inliers or outliers. In the implementation for the web only assign the geometric distance [16] threshold, t, between **x'** and $H\mathbf{x}$ was enough. Hartley and Zisserman [22] provides more details about RANSAC.

Another issue is to decide how many iterations to run the algorithm, it's not

required to try every combination of 4 correspondences. The goal becomes to determine the number of iterations, N, that ensures with a probability p that at least one of the random samples will be free from outliers. $N = 100$ was used on the web implementation.

3.3 Rapid Object Detection (Viola Jones)

3.3.1 Contextualization

Rapid Object Detection [61] technique, much known as Viola Jones [61], brings together new algorithms and insights to construct a library for robust and extremely rapid object detection. What has motivated this technique to be added to *tracking.js* library was the task of face detection. The algorithm implementation became robust enough to detect any training data [61], not only for faces. Currently, *tracking.js* supports, face, eyes, upper body and palm detection.

In order to scan faces, eyes or palm from images, a training phase is required. The training phase generate cascading stages. The cascade are constructed by training classifiers using AdaBoost [61] and then adjusting the threshold to minimize false negatives. OpenCV library [6] has some open-source training data, therefore doubling efforts on training is unnecessary, i.e. the face training set consisted of 4916 faces, extracted from images downloaded during a random crawl of the world wide web [61]. Those faces were scaled and aligned to a base resolution of 24 by 24 pixels [61], see Figure 3-9. Training is not the focus of this work, the algorithm to scan the faces is. The training data itself is useless if a Scanning Detector [61] is not available, the scanning is what makes the rapid object detection.

A scanning detector was implemented in JavaScript [36] and is available on *tracking.js*. The training data used is from OpenCV library [6] converted from Extensible Markup Language (XML) [7] to JavaScript Object Notation (JSON) [14]. JSON [14] has much superior performance since it's interpreted by JavaScript language [36, 14]. The results of the JavaScript [36] implementation can be used in real-time appli-

Figure 3-9: Example of frontal upright face images used for training [61].

cations, the detector runs at 15 frames per second. For more information about performance see Chapter 4.

The overall idea of the detection process is that it uses a degenerate decision tree, what Viola and Jones [61] call "cascade". A positive result from the first classifier triggers the evaluation of a second classifier which has also been adjusted to achieve very high detection rates [61]. A positive result from the second classifier triggers a third classifier, and so on [61]. The main steps of the scanning algorithm are:

1. Create or scale a squared block, initially set to 20×20 pixels, by 1.25 per iteration;

2. Loop the squared block by Δ pixels over the image;

3. For each squared block location, loop the decision tree and evaluate each stage;

4. A positive result of the stage [61] triggers the next stage, otherwise stops the stages loop;

5. If all stages were positively evaluated store that rectangle as a possible face;

6. Once the decision tree is done, group the overlapping rectangles;

7. Find the best rectangle of each the group to represent the face. This phase is also known as "merging phase".

The final detector is scanned across the image at multiple scales and locations of the image. This process makes sense because the features can be evaluated at any scale with the same cost [61]. Good results were obtained using a set of scales a factor of 1.25. Subsequent locations are obtained by shifting the window some number of pixels Δ, for a better accuracy $\Delta = 1$ is recommended. We can achieve a significant speedup by setting $\Delta = 2$ with only a slight decrease in accuracy, thus this value was set as default value of the JavaScript [36] implementation.

Viola and Jones [61] proposed that for each found possible rectangle representing the face to be partitioned into disjoint subsets data structures. Two detections are in the same subset if their bounding regions overlap [61]. The corners of the final bounding region are the average of the corners of all detections in the set. In order to perform well on the web, some optimizations were made in the implementation level of the scanning detector. The disjoint set was replaced by an alternative logic that is called "Minimum Neighbor Area Grouping" by this dissertation. Minimum Neighbor Area Grouping has $O(N^2)$ performance [5] and consists in a loop trough the possible rectangle faces returned by the scanning detector. For each step of the loop compare the current rectangle with all other not yet compared rectangles. If the rectangle area overlaps more than η with the compared, by default $\eta = 0.5$ (or 50%), select the smallest rectangle in area of the comparison. Using the smallest rectangle, guarantees that the best match is much centralized in the face.

For more information about the JavaScript [36] implementation, such as evaluation and results, see Chapter 4.

3.4 Color Tracking Algorithm

3.4.1 Contextualization

Colors are everywhere in every single object. Being able to use colored objects to control your browser using the user camera is very appealing. For that reason, *tracking.js* implemented a basic color tracking algorithm that resulted in an real-time frame rate trough a simple and intuitive API. Color has been widely used in real-time tracking systems [53]. It offers several significant advantages over geometric cues such as computational simplicity, robustness under partial occlusion, rotation, scale and resolution changes.

In the tracking system implemented, the color blobs are being tracked. The notion of blobs as a representation for image features has a long history in computer vision and has had many different mathematical definitions. It may be a compact set of pixels that share a visual property that is not shared by the surrounding pixels [42]. This property could be color, texture, brightness, motion, shading, a combination of these, or any other salient spatio-temporal property derived from the signal, in our case the image sequence. Colour perception is a difficult and little understood problem, which seems to defy even the most ingenious mathematical expressions. Evaluating color differences is subjective: when asked to pick the "closest" match for a specific color from a small palette, the selections by the test persons differ, therefore automatize a color tracking technique is not as trivial as find the specific RGB [20] value to be classified as a pixel of interest.

3.4.2 Color Difference Evaluation

When a true color (photographic) image is mapped to a reduced palette, every true-color pixel must be mapped to the palette entry that comes closest to original color. The difference between the original color and the quantized color should remain below some threshold. For that reason, this work determines whether the color is close to the tracked color based on Euclidean distance.

