Graz University of Technology

Institut für Maschinelles Sehen und Darstellen

Master's Thesis

Web-based Augmented Reality

Christoph Oberhofer

Graz, Austria, January 2012

Thesis supervisors

Prof. Dr. Gerhard Reitmayr

Jens Grubert

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)

Abstract

Augmented Reality (AR) applications are usually built on top of dedicated visual tracking pipelines implemented in a high performance compiled programming language, such as C++, or executed inside optimized runtime-environments, like Adobe Flash. Such implementations tie applications to specific platforms and vendors, making it difficult to provide a single solution for multiple systems. Today's cross-platform AR solutions are mainly based upon proprietary web-technologies lacking of mobile device support or open standards.

Within this thesis, the development, implementation and evaluation of an AR tracking pipeline using natural features is presented. The whole pipeline, starting from camera access to final 3D real-time rendering, is solely based on standard web technologies including HTML5, JavaScript and WebGL. The novelty lies within the completely plugin-free manner of the solution, running in basically each modern web browser on the PC and even on mobile phones.

An extensive evaluation shows that real-time framerates are achieved on entry-level PCs whereas interactive experience is made feasible on high-end smartphones.

Keywords. JavaScript, HTML5, Augmented Reality, AR, Video, WebGL, getUserMedia, Adobe Alchemy, Google Native Client

Acknowledgments

This thesis would not have been possible unless my advisor, Jens Grubert, who introduced me to Augmented Reality, supported me from the initial to the final level with his ideas and suggestions to create a deep understanding of computer vision and computer graphics related topics. He always encouraged me to stay interested in the topic for which I am very grateful.

It is an honor for me to thank Gerhard Reitmayr, my thesis supervisor, whose knowledge and experience played an important part in achieving my goals. His contributions and ideas led to a very successful outcome for which I am very thankful. His effort made it possible for me to travel to Basel attending ISMAR 2011, my first conference visit ever.

I would also like to show my gratitude to Michael, my twin-brother, who dedicated some of his time to assist me in during testing and evaluating the implementation throughout the whole development process. My heartily thank goes to the rest of my family, who supported me during my entire studies. Further gratitude is due to all my friends who reminded me of not forgetting about real-life and social needs. My thanks also go to Barbara, my girlfriend, whose patience and understanding during this time I cannot appreciate enough. Her knowledge of the English language enabled me to improve my writing and greatly increase my competence in the language of computer science.

Lastly, I offer my regards and blessings to my godmother, Elisabeth Schlocker, who spent much of her precious free-time to proofread my work. Due to her linguistic revision I was able to eliminate my errors and complete my thesis successfully.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As Augmented Reality (AR) is rapidly gaining popularity it may be finally accepted by a critical mass of users [met11a]. With the growth of people's awareness of AR, it is considered being not only a research field any more.

Today's consumer AR solutions can be experienced through various channels, such as native applications for PCs, apps for smartphones or even web applications for both of these platforms. Apart from the platforms served, the publicly available AR solutions may be divided into two categories. First, AR-browser that try to annotate the surrounding of the user based on their current position and orientation [lay11] [Wik11]. And second, AR entertainment solutions such as games or magazine overlays [Met11c] [Qua11b]. Whereas applications in the first category often rely on inertial sensors such as GPS, accelerometer and compass, solutions found in the second category generally apply visual tracking techniques to follow the user's view. This thesis focuses on the latter approach since tracking from natural features (NFT) is the main focus of interest.

Employing NFT in AR solutions is a computationally complex task, which often requires optimizations in low-level software components to achieve satisfactory performance. For that purpose, solutions are largely implemented using native programming languages such as C & C++ which have to be adapted and compiled for the individual systems. As long as the diversity of soft- and hardware platforms exists, developing AR solutions that target different systems is not only cost intensive but may not even be possible at all. Even though cross-platform development was recently improved through techniques like intermediate language representation [Lat11] and virtual machines [Ado11a], creating homogeneous software is still a time-consuming task [Tri11]. One example of such a cross-platform approach for AR applications is Total Immersion's D'Fusion Studio [Tot11a].

From the user's point of view, the expectation of the same experience throughout the entire range of devices is high, regardless of whether using a mobile phone, a desktop PC or a tablet computer. Currently existing solutions are generally provided as software packages that have to be installed locally, which may limit user's acceptance. [Tec10]

Web technologies may be considered as a possible solution to the cross platform dilemma (see figure 1.1). Especially the combination of JavaScript, HTML5 and CSS3 is deemed to be platform independent offering powerful tools to develop native-like applications. This approach is still suffering from the bad reputation it received in the earlier days, when Microsoft's Internet Explorer was the damper on evolving web technologies [Mic11d]. Until then, the web as a software development platform was not taken seriously [WR01] for rich client applications. In contrary, nowadays most of those aspects are proven wrong by many software companies that offer web-based counterparts of their own software products. With the appearance of GMail [Goo11d], Google Docs [Goo11c] and Office Web Apps [Mic11g] the web may be seen mature enough to be used as a serious platform for software development. Even Microsoft has favored web technologies over their proprietary platform for developing extensions for Windows 8 [Ful11]. This indicates a shift away from native towards web applications which is not only taking place on the PC, but also in the mobile world [Bra11].

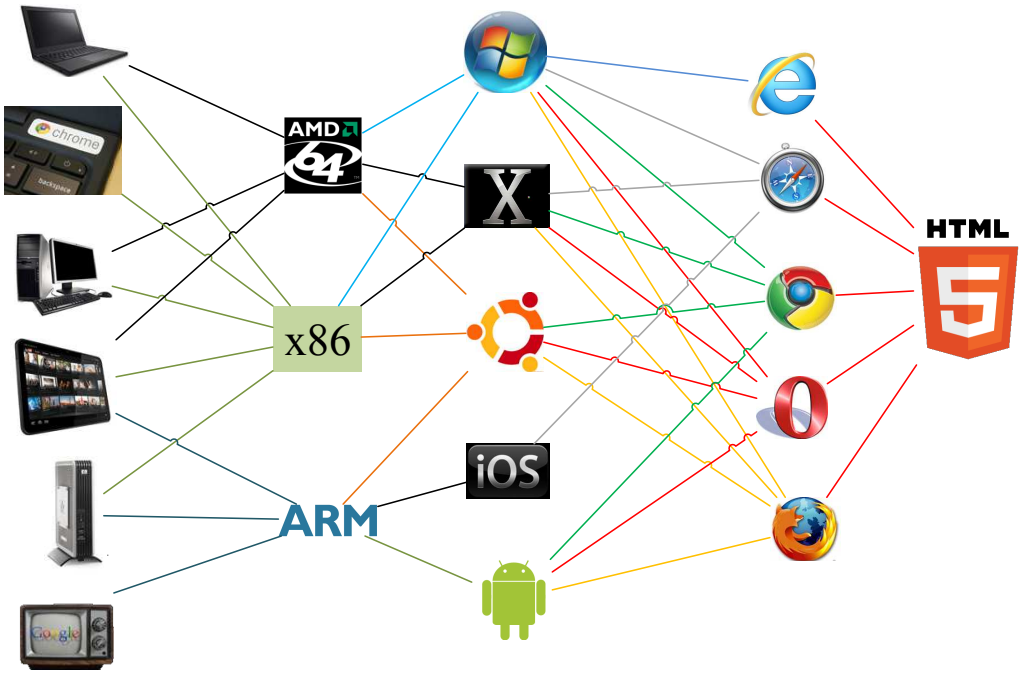


Figure 1.1: HTML5 Cross-Platform Software Development

Efforts are being made not only to run applications but also to create whole operating systems based on web technologies. Google's ChromeOS [Goo09] is a notable candidate, as well as Mozilla's attempt named Boot-to-Gecko [Moz11i], a streamlined and efficient web-based operating system.

Because earlier web browsers lacked access to hardware from within the HTML environment, parts of a typical AR-pipeline were not feasible to be implemented with standard web technologies when using HTML4 [W3C99]. Particulary, web-cam access and rendering 3D graphics were not foreseen in the latest finalized HTML specification. Hence, today's web-based AR applications are highly dependent on plugin technology such as ActiveX [Met11c], Adobe Flash [Ima11] or Microsoft Silverlight [Sch10]. Recent developments and initiatives of browser developers are trying to close these gaps with the introduction of WebGL [Khr11f] and getUserMedia [WHA11b] in HTML5 [W3C11a]. A port of the popular ARToolKit [KB99] [Kat04] called JSARToolKit [Hei11] makes use of WebGL and demonstrates that an almost complete AR tracking pipeline can be built with native web technologies only.

Since the ARToolKit is based upon marker tracking techniques, the computational complexity can be handled by low-end PCs and mobile devices [WS07]. In contrast, tracking natural features still requires optimized algorithms and state-of-the-art computer vision techniques to run smoothly on standard PCs or high-end mobile phones [WRM$^+$10].

In this thesis, the development, implementation and evaluation of an AR tracking pipeline using natural features is presented. The whole pipeline, starting from camera access to final 3D real-time rendering, is solely based on standard web technologies including HTML5, JavaScript and WebGL. The novelty lies within the completely plugin-free manner of the solution, running in basically each modern web browser on the PC and even on mobile phones. Real-time framerates are achieved on entry-level PCs whereas interactive experience is made feasible on high-end smartphones.

1.1 Problem description

The underlying questions for this thesis are:

- Is it *feasible* to implement a natural feature tracking AR pipeline relying only on native web-technologies such as JavaScript and HTML5?
- What are the current *limitations* of realizing an AR system with the mentioned technologies?
- If possible, how does the implementation *perform* compared to other web-based approaches?

Thesis organization

Chapter 2 contains related work and background information on AR and web technologies. After that, chapter 3 presents the selection of components and mechanisms as well as the approach towards the later implementation. Chapter 4 goes into detail how the system was implemented in JavaScript, C/C++, Adobe Alchemy and Google Native Client. The results after testing and evaluation are shown in chapter 5 followed by a conclusion in chapter 6 including an outlook on further work and ideas. Additional work which could not fit into the thesis itself is put into the appendix, where further evaluations did not fit.

Chapter 2

Related Work

This section is dedicated to the work already done in the fields of AR and the web, in combination with specific background information. Some fundamental knowledge is presented to lead into the topic.

Section 2.1 gives an introduction to Augmented Reality, covering topics such as applications, output devices and tracking modalities. Special focus is put on AR pipelines using natural feature tracking (NFT) by describing the methods and algorithms they apply.

A detailed overview of JavaScript is given in section 2.2, where the evolution of the programming language, as well as the execution environments are depicted. Particular attention is paid to the JavaScript engines and their mechanisms to speed up the execution of JavaScript code.

Section 2.3 deals with the new HTML5 standard, the layout engines that parse and render the markup and some hand-picked elements which are used inside an AR pipeline. In addition, a comprehensive feature-matrix on web browsers is presented allowing quick comparisons between different developments.

Lastly, section 2.4 concentrates on Augmented Reality solutions currently available on the web, their implementation details, advantages and disadvantages. Particular focus is set on proprietary web-based approaches.

2.1 Augmented Reality

Augmented Reality (AR) presents the user a real world view showing virtual objects combined with the real environment. It is not only about adding virtual objects, but also removing real world content. AR is therefore closely related to Virtual Reality (VR),

where, in contrast, the user is completely enclosed inside a virtual environment without seeing the real world. In AR the composite of virtual and real objects should look as real as possible in order to appear to the user as a whole [MK94]. No general accepted definition of AR exists, because the separation of the fields AR and VR is somewhat blurry as illustrated in figure 2.1.

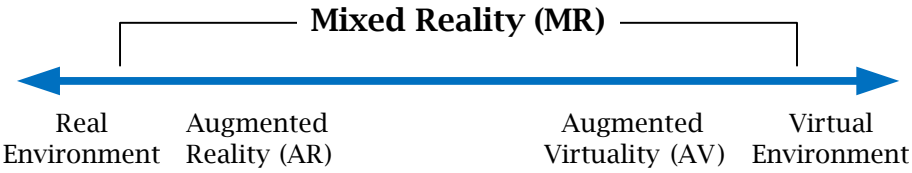


Figure 2.1: Milgram's virtuality continuum [MK94]

According to [Azu95] an AR system has to fulfill the following three criteria:

- Combining real and virtual content
- Being real-time interactive
- Registered in three-dimensional space

A system meeting all these criteria is considered being an AR system. This does not limit AR systems for creating graphical content only, but may also generate different virtual content such as audio as long as it is aligned and one can interact with.

Owing to the fact that the definition given in [Azu95] is very broad, many systems may be categorized as AR systems. A typical pattern for classifying these systems is either through their output device or through their tracking modality.

2.1.1 Output devices

Presenting the augmented reality to the user can be approached in various ways. The most natural device is a head-mounted display (HMD) which is attached to the user's head and projects the combination of real and virtual objects in front of their eyes. A slightly different attempt is the use of a handheld device, which acts see-through and gives the user an impression of a "magic window". Apart from mobile usage, stationary AR systems, such as computers with a built-in webcam, are less portable but provide more computing power and applications for multiple viewers.

2.1.1.1 Head mounted display (HMD)

HMDs are image generating devices which are worn by the user. For presenting the combined image to the user, optical systems are incorporated and are positioned directly in front of the eyes. Because of the short distance between the image plane and the eyes, optics are used to move the focal point further away. There coexist two different types of HMDs, namely optical see-through and video see-through. A more concise comparison between the two types can be found in [RF00].

Optical see-through A user wearing an optical see-though HMD observes the real world directly through the display. The blending of the real-world and virtual content is achieved through a half-silvered mirror which lets in the light from the outside but also acts as surface for reflecting the generated projections. Because of physical limitations, the HMD is not able to fully occlude real objects, which makes them still shine through. The current focus of optical see-through development lies within a more compact and lightweight form factor [CWHT09] [Tot11b].

Video see-through In contrast to the optical see-through technology, the video see-through HMD does not employ a direct view of the real world. Hence, the outside is presented through displays mounted on the inside of the glass. In order to accomplish the mixing of real and virtual objects, cameras are attached to the head capturing the user's view. The resulting images are then augmented with computer generated content and displayed to the user. Due to the replacement of the eyes with cameras, the viewpoint of the user may not be matched. Current work tries to tackle the challenges in expanding the field of view as well as correct visualization of virtual objects that are close to the user's point of view [CC09].

2.1.1.2 Handheld devices

The smart-phone market has become a widespread platform for AR applications [met11a] owing to its exponential growth [Bus11] in recent years. The screen of the phone serves as an output device presenting an augmented view of the real world to the user. Due to the vast number of sensors today's mobile devices are equipped with, the platform is not only an output device, but also offers capabilities for tracking. Featuring inertial sensors like accelerometer, gyroscope and compass for determining orientation as well as GPS for gathering the position of the phone. In addition, the integrated camera may be used for

visual tracking which lets the device act as “magic window”. This combination of output device and different tracking modalities makes PDAs, tablet computers and mobile phones an ideal platform for Augmented Reality.

2.1.2 Tracking

Augmented Reality applications require real-time 6DOF pose tracking of devices such as head-mounted displays or mobile cameras. These six degrees of freedom are also known as extrinsic camera parameters and consist of three rotation and three translation values. To recover the camera's position in an unknown environment, various mechanisms and methods exist for different applications. These tracking modalities try to capture the motion and position of the current viewpoint in order to align the virtual content with the real world. A comprehensive classification of devices and systems is presented in [AWW01].

2.1.2.1 Single sensing

Tracking modalities employing only one kind of sensors for determining the extrinsic camera parameters.

Acoustic trackers employ ultrasonic sound waves, which have a frequency just above the audible range of human ears, to measure range. A method called phase coherence determines the phase difference between the sound wave at the receiver and the transmitter. Since the frequency of the signal is known, the distance can be computed directly from the phase difference between the transmitted and received signals. An exact position can be determined when using at least 3 transmitters or receivers [FHP98].

Magnetic trackers use, in contrast to acustic trackers, magnetic fields to sense range and orientation. The transmitter emits magnetic fields which the receiver picks up for measurement. In order to determine distance and orientation, both sides have three orthogonal triaxial coils incorporated [Bha93].

Inertial tracking makes use of two separate devices, both reacting to movement. First accelerometers to sense acceleration for position and second gyroscopes to measure the orientation.Since these sensors rely on Newton's law's, no external devices are needed and act completely passive [Fox96].

Mechanical trackers are used to measure movement directly from attached sensors on a human body. Sensing relative changes in joint angles and lengths between joints, the absolute position can be derived. This form of tracking is also known as motion capturing technique where a human being wears a full body suit, or just arm gloves to measure locations of the joints [WF02].

Optical trackers make use of optical sensors such as CCD or CMOS which are incorporated in video cameras. This type of trackers make use of either active (light-emmiting), passive (not powered) or no targets at all (natural features) to measure the position and orientation of the camera (see Section 2.1.4.

2.1.2.2 Hybrid sensing

Since each single tracking device has limitations on its own, it is practically to combine multiple trackers in order to increase robustness and accuracy [YNA99]. For example, using GPS along with inertial sensors and video, tracking performance increases due to the distinct advantages each sensor carries within [RD06a].

2.1.3 Applications

Augmented Reality systems can be employed in various situations helping the user to achieve certain tasks, work collaboratively or just for entertainment purposes.

AR may be used for *medical assistance* to aid during surgery. The basic idea is to provide the doctor an "X-ray vision" in order to operate with minimally-invasive techniques [RBBS06]. This application relies on very detailed and accurate 3D models of the patient, as well as a robust and precise registration. 3D models may be generated using MR or Ultrasound.

Similar to the medical tasks, *manufacturing* and *repair applications* benefit from AR. Instead of having a user's manual besides a device where illustrations indicate the parts to be replaced or repaired, an HMD allows to annotate and label the parts directly on the device. In additions, for aiding the user, correctly registered animations may demonstrate the process of repair [HF09].

Another approach is using mobile phones for vision-based tracking and on-screen visualization. These mobile devices already provide enough processing power to perform computer vision and graphic tasks to register a real-time 3D pose [WRM+10], many applications and SDKs have already been released [Qua11a] [met11b] since then.

AR applications are not restricted to single-user work environments, but may also be used in collaboration with other participants [HBO05].

For entertainment purposes, AR is already used in TV where live sport events are broadcasted with real-time augmentations. A notable pioneering task was carried out by FoxTrax [Cav97] with the idea of increasing the visibility of the puck on the ice. For this task, the environment was modeled in 3D ahead of time in addition to calibrated and precisely tracked cameras. A specially manufactured puck with infrared LEDs incorporated is tracked by the cameras. During the final step, the image is chroma-keyed and a yellow line is drawn on the field appearing real to the TV viewers.

2.1.4 Vision-based tracking

Vision based tracking techniques completely rely on the acquired images from a camera to measure the current position and orientation of the viewpoint. There exist various ways to address this problem such as tracking fiducial markers or natural features. The basic issue lies within the requirement for being real-time interactive (see 2.1). Many vision-based algorithms are too expensive to be used for real-time experience and have to be selected and optimized carefully.

Marker tracking vs NF-tracking

Both, marker tracking and natural feature tracking, are widely employed techniques to measure the 3D position of the camera. Earlier attempts rather used aritificial features such as active light sources or passive fiducials. Detecting and tracking those features requires much less effort than relying on natural features only [NY99]. The advantage of NFT over tracking fiducial markers is that nearly any structure with sufficient features can be used for tracking. In this context features are known as a collection of areas or points in an image that can be easily redetected under varying viewing conditions.

2.1.5 NF-Tracking

Natural feature tracking relies on features found in a video stream, which are then further processed to determine the current position of the camera. Positions of these features are not known a priori and determined to be found in a separate initialization step. After successfully detecting the initial position, the tracking phase can take over.

2.1.5.1 Tracking vs. Detection

During the initialization phase, no prior information about the scene is known, except the target image or model to detect. The initialization takes care of determining the initial absolute camera position only by detecting a previously defined image or model. This process is generally slow and is only required for initialization or recovering after tracking loss. Because the detection does not rely on prior pose information, sudden changes applied to the viewpoint do not affect the performance which makes it more robust compared to tracking.

Since the initialization step is usually too slow for real-time experience [WRM+10], an additional tracking phase may take over. The tracker makes use of the already known pose and extracts new information on a frame-to-frame basis to keep the camera position up to date. Under the assumption that the change between two subsequent frames is kept small, algorithms may be applied taking advantage from it. This approach can significantly speed up the process of pose tracking [KM07]. Since the tracker purely relies on relative position changes acquired from two subsequent frames, it is not as robust to large inter-frame changes as the detection.

2.1.5.2 Initialization

Several methods exist for initializing the camera pose without prior knowledge. Approaches may rely on detection of a target image in the current video [WRM+10], while others make use of Structure-from-Motion [KM07]. For outdoor usage, location based information [RD06a] may also assist during the initialization process, such as GPS.

Methods for detecting known plane 2D target images are employed in many AR applications. The basic idea is to find corresponding points in both images, the target and the search-image, which are then used to compute the 6DOF camera pose by solving an overdetermined system of equations.

Finding correspondences is a nontrivial task for the reason that scaling, translation, rotation and perspective deformations make the target appear dissimilar. One way of addressing this problem is to start with interest point detection followed by description and matching of those points. Because it cannot be assumed that all resulting correspondences are matched correctly, outliers have to be removed. In the final stage, the homography, or the pose itself is calculated.

Feature Detection Detection of features inside an image can be done in various ways. Features may be points, edges, regions or even lines. Most importantly, the extraction of features should detect the same features on the same object, even if the view has changed or presence of noise.

For accurate detection of keypoints, methods such as the Hessian detector [Bea78] or Harris operator [HS88] may be used. The former is based on the matrix of second derivatives, the Hessian matrix, and locates points with large differential in two orthogonal directions. The latter finds points where the second moment matrix applied in the neighborhood results in large eigenvalues. While the Hessian detector responds to high textured regions and may result in a denser outcome, the Harris operator often corresponds to corner-like structures. Another approach for keypoint detection is described in [RD06b] where radial surroundings are inspected and evaluated if being a corner or not. This method is known to be very fast to compute, but suffers from low repeatability.

None of these detectors is neither invariant to scale nor to affine transformation. Scale invariance can be achieved in combination of a Laplacian scale selection [MS02].

Interesting regions may be detected using Laplacian-of-Gaussian [Lin98] or Difference-of-Gaussian [Low04]. These methods are built upon image pyramids and respond to blob-like structures. Extending these scale-invariant detectors to result in affine covariant regions, can be achieved by iterating over the region's second-moment matrix to compute the eigenvalues. The position of the region is then refined and yields to an elliptical shape.

Keypoint Description After extracting interesting features from the image, they get assigned a descriptor for later discriminative matching. The descriptor is a unique identifier for the same feature region or point under different viewing conditions. This allows for the later matching stage to find corresponding points in different images.

The very popular method *Scale Invariant Feature Transform*, known as SIFT [Low99], is a combination of DoG region detector and a feature descriptor. It is known to be very robust against changes in lighting, scale and rotation, as well as variations in the viewport. The basic idea of SIFT is storing the image information in a localized set of gradient orientation histograms. At first, a scale and rotation normalized region, extracted by a region detector, is acquired, followed by sampling the image gradients into an orientation histogram. A coarse 4x4 grid of image gradients and 8 orientations per histogram bin results in a 128 dimensional, highly discriminative, descriptor.

Since SIFT is computational expensive, various alternatives have been developed in order to increase efficiency, of which SURF [BETG08] is a notable method. Its speed-up

compared to SIFT is reported up to a factor five while still achieving comparable repeatability. The speed enhancement is realized by replacing the Gaussian-model with simple Haar wavelets. The feature region is again divided into a 4x4 grid with binning summary statistics, instead of a gradient orientation histogram, resulting in a 64 dimensional descriptor.

Both of these methods demand lots of computational power during the description and matching stage. In contrary, a very efficient approach using Binary Robust Independent Elementary Features (BRIEF) targeting real-time applications is presented in [CLSF10]. With the help of binary tests and a descriptor represented as binary string, matching of keypoints is efficiently implemented using the Hamming-Distance [Ham50] function. Due to the low complexity, the descriptor is neither invariant to scale nor to rotation. BRIEF was developed further to the BRISK [LCS11] descriptor, extending the original version with scale and rotation invariance.

Keypoint matching After the extraction and description of features, the next step in finding correspondences is the matching of already known points from the target with the new features gathered from the unknown image. The matching has to be carried out very fast with high reliability and distinctiveness. This process applies an a-priori defined distance function to each pair of known and unknown features, which gives information on the similarity of the two features. If one pair is found with minimum distance, a correspondence is established. In order to avoid comparing each found feature with each known one, measures such as tree- or hash-based search may be employed.

Outlier removal During the matching stage, where correspondences get assigned to by the means of their similarity, wrong matches are likely to occur. This may happen due to similar features which are present more often, like a pattern, or just because features found in one picture were not extracted from the other and being matched to the closest candidate. Those false positives have to be eliminated during the outlier removal process. During this stage, the geometric error of the correspondences is being measured and matches succeeding a certain threshold level are being removed. The process of geometric verification is based upon the assumption that all correspondences must share a consistent geometric transformation. In case of the projection of one plane onto another, the homography can be computed out of four correspondences. Because images are impaired with noise and other artifacts, it is recommended using more than four correspondences. For finding and rejecting outliers methods such as RANSAC [FB81] or Hough

Transform [DB81] may be employed.

Random Sample Consensus (RANSAC) is a non-deterministic approach and works in a hypothesize-and-test manner. The probability in finding a good model increases with the number of iterations. The algorithm samples a minimal subset of four correspondences and estimates the homography from these. It then verifies the remaining correspondences and counts the number of inliers. If the number of inliers is greater than the largest one alreay found, then store the new model, otherwise reject and continue until a certain criteria is met. A termination criteria may be the maximum number of iterations, or a minimum number of inliers found.

2.1.5.3 Tracking

While the initialization phase does not rely on any known parameters of the current camera pose, the tracking phase depends on accurate a priori information. Due to the provided input, the tracker is much faster and may be even more robust in certain situations than the detector. In contrast to marker tracking, where a fiducial marker is detected in each single frame, natural features are harder to detect and require much more computational capacity.

The method presented in [NY99] is a closed-loop motion tracking architecture which is capable of detecting and tracking natural features in video images. It combines three functions, namely feature selection, motion estimation and evaluation. The feature selection stage detects and selects point and region features, whereas the motion estimation relies on affine region warping and SSD evaluation. The whole architecture is designed to work iteratively and automatically selects the best features, depending on the model's error, to discriminate between good and bad ones.

A more advanced technique working in real-time is presented in [KM07] for parallel tracking and mapping. The method does not rely on any predefined map or template which could limit the range of operation. It is a 2-threaded approach, where one thread tracks the motion of the camera, the other produces a 3D map of point features. While other techniques depend on non-separable tasks, this method allows for optimal resource usage of today's multicore computer systems . The motion tracker itself is implemented in a two-stage procedure, where a combination of patch-tracking and pose estimation is applied.

Most AR tracking approaches work only in small workspaces or in limited environments because of restrictions to 2D-plane tracking or dependence on already known features. A

method worth mentioning is described in [RD06a]: a model-based hybrid tracking system for outdoor AR in urban environments. The system combines individual components such as an edge-based tracker, gyroscope measurements, sensing gravity and magnetic fields and a back store.

Not all tracking algorithms are computationally expensive, selected may even run on mobile phones [WRM+10]. The approach presented in [WRM+10] implements a very efficient patch-tracker for motion tracking, combining a keypoint-less detector, affinely warped patches and normalized cross correlation for matching results.

2.1.5.4 Pose estimation

The estimation of the 3D position of an object is the final stage before rendering the virtual objects. This problem is also known as extrinsic calibration and is decoupled from the intrinsic calibration where parameters such as focal length, skew and principal point offset are determined. The basic idea is to recover the extrinsic parameters R and t (see figure 2.2) from three correspondences (perspective-3-point-problem) or more (perspective-n-point).

To solve the perspective-n-point (PnP) problem techniques such as the direct linear

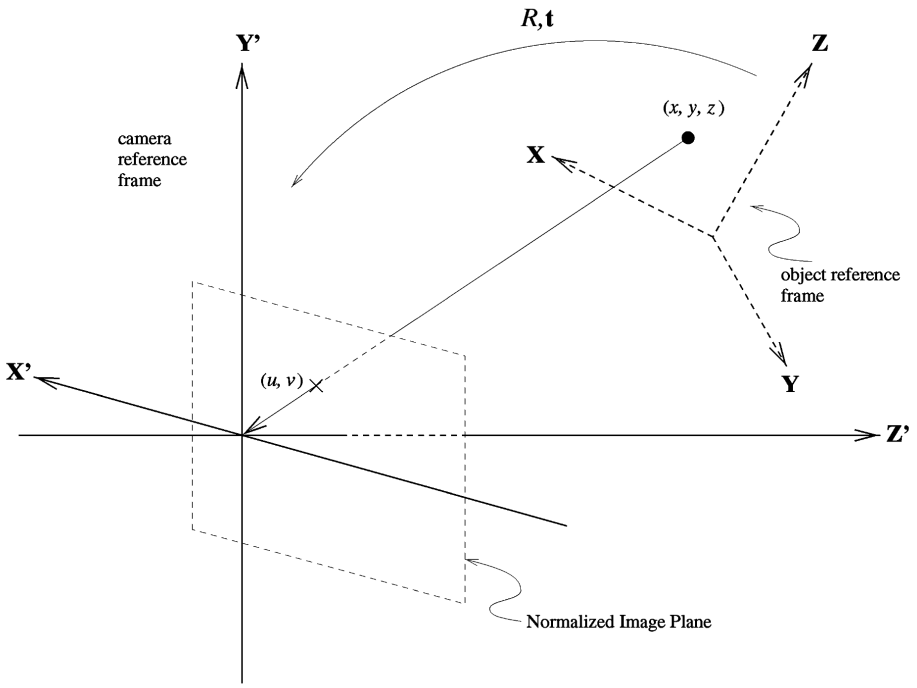


Figure 2.2: Camera Pose, taken from [LHM00]

transform (DLT) and iterative algorithms have been developed. The DLT approach recovers the pose from a set of linear equations. This requires at least six known correspondences between 3D and 2D locations. Those 3D to 2D point collectives are connected through the camera's perspective projection matrix P from which the 11 unknowns are computed. In order to decompose P to recover R and t, RQ factorization is applied. Because this method is highly noise sensitive, an alternative approach using iterative techniques is preferred.

Iterative algorithms estimating the camera's pose are more accurate and resistant to noise in comparison to the DLT approach. The basic idea is to minimize the reprojection errors of the 3D to 2D correspondences iteratively .

2.2 JavaScript

JavaScript it is known to be one of the world's most misunderstood programming language [Cro01]. Nevertheless, it is one of the most popular programming languages among web developers [Mon11] [Way10].

For implementing an AR system in web technologies, JavaScript offers the basic building blocks for algorithms and data processing. It is a multi-purpose programming language, supporting functional, prototypal and object-oriented programming. Different mechanisms such as closures and high-order functions make it powerful and very flexible in usage.

2.2.1 History

A look back at the beginning of JavaScript helps to understand the current limitations, the good and the bad parts of the language. JavaScript was originally developed in 1995 in only ten days by Brendan Eich [Eic11]. During that time it was named Mocha, then LiveScript and finally JavaScript as a scripting language counterpart to Java. In December the same year, it was licensed to Netscape and Sun and was added to Netscape Navigator 2.0 [Net95]. [Kri08] [Ham08]

Without the chance for revision and optimization, the original version of the language was integrated into Netscape Navigator, which made it nearly impossible to enhance and extend the initial functionality. Many reserved keywords in the language are still unused giving clues about the original intentions.

In 1995, the HTML standard had reached its version 3.2 and the web was still very static at that time. No DOM operations, no calculations, no animations were possible. The

need for easy programming tasks, without the bloat of packaging, compiling and linking, was the main goal during development of JavaScript. Because of licencing problems, Microsoft was not allowed to call their implementation JavaScript and named their language JScript [MC07]. It was first shipped along with Internet Explorer 3.0 in 1996 [Mic11c].

2.2.2 Evolution

In the early days JavaScript had a bad reputation and was mainly referred to as being a toy-language for "script kiddies". JavaScript was seen as a link between Java-Applets and the HTML document. Since Java-Applets did not gain the popularity which had been expected at the beginning, JavaScript was becoming more and more the preferred language for development of web applications [Ham08].

Firstly, JavaScript was used for simple calculations and DOM-manipulations to offer the user a new browsing experience. Due to the bad performance of the first interpreter implementations, the efficiency of the execution of JavaScript code was quite low compared to solutions offered nowadays [Gol11].

With the increasing popularity, many libraries and frameworks were sprouting up and promised easier to use functionality. The biggest disadvantage of JavaScript is still the lack of cross-browser compatibility. Although the language is specified in ECMA standard, different browsers produce different output. Notable libaries such as jQuery [jQu10], Dojo [Doj11] or prototype [Pro10] try to cover up those cross-browser incompatibilities.

The introduced browser-war among Firefox, Opera, Chrome, Safari and Internet Explorer, which was ultimately started with Firefox' release, brought the web's evolution up to speed again [Wha05]. After years of stagnation, ruled by Microsoft's Internet Explorer, the users are today no longer bound to use the browser by default, but may choose on their own instead [Mic10b]. Along with the introduction of new browsers, mechanisms were introduced to further increase the speed and conformity of JavaScript interpreters for a better experience. Finally heavy-weight web applications offering real-time multimedia content were ready to be experienced.

2.2.3 Specification

JavaScript is a dialect derived from ECMAScript and currently specified and maintained within Mozilla [Moz11b]. The ECMAScript standard is standardized in ECMA-262 [ECM11] which's latest version was released in 2009 in its 5th edition. Not all browser built-in scripting engines are based upon JavaScript itself, but use other

ECMAScript dialects. Opera [Ope11], Safari [Web11a] and Chrome [Goo11f] directly offer ECMAScript compliant engines, whereas Firefox [Moz11f] builds upon JavaScript and Internet Explorer [Mic11f] on JScript. The main differences between those dialects are mostly found in APIs such as the Document Object Model (DOM). The deviations in the language itself are described in greater detail in [Lak07].

2.2.4 Engines

The purpose of a JavaScript engine is the interpretation and execution of programming code, where not only the speed is of importance, but also the conformity to the ECMAScript specification.

JavaScript is considered being much slower than Java when interpreted. The reasons are twofold. First, JavaScript is a dynamic weakly typed language that is not aware of types before run-time. Second, it is executed by an interpreter which adds a lot of overhead to the actual code. The illustration in figure 2.3 shows a side-by-side comparison of different execution environments, when executing a simple addition. On the left side, the SpiderMonkey JavaScript interpreter incorportated in Firefox 3 is shown. The center is occupied by a JIT (Just-in-time) compiled Java or C code whereas Firefox 3.5 with the TraceMonkey JIT extension to SpiderMonkey is placed on the right side.

```
1 var a = 3, b = 5;
2 var c = a + b;
```

Listing 2.1: Simple Addition

Most of the execution time is spent during type checking and converting values to appropriate types in order to execute the add operation. In contrast, in languages such as Java and C this information is already known at compile-time and the appropriate machine instruction can be executed without checking the types. [Man09]

Hence, optimization strategies have to reduce or eliminate the overhead caused by the interpreter in order to increase speed. Each engine applies different mechanisms to achieve the desired speed-ups. which means that some parts of the code might run fast inside one engine, but not in another.