Mapping a RGB color in an orthogonal three-dimensional space, the distance between two colors is denoted by the Euclidean distance $\|C_1 - C2\|$. For a three-dimensional space (with dimensions R, G and B) the Euclidean distance between two points is calculated as follows:

$$\|C_1 - C2\| = \sqrt{(C_{1,R} - C_{2,R})^2 + (C_{1,G} - C_{2,G})^2 + (C_{1,B} - C_{2,B})^2}.$$

Graphic applications for computers usually employ the Red-Green-Blue (RGB) color space. This model maps well to the way the common Cathode Ray Tube (CRT) and Liquid-Crystal Display (LCD) display works. These displays have three kinds of elements that emit red, green or blue light. Another advantage of the RGB model is that it is a three-dimensional orthogonal space, precisely what we need for the Euclidean distance function.

To determine if the pixel is a possible color it should be inside the tracked color neighborhood, $\|C_1 - C2\|$ must be lower than a configurable $threshold = 100$. Imagine that the tracked color C_1 is the center of a sphere in a three-dimensional space, any point C_2 around C_1 that its Euclidean distance is lower than a *threshold* is considered a color that "looks like" C_1. Figure 3-10 exemplifies a green color neighborhood represented in a RGB orthogonal three-dimensional color space. The mentioned technique resulted in a robust and simple color tracking algorithm, since considering multiple values around C_1 as the tracked color increase robustness against illumination changes.

3.4.3 Color Blob Detection

Once all pixels of the tracked color are collected two steps are still required: (1) Detect outliers; and (3) Find the coordinates that represents the color blob.

Detecting Outliers

Outliers pixels are determined if they are not close enough to other pixels with the same characteristics. For each pixel $p = (x, y)$, the mean distance with the other

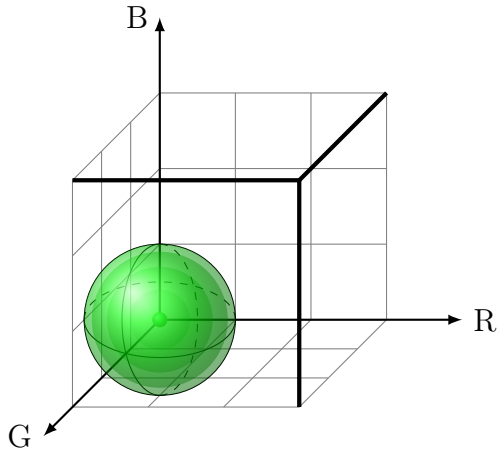

Figure 3-10: Green color neighborhood represented in a RGB orthogonal three-dimensional color space.

found pixels is calculated as follows:

$$meanDistance = \frac{\sum_1^n \sqrt{(x_n - x)^2 + (y_n - y)^2}}{n},$$

If the $meanDistance$ is greater than a $blobThreshold = 30$ the tested pixel is considered "too far" from the color blob and is discarded (outlier). The non-discarded pixels (inliers) are in average closer than $blobThreshold$ from each other, representing the tracked color blob.

Finding Color Blob Coordinates

After the color blob is detected, the found pixels are close enough to represent a color blob. To determine the blob central coordinates $C_b = (x_b, y_b)$, the value for each axis is determined based on the average of each pixel axis value $p_n = (x_n, y_n)$, the values of x_b and y_b are calculated as follows:

$$x_b = \frac{\sum_1^n x_n}{n}, y_b = \frac{\sum_1^n y_n}{n}.$$

Discussion

The described color technique results in a fast and robust solution for color blob detection on the web. On Figure 3-11 is shown the result running on the browser with and without possible outliers interference in the scene.

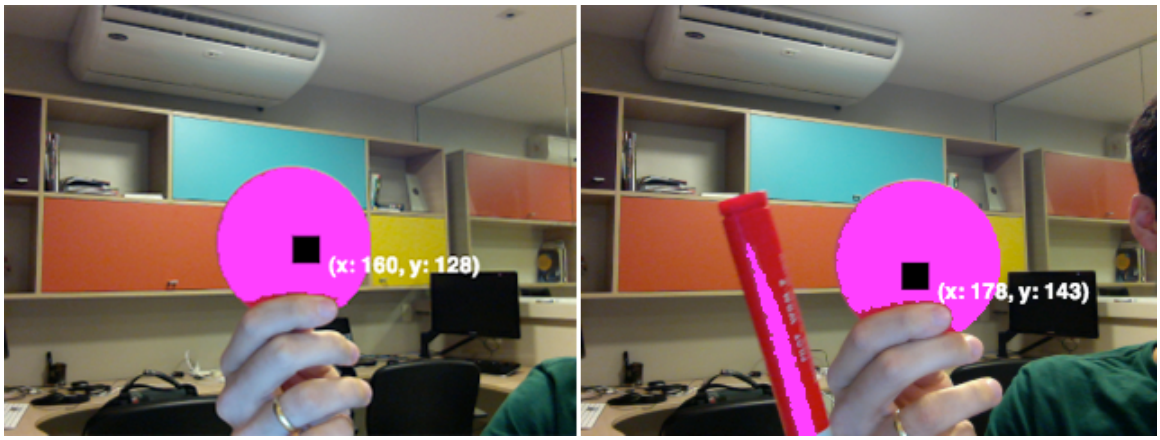

Figure 3-11: Example of color tracking technique. The black square represents the central color blob coordinate C_b. On the left, the magenta circle is tracked without outliers interference. On the right an outlier object of the same color is introduced in the scene without causing issues to the found circle.

Chapter 4

Evaluation

4.1 Evaluation Methodology

This chapter present the results of each technique described on Chapter 3. Two metrics were utilized to analyze the solutions available on *tracking.js* library:

1. Frames per second (FPS): present FPS metric of how the implemented techinques performs on the web environment in order to reach real-time capability. On the United Kingdom popular video format known as Phase Alternating Line (PAL) [8], real time video is represented by 25 FPS, therefore this value is used to define whether the tests can or cannot be considered real-time.