Most modern JavaScript engines use techniques such as just-in-time (JIT) compilation and native code generation. This helps to improve the execution speed of the resulting code significantly, but makes it also more difficult to port the engine to different hardware platforms. Since native code is bound to the instruction set of the used CPU, special adaptations have to be made. If a CPU platform is not supported, a fallback to the

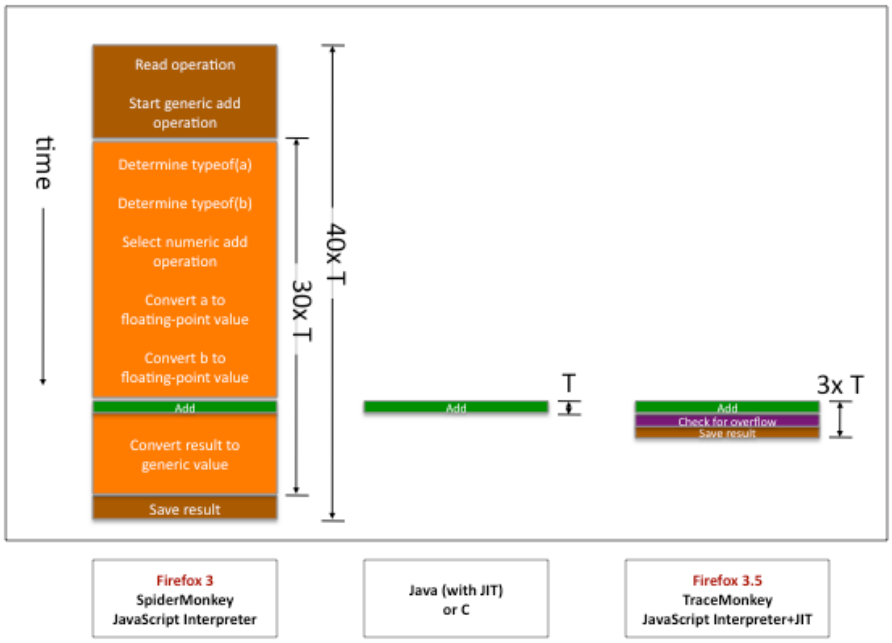


Figure 2.3: JavaScript interpretation overhead taken from [Man09]

interpreter takes place and the device cannot benefit from these features any more. The majority of engines are already supporting the most frequently used platforms such as x86, x64 and ARM. The only exception currently is Chakra [Mic10c] which can only be used within the 32bit version of the Internet Explorer [Law09].

Currently there are five major JavaScript engines available powering different browsers. Namely, SpiderMonkey (Mozilla), V8 (Google), Nitro (Apple), Carakan (Opera) and Chakra (Microsoft). These are explained in greater depth in chappters 2.2.4.1 to 2.2.4.5.

2.2.4.1 SpiderMonkey

SpiderMonkey [Moz11f] is the JavaScript interpreter used within Gecko-based layout engines, such as Firefox and Thunderbird, developed by Mozilla. It takes raw JavaScript instructions as input, compiles it to intermediate bytecode which is then interpreted. This does not achieve satisfactory execution speed, because interpretation is still neccessary.

With the introduction of TraceMonkey [Man09], an extension to the SpiderMonkey interpreter, techniques such as tracing and just-in-time compilation were added. These mechanisms are targeted to reduce the amount of overhead caused by interpretation and result in speed-ups of 2x -40x depending on the complexity of the actual operation (see Fig-

ure 2.3).

Tracing is a method which records paths of executions when certain parts are visited multiple times and keeps track of the types used. The moment a loop gets executed more than two times, the loop is considered hot and the trace is being compiled to native machine code which is then executed directly on the CPU. The compilation task is carried out by nanojit [Moz11d], a component extracted from the Tamarin [Moz11h] project which is also used by Adobe within the ActionScript Virtual Machine.

Furthermore, to increase speed of general code which is not being compiled by nanojit, JägerMonkey [Bli10] introduced the Nitro-Assembler borrowed from Apple's WebKit. It takes all the raw JavaScript instructions and generates native code without optimizations. The Nitro-Assembler is not to be confused with nanojit, while the former does not optimize the code generated, but works fast, the latter optimizes often used code-branches and generates highly efficient code. In brief, all JavaScript get compiled with Nitro-Assembler to native code, while tracing takes on tight inner loops and further optimizes them using nanojit.

In addition to all these extensions to SpiderMonkey, a new project called IonMonkey [Moz11j] is currently under development which promises even more speed improvements on JavaScript. The speed-ups are mainly achieved by Type Inference (TI), a mechanism which staticly and dynamically analyses the type of each single variable and property in the whole program [Hac11] [Syn11]. This obviates the need for further type checking which massively decreases the execution-overhead.

2.2.4.2 V8

The JavaScript engine used inside the Chromium browser is an open source project called V8 [Goo11f]. The engine's approach to enhance JavaScript execution builds upon three mechanisms, namely fast property access, dynamic machine code generation and stop the world garbage collection. [Goo11b]

To overcome the overhead associated with property lookup, a feature called *fast property access* is employed. This optimization's basic idea is to create hidden classes which are created during runtime when properties are added or deleted from objects. This mechanism tries to infer the internal class structure of the program which is then created in native code. Each time a property is added or deleted, a hidden class representing the object is searched for, or created on demand. Each single object is linked to a hidden class which may be shared among many others. Having the JavaScript objects linked to hidden

classes makes access to properties really fast, because no dynamic lookup is needed any more. This improvement relies on the fact that many scripts are written in object-oriented manner where structures are being reused.

The second technique for significant enhancement in execution speed is called *dynamic machine code generation*. As the name already implies, the code is not interpreted in a classic manner any more, but compiles to machine code before execution. There is no intermediate bytecode generation, hence no interpretation of the code. In addition to the code generation, the property access is inline cached, which is made possible due to the hidden classes. There may be cases where the prediction of an underlying property is wrong (when a value of a different type is assigned to the property) then deoptimization is applied. These runtime patches are again directly compiled to machine code.

Because the JavaScript language specification does not provide any kind of manual memory management, an efficient garbage collection mechanism is crucial. The implementation of the GC in V8 is built in a *stop the world* manner which acts generationally and accurately. The program's execution pauses on a regular basis and parts of the heap are processed and cleaned out. Afterwards, the program continues. These relgular pauses can cause noticable slow-downs in interactive applications. In order to avoid these performance impacts, a new incremental garbage collector is in development promising significantly reduced pause times. [EC11]

With the introduction of Crankshaft [Chr10], extended optimization techniques were added, such as a baseline compiler, a runtime profiler, an optimizing compiler and support for deoptimization. These additional mechanisms basically try to identify and enhance crucial parts of the code which are frequently executed.

The baseline compiler generates machine code for all JavaScript before the execution even starts. This must be done very quickly, because page-load times are one of the most important factors. This is achieved through very fast, non-optimizing machine code generation during startup. During runtime a profiler identifies hot code, code which is executed more than once, and tells the optimizing compiler to improve these parts. In the case that the compiler was too optimistic, support for deoptimization is included.

2.2.4.3 Nitro

Apple's JavaScript engine implementation used within the Safari web browser. The first version of Nitro, called Squirrelfish [Gar08], was a major improvement over the old JavaScriptCore available in previous WebKit releases. It was given a bytecode interpreter

which was register-based and direct-threaded. The bytecode generator was implemented to behave lazily and compiled only code from the syntax tree on demand. This approach resulted in a 50% speedup compared to the older WebKit version (3.1).

Further work resulted in SquirrelFish Extreme [Sta08] which contained many additions and mechanisms to deliver even more performance. First, the bytecode generator was optimized by implementing more efficient opcodes in order to execute faster inside the VM.

Second, a technique called *polymorphic inline cache* was added. This module is responsible to detect JavaScript objects with the same underlying structure and group them together in common classes with an unique StructureID. When an object's property is accessed for the first time, the lookup of the property is carried out using a HashTable which is usually a very expensive process, even when highly optimized. When the property is found, the StructureID is cached and used for subsequent calls.

Third, SquirrelFish Extreme was equipped with a *context threaded JIT* with the task of generating native code. While having the direct-threaded bytecode interpretation, the JIT compiler uses the bytecode to convert native code from it, opcode by opcode. This lightweight JIT does not optimize any form of code, but instead targets fast compilation.

2.2.4.4 Carakan

Opera introduced a new JavaScript engine called Carakan [Lin09] with its initial release being part of the browser in version 10.5. Until then Futhark, a stack-based interpreter which was optimized in memory consumption and resource management, was used in Opera browsers.

The stack-based interpreter within Futhark was then replaced by a register-based bytecode instruction set in Carakan. A stack-based instruction set is based around a stack of values, where operands are poped from the stack, are processed, and the result is pushed back onto the stack again. This type of set is really memory efficient, because only one dynamically sized stack of values is needed. In contrary, register-based instruction sets work on fixed size memory blocks, called registers. The advantage is that values and operands can be accessed directly without copying the data from the stack. This results in fewer instructions being executed, speeding up the JavaScript engine.

For optimal performance the engine is equipped with a hot-spot detecting JIT which is activated when a code is executed multiple times. It compiles native code directly from JavaScript without an intermediate bytecode representation. Supported by compile

time static analysis and runtime profiling of the bytecode it further optimizes code parts focusing on arithmetic operations. This helps to eliminate type-checks at runtime and reduce the overhead of interpretation significantly. Another component which increases the performance of Carakan is the use of function inlining, which is not only carried out on user-written scripts but also on built-in functions.

In order to reduce the memory footprint of the running script, an automatic object classification mechanism is employed. Meaning, an internal representation of the JavaScript objects is constructed using class-like structures. Information about an object, its prototype and properties are collected and stored internally in a class. Objects which use the same set of properties are then sharing a common class. These shared structures allow caching the results of individual property lookups of objects sharing the same class.

2.2.4.5 Chakra

With the introduction of the Internet Explorer in Version 9 a completely new JavaScript engine called Chakra [Mic10c] was delivered. Until IE8, Microsoft's web browser made only use of the traditional script interpreter and lacked modern adaptations such as bytecode interpretation and native code compilation. Microsoft finally realized the need for fast JavaScript code execution and implemented Chakra, an engine just as powerful as the competitors.

In order to achieve a quick start-up of the page, the JavaScript code is immediately executed in a register-based interpreter when the page is loaded. In the meantime, a background code generator is activated and compiles methods to native code which are then pushed back to the application. This design greatly benefits from multi-core configurations and makes it possible to render and optimize the page at the same time.

Chakra not only compiles JavaScript code, but also implements techniques to optimize type information. Namely *type representation* and *polymorphic inline caching* for fast property access. Another feature which results in speed improvements is the ability to *remove dead code* in order to avoid useless code to be executed. Because of this optimization technique, Microsoft scored much higher in the popular SunSpider [Web11b] JavaScript benchmark than the competitors in certain tests. As a matter of fact Microsoft was even accused of cheating [Hol10].

2.3 HTML5

The building blocks which make up the link between the processing part of the application and the user are defined in HTML.

In the past, HTML was developed as a markup language for mostly static content, supporting only elements for layouting and formatting text and images. Since the appearance of JavaScript, the document object model was no longer supposed to stay static, but subject to be altered in numerous ways yielding in a new area of user experience inside the web browser. The most recent finalized HTML specification is HTML 4.01 which was approved in 1999 [W3C99]. Since then, the web has undergone major changes triggered off by different parties. Due to massive increase of available Internet bandwidth, multimedia content was no longer restricted to be consumed off-line, but on-demand. The features available through HTML4 were not designed to deliver such an experience to the user, such as video playback and dynamic 2D & 3D graphics. As a consequence, many companies started building their own, proprietary, technologies to close the gap. Until today, Adobe's Flash Player is one of the most popular platforms to present multimedia content to the end-user [Ado11b].

In order to eliminate the dependence on additional software and to standardize the use of multimedia elements, the HTML consortium started working on HTML5 in mid 2004 [Hic04]. Although many parts are already considered to be finished the document is still in draft.

Since then the document remains still in draft, but many parts are already considered to be finished. The differences between HTML4 and HTML5 can be obtained from [KP11].

The most interesting features of HTML5 for building an interactive AR pipeline in JavaScript are the support for *video*, bitmap drawing with *canvas*, realtime 3D rendering in *WebGL* [Khr11f] and media-streaming through *getUserMedia* [WHA11b]. These four building blocks are explained in greater detail in the following sections (see 2.3.2).

2.3.1 Layout engines

While JavaScript engines are responsible for parsing, interpreting and executing JavaScript code, layout engines are determined to parse, interpret and render the Document Object Model (DOM) written in HTML. *Gecko*, *WebKit*, *Trident* and *Presto* are the four most popular layout engines used throughout the browsers.

Gecko was developed by Mozilla and is used within a wide range of applications, such as Firefox and Thunderbird. The most recent version is 8 [Moz11a] which is used in

Firefox 8. Google Chrome and the Safari browser both use Webkit as as layouting engine. The former builds on top of Webkit1 [Goo11i], whereas the latter uses Webkit in its most recent version v2 [App11c] in Safari 5.1. The main difference between those two versions is that the latter is designed to seperate the processing of the rendering UI from the layout-engine. This mechanic is similar to the approach used in Chrome [Chr08]. The Opera web browser relies on the Presto layout engine, whose current version is 2.9.168 used in Opera 11.5 [Bov11]. Lastly, Trident is used within Microsoft's web browser whose latest version 5.0 is used in Internet Explorer 9.0 [Mic11j]. More details on the HTML5 compliance can be found in the browser-feature matrix (see 2.3.7).

2.3.2 Video

The video element allows the browser to play video files without the need for additional plugins. It offers a wide range of attributes common to media players to provide rich user experience. For further customizations and advanced usage a JavaScript API is available.

A first proposal for a video element was added to the HTML5 specification on behalf of Opera in early 2007 [KI07]. A few days later a demonstration of playing back ogg-encoded video directly inside the browser was given, using an experimental build of Opera [Jam07]. One year later, in March 2008, Apple released Safari 3.1 being the first public web browser featuring the video element [App11b]. Since then, one browser after the other caught up, including Microsoft with its release of Internet Explorer 9.0 in 2010. A summarized overview of video support among the different browsers can be found in section 2.3.7.

Altough the element is standardized, there are still compatibility problems whith playing media files in different browsers. The root of the problem lies in the various formats media files may be encoded in, which is mostly up to the developers to decide. There exist three widely accepted standards for video encoding namely *ogg*, *H.264* and *WebM*. [WHA11a] Since no single format is supported by all browsers (see 2.3.7), there is still the need to provide at least two different versions of a videoclip. A combination of *ogg* or *WebM* along with h.264 is accepted by the majority of browsers.

HTML5 video is considered being a serious competitor to the Adobe Flash Player since more and more news-sites and video-portals are offering HTML5 video natively [CZ10] [Dou10]. Another factor for increasing popularity is the availablility on mobile phones, especially since Apple favors HTML5 over Adobe's products [Job10].

Example The code in Listing 2.2 shows a simple markup of how to use the video element providing two differently encoded clips.

```
<video src="video.webm" width="320" height="240" controls="controls">
    <source src="video.ogg" />
</video>
```

Listing 2.2: Video-Element

2.3.3 Canvas 2D

The canvas element allows drawing of 2D shapes and images dynamically by updating a bitmap area. No markups are available for rendering shapes, instead the only way of drawing is made available through a JavaScript API. The canvas element features an extensive set of drawing tools as well as low-level pixel access to the underlying data.

Altough, drawing vector graphics using SVG or VML (IE) was supported, drawing bitmap graphics inside the browser window was by no means possible. Remedy was found by using plugins, or server-side image generation, even attempts were made to draw lines using the DOM combining tables and borders.

The canvas element was first proposed by Apple in 2004 [Hix04] and made available with the realease of Safari 2.0 in the year 2005 [App11a]. In late 2005 Mozilla followed with Firefox 1.5 [Rud05] and Opera in 2006 [Ope06] to support canvas. Because Google Chrome builds upon WebKit, canvas has been supported since its first release. The last one to follow was Microsoft and finally added support for its newest member of the IE family in version 9.0 [Mic11e] released in 2010.

Example The code in Listing 2.3 outputs a red-stroked rectangle.

```
<html><head>
    <script type="application/javascript">
      window.onload = function run(){
        var ctx = document.getElementById("canv").getContext("2d");
        ctx.lineStyle = "red";
        ctx.strokeRect(20,20,20,20);
      };
    </script></head>
  <body>
    <canvas id="canv" width="640" height="480"></canvas>
  </body></html>
```

Listing 2.3: Canvas-Element

2.3.4 WebGL (Canvas 3D)

WebGL is the latest installment of rendering real-time 3D content inside the webbrowser. Its major advantage over frameworks like Stage3D (Adobe) or Silverlight (Microsoft) is simply the native implementation. There is no need for additional software and it runs on all platforms if the hardware is supported. The latest stable version is 1.0 which was released in February 2011 [Khr11f].

WebGL is the result from an initiative within the Khronos Group to create a cross browser standard for rendering 3D graphics inside the web browser. The working group started in August 2009 supported by many industry leaders such as AMD, Ericsson, Google, Mozilla, NVIDIA and Opera [Khr09a]. Two years earlier, Mozilla and Opera demonstrated separate implementations of a 3D canvas, which acted as a basis for further development. Mozillas implementation was already based on OpenGL ES 2.0 [Vuk07] whereas Opera used an abstract model to support non-OpenGL platforms such as Direct3D [Joh07]. During development of WebGL the important aspects were hardware acceleration, cross platform compatibility and security. All major browser brands have already implemented or previewed their attempts (see 2.3.7) with the exception of Microsoft, who does not want to include WebGL inside their Internet Explorer due to security concerns [Mic11i].

Basically WebGL is a JavaScript binding to OpenGL ES 2.0. OpenGL ES 2.0 is a subset of the OpenGL graphics API which is designed for the use within embedded devices such as mobile phones, tablets and small form factor computers. Differences between OpenGL ES 2.0 and WebGL can be obtained from [Khr11b]. The API is extremely streamlined and mostly relies on shader programs written by the developer. Because of this low-level design, many frameworks and scenegraphs have recently been released to reduce the complexity and lower the entry bar for developers entering the field. For example, three.js [Cab11], scenejs [Kay11] and spidergl [Ben10].

2.3.5 getUserMedia

Owing to the fact that HTML and JavaScript are running inside a sandbox, there is no direct access possible to the computer's hardware. Platforms such as Java, Silverlight and Flash offer limited access to media devices such as webcams and microphones. In order to close the gap, the HTML5 standard has already drafted a specification for such an API. The MediaStream API, as part of the Web Real-time Communication initiative (WebRTC) [Goo11h], provides an interface named getUserMedia to get access to the user's

media devices [WHA11b]. The specification is still in a very early stage and not likely to released in the near future.

Most of the pioneering work was achieved by the Ericsson Labs which created several prototypes for live communication purposes in HTML5 in a series called *Beyond HTML5* [Eri11b]. Older work built upon the former specified *device* element which was removed in the favor of a JavaScript only API [htm11]. The first preview of a web browser supporting the MediaStream API and getUserMedia interface was released to the public in March 2011 by Opera [Ric11]. The Android based Opera mobile web browser was capable of requesting a media device, such as the built-in camera, from the user. Two months later, in May 2011, Ericsson labs released their modified WebKit library, which contained a developers preview implementing the WebRTC proposal including camera access [Eri11a].

Another initiative is the WebRTC community, which is an open source project supported by Chromium, Mozilla and Opera to implement the proposal [Goo11h]. An already existing C++ API can be tested by active browser developers, but there is no access through JavaScript yet available. [Bev11].

Example The code in Listing 2.4 prompts the user to select an available device (preferably the camera facing the user) and attaches the web-camera stream to an embedded *video* element.

```
1 <script type="application/javascript">
2   navigator.getUserMedia("video user", function(stream){
3     video.src = stream;
4   });
5 </script>
```

Listing 2.4: Canvas-Element

2.3.6 Notable HTML5/JavaScript projects

This section presents some HTML5 & JavaScript projects which demonstrate the capabilities of today's webbrowsers.

JSMad - MP3 decoding

JSMad [WN11b] is a JavaScript port of the libmad library allowing mp3 files to be decoded directly in JavaScript. In order to actually play the decoded content, either the Web Audio

API provided by Chrome, or Audio Data API from Mozilla is used. The W3C has not yet decided upon which API to be specified in the HTML5 standard [W3C11b].

The JavaScript port was created by four people which contributed 15,000 lines of code. The JavaScript library was published open source under the GLPv2 licence and has been made available on GitHub [WN11a].

PDF-Rendering

While millions and billions of PDF files are floating around the web, there is still a need for additional plugins to be installed in the web browser to view those files. With the idea of rendering PDFs directly in the browser, a library called *pdf.js* [Moz11l] was developed with the help of Mozilla Labs and their community. Although the renderer is not yet completed and still in an early stage, it is already able to display quite a few files. The code is also available on GitHub [Moz11k]

The parser is basically implemented as a PDF to JavaScript compiler which takes the PDF bytestream, translates the draw commands to JavaScript code and executes it. The resulting JavaScript code mainly consists of draw calls to a HTML5 canvas. One particular problem with canvas is the fact that there is no text selection possible. Using other rendering techniques such as SVG and WebGL is in planning in order to combine speed with usability.[Pit11]

Node.js - Event-driven server

A totally different use of JavaScript on the web is instead of running in the clients browser being employed on the server's side. *Node.js* [Joy11b] follows such a paradigm and provides a rich environment for event-driven application development. It is mainly used as a web-server for real-time application. The code runs on top of Google's JavaScript Engine V8 and provides many extensions such as access to the filesystem. This project is mainly developed by the community and also available through github [Joy11a].

Emberwind - 2D Style cartoon game

A small team within Opera [Möl11b] is currently developing and evaluating a 2D style cartoon game which completely relies on HTML5 technology. The basic idea behind the project is to replace Adobe Flash technology, used for many games, with native web technology. The current developmental process can be followed at github, where the code is actually published [Möl11a].

The game engine inhabits two different renderers based on either `canvas` or WebGL. The former is used within browsers which do not support WebGL and the latter for browsers which benefit from hardware-accelerated rendering . The game incorporates a collision detection and movement system, as well as support for Flash-free audio.

2.3.7 Browser/Feature Matrix

The table in 2.1 summarizes the browsers' support for `canvas`, `video`, WebGL and `getUserMedia`. The number indicates the first publicly available version implementing the particular feature.

Table 2.1: Cross-browser feature-matrix

Browser	<canvas>	<video>	WebGL	`getUserMedia`
Firefox	1.5[fc]	3.5[fv]	4.0[fw]	no[fg]
Chrome	1.0[wc]	1.0[wv]	9.0[cw]	no[cg]
Opera	9.0[oc]	10.5[ow]	Alpha (10.5[ow1] & 12[ow2])	Alpha 12[og]
Safari	2.0[wc]	3.1[sv]	5.1 (disabled)[sw]	Labs[sg]
IE	9.0[ie]	9.0[ie]	no[iw]	no

[fc] `http://www.squarefree.com/burningedge/releases/1.5-comprehensive.html`
[fv] `https://developer.mozilla.org/en/HTML/Element/video`
[fw] `http://www.mozilla.org/en-US/firefox/4.0/releasenotes/`
[fg] See [Nar11]
[wg] `http://developer.apple.com/library/safari/#documentation/AudioVideo/Conceptual/HTML-canvas-guide/Introduction/Introduction.html`
[wv] `http://www.webkit.org/blog/140/html5-media-support/`
[cw] `http://chrome.blogspot.com/2011/02/dash-of-speed-3d-and-apps.html`
[cg] See [Bev11]
[oc] `http://www.opera.com/docs/changelogs/windows/900/`
[ov] `http://my.opera.com/core/blog/2009/12/31/re-introducing-video`
[ow1] `http://my.opera.com/core/blog/2011/02/28/webgl-and-hardware-acceleration-2`
[ow2] `http://my.opera.com/desktopteam/blog/2011/10/13/introducing-opera-12-alpha`
[og] See [Øde11]
[sv] `http://developer.apple.com/library/safari/#documentation/AudioVideo/Conceptual/Using_HTML5_Audio_Video/Introduction/Introduction.html`
[sw] `http://fairerplatform.com/2011/05/new-in-os-x-lion-safari-5-1-brings-webgl-do-not-track-and-more/`
[sg] Ericsson Labs - WebRTC built upon WebKit (see [Eri11b])
[ie] `http://msdn.microsoft.com/en-us/ie/ff468705`
[iw] WebGL Considered Harmful (see [Mic11i])

2.4 AR on the web

The web is not be seen mature enough to serve as a platform for AR applications based only upon native web-technology. Techniques such as the access of live video streams or real-time 3D rendering were just not available. Hence the majority of the solutions are not based upon pure HTML standards but rather on proprietary technologies in combination with browser plugins.

2.4.1 Proprietary technology

AR solutions based on proprietary code need specific plugins installed within the browser in order to work properly. The lack of accessing hardware components, poor execution speed of JavaScript and the absence of rendering capabilities left no choice but to implement solutions in proprietary technologies. This approach has certain advantages: Already existing code can easily be reused without or little modification and there is also a performance advantage in comparison to HTML5 (see Section 5.3). On the other hand, these benefits cause serious problems when trying to support different platforms as well as different devices. Cross platform compatibility greatly depends on the availability of plugins.

Proprietary web-components can be divided into two categories. First, plugin development to extend the browsers capabilities and second, building on existing frameworks such as Adobe Flash Player or Microsoft Silverlight. While the former approach requires the user to explicitly download and install the software, hence decreasing the users acceptance, the latter works without user interception if the appropriate framework is installed.

2.4.1.1 ActiveX

Mirosoft's ActiveX offers the most popular approach for plugin development although the controls only run inside Internet Explorer. Companies such as Bitmanagement [Bit10] and metaio [Bau07] offer solutions built upon ActiveX browser plugins to display and render 3D content.

A technology created by Microsoft in 1996 [Mic96b] to extend the static web pages with active content to deliver media-rich and interactive content to the user. Since the introduction to the Internet Explorer 3.0 it is a core component and used as a plugin system to extend the functionality of the browser. ActiveX is not bound to the usage inside the Internet Explorer, due its COM and OLE component model, many other applications can

make use of them.

The controls may be written in different languages which support the COM component development, like C++, Visual Basic or .NET [Mic96a]. Since all those components get executed natively, without the use of a sandbox, certain advantages and disadvantages arise. First, the delivered performance is the same as for native software and second, access to system services is granted. These facts make it easy to re-use already existing software components without any or little adaptations. On the other hand, using native technology leads to security risks due to malicious software.

Since ActiveX components do not run in any kind of sandbox, executing malicious code is rather easy. In order to prevent this, a security mechanism called `Authenticode` [Mic11b] is implemented in order to trace back the creator of the components as well to maintain a database for valid controls.

2.4.1.2 Adobe Flash

Adobe Flash technology is widely used within the AR community. Popular AR engines such as FLARE* [Ima11], IN2AR [IN211] and FLARToolKit [Saq11] are built ontop of Flash offering either tracking fiducial markers (FLARToolKit, flare*tracker) or natural features (flare*nft, IN2AR).

Owing the fact that Adobe's Flash Player is widely accepted [Ado11b] it is one of the most promising platforms to develop AR applications on. Since it provides direct access to the user's web-cam, an efficient scripting language, and the capability to render real-time 3D graphics it offers all the crucial building blocks for AR solutions. Using the Alchemy Toolkit [Ado09c], developers may even write C++ instead of ActionScript code, in order to reduce resources required for cross-platform development. One major drawback is still the requirement for an additional plugin which does not offer all the functionalities throughout the supported platforms. While Flash Player support is available for many Android devices [Goo11a], Apple on the other hand refuses [Job10] the deployment of the Flash Player in the web browser on their mobile devices. Due to the increasing popularity of HTML5 development, it was recently announced that the Flash Player is no longer developed for the use in mobile browsers [Win11b]. The PC version is not affected and will be further developed for advanced gaming and delivering premium video.

2.4.1.3 Adobe Alchemy

Alchemy [Ado09c] is a research project within Adobe Labs with the goal to run general C/C++ code inside the ActionScript virtual machine which powers Adobe Flash Player and Adobe AIR. This approach helps to port already existing libraries written in C/C++ to Adobe's platform without the need to rewrite the whole software. The ActionScript virtual machine does not run the compiled C/C++ code directly, because the runtime environment is protected by a sandbox which does not allow to run arbitrary code. Hence, the code gets first compiled to LLVM, an intermediate bytecode representation, and finally to ActionScript-Bytecode. The resulting code is nothing more than compiled handwritten ActionScript code. Which means that the original C/C++ code does not run natively on the system, rather being interpreted inside the ActionScript VM. Consequently, the resulting performance cannot compete with native compiled code.

In short, this framework's purpose is not to execute high performance code, but to reuse already existing C/C++ modules in other ActionScript projects. Libraries such as the OggVorbis audio de/encoder or JPEG encoding based on the libjpeg C-library were already ported to ActionScript using Adobe Alchemy [Ado08].

Adobe Alchemy was released as an experiment in 2008 and did not receive any updates or bug-fixing since then. On the behalf of customer feedback Adobe received, a next-generation successor of the Alchemy platform is already in development [Ngu11].

Portability Another important aspect in proprietary solutions for the web is the provided portability. Since the resulting library is compliant to any other ActionScript library, it runs wherever the Flash Player is available.

2.4.1.4 Google Native Client

Google's way of reusing existing C/C++ code is to run it with the Native Client (NaCl) technology [Goo11e] present in the Chromium Browser. The approach is different to Adobe's because instead of compiling C/C++ to an intermediate representation for running inside a virtual machine, NaCl compiles to native code. Thus the name native client. As the code is compiled into native machine code, it does not offer the same degree of portability as Adobe's system, but manages to execute it almost the same performance as native applications.

NaCl does not target the creation of entire web applications, but concentrates on certain tasks which have to be carried out very fast. Besides, already existing C/C++

code may be reused.

Portability Due to the fact that the code gets compiled natively to x86 (32bit) and x64 (64bit) platforms, the usage is limited to standard PC configurations and excludes modern Smartphones which are mostly based upon the ARM architecture.

2.4.1.5 Microsoft Silverlight

Silverlight [Mic11h] is Microsoft's counterpart to Adobe's Flash Player offering a platform to embed rich-client functionality inside the web browser. The whole framework is built upon the .NET runtime environment, hence supporting all CLI languages. The compatibility is limited to Windows and Mac OS X operating systems as well as Windows Phone 7 and Symbian [Mic10a]. Due to this narrow support of platforms, the current penetration (66%) is still much lower compared to Java (76%) and Adobe Flash (95%) [Sta11].

While AR applications are still rarely for Silverlight, a port of the popular ARToolkit is available and published under SLARToolkit [Sch10].

2.4.1.6 Java Applets

Java applets [Ora10] deliver rich internet applications to the user, featuring custom controls and graphics support. Applets are developed in the Java programming language and executed in the Java virtual machine. Usually, applets run inside a sandbox of the browser for security reasons.

Applets were introduced in 1995 [Sun96] along with the first version of Java. Since that time, applets have often been used for visualization taks of complicated processes. Because of its poor multimedia capabilities, fairly strong competitors and the missing support on mobile devices the popularity of Java applets is declining.

Nevertheless, AR solutions may be implemented using Java applets. A port of the ARToolKit called NyARToolkit [nya11] is implemented in pure Java.

2.4.2 HTML5

2.4.2.1 JSARToolkit

The somewhat most interesting project recently released is the port of the popular ARToolkit to JavaScript created by Ilmari Heikkinen [Hei11]. This project is part of Mozilla's Web O'(pen) Wonder named Remixing Reality [Moz11e] showing the power of today's JavaScript engines and creativity of developers.

The novelty of this work is, that is was the first attempt implementing an AR system in native HTML5 without any use of prorietary plugins or third-party software.

2.5 Discussion

This chapter presented an overview of Augmented Reality including various tracking techniques, in-depth information about web-technologies and existing web-based AR solutions.

Tracking is a fundamental element of AR, regardless of whether the tracker makes use of mechanical sensors, inertial sensors or vision-based approaches. The main interest lies within tracking of natural features (NFT) and their algorithms to achieve efficient and robust tracking. While methods exist which only run on decent PC hardware [KM07] or require special equipment [RD06a], others even run on mobile phones [WRM+10]. These visual tracking pipelines are mainly based on high performance, compiled programming languages like C++.

For this particular reason, existing web-based AR solutions are mostly built upon proprietary web-technologies, such as ActiveX [Bau07] [Bit10] or Adobe Flash [Ima11] [IN211]. Proprietary solutions benefit from high performance (ActiveX, Google Native Client) and a multimedia-rich feature-set (Flash, Silverlight), yet they suffer from security vulnerabilities (ActiveX), limited portability and dependence on the respective software producer.

When focusing on scripting languages for high-performance computing tasks, their environments are required to run the code efficiently. Modern JavaScript engines reduce the overhead of interpreting the language by applying mechanisms such as tracing [Man09] and native code generation [Goo11b] with the help of just-in-time (JIT) compilers. These techniques can speed up the execution up to a hundredfold and demonstrate their power in tasks such as PDF rendering [Moz11l], h.264 video decoding [Beb11] and games [Möl11b].

The realization of an NFT AR pipeline using native web-technology not only requires high-performing JavaScript engines, but also a rich set of multimedia-components starting with camera image retrieval to the final 3D rendering stage. The draft HTML5 [W3C11a] specification offers elements for device access (`getUserMedia`) [WHA11b], video playpack (`video`) [WHA11a] and real-time 3D rendering (WebGL) [Khr11f]. A JavaScript port of the popular ARToolKit, JSARToolKit, is based upon some of these components and demonstrates that the web is already capable of providing simple AR solutions.

A concept of how to realize a web-based natural feature detection and tracking pipeline without the use of plugins is presented in the next section.

Chapter 3

Concept

Against the background of the leading questions introduced in 1.1, this chapter presents the approach chosen. First, the basic components needed for a vision-based AR-pipeline are given, followed by approaches using proprietary technologies to realize such a pipeline. The third part focuses on the feasibility of implementing a natural feature tracking AR-pipeline using native web-technologies. After that, the current limitations of the presented approach are depicted in detail. Then, the evaluation framework is introduced and the concept is compared to other web-based approaches. Finally, the concept of the NFT-pipeline is presented, covering the two-stage approach which divides the pipeline into a detection and tracking phase.

3.1 Components of a vision-based AR-pipeline

Typically, a simplified AR system consists of three major parts as illustrated in Figure 3.1. First, an image acquisition component which captures the camera's current view and buffers the incoming data. The second part contains the programming logic where computer vision algorithms are applied to the image data to determine the current position of the camera. The position data is finally provided to the rendering stage, the last component of the pipeline.