2. Partial occlusion robustness: present tests how the implemented techinques reacts to partial occlusions.

In addition to the two metrics described above, a comparison between *tracking.js* and existing solutions involving server-side tracking is presented [54], showing benefits of a pure JavaScript client-side tracking solution. All tests were executed on Google Chrome browser version 28.0.1500.71 [30], running on Mac OS X 10.8.3, 2.6 GHz Intel Core i7 16 GB 1600 MHz RAM.

4.2 Results

4.2.1 Rapid Object Detection (Viola Jones)

Discussion

Having Viola Jones rapid object detection as part of the library resulted in interesting examples for web applications, such as detecting faces, mouths, eyes and any other training data [61]. On Figure 4-1 is shown different examples of training data being used by the library implementation of Viola Jones.

Figure 4-1: Library implementation of Viola Jones using different training datas for detecting faces, eyes and mouth.

Figure 4-2: Library implementation of Viola Jones detecting multiple faces inside the real-time limit of 25 FPS.

Augmented reality and tracking applications for advertising and entertainment are gaining more space on the web enviroment. The media used in this kind of application needs to be as appealing as possible in order to catch consumers' attention, thus detecting faces, or augmenting the scene with objects are attractive possibilities. In order to demonstrate that concept, a simple chat application was created. In this chat application, while talking in real-time, the users could augment their faces

with objects, such as a fake glass with mustache. In order to extract the users face coordinates Viola Jones was used. Listing 4.1 shows the simplified JavaScript API provided by *tracking.js* in order to extract face coordinates and draw the image. The fake glasses are positioned over x and y coordinates on the canvas axis based on the extracted values. Figure 4-3 demonstrates the described example.

Figure 4-3: Augmenting users faces with objects using *tracking.js* Viola Jones eyes detection.

```
1   var img = new Image();
2   img.src = 'img/glasses.png';
3   var videoCamera = new tracking.VideoCamera();
4   videoCamera.track({
5       type: 'human',
6       data: 'frontal_face',
7       onFound: function(track) {
8           videoCamera.canvas.context.drawImage(img, track[0].x, track[0].y, track[0].size, track[0].size);
9       }
10  });
```

Listing 4.1: Example of *tracking.js* API of augmenting users faces with objects using Viola Jones face detection.

F

The United Kingdom popular video format known as Phase Alternating Line (PAL) [8] defines that real time video is represented by 25 FPS, therefore this value is used to define whether the tests can or cannot be considered real-time. On Figure 4-4, the Viola Jones implementation were tested with different numbers of detected faces. Fifteen faces were gradually added, for each addition the FPS average was recorded. Note that, until five faces detected the web implementation still runs inside the real-time limit defined by PAL [8].

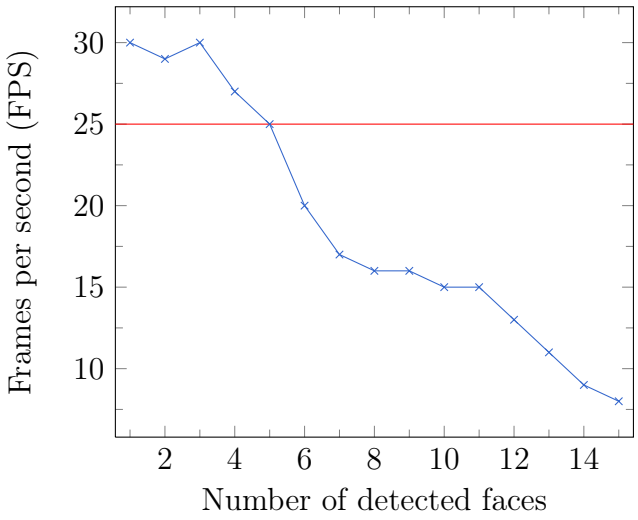

Figure 4-4: Library implementation of Viola Jones tested with different numbers of detected faces.

Oclusion Robustness

Viola Jones implementation of *tracking.js* shows d results for partial occlusions. Figure 4-5 demonstrates occlusions variations that still allows the face to be detected.

4.2.2 Color Tracking Algorithm

Discussion

Being able to use colored objects to control your browser using the user camera is very appealing. Any colored object could be used to enable user interaction with your web

Figure 4-5: Library implementation of Viola Jones partial occlusion robustness.

application using *tracking.js* color tracking technique trough a simple and intuitive JavaScript API (Listing 4.2). On Figure 4-6 is shown different colored objects being tracked.

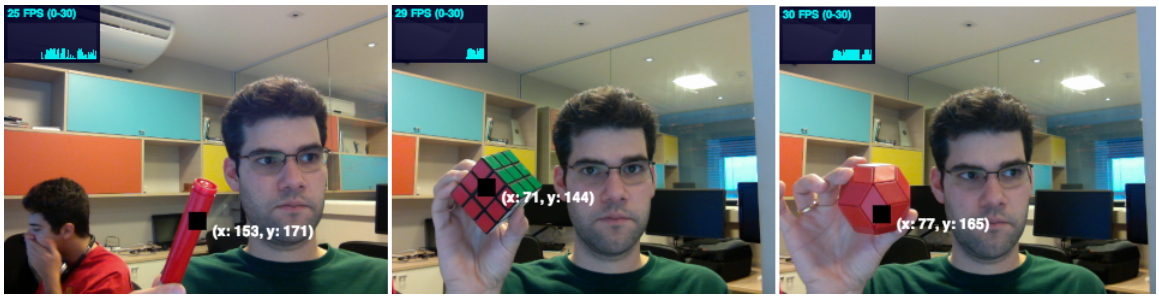

Figure 4-6: Library implementation of color tracking for different objects: On the left a red pencil marker; on the center a Rubik's magic cube [59] from the red face; and on the right a red Ball of Whacks [64].

```
1  var videoCamera = new tracking.VideoCamera();
2  videoCamera.track({
3      type: 'color',
4      color: 'magenta',
5      onFound: function(track) {
6        // do your logic here.
7      }
8  });
```

Listing 4.2: Example of *tracking.js* color API.

Enable entertainment web applications is also possible using color tracking. Few examples were implemented using color tracking and a PlayStation move controller (Figure 4-7) that is basically a colored sphere that can emit light, the color of the sphere can be set via Bluetooth. This controller have four buttons, square, triangle, cross, circle, which are color coded as pink, green, blue and red. The controller has also three types of built-in sensors, temperature, accelerometer, gyroscope and magnetometer. In this work only the emitted light is used to track the scene.

Figure 4-7: PlayStation move controller.

On Figure 4-8, the two bottom images are examples of how games could be developed to the web trough color tracking. On the bottom left, a multi-player game that allows the user to draw in order to competitors guess what is the drawing meaning is demonstrated. On the bottom right, multiple PlayStation move controllers are used to control the user interactions into a 3D environment rendered trough the GPU using WebGL [29]. Note that, the available *tracking.js* techniques could be combined with WebGL rendering in order to reach real-time 3D rendering.

Another example of color tracking on user interactions is demonstrated on Figure 4-9. Using a HTML5 audio element [36] a music is played and trough any colored object, the user can control the volume of the player sliding the object from the left to right and vice-versa, left most coordinates means volume set to zero, right most coordinates means volume set to maximum.

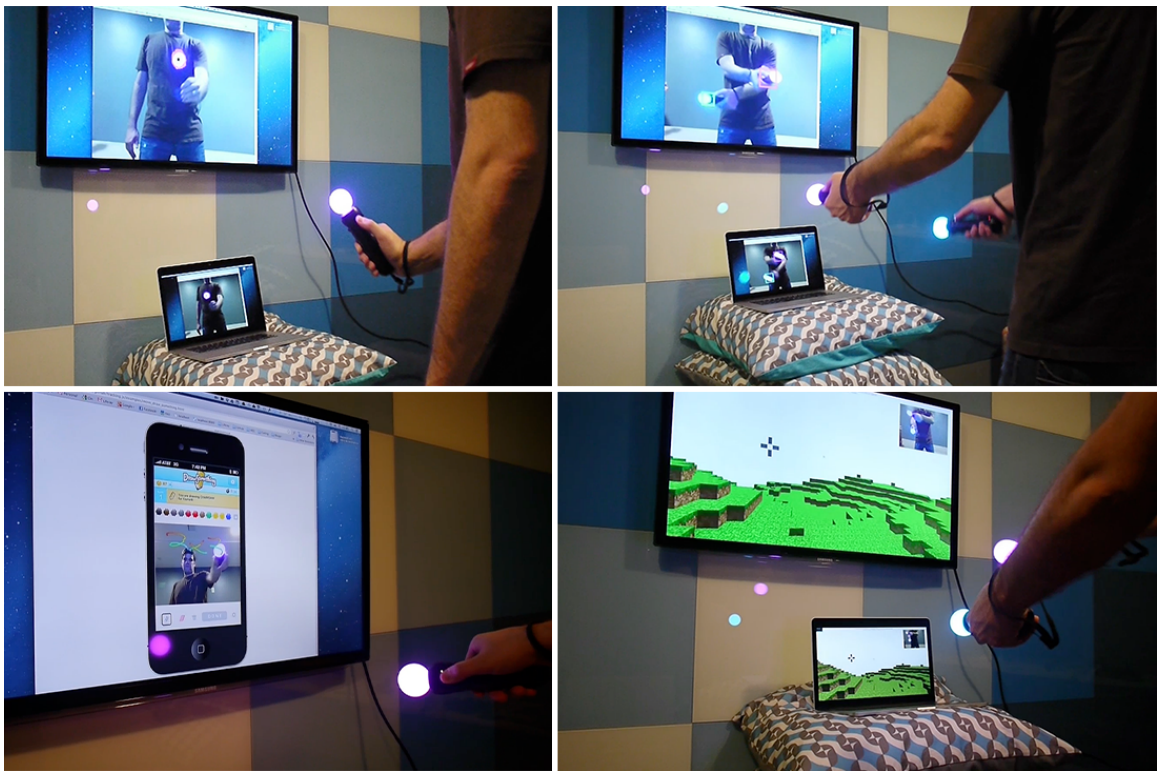

Figure 4-8: Library implementation of color tracking used in games running on the web. On the Bottom left, a multi-player game that allows the user to draw using the camera. On the bottom right, multiple PlayStation move controllers are used to control the user interactions into a 3D environment.