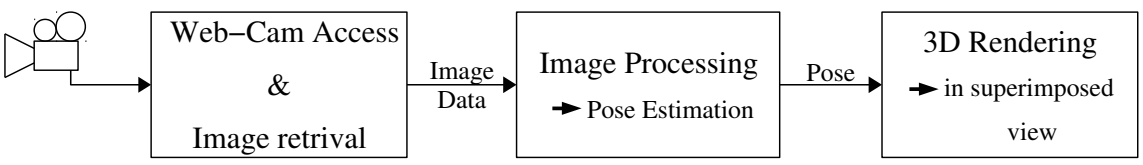


Figure 3.1: Components of a vision-based AR-pipeline

Now that all the elementary components are defined, the question arises how such a sytem could be implemented with web-technologies?

3.2 Proprietary approaches

In this section, six different approaches are presented using proprietary technology to implement the previously described AR-pipeline. Starting with ActiveX, Microsoft's extension technology for the Internet Explorer, followed by the Adobe Flash Player and Alchemy running in the ActionScript VM. After that, approaches using Google Native Client and Microsoft Silverlight are described. Lastly, a possible AR system using Java Applets (see Section 2.4.1.6) is shown. For each of these technologies an implementation approach is presented and the respective advantages and disadvantages are pointed out.

3.2.1 ActiveX

The development of an AR system within ActiveX (see Section 2.4.1.1) is not very different to programming native windows applications.

The web-cam can be accessed directly through DirectShow [Tom10] whereas the image processing and business logic can be implemented in languages such as C++ or .NET. For real-time 3D graphics, either OpenGL or DirectX may be adopted.

ActiveX components get executed natively on the system, which yields to a high performance, but introduces security holes which may be exploited by malicious software. In addition, these components can only be used within the Internet Explorer excluding all other browsers, including mobile versions.

3.2.2 Adobe Flash

Implementing AR applications in Flash (see Section 2.4.1.2) is relatively easy [Riv09] since all the building blocks are already provided. The user's web-cam can be accessed using the media package, whereas direct access to the underlying video stream is prohibited. Instead the video can be drawn onto a bitmap which gives access to the video-frame's pixels. Image processing algorithms can then be directly implemented in ActionScript. When it comes to rendering real-time 3D graphics, the newly introduced Stage3D framework [Ado11c] can be employed.

The ActionScript bytecode gets executed in a virtual machine, providing a high level of security. While Adobe's Flash Player is installed on most PCs and Android

smartphones, other platforms such as iOS do not support Flash in their respective web browsers [Job10]. Also, the recently announced stop of the development of Flash for mobile web browsers [Win11b] makes the technology only interesting for PC usage.

3.2.3 Adobe Alchemy

Since Adobe Alchemy (see Section 2.4.1.3) is based upon the Adobe ActionScript VM, implementing an AR system is no different than using Adobe Flash (see Section 3.2.2). More information on employing Adobe Alchemy in an AR solution can be found in Section 4.3.

3.2.4 Google Native Client

Google Native Client (see Section 2.4.1.4) is currently only supported in Google Chrome web browser, which is also available only on the PC. For this reason, the possible audience is seriously limited, inspite of the fact that the technology features almost native performance and mechanisms to ensure secure execution.

For more details on the implementation of an AR solution using Google Native Client see Section 4.4.

3.2.5 Microsoft Silverlight

Due to the rich multimedia features Silverlight (see Section 2.4.1.5) offers, all of the components needed for an AR application are already present. Accessing the user's media devices is made available through the `VideoCaptureDevice` and `CaptureSource` interfaces. Grabbing the video-frames from the `CaptureSource` is made possible directly through `CaptureImageAsync`. For creating the programming logic, CLI languages such as C# and VB.NET are supported. With the introduction of Silverlight 5, the ability to use hardware accelerated 3D graphics was added [Mic11a].

On the one hand, Microsoft Silverlight benefits from a rich feature set and provides easy access to multimedia devices. But on the other hand, the compatibility is limited to Windows, MacOS X, Windows Phone 7 and Symbian excluding other popular platforms such as Linux, Android and iOS. Thus, the current penetration cannot keep up with Adobe Flash [Sta11].

3.2.6 Java-Applets

Since Java does not provide any interface for capturing media devices, an additional framework, the Java Media Framework [Ora11], is required. The JMF Framework extends the

Java 2 Platform and is considered being outdated. Once the video image is captured, the processing logic implemented in Java takes over. Since Java does only natively support drawing 2D graphics, the Java3D [jav11] extension is required for hardware accelerated 3D graphics.

Java Applets suffer from their limited multimedia capabilities and their narrowed support of platforms being only available on desktop web browsers. An advantage of Java Applets compared to ActiveX is the presence of a virtual machine executing the Java code providing a high level of security.

3.3 HTML5 approach

The approaches for realizing AR systems described in the previous section (Section 3.2) are all based upon proprietary web technologies. Due to their need for separate installation, security vulnerabilities and platform dependence, a uniform solution using standardized web technologies is from interest.

This section describes the potential of native web technologies such as HTML5, JavaScript and WebGL. To address the problems mentioned above: In Section 3.3.1 the camera access and image retrieval components are depicted. Then the focus is put on the image processing part of the pipeline describing various ways of handling computer vision algorithms. And finally, the capabilities of HTML5 of rendering 3D objects in real-time are presented in Section 3.3.2.2.

3.3.1 Camera access

The initial component of the AR-pipeline needs access to an external capturing device, such as a web-cam.

The `getUserMedia` (see Section 2.3.5) interface, specified in a draft document in [WHA11b], may be used for that purpose. Since the specification is still in an early stage, only a few web browsers actually support this interface (see Section 2.3.7). With the help of this interface, access to a multimedia device can be granted by the user. The resuling video-stream can then be attached to a `video` element.

Another approach is offered by Mozilla's Rainbow [Nar11] project which was started in 2010. It provides an addon for Firefox offering real-time video and audio recording capabilities through a JavaScript API. Due to it's experimental status, it works only under certain conditions which makes it hard to use. Due to the increasing popularity

of the WebRTC [Goo11h] initiative, the Rainbow project is currently being adopted to feature a new API making it compatible with the `getUserMedia` interface [Nar11].

3.3.2 Image processing

Once the image's data is aquired, the program's logic is executed and returns the camera's pose. web browsers offer various ways of carrying out complex computation tasks, needed e.g. for image processing. First of all, there is JavaScript, a multi-purpose language also capable of employing image processing algorithms. Second, WebGL in combination with GLSL (OpenGL Shading Language) might be used to carry out calculations on images. And lastly, WebCL provides another way of utilizing the CPU and GPU for complex computation tasks.

3.3.2.1 JavaScript

JavaScript, as thoroughly described in Section 2.2, offers constructs to manipulate and process images. Since JavaScript is a scripting language, the achived performance might not be satisfiable for real-time applications. However, JavaScript is available in all web browsers by default and provides widespread support regardless of different devices, platforms and systems.

3.3.2.2 WebGL

Apart from JavaScript, WebGL may be used for image processing as well, making use of the GPU. Since the whole graphics pipeline is based upon programmable vertex and pixel shaders, custom programs may be written within GLSL (OpenGL Shading Language) [Khr09b] to apply computer vision algorithms. Through the use of texures, the pixel-shader and a framebuffer, computation can be outsourced to the graphics card.

Due to limitations such as the reduced precision of the framebuffer (32bit integer [Khr11f]) and the overhead of copying the images from and to the GPU-memory, makes WebGL not an ideal candidate for carrying out image processing tasks. Compared to JavaScript, WebGL is not as widely adopted and is activated in only a few web browsers (see Section 2.3.7).

3.3.2.3 WebCL

Along with WebGL, WebCL is another approach utilizing the GPU for carrying out complex computations. It is a JavaScript binding defined by the Khronos Group to

use the OpenCL (Open Computing Language) [Khr11c] standard from within the web browser [Khr11e]. The initiative started in March 2011 and they are currently working on finalizing the API definition. Although the specification is still in draft, there already exist two open source prototypes, one provided by Nokia [AN11b] for the use in Firefox, and one by Samsung [Sam11] based on WebKit. These prototypes still suffer from low maturity but already demonstrate the potential of employing the GPU instead of JavaScript. Speed-ups up to 100x are promised when using a modern graphics card [Khr11e].

OpenCL is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. This platform is very well suited for carrying out all kinds of computations, including image processing. The major problem is that neither a finally released web browser exists that implements the WebCL standard, nor is the specification fully stable yet. In the future, not only desktop platforms may benefit from WebCL, but there are also plans to implement a version for mobile web browsers do already exist [AN11a].

3.3.3 3D rendering

The third stage of the AR-pipeline requires real-time 3D rendering of objects in the superimposed view. Basically, there exist two possibilities to draw 3D objects inside the web browser. Either, rendering hardware-accelerated 3D graphics with WebGL, or creating a custom 3D rendering pipeline which is executed on the CPU and uses Canvas2D (see Section 2.3.3) for output.

Due to utilizing the GPU, the performance of WebGL is significantly higher compared to the software rendering approach in combination with `canvas`. Another advantage is the existence of many WebGL frameworks [Khr11d] facilitating development while renderers based on Canvas2D are rarely available [Cab11]. While WebGL is officially only supported in Firefox and Chrome (see Section 2.3.7), the latter approach can be employed in all modern browsers.

3.3.4 Discussion

After weighing the pros and cons of the different HTML5 approaches presented in the previous section, the following components were selected.

For the camera access, the `getUserMedia` interface was chosen because of its initiative for standardization [WHA11b] and available prototype implementations.

Due to the widespread support and highly portability of JavaScript, it was preferred

over WebGL and WebCl in image processing tasks. Although latter two approaches provide higher performance they either dot not support floating-point precision (WebGL) or they suffer from low maturity and limited availability (WebCL).

As for the rendering part, WebGL was selected due to its hardware accelerated nature and higher availability of frameworks to build upon.

3.4 HTML5 AR-pipeline technology stack

In order to glue together the building blocks described in the previous section to create a working pipeline in HTML5, additional components are required (see figure 3.2). The entry point of the system is formed by the camera stream acquired from the `getUserMedia` interface. Because the stream is encapsulated and is not intended to be intercepted, no access to the raw image data is possible. A workaround through two supplementary components can be established to capture the image's pixels. First, a video element acting as a data-sink for the stream. And second, a canvas element employed as an image buffer offering methods to acquire the bitmap's pixel. Once the array of pixels has landed in the image processing module, the actual computation of the pose begins. The resulting pose is then pushed to the final part of the pipeline, the rendering stage, where the viewpoint is adjusted accordingly.

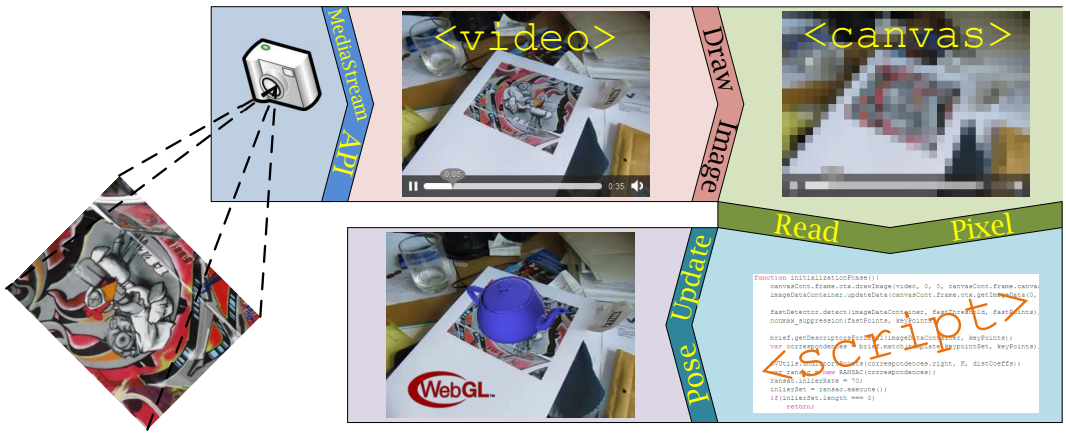


Figure 3.2: HTML5 AR-Stack

3.4.1 Which criteria have to be fulfilled by the web browser?

In short, a web browser is capable of runnning the designed AR-pipeline if the following features are present:

- Camera access: getUserMedia
- Image Processing: JavaScript
- Rendering: WebGL

3.5 Current limitations

While all of the HTML5 components needed within the AR-pipeline are either finally specified (`video`, `canvas`, WebGL) or still in draft (`getUserMedia`), the availability is still limited. The camera access through the `getUserMedia` interface is the least supported element among all. Currently only two prototypes exist (see Section 2.3.7). WebGL on the other hand is already in use by many browser manufacturers but in some cases officially not unlocked (Apple Safari) or only available as prototype implementation (Opera). The only company refusing to implement WebGL in their browser is Microsoft [Mic11i].

Besides the limited support of WebGL and `getUserMedia`, another restriction may be caused by JavaScript itself. Despite the fact that the JavaScript performance is already highly optimized in the respective engines, the difference to native code is still significant (see Section 5.3).

3.6 Evaluation of performance

Since using JavaScript and HTML5 is not the only way of creating an AR-pipeline in web-technologies, alternative approaches are considered employing proprietary technologies. The focus is set to re-implement the detection pipeline in C/C++ in order to reuse that code in Google Native Client (GNC) and Adobe Alchemy. The interest of comparing JavaScript to Adobe Alchemy is to evaluate the typed ActionScript bytecode against untyped JavaScript code, both running in virtual machines. GNC, on the other hand, runs the code directly on the CPU and promises equal performance to native applications [Goo11e]. Comparing these technologies might provide information of possible bottlenecks and issues inherent in JavaScript helpful for further optimizations.

3.7 NF-Tracking pipeline

The goal of this work is to design and implement a complete AR system using natural features for tracking. Most of the design decisions are based upon two requirements, *efficiency* and *simplicity*. Efficiency because of the limitations in computational power

provided by JavaScript and considerations for making it available on mobile phones as well. Since computer vision libraries for JavaScript are non-existend (compared to OpenCV and the bindings to C/C++ and other languages) all algorithms have to be implemented from scratch.

Basically, the pipeline is designed in a two-stage manner (see Section 2.1.5.1) where the initialization is decoupled from the tracking part as illustrated in Figure 3.3. During the initialization phase, the tracker tries to detect a known target in the current image and determines the initial position of the camera. If this stage returns a valid pose, the pipeline switches over to the second, the tracking phase, and keeps track of the camera by incrementally updating its position. Whereas the tracking is not as robust to sudden scene changes as the initialization phase, it may fail under certain circumstances such as fast movement or severe light changes. If this happens, the tracker falls back to the detection stage where it tries to re-initialize the pose.

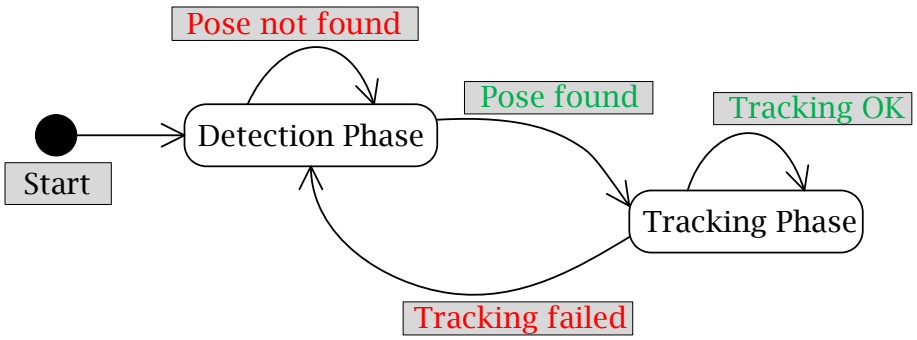


Figure 3.3: Two-Stage AR Tracking pipeline

This approach allows more time-consuming methods for finding the initial pose, since it is only carried out at the beginning, or when the tracking is lost, which happens only occasionally. In addition, a separate tracking phase benefits from the information extracted from the detection stage and uses that knowledge to achieve speed improvements.

3.7.1 Detection phase

The purpose of the detection phase is to determine the initial pose of the camera without any prior knowledge of the scene. The only available information is the target image to search for. Hence, this task requires image detection and matching techniques. The usual procedure addressing this problem consists of three subsequent steps. First, the extraction of local features which can be repeatedly found under different conditions. Second, creation of discriminative and unique descriptors for each feature which are then used in the final

step, the matching. The matching stage tries to infer correspondences between two sets of descriptors, based on similarity measures, in order to re-detect an already known image.

As already mentioned, the detection does not occur very frequently, thus the requirements are less strict than for the tracking stage (see Section 3.7.2):

- running interactively (approximately 10 fps)
- being robust to scale (to some level) and translation
- being robust to lighting and slight viewport changes

The final design of the detection pipeline, meeting the above mentioned requirements is illustrated in Figure 3.4. The reasons why each individual approach was chosen is explained in the following sections.

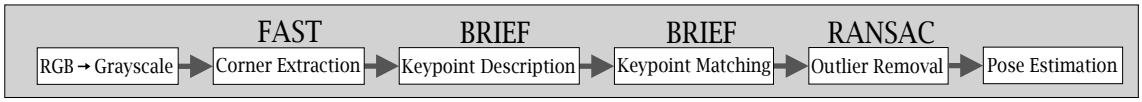


Figure 3.4: Design of the Detection-Pipeline

3.7.1.1 Feature detection

Many different mechanisms exist to extract local features from an image (see Section 2.1.5.2). For the sake of efficiency, only simple feature detectors are considered being useful, in particular those which respond to corner-like structures. In order to achieve high-speed corner detection, some restrictions in visual performance have to be taken into account. While the Harris [HS88] detector responds to corner-like structures and acts very precisely, it is quite expensive in terms of CPU-time. Calculating derivatives and weighing with a Gaussian kernel is very time consuming-because several convolutions are applied on the entire image. First implementations in JavaScript revealed (see Section A.1), that the Harris detector does not meet the desired requirements. The time needed for detecting Harris features in a single image is around 50-80ms.