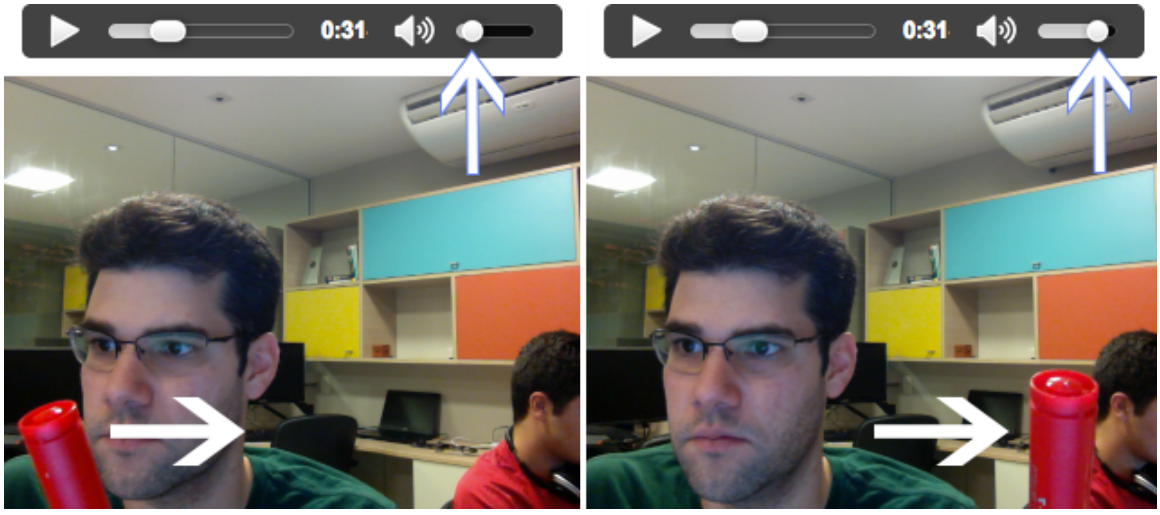

Figure 4-9: Library implementation of color tracking controlling the HTML5 audio element volume.

FPS

On Figure 4-10, the color tracking implementation were tested with different numbers of pixels detected. The size of the detect object were gradually increased, resulting in

a bigger number of pixels found, for each incrementation in the size the FPS average was recorded. Note that, until 2138 pixels detected the web implementation still runs inside the real-time limit defined by PAL [8].

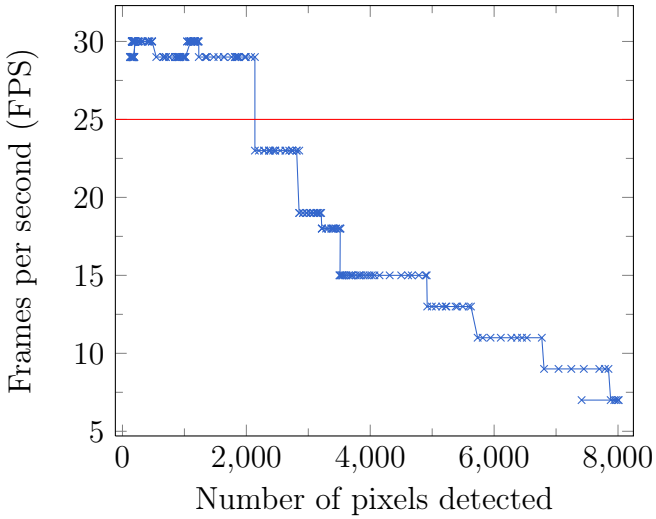

Figure 4-10: Library implementation of color tracking technique tested with different numbers of pixels detected.

Oclusion Robustness

Color tracking technique implementation of *tracking.js* shows good results for partial occlusions. Figure 4-11 demonstrates occlusions variations that still allows the colored object to be detected.

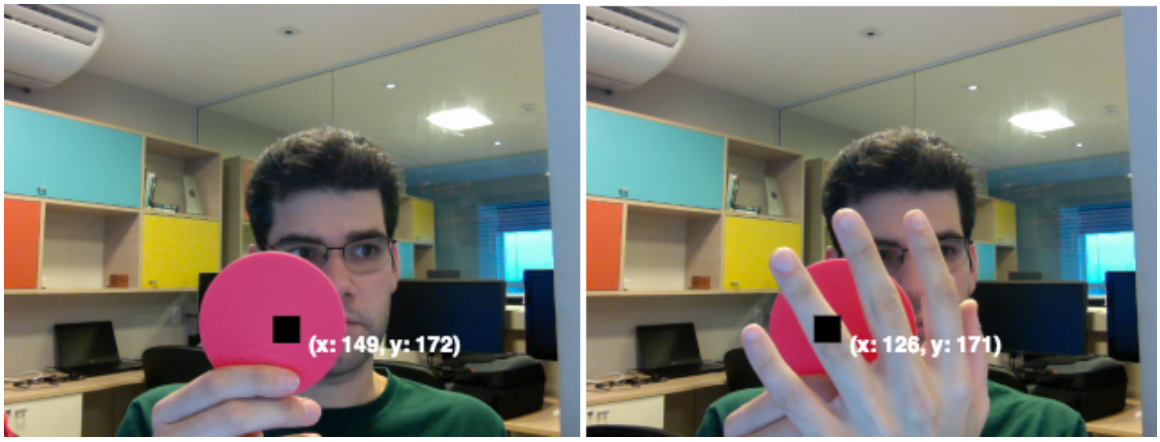

Figure 4-11: Library implementation of color tracking technique partial occlusion robustness.

4.2.3 Markerless Tracking Algorithm

Discussion

Being able to use track any objects based on invariant natural features characteristics is also demonstrated by the markerless tracking technique. On Figure 4-12 is shown features extracted from different scenes. Note that most part of the keypoints are invariant to the camera rotation. Based on the features detected, the best H homography matrix is estimated. All found points in the first frame is multiplied by H in order to be plotted in the second frame. The found H is defined as follows:

$$H = \begin{pmatrix} 1.6101363130102753 & -0.5346783206945362 & -33.21640286774384 \\ 0.18027586959550446 & 0.8414271531603306 & -7.530803625100701 \\ 0.0021658540103673164 & -0.0025444273054841503 & 1 \end{pmatrix}.$$

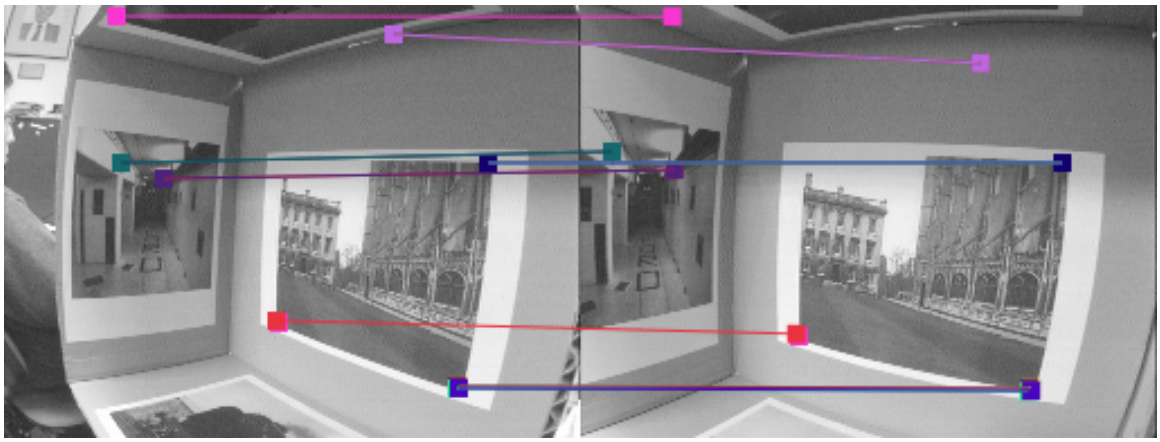

Figure 4-12: Library implementation of markerless tracking technique. Features are extracted from different scenes. On the first frame, 49 features were detected by FAST [57], on the second frame 17 of those were matched by BRIEF [9] and plotted by the H matrix.

The markerless tracking web implementation have also showed to be robust for illumination and scaling differences. On Figure 4-13 is shown the features detected by FAST (top images) and features extracted by BRIEF (bottom images) plotted based on the estimated homography H.

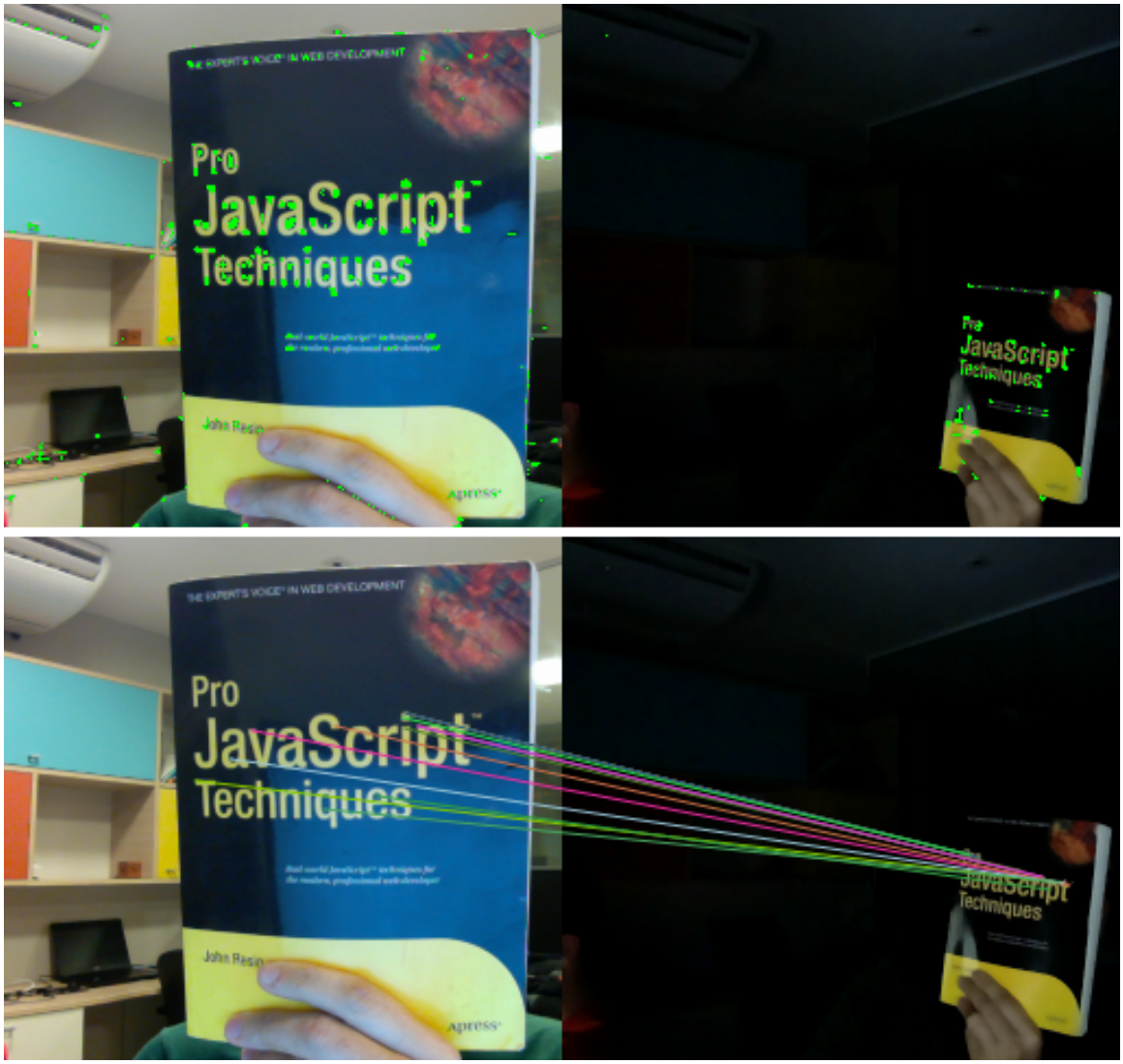

Figure 4-13: Library implementation of markerless tracking technique. Illumination and scaling robustness for features detected using FAST (top images) and extracted using BRIEF (bottom images) plotted on the second frame by the estimated homography matrix H.

FPS

On Figure 4-14, the markerless tracking implementation were tested in an interval of 1000 seconds. The FPS average for this technique is 6 FPS, which is not ideal for real-time applications yet. Some optimizations could be applied on future work, such as implementing FAST-ER [57] for feature detection.