FAST was then chosen as an alternative to Harris. FAST, described in [RD06b], speeds up the extraction of local features compared to Harris. An experimental implementation showed significant improvement in performance yielding to an average of 10ms for one frame (see Section A.1), which was considered being fast enough.

In order to fulfill the requirement of being robust to scale, the features of the target image are detected on multiple scales. This approach is only considered useful for being

applied to the target image and not for each incoming camera frame. Thus, the target image is detected even if it is further away from the camera, but does not work for close-ups.

3.7.1.2 Keypoint description

As soon as a set of local feature-points has been extracted, a discriminative description needs to be created for later matching. Some of the most popular methods used in this step are SIFT [Low04] and SURF [BETG08] in various adaptations. Both of these are highly discriminative and offer state-of-the-art techniques for image matching. The discriminative nature lies within the multidimensional descriptor which uses a 128 dimensional feature vector for SIFT whereas SURF results in 64 dimensions. Due to the complexity of those approaches, real-time detection cannot be easily achieved. As pointed out in [CLSF10] the efficient SURF descriptor needs over 300ms implemented in C++ and is consequently not considered being part of the JavaScript detection pipeline.

A keypoint descriptor promising real-time description and matching techniques is BRIEF [CLSF10]. Since BRIEF has a very straightforward model, it is considered being suitable for a JavaScript implementation. The descriptor purely relies on binary string representation and features a very small memory footprint ranging from 64 to 512 bits for each interest-point. The size of the descriptor is based on the number of binary brightness tests carried out per keypoint. Each test returns either 0 or 1, which for example means that 128 tests yield to 128 bits stored within a binary string. A sample pattern of those randomly generated brightness tests is shown in Figure 3.5.

Since initially conducted tests were statisfactory BRIEF has been chosen to be part of the pipeline.

3.7.1.3 Keypoint matching

Given two sets of local descriptors, one extracted from the scene and the other acquired from the target image to be matched with. In order to find corresponding points between them, a similarity measure is applied to all pairs of features. This measure is basically a distance-function which gives information about the similarity between two features. The computational complexity of comparing all possible pairs is the fundamental problem of this stage. Elaborate algorithms and classification techniques may be used to speed up the matching process when relying on multi-dimensional descriptors [LF06]. These mechanisms significantly reduce the time for finding correspondences for SIFT or SURF

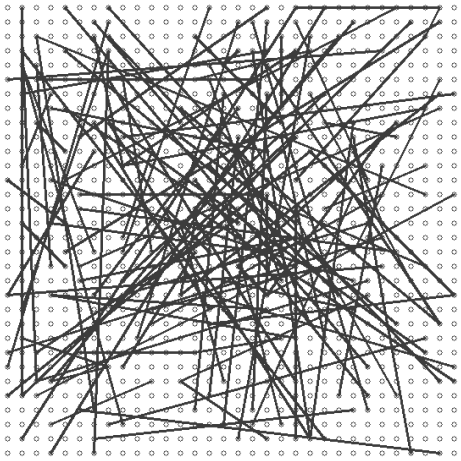


Figure 3.5: BRIEF: Binary Brightness Tests taken from [CLSF10]

descriptors, but cannot be easily applied to binary descriptors such as BRIEF. An analysis of Approximate Nearest Neighbor search on binary vectors is presented in [TLF11] demonstrating the techniques to speed-up matching of this particular type of descriptors.

The similarity measure employed in BRIEF builds upon the Hamming-distance [Ham50] function. Hence, two features are considered being a match if the number of unequal bits is minimal. Since the calculation of the hamming-distance is very cheap in terms of CPU-time (XOR followed by bitcount), brute-force matching is considered being suitable, although mechanisms exist reducing the number of possible candidates [TLF11].

3.7.1.4 Outlier removal

Since keypoint matching relies on noisy data and ambiguities, it is quite likely that the process may establish wrong correspondences. To detect and remove these outliers, the RANSAC [FB81] approach is considered being appropriate. RANSAC is also employed in [WRM+10] demonstrating that even low-end devices are able to handle the hypothesize-and-test framework.

3.7.1.5 Pose estimation

Once a valid set of corresponding points is available, the initial camera position can be estimated. This step requires the intrinsic parameters K of the camera which are identified in an off-line calibration process. A rough estimation of the camera's pose can be direcly extracted from the homography H between the target image and the current scene.

determining H Given the set of valid 2D to 2D point correspondences $x_i \leftrightarrow \acute{x}_i$ the transformation H is given by the equation $\acute{x}_i = Hx_i$. Because more than four correspondences are given, the set of equations is over-determined which means that no exact solution for H exists. Instead, a solution with the minimum error is from interest. This can be obtained using the DLT algorithm described in [HZ04].

recovering R & t The rotation R and translation t of the camera orientation can be directly obtained from the homography H according to [SB02] and [XKM09]. The equations 3.1 to 3.3 show the relation between R, t and H. As shown in Equation 3.1 the projection matrix P is a combination of the intrinsic camera matrix K and the extrinsic parameters R and t. The rotation part may also be split up into separate column-oriented vectors whereas the third column is the cross product between r_1 and r_2. The homography H implicitly contains the calibration matrix K and the two independent rotation vectors r_1 and r_2 in addition to t (3.2). The final extraction of the parameters is shown in Equation 3.3 as the rotation vectors are being normalized. The notation (e.g. $(K^{-1}H)_{1t}$) used in (3.2) is read as the following: Select the first column of the resulting matrix.

$$\begin{aligned} P &= K \begin{pmatrix} R & t \end{pmatrix} \\ &= K \begin{pmatrix} r_1 & r_2 & (r_1 \times r_2) & t \end{pmatrix} \end{aligned} \tag{3.1}$$

$$\begin{aligned} K \begin{pmatrix} r_1 & r_2 & t \end{pmatrix} &= H \\ \begin{pmatrix} r_1 & r_2 & t \end{pmatrix} &= K^{-1}H \end{aligned} \tag{3.2}$$

$$\begin{aligned} r_1 &= \frac{(K^{-1}H)_{1t}}{\|(K^{-1}H)_{1t}\|} \\ r_2 &= \frac{(K^{-1}H)_{2t}}{\|(K^{-1}H)_{2t}\|} \\ t &= \frac{2(K^{-1}H)_{3t}}{\|(K^{-1}H)_{1t}\| + \|(K^{-1}H)_{2t}\|} \end{aligned} \tag{3.3}$$

3.7.2 Tracking phase

Once the initial camera position has been determined, there is no need for re-detecting the template image at each frame. Based on the assumption that the video-framerate is high enough and the applied motion is adequate, the change between two subsequent frames is

considered being very small. Tracking mechanisms benefit from this assumption and are able to work more efficiently than in the detection phase.

The method proposed in [KM07] makes use of the the small-motion assumption and applies a patch-tracker to follow keypoints. Instead of relying on a target image, the surrounding is created on-line with a SLAM approach. Due to the parallel mapping and tracking, this method benefits from multi-core architecture and is known to perform well on systems where multi-threading can be implemented efficiently. Since JavaScript's ability to run code in parallel is rather limited (see Section A.2) this approach is considered being too complex for the time being.

A similar approach, using a target image instead of SLAM, was proposed in [WRM+10]. The novelty of this method is that it runs in real-time on mobile phones, which is achieved by elaborate algorithms and heavily optimized data processing. Because of the efficiency of the approach the usage in the tracking pipeline was considered being suitable.

The tracking's system is made up of two separate components, the patch-tracker and the pose refinement (see Figure 3.6). Former determines the motion applied from one frame to another while the latter adjusts the pose accordingly.

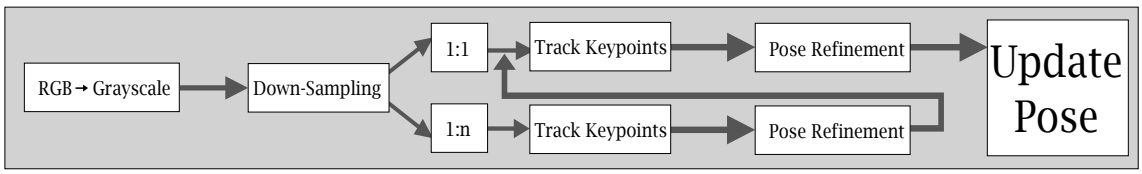


Figure 3.6: Design of the Tracking-Pipeline

3.7.2.1 Patch-tracker

As soon as the detection phase recovers the initial camera pose, subsequent changes are assumed to be quite modest. This helps to simplify and speed up the process of finding the same interest points in subsequent images. The basic idea is illustrated in Figure 3.7. Instead of searching the entire image for a certain template, only a small neighborhood area is taken into consideration. A tiny image patch (8x8 pixel) is extracted at the feature's last known location and searched within the immediate neighborhood of the expected area. The neigborhood area where the patch-search actually takes place is also referred to as search window. Locating the feature points within this defined window significantly speeds up the re-detection process.

Since the size of the window is kept small, neighborhood template matching may fail

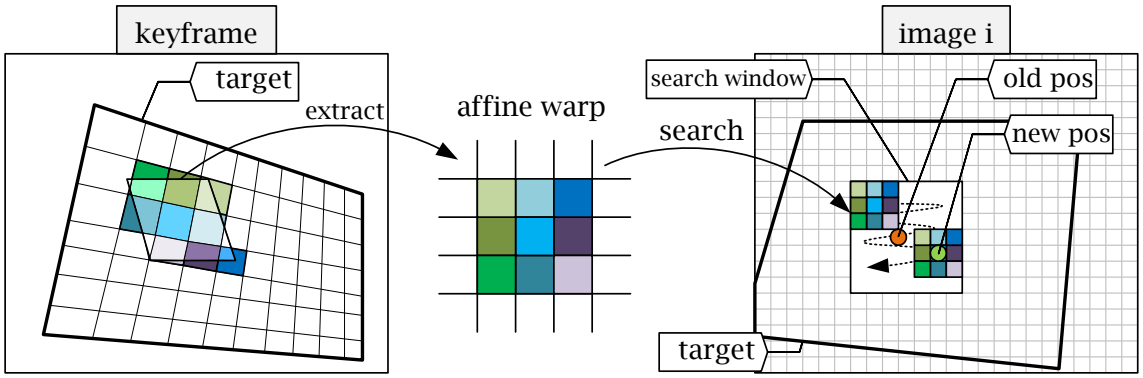


Figure 3.7: Affine warp extracted from the keyframe and matched in the current frame

during fast movement. A selected search radius of 4px around the patch results in a very small maximum possible motion of 80px per second when runnning at 20 fps. In order to increase the robustness during fast movements, the patches are tracked on a multi-scale level. Having the radius set to 5px on the second image level, motion up to $5px * scale(4) * 20fps = 400px/s$ is still being detected.

Keyframe As soon as the detection phase returns a valid pose, the tracker remembers the initial pose in combination with the frame it was extracted from. This particular frame is also referred to as keyframe and is used as reference for all subsequent patch searches. The keyframe gets assigned a to set of feature points and their position in world coordinates as well as the feature's plane.

Determining affine transformation Instead of extracting the patch from the previous image and matching it within the current, the keyframe provides the source of image patches. In order to determine the correct transformation between the keyframe and the current view, an affine warp is computed.

At first, an a-priori defined pattern gets re-projected from the current view on to the keyframe where the pattern's location is determined. The process is illustrated in Figure 3.8 and Figure 3.9 and works as follows: Each randomly selected feature point is projected from its center position into the current camera (red to red). The intersection with the camera's image plane marks the center of the feature point in the current video frame. In order to fetch the neighborhood region in the keyframe for later search and determining the correct scale, rays are sent back into the scene through the four neighboring pixels. These rays are then intersected with the feature plane and reprojected into the image plane of the keyframe.

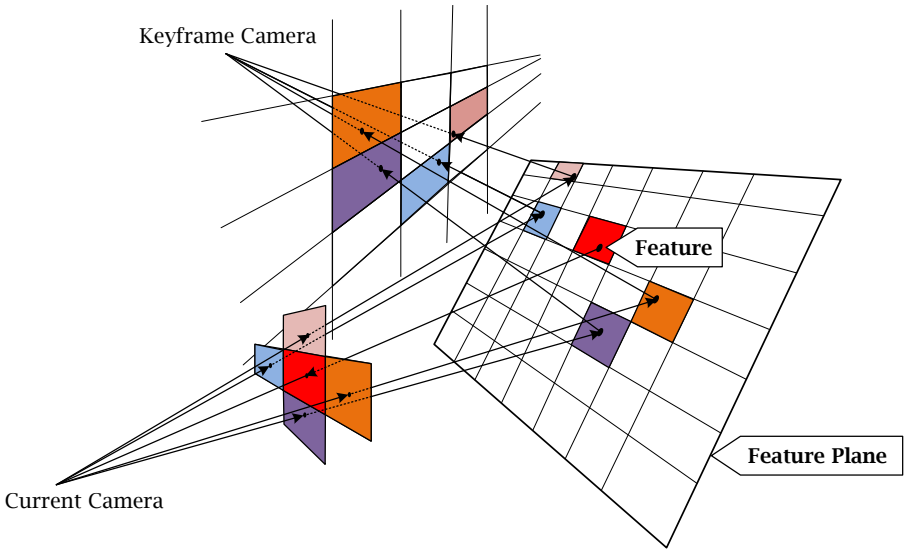


Figure 3.8: Reprojection of known pattern into keyframe around a feature-point

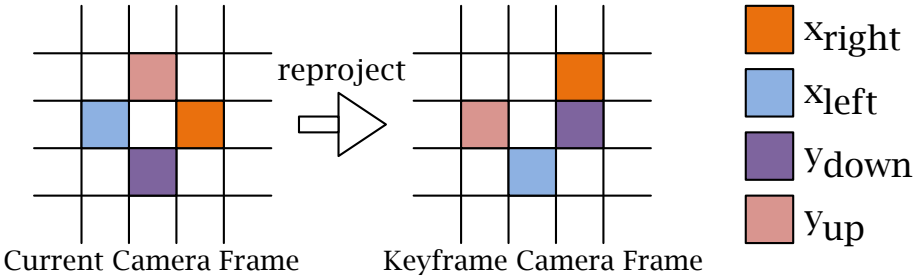


Figure 3.9: Affine transformation of the pattern

$$m = \begin{pmatrix} \frac{x_{right} - x_{left}}{2} & \frac{x_{down} - x_{up}}{2} \\ \frac{y_{right} - y_{left}}{2} & \frac{y_{down} - y_{up}}{2} \end{pmatrix} \tag{3.4}$$

Once the 2D image points have been recovered in the keyframe (see Figure 3.9), the central differences, denoted m, are calculated according to Equation 3.4. This is the basis for the later selection of the correct scale level as well as for creating an affine warp from the keyframe. The correct scale is determined with the help of m. The basic idea is to up- or downscale m until the resulting determinant lies between 0.5 and 2 with respect to the current image level.

Once the correct scale is selected, a warped patch is extracted from the keyframe with the aid of the transformation matrix m. The resulting patch is then used as a template for searching the neighborhood of the feature in the current image frame (see Figure 3.7).

NCC-based template matching The template patch is matched based on the NCC [Lew95] score. If a predefined threshold is exceeded, the patch is considered being found. Else, the current feature point is considered being out of reach and therefore discarded. Each newly found 2D position is then stored along with its 3D feature point representing the updated correspondences. With the help of these correspondences the current pose can then be refined.

3.7.2.2 Pose refinement

While the current camera view still looks in the direction of the previous frame, the position has to be refined according to the updated point correspondences. This is achieved by minimizing the re-projection error caused by projecting the 3D feature points into the camera frame.

Let the camera pose C be a rigid transformation from world coordinates to camera coordinates. The following observation function

$$p = (u, v) = proj(CX) \tag{3.5}$$

maps a point $X = (x, y, z, 1)$ into the camera view to the point $p = (u, v)$. The function $proj(.)$ represents the projection of a standard camera frame to image coordinates as shown in Equation 3.6. The parameters f_x, f_y, t_x and t_y are part of the intrinsic calibration matrix K.

$$proj\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \overbrace{\begin{pmatrix} f_x & 0 & t_x \\ 0 & f_y & t_y \end{pmatrix}}^{\mathrm{K}} \begin{pmatrix} \frac{x}{z} \\ \frac{y}{z} \\ 1 \end{pmatrix} \tag{3.6}$$

The re-projection error $e(C)$ is the sum of individual errors d_i^2 for each observation of the model, where p_i denotes the 2D point correspondence and X_i the 3D point respectively.

$$e(C) = \sum_{i=1}^{N} d_i^2 = \sum_{i=1}^{N} (p_i - proj(CX_i))^2 \tag{3.7}$$

The least-squared-error of the camera pose is supplied by minimizing $e(C)$ in terms of C. A small motion M is added to C resulting in

$$C^+ = MC. \tag{3.8}$$

The motion M is parameterized by the vector δ as exponential map parameterization. Instead of $e(C)$ the parameterized error $e(MC)$ is then minimized with respect to δ. This requires the over-determined system of linear equations (3.9) where $J_{pose} = J_{proj} J_{trans}$ to be solved.

$$J_{pose}\delta = d \tag{3.9}$$

The Jacobian of (3.6) gives

$$J_{proj} = \begin{pmatrix} \frac{1}{2} & 0 & \frac{-x}{z^2} \\ 0 & \frac{1}{2} & \frac{-y}{z^2} \end{pmatrix} \tag{3.10}$$

whereas the Jacobian of C results in

$$J_{trans} = G_i C X_j. \tag{3.11}$$

For each individual transformation CX_j a generator field G_i is added denoting the differentials around $(0, 0, 0)$ for each degree of freedom (DOF).

The minimizing six-vector parameter δ is determined after rearranging (3.9)

$$\delta = \left(J^T J\right)^{-1} J^T d \tag{3.12}$$

The minimizing parameter is then added to the current camera pose resulting in $\acute{C} = \delta + C$. This may be iteratively applied until the error falls below a certain threshold.

3.8 Discussion

This chapter presented a concept of a 2-stage natural feature detection and tracking (NFT) pipeline which can be realized with the help of HTML5 and JavaScript.

Proprietary web-based approaches offer efficient ways of implementing an AR pipeline, however, due to their inherent drawbacks, such as limited portability and security, native web-technologies constitute an interesting alternative.

Basically, a vision-based AR pipeline consists of three main parts. Firstly the image retrieval from the web-cam, secondly image processing for pose estimation and finally real-time augmentation of the video with 3D objects. For each of those steps a component of the HTML5 proposal [W3C11a] was chosen. As far as the access to the user's web-cam is concerned, the `getUserMedia` [WHA11b] interface was favored above

Mozilla's Rainbow [Nar11], because of an already existing standardization draft. For the image processing task, JavaScript was chosen, despite the fact that WebGL [Khr11f] and WebCL [Khr11e] benefit from hardware acceleration. However, WebGL does not offer floating-point precision [Khr09b] and WebCL is only available as a prototype [AN11b]. When it comes to 3D rendering WebGL was preferred to Canvas2D [WHA10], since the latter only employs a 3D software rendering pipeline.

In order to find out the strengths and weaknesses of the HTML5 solution, comparisons to Google Native Client (GNC) [Goo11e] and Adobe Alchemy [Ado09c] are considered. While the former provides an in-depth comparison to code running directly on the CPU, the latter is selected due to the comparison with another virtual machine based approach.

Besides the chosen web-technologies, algorithms and techniques for the detection and tracking pipeline were picked out. The designed detection pipeline starts off with the FAST [RD06b] corner detector, followed by the BRIEF [CLSF10] keypoint descriptor. Both approaches were chosen upon their low computational complexity compared to state-of-the-art algorithms such as SIFT [Low04] and SURF [BETG08]. The initial camera pose is considered to be extracted from the estimated homography between the target and the detected image [SB02] [XKM09].

As for the tracking pipeline, a patch-tracker approach was agreed upon, similar as presented in [KM07]. Since this method employs a 2-threaded approach which cannot be efficiently realized in JavaScript (see Section A.2), the mapping step was omitted in favor of a separate pose estimation as described in [WRM+10].

The next chapter gives an in-depth description of the implemented AR-pipeline following the concept presented here.

Chapter 4

Implementation

In this chapter, the realization and implementation of the previously described concept are focused on. Furthermore, some alternative approaches will be presented.

The first section focuses on the feasibility of implementing the AR detection and tracking pipeline with the help of HTML5 and JavaScript (see Section 1.1). Then, limitations and weaknesses of this approach are discussed. As for the evaluation of the implementation, the detection pipeline was ported to C/C++ for further use in Adobe Alchemy and Google Native Client. This evaluation approach is employed since both platforms accept C/C++ code for application development.

4.1 JavaScript

The implementation of the detection and tracking phase was first carried out in JavaScript in order to make a proof-of-concept. The whole developmental process was carried out within the Eclipse IDE using the JavaScript Developer edition which proved to be a reliable environment for JavaScript development providing good coding assistance. This section first explains how video data is aquired from different sources, including read-out and pre-processing. Followed by implementation details given for the detection and tracking phase of the pipeline, describing the lower-level facts. Finally, the rendering step using WebGL and a selected framework are depicted completing the pipeline.

4.1.1 Input: getting access to <video>data

The entry point of an AR-system is formed by the aquisition of a video stream which is then used for tracking the camera pose as well as for later augmentation. The video stream

may be provided either by a web-cam attached to a computer system or an integrated camera in a mobile's device. As already pointed out in Section 3.4 accessing the video stream's data is only possible through the use of the <video>element in combination with the <canvas>element.

4.1.1.1 Attaching a stream to a video element

Generally speaking, the <video>element acts as a sink for all incoming streaming data, regardless of the source (local file, http stream, user's webcam). This fact makes it easy to attach different kinds of sources without changing the behavior of the subsequent stages. The most favorable source is the webcam, on condition that the appropriate interface is offered, which however is the exception rather than the rule (see Section 2.3.7).

WebCam: `getUserMedia` One way of acquiring a live stream directly from the user is through the `getUserMedia` (see Section 2.3.5) interface. Since the document of specification [WHA11b] is still in draft, the code may work in one browser but not the other.

```
1 <script type="application/javascript">
2   navigator.getUserMedia("video user", function(stream){
3     video.src = stream;
4   });
5 </script>
```

Listing 4.1: Code snipped demonstrating the simplicity of aquiring the user's webcam

The code in Listing 4.1 demonstrates how to request a video stream (without audio) from the user's webcam and prefers, if there is more than one camera available the one facing the user. In the process of execution, the browser prompts the user to select an appropriate device and confirms that the access is allowed. Similar to Adobe Flash Player, an explicit confirmation from the user is needed to gain access in order to prevent malicious websites spying on the user.

During the implementation phase, two desktop web browsers have been made available providing the `getUserMedia` interface. First the modified Web-Kit library from the Ericsson Labs [Eri11a], and second an alpha version of Opera 12 [Øde11].

Ericsson Labs - WebRTC Tests with the MediaStream API were carried out using the experimental implementation of the WebRTC specification offered by the Ericsson Labs. They provide a modified WebKit library only running under Ubuntu 11.04 within the epiphany-browser which is built upon WebKit. Because the specification is still in draft

stage, all experimentally implemented operations are suffixed with the keyword `webkit`. Apart from the offered HTML5 features, the JavaScript engine used is borrowed from Safari, thus it does not perform as well as the competitors. Some minor problems are still present, causing the browser to exit due to unexpected crashes.

Opera 12 Alpha At a later stage, in October 2011, Opera made available their browser supporting direct camera access. In contrast to the WHATWG specification, the browser does not ask for permission while accessing the webcam, neither does it offer a selection of the desired device if more than one camera is present. The developers stated [Øde11] these features are planned for the final build. Interestingly, this release does not only work on Windows, but also on Linux and Android powered devices.

HTTP-stream If a browser does not yet support the access through the `getUserMedia` interface, a workaround can still be established using an http-stream attached to the <video>element. This works as follows: First, the live camera stream is transcoded in real-time to a format the webbrowser can work with (see Section 2.3.7) and then published over HTTP. This whole procedure can be realized with the help of VLC media player [Vid11] for example. There exist two major problems with this approach. First, it is only usable for experimental purposes because it is not standardizised and requires additional software which the end-user might not be able to access or operate. Second, due to the process of transcoding and buffering, the input delay is too high (from 2 to 5 seconds) to work with.

Static video file If there is no camera directly attached to the computer, a static video file can help out during testing but may neither reflect real-time performance nor experience.

4.1.1.2 Reading pixel data from the <video>element

The next step, after having a video-stream attached to the <video>element is to readout the pixel data. As already mentioned before, a direct access to the streaming data is not provided and therefore a workaround has to be established. A brief glance at the <canvas>element [WHA10] reveals that it offers two important operations. First, the draw method does not only allow static images to be drawn, but also the current frame of a video. Second, the canvas image buffer can also be read back in order to access already

drawn data. Hence, successive draw and read operations on the <canvas>element allow continuous access to the video stream.

```
1 var ctx = canvas.getContext("2d");
2 ctx.drawImage(video, 0, 0, canvas.width, canvas.height);
3 var pixels= ctx.getImageData(0, 0, canvas.width, canvas.height).data;
```

Listing 4.2: Read-Pixel

The code in Listing 4.2 illustrates the process of obtaining pixels from a video. First, the underlying canvas-context is retrieved (line 1) which is then used to draw the current video frame on to the canvas element (line 2). Finally, the resulting pixels are returned (line 3) in the form of a `CanvasPixelArray` representing a 1D array containing $4(RGBA) * width * height$ elements as illustrated in Figure 4.1.

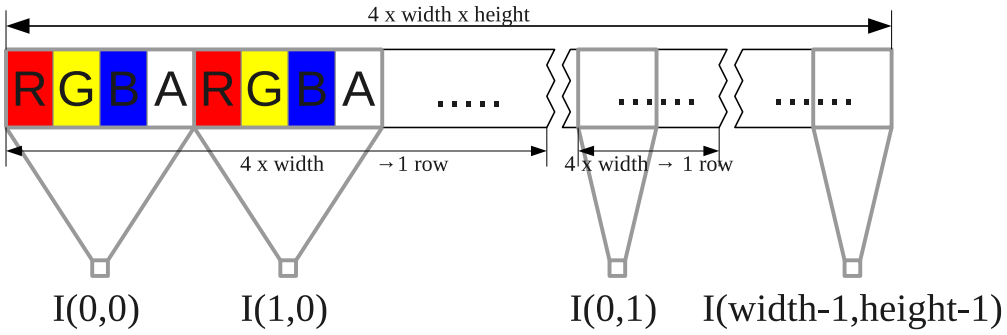


Figure 4.1: Pixel representation in CanvasPixelArray

4.1.1.3 Preparing frame-data for further processing

Since the CanvasPixelArray contains 32bit color information and most computer vision (CV) algorithms are designed to work with one color channel only (8bit) it has to be converted to a gray-scale representation before further processing can take place. The RGBA image is converted to gray-scale using the definition in (Equation 4.1).

$$I_{gray}(x,y) = 0.299 * I_{red}(x,y) + 0.587 * I_{green}(x,y) + 0.114 * I_{blue}(x,y) \quad (4.1)$$

The 8bit image is then stored using an one-dimensional array being either a native JavaScript array or a TypedArray in the form of an `Uint8Array` (for differences see Section A.5). Once the current video frame is available as an array of gray-scale pixels, the subsequent stages, detection or tracking, are being supplied with.

4.1.2 Detection phase

The first stage, as described in Section 3.7.1, of the AR-pipeline is designed to detect a known image in the current video frame in order to determine the camera's initial position. While the idea of the approach is already depicted in Section 3.7.1, this chapter concentrates on the details of how the solution was actually realized.

The detection pipeline starts with the extraction of interest points using the FAST corner detector followed by BRIEF keypoint description and matching. The outlier removal is carried out with a RANSAC approach whereas the final camera pose is estimated using the homography between the target- and video-image.

Initial target processing/preparation Due to the requirement (see Section 3.7.1) of being robust to scale, an image pyramid for the target image is initially constructed before the pipeline is executed (see Figure 4.2). The pyramid is built-up using sub-sampled images with step-size of $1/\sqrt{2}$ in height and width. This approach was given preference over half-sampling because of the sensitivity of the BRIEF descriptor to scale changes.

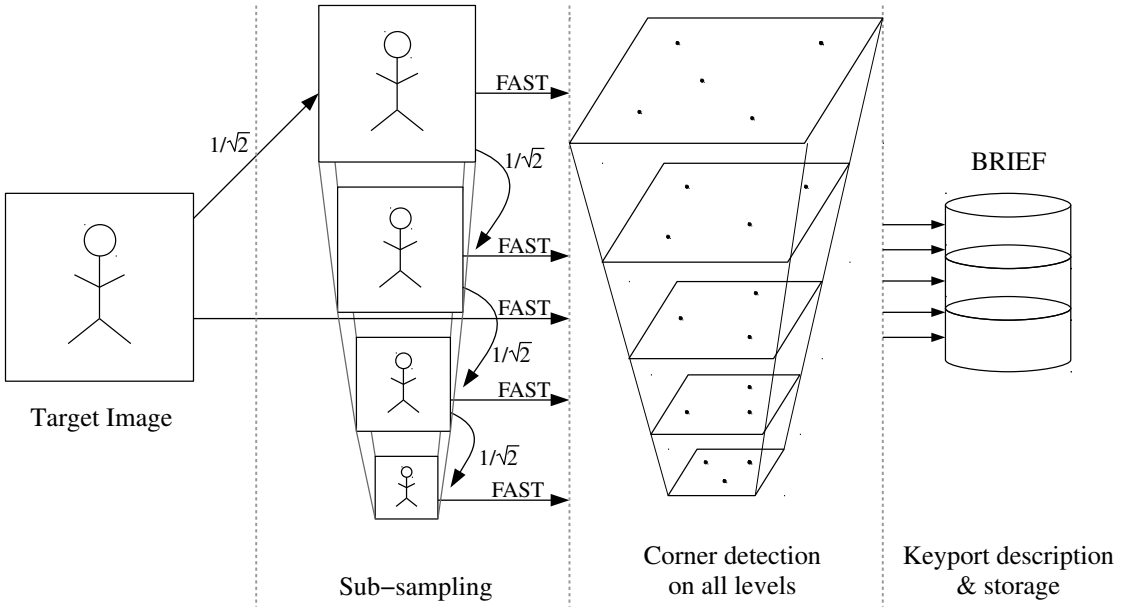


Figure 4.2: Initial target preparation

4.1.2.1 Interest-point detection

As stated in Section 3.7.1.1 the FAST [RD06b] corner detector is used for detecting points of interest. This approach makes use of machine learning techniques to efficiently imple-

ment the segment-test algorithm in a decision tree. Pre-made trees are available at the author's website [Ros11] for further usage. These trees are prepared for conversion into another programming language such as C, or Matlab. Because these conversion scripts are only provided for a small set of languages, a JavaScript target had to be created from scratch.

The selection of a particular version, ranging from FAST-9 to FAST-12, was determined by the results of the evaluation on speed. Initial tests (see Section A.1) revealed that JavaScript engines have difficulties processing arcs smaller than 10 px. Hence, employing FAST-10 was agreed on based on these results. In order to eliminate multiple responses occuring too close to each other, non-maxima supression is applied.

To maintain a roughly constant number of keypoints throughout the detection phase, a simple automatic-threshold adjustment is applied. The mechanism counts the number of detected keypoints and adjusts the FAST threshold according to a pre-defined lower and upper boundary.

4.1.2.2 Keypoint description

Once the keypoints are extracted, the description is performed by the BRIEF [CLSF10] approach mentioned in Section 3.7.1.2. The descriptor relies on simple binary brightness tests carried out around each detected interest point. Initial performance tests were carried out on the matching stage and yielded to the conclusion that the number of tests ought to be limited to 128. As stated in [CLSF10] 128 binary tests are still sufficient to be robust against small changes applied to the viewport.

To improve robustness against noise, the input images have to be smoothed before the binary tests can be carried out. For this purpose an integral image is calculated instead of applying computational expensive methods such as Gaussian blur or similar techniques based on convolution kernels. To further increase the speed of describing the keypoint, only the areas around the actual tests are blurred, not the entire image.

The results of the binary tests are then stored inside a binary string for later usage during the matching stage. Each test results in either 0 or 1 representing one bit in the descriptor's binary string. Since JavaScript does not offer constructs for storing binary strings, like C++ does with bitsets, an alternative solution is employed. Binary operations, as needed for later descriptor matching, are only available to the `Number` data type. Thus, a combination of `Number` values and an array yields to a practicable solution. In order to represent a 128bit binary string, a certain number of `Number` values are needed. In

JavaScript, a `Number` is defined as a double precision floating point number taking up 64bit of space [ECM11]. Because binary operations only operate on fixed size numbers, the JavaScript engine internally converts the double precision value to a 32bit integer. Due to varying optimization techniques throughout the different JavaScript engines, they may not work with full 32bit integers, but with 31 or even 30 bit. This problem is further discussed in Section A.3. Hence, to circumvent the problems caused by using full-sized integer values, the binary string is ultimately stored in five `Number` values encapsulated in an array as illustrated in Figure 4.3.

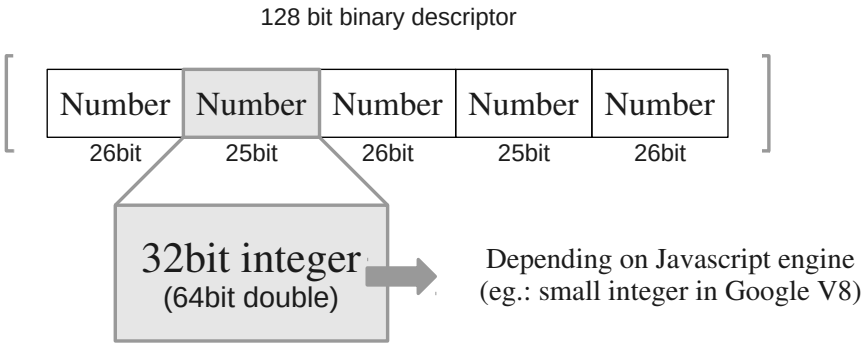


Figure 4.3: Representation of the BRIEF-Descriptor in JavaScript

4.1.2.3 Keypoint matching

The keypoint matching process compares the descriptors obtained from the target image and the current frame in order to establish correspondences between two similar looking features.

As pointed out in Section 3.7.1.3 BRIEF descriptors are matched according to the their Hamming-Distance. A brute-force matching approach is applied, since computing the Hamming-Distance is usually not an issue. As stated in [CLSF10] a special CPU instruction named POPCNT may speed up the process up to 15-fold using a SSE 4.2 enabled processor. Unfortunately, JavaScript does not offer any tools for optimizing code on CPU level, this is all up to the underlying engine to improve and optimize during execution.