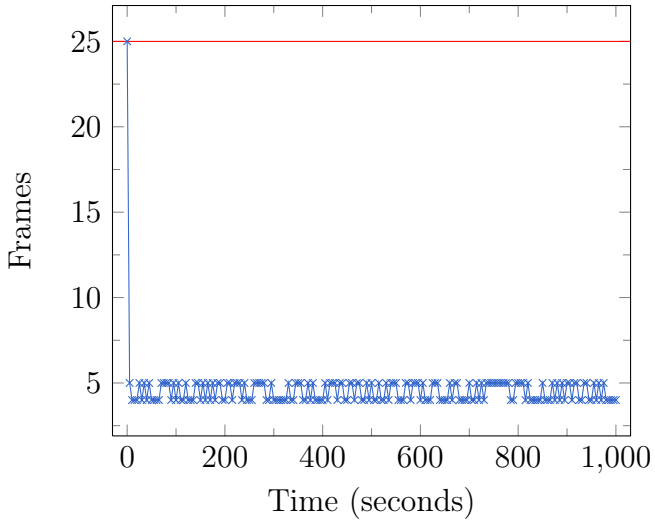

Figure 4-14: Library implementation of markerless tracking FPS metric. Maximum rate using this technique on the web is 6 FPS.

Oclusion Robustness

Markerless tracking technique implementation of *tracking.js* shows good results for partial occlusions. Figure 4-15 demonstrates occlusions variations that still allows the object to be detected.

Figure 4-15: Library implementation of markerless tracking technique partial occlusion robustness.

4.2.4 Benefits of a JavaScript tracking solution

Discussion

On Markerless Tracking Solutions for Augmented Reality on the Web [54], a server-side tracking solution was implemented and the developed solution showed to be not scalable, causing a delay in the exhibition of the AR result when having more than five simultaneous users. This way, the adopted distributed approach turned out to be not adequate to web targeted applications, where hundreds of simultaneous users should be capable of using it. In the same work [54], Pablo et al., proposed a client-side solution that requires a third-party plugin installation called OSAKit [2]. The OSAKit plugin allows desktop applications to run without loss of performance inside most current browsers on the Windows platform. The results were promising in terms of FPS reached (25 FPS), although the average load time of the application was 11 minutes (due to the size of the precomputed data, 22 MB), considering a broadband connection of 256 Kbps. Using a pure JavaScript tracking solution has shown to be very effective since the FPS average is the same with a smaller size. On Table 4.1, a comparison between the size, loading time and FPS between the OSAKit solution and *tracking.js* is presented.

	OSAKit	*tracking.js* + color	*tracking.js* + face
Average frame rate	25 FPS	30 FPS	25 FPS
Average file size	22 MB	8 KB	187 KB
Average load time (256 Kbps)	11 min	0.25 seconds	5 seconds

Table 4.1: Comparison between *tracking.js* and OSAKit client-side tracking solution.

Chapter 5

Conclusion

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

5.1 Contributions

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

5.2 Future Work

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

Bibliography

[1] Eduardo A Lundgren Melo. Typed Arrays Performance, 2013.

[2] Soft Ancient. OSAKit. 2013.

[3] Ronald T Azuma. A Survey of Augmented Reality. *Media*, 6(August):355–385, 1997.

[4] Steve Benford, Chris Greenhalgh, Gail Reynard, Chris Brown, and Boriana Koleva. Understanding and constructing shared spaces with mixed-reality boundaries. *ACM Transactions on Computer-Human Interaction*, 5(3):185–223, 1998.

[5] Paul E Black. Big-O Notation. *Dictionary of Algorithms and Data Structures*, 2007.

[6] G Bradski. The OpenCV Library. *Dr Dobbs Journal of Software Tools*, 25(11):120–125, 2000.

[7] Tim Bray. Extensible Markup Language (XML), 2013.

[8] Walter Bruch. Phase Alternating Line (PAL), 1962.

[9] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. BRIEF : Binary Robust Independent Elementary Features. *Computer*, 6314(3):778–792, 2010.

[10] C Cedras and M Shah. MOTION-BASED RECOGNITION - A SURVEY. *Image and Vision Computing*, 13(2):129–155, 1995.

[11] Y Cho, J Lee, and U Neumann. A Multi-ring Color Fiducial System and an Intensity-Invariant Detection Method for Scalable Fiducial-Tracking Augmented Reality. *In IWAR*, pages 1–15, 1998.

[12] WHATWG Community. The canvas element, 2013.

[13] Intel Corporation. Intel® SSE4 Programming Reference, 2007.

[14] Douglas Crockford. JSON. 2013.

[15] Voxeo Daniel C. Burnett, Ericsson Adam Bergkvist, Cisco Cullen Jennings, and Anant Narayanan. Media Capture and Streams, 2013.

[16] Elan Dubrofsky. *Homography Estimation*. Essay, The University of Britsh Columbia, 2009.

[17] Xiph.Org Foundation. Vorbis specification, 2012.

[18] B Gloyer, H K Aghajan, Kai Yeung Siu, and T Kailath. Vehicle detection and tracking for freeway traffic monitoring. In *Asilomar Conference on Signals Systems and Computers*, volume 2, pages 970–974, 1994.

[19] Jonas Gomes and Luiz Velho. *Fundamentos da Computação Gráfica*. 2009.

[20] Rafael C Gonzalez and Richard E Woods. *Digital Image Processing (3rd Edition)*. Prentice Hall, 2007.

[21] Alan Grosskurth and M.W. Michael W Godfrey. A reference architecture for web browsers. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 661–664. IEEE, IEEE, 2005.

[22] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*, volume 2. Cambridge University Press, 2004.

[23] David Herman and Kenneth Russell. Typed Array Specification, 2013.

[24] Ian Hickson. HTML 5 Nightly Specification (W3C), 2013.