Hamming distance counts the number of different bits of one string to another having the same length [Ham50]. In general, the distance can be computed as shown in (Equation 4.2) where both binary strings are first XORed followed by counting the high bits.

$$d_{Hamming} = \sum_{n=0}^{\#bits} a_n \otimes b_n \tag{4.2}$$

Because the Hamming-Distance can be computed independently for strings of the same length, the sum of these partial results is the same as for calculating the string as a whole. This fact makes it possible to easily implement the distance function using the 5-numbers-array construct. The resulting distance is a number between between 0 (the same) to 128 (all bits different) and describes the similarity of one interest point to another. The lowest score is considered as a correspondence and is remembered. In order to reduce falsely assigned correspondences, a threshold based on Figure 4.4 is set to allow a maximum of 32 bits to be different. After evaluation of all distances, each interest point has a corresponding point in the other picture assigned to.

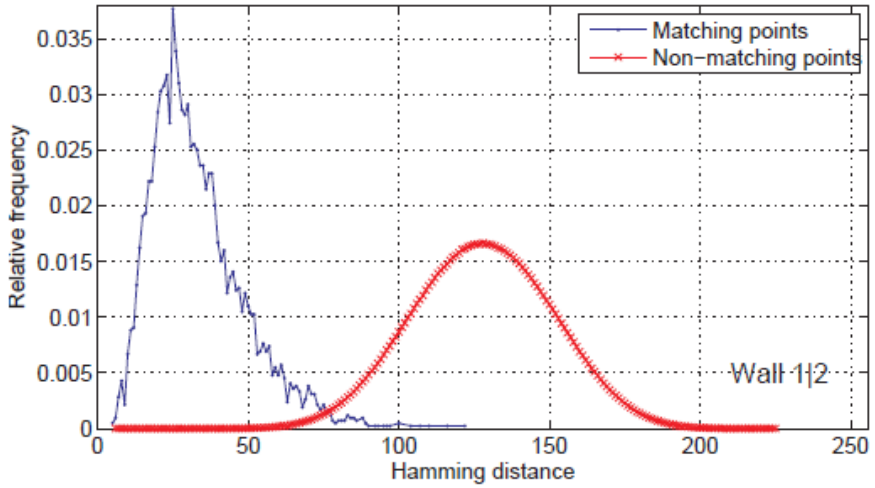


Figure 4.4: Distribution of hamming distances, taken from [CLSF10]

Bitcounting code The JavaScript code listed in Listing 4.3 was implemented during the first iteration. It takes a JavaScript `Number` object and determines how many bits are set. Initially conducted tests revealed that this function performs satisfactory when running in Firefox or Chrome. Using Opera instead the execution speed declined by a factor of 10. The reason for this significant slow-down was found with the help of a developer at Opera. Their JavaScript engine Carakan is not able to compile these statements into native code and falls back into interpretation mode.

An alternative implementation (see Listing 4.4), replacing the last line of code by some

additional binary operations instead of a mutliplication, fixed that problem. Not only the Opera browser worked 4 times faster than before (300 vs. 1200ms) but also Firefox (68 vs. 37ms) and Chrome (130 vs. 142ms) showed noticable speed improvements.

```
HammingDistance.numberOfSetBits = function (i){
  i = i - ((i >> 1) & 0x55555555);
  i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
  i = (i + (i >> 4) & 0xF0F0F0F;
  return (i * 0x1010101) >> 24;
};
```

Listing 4.3: Old Code

```
HammingDistance.numberOfSetBits = function (i){
  i = i - ((i >> 1) & 0x55555555);
  i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
  i = (i + (i >> 4)) & 0x0f0f0f0f;
  i = i + (i >> 8);
  i = i + (i >> 16);
  return (i & 0x3f);
};
```

Listing 4.4: New Code

Scale invariance In order to fulfil the requirement of being robust to scale, this matching procedure is carried out on all image levels of the target as illustrated in Figure 4.5. The image levels are chosen on the basis of the average distance of the matching score of each level. An image level having a lower average score is given preference over a higher score.

4.1.2.4 Outlier removal

The matching stage returns a list of correspondences between the target image and the frame investigated. Since the matching algorithm does not recover all corresponding keypoints correctly, wrong matches have to be removed. As described in Section 3.7.1.4 the RANSAC [FB81] approach is used to eliminate those false positives.

In prior to executing RANSAC, the keypoints acquired from the current frame are undistorted in order to provide better results. The distortion coefficients were determined during an offline-camera calibration carried out during project setup.

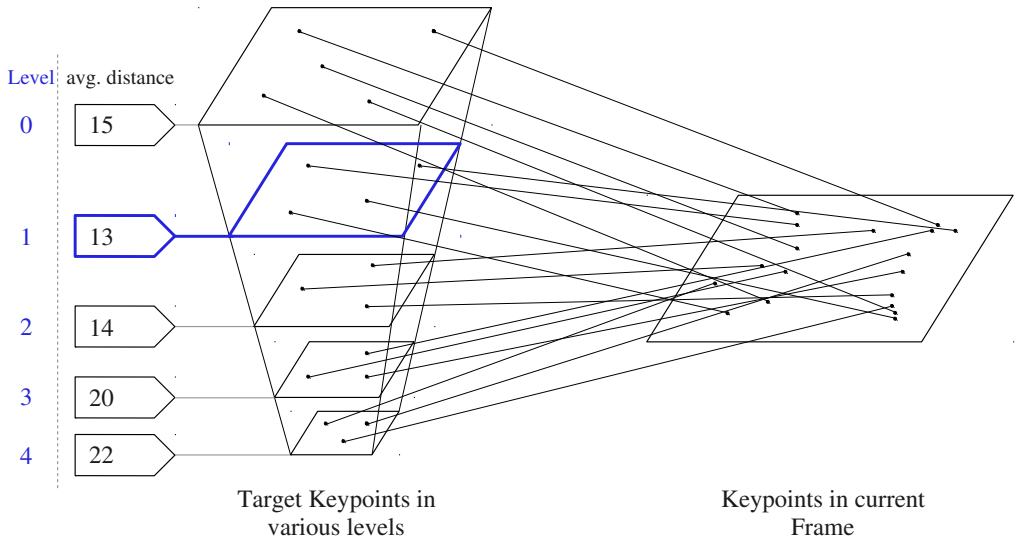


Figure 4.5: Scale selection: The level with the lowest average Hamming-distance is selected (level 1 in this case)

For performance reasons, the RANSAC implementation uses a different approach to calculate the homography between four correspondences, compared to the later pose estimation (see Section 4.1.2.5). The algorithm employed is presented in [Far05] demonstrating how to obtain the parameters by inversion of a 3x3 equation system.

The condition to indicate success is set to 80 or more inliers. Because RANSAC is based on random numbers, the outcome is neither predictable nor deterministic. Hence, an upper bound for the maximum number of iterations has to be set in advance. Empirical tests were carried out and resulted in a maximum of 300 iterations before the loop breaks. This stage results in a set of correspondences, cleared from outliers, ready for further usage in the process of initial pose estimating of the camera.

4.1.2.5 Pose estimation

As already described in Section 3.7.1.5 the initial pose is estimated by decomposing the homography H with the help of the calibration matrix K into the rotation R and translation t part.

The overdetermined system of equations $H^{-1}x = 1$ may be solved using the Singular value decomposition (SVD) or the Choleksy decomposition. Since no generic JavaScript implementation was found for any method, the latter approach was implemented owing to its greater efficiency over the SVD. For this purpose existing JavaScript matrix libraries

were evaluated for further usage in linear algebra related tasks. The decision was taken in favor of the *sylvester* matrix library [Cog11] which offers many matrix and vector related operations necessary for implementing the homography estimation.

Once the homography H is computed, represented by a 3x3 matrix, it is decomposed into R and t based on the idea depiced in Section 3.7.1.5. In contrast to the implementation of the homography computation where the sylvester matrix library is employed, recovering R and t makes use of another library, the *glMatrix* [Jon11] library. This library's primary purpose is to serve as a high-performance matrix library for implementing real-time graphic related tasks. Therefore, only fixed-sized matrices and vectors in the size of 2, 3 and 4 dimensions are offered. Since H, R and t fit in this schema, glMatrix is used during the pose recovering process.

4.1.3 Tracking phase

As soon as the detection phase reaches satisfactory results, the tracking phase takes over as long as the tracker's quality is maintained. If the quality falls below a certain threshold, the pipeline falls back into the detection phase and tries to reinitialize the camera's position. More details are to be found in the concept in Section 3.7.2.

The implementation of the tracking phase consists of two major components. First, the patch-tracker, implementing the idea of locating the feature's positions in the current frame. And second, the pose-refinement step, updating the camera's position according to the latest set of $2D \leftrightarrow 3D$ point correspondences.

4.1.3.1 Patch-tracker

The patch-tracker implementation follows the concept described in Section 3.7.2.1 and explains the measures taken to run the code as efficiently as possible. The main focus is put on optimization techniques regarding high-speed array access, such as dense and typed arrays. In addition, the adaptations applied to the `glMatrix` library, required to work properly in linear algebra environments, are depicted.

Dense vs. sparse arrays: During the processing of each single frame most of the execution time is spent on accessing pixels in images and their patches. The underlying image-data is organized in flat JavaScript arrays. In JavaScript an array can either be dense or sparse depending on the position of the elements and the respective length. E.g. when assigning a value to the 100th element of a previously empty array the length is

incremented to one creating a sparse array. This is because of the internal representation of arrays in the respective JavaScript engines. Usually an `Array` object is nothing but an object having properties attached like any other JavaScript object. While objects have properties with names, arrays are indexed with numbers instead. Typically, these properties are stored within a hash-table and looked up when accessed. This fact makes property-access usually very expensive. Optimization mechanisms inherent in JavaScript engines [Moz11g] try to resolve that issue by mapping dense `Array` objects internally as arrays (connected low-level memory blocks) [Moz11c]. The JavaScript developer cannot decide or analyse himself if the optimizations are actually applied or not. In short, sparse arrays are avoided throughout the patch-tracker implementation by either initializing arrays properly, or carefully appending of elements.

Type restriction in arrays: Another strategy when working on high-performance arrays is the restriction of the same data-type for each element. Since JavaScript does not distinguish between different types, implicit type conversations may be performed and valuable time wasted. This effect can be observed by the unsystematically usage of fixed-point numbers only (`Math.ceil`) and the combination fixed- and floating-point values. Using the extracted values as an index for accessing other arrays requires the value to be converted into fixed-point when stored as floating-point.

Breaking the rule of using same types might cause the JIT compiler in the respective JavaScript engine to deoptimize, or even fall back to interpreting the code. Therefore, the scheme of having the same data-types in arrays is applied throughout the project in order to benefit from the engine-specific optimization techniques.

Typed vs. untyped arrays Along with the introduction of WebGL the usage of strictly-typed arrays [Khr11f] has been made available. In contrast to native JavaScript arrays where the data-type of the stored values is not restricted, *TypedArrays* only allow values of a predefined data-type (signed/unsigned, 8,16,32bit fixed point or 32,64bit floating-point precision) to be assigned. This strict behavior is well known in programming languages such as C++ or Java and helps the underlying JavaScript engine to further enhance the performance by making use of the provided type information. Because of the young age of TypedArrays, optimizations have not yet taken place as observed in Section A.5. This means that using TypedArrays may yield to the same speed (or even slower) as already known from traditional JavaScript arrays.

Due to the current performance impact (see Section A.5) caused by TypedArrays, the

implementation of the patch-tracker uses abstract array types to change the concrete types on demand.

Reusing arrays: Another optimization strategy applied is the reuse of most of the arrays for the subsequent frames in order to avoid the creation of new objects. This avoidance is important because to the behavior of the garbage collector which pauses (see Section 2.2.4.2) the execution of the program and cleans up the memory. This causes the program to slow down from time to time limiting the real-time experience.

glMatrix adaptations: As already pointed out in Section 4.1.2.5 the *glMatrix* library is used for high-performance vector operations. But since this library is primarily created for the use within WebGL related tasks, the multiplication of a vector with a matrix is transposed. For the purpose of employing this library in linear-algebra related operations alternative implementations were added.

4.1.3.2 Pose refinement

After tracking the keypoints on a certain level, the camera's pose is refined in respect to the new positions. The implemented pose optimization algorithm is based on the description given in Section 3.7.2.2. Since the minimization of the re-projection error is based upon an iterative algorithm, the number of maximal iterations has to be defined. Experimental results showed that five iterations are enough for maintaining a stable pose.

In order to ensure that the pose optimization step does not create a bottleneck during the tracking phase, certain optimizations were applied. First, the whole refinement step is implemented using fixed-size arrays in order to operate as fast as possible. For this purpose, a variant of the Cholesky decomposition to efficiently compute $(J^T J)^{-1}$ was created to work with 6x6 matrices only. And second, the remaining parts rely on the glMatrix library and custom-built extensions to it to further improve the performance.

4.1.4 Rendering

The final stage in the AR pipeline, after determining the camera's current position, is the augmentation of the video with generated 3D content. The rendering of 3D objects on top of the video is realized with WebGL. More on WebGL can be found in Section 2.3.4

WebGL relies entirely on shader programs and does not offer helpers like primitive types or lighting models, known from OpenGL. This means that creating 3D models and

scenes completely from scratch without a proper framework is hard work and mostly results in error prone code. Hence, the usage of an appropriate framework was agreed upon. Despite the young age of WebGL, various scenegraphs and frameworks based on WebGL are currently in development or have just been released [Khr11d].

A framework used in an AR system has to fulfill two requirements. First, the ability to customize the view-frustum in order to reproduce the real-world camera with its intrinsic parameters. And second, the capability of updating the model-view matrix according to the current camera pose. A promising candidate meeting both of the requirements is SceneJS [Kay11], offering a very customizable scenegraph based on JSON [Cro11].

4.1.4.1 SceneJS

SceneJS [Kay11] is a very unique WebGL framework using JavaScript Object Notation (JSON) for creating scenegraphs. Whereas the popular COLLADA [Khr11a] format uses XML for scene description, in SceneJS all objects, nodes, transformations and their properties are written in JSON. This makes the rendering and scene-update process very efficient. Converters exist to transform COLLADA or OBJ models to JSON and use it within SceneJS. Since the framework has not reached its full maturity, some minor bugs and rendering issues are still present. An impressive showcase demonstrating the power of SceneJS is the biodigital human [Bio11], an interactive 3D model of a human being allowing the user to scan through different layers as well as to show animations of the internal body parts.

Implementation SceneJS was used to create a scenegraph for demonstration purpose. A transformation node is set up at the top of the graph acting as model-view matrix which is subject to be adjusted according to the currently tracked camera pose. The 3D model to be rendered on top of the video was taken from the developers' website, which was converted from COLLADA to JSON. The result of the tracked target combined with a 3D plane model is shown in Figure 4.6.

Camera frustum The intrinsic camera matrix K, which was determined in an off-line calibration process, cannot be used directly in the framework. SceneJS instead allows a custom camera frustum to be used. For this purpose the matrix K has to be transformed into a frustum applying the specification in Equation 4.3. The near and far clipping plane were set to 0.0001 and 2 respectively.

$$K = \begin{pmatrix} f_x & 0 & t_x \\ 0 & f_y & t_y \end{pmatrix} \qquad \begin{aligned} frustum_{left} &= -near * t_x / f_x \\ frustum_{top} &= near * t_y / f_y \\ frustum_{right} &= near * (size_x - t_x) / f_x \\ frustum_{bottom} &= -near * (size_y - t_y) / f_y \end{aligned} \tag{4.3}$$

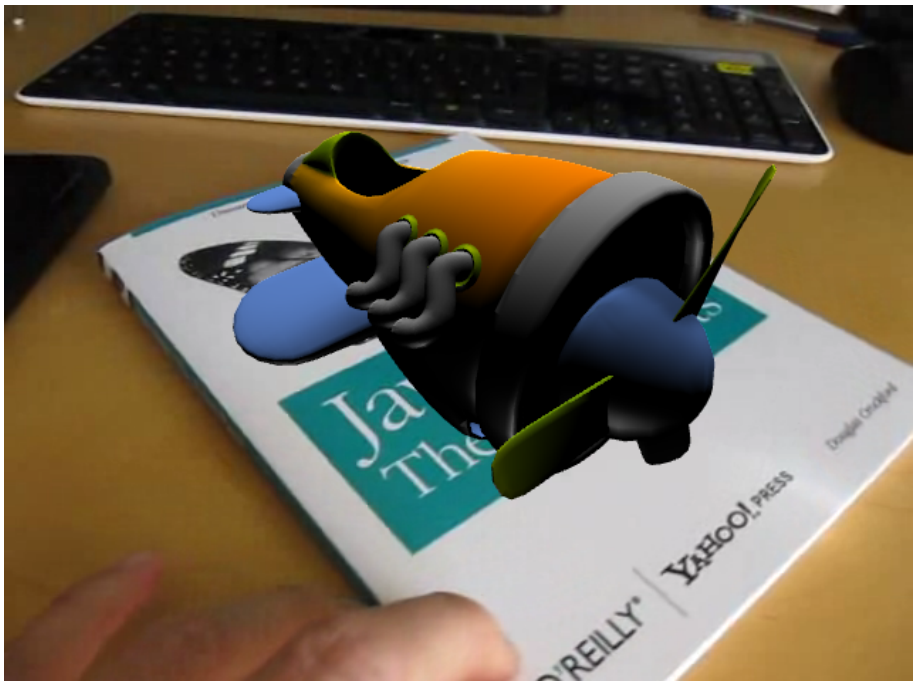


Figure 4.6: Result showing seymor plane ontop of a JavaScript Book

4.2 C/C++ core module

As an alternative to the AR-pipeline implemented in JavaScript an approach using C/C++ as programming language was taken. The code does only cover the detection pipeline, because