[25] Kato Hirokazu. ARToolKit: Library for Vision-based Augmented Reality. *IEIC Technical Report Institute of Electronics Information and Communication Engineers*, 101(652(PRMU2001 222-232)):79–86, 2002.

[26] Adobe Inc. Adobe Flash, 2013.

[27] Apple Inc. Safari Browser, 2013.

[28] Apple Inc. The WebKit Open Source Project, 2013.

[29] Apple Inc. WebGL Specification, 2013.

[30] Google Inc. Google Chrome Browser, 2010.

[31] Google Inc. Blink, 2013.

[32] Google Inc. HTML5 Rocks Tutorials. 2013.

[33] Google Inc. Project Glass, 2013.

[34] Google Inc., Mozilla Inc., and Opera Inc. WebRTC, 2013.

[35] Metaio Inc. Metaio Unifeye Viewer. 2009.

[36] Ecma International. ECMA-262 ECMAScript Language Specification. *JavaScript Specification*, 16(December):1–252, 2009.

[37] ISO. Information technology – Generic coding of moving pictures and associated audio information – Part 7: Advanced Audio Coding (AAC), 2006.

[38] Omar Javed and Mubarak Shah. Tracking and object classification for automated surveillance. *History*, 26:343–357, 2002.

[39] Xu Jia and Huchuan Lu. Visual Tracking via Adaptive Structural Local Sparse Appearance Model. *IEEE Conference on Computer Vision and Pattern Recognition (2008)*, pages 1822–1829, 2012.

[40] Zhen Jia, Arjuna Balasuriya, and Subhash Challa. Vision Based Target Tracking for Autonomous Land Vehicle Navigation: A Brief Survey. *Recent Patents on Computer Science*, 2(1):32–42, 2009.

[41] Hirokazu Kato and Mark Billinghurst. JSARToolkit a JavaScript port of FLARToolKit, 2011.

[42] Vladimir Kravtchenko. Tracking Color Objects in Real Time. 1999.

[43] D W F Van Krevelen and R Poelman. A Survey of Augmented Reality Technologies , Applications and Limitations. *International Journal*, 9(2):1–20, 2010.

[44] Vincent Lepetit and Pascal Fua. Monocular Model-Based 3D Tracking of Rigid Objects. *Foundations and Trends® in Computer Graphics and Vision*, 1(1):1–89, 2005.

[45] Guorong Li Guorong Li, Dawei Liang Dawei Liang, Qingming Huang Qingming Huang, Shuqiang Jiang Shuqiang Jiang, and Wen Gao Wen Gao. Object tracking using incremental 2D-LDA learning and Bayes inference, 2008.

[46] Joao Paulo Lima, Veronica Teichrieb, and Judith Kelner. A Standalone Markerless 3D Tracker for Handheld Augmented Reality. *Virtual Reality*, pages 1–15, 2009.

[47] C Mei, S Benhimane, E Malis, and P Rives. Efficient Homography-Based Tracking and 3-D Reconstruction for Single-Viewpoint Sensors, 2008.

[48] Microsoft. Silverlight, 2013.

[49] Pranav Mistry, Pattie Maes, and Liyan Chang. WUW - Wear Ur World - A Wearable Gestural Interface. *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, 68(3):4111–4116, 2009.

[50] Inc. Mozilla. Gecko, 2013.

[51] Inc. Mozilla. Mozilla Developer Network, 2013.

[52] Inc. Mozilla. Mozilla Firefox Browser, 2013.

[53] G Paschos. Perceptually uniform color spaces for color texture analysis: an empirical evaluation, 2001.

[54] João Paulo, S M Lima, Pablo C Pinheiro, Veronica Teichrieb, and Judith Kelner. Markerless Tracking Solutions for Augmented Reality on the Web.

[55] Les A Piegl. *Fundamental developments of computer-aided geometric modeling*. Academic Pr, 1993.

[56] Jinchang Ren Jinchang Ren, T Vlachos, and V Argyriou. Immersive and perceptual human-computer interaction using computer vision techniques. *Computer Vision and Pattern Recognition Workshops CVPRW 2010 IEEE Computer Society Conference on*, pages 66–72, 2010.

[57] Edward Rosten, Reid Porter, and Tom Drummond. Faster and better: a machine learning approach to corner detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(1):105–119, 2010.

[58] Edward Rosten, Reid Porter, and Tom Drummond. Machine learning for high-speed corner detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(1):1–14, 2010.

[59] Rubiks. Rubiks Magic Cube, 2013.

[60] Veronica Teichrieb, Monte Lima, Eduardo Lourenc, Silva Bueno, Judith Kelner, and Ismael H F Santos. A Survey of Online Monocular Markerless Augmented Reality. *International Journal of Modeling and Simulation for the Petroleum Industry*, 1(1):1–7, 2007.

[61] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–511–I–518. IEEE Comput. Soc, 2001.

[62] W C. The World Wide Web Consortium (W3C), 2006.

[63] W3C. Descriptions of all CSS specifications. 2013.

[64] Creative Whack. Ball of Whacks. 2013.

[65] Wikimedia. Wikimedia Traffic Analysis Report - Browsers, 2013.

[66] Joao Marcelo Xavier Natario Teixeira. Analysis and Evaluation of Optimization Techniques for Tracking in Augmented Reality Applications. 2013.

[67] Xiph.Org Foundation. Theora Specification, 2011.

[68] Yongxin Yan and Xiaolei Zhang. Research and analysis of the Virtual Reality with FLARToolKit, 2011.

[69] Alper Yilmaz, Omar Javed, and Mubarak Shah. Object tracking: A survey. *ACM Computing Surveys*, 38(4):13, 2006.

[70] Feng Zhou, Henry Been-lirn Duh, and Mark Billinghurst. Trends in augmented reality tracking, interaction and display: A review of ten years of ISMAR. *2008 7th IEEEACM International Symposium on Mixed and Augmented Reality*, 2(4):193–202, 2008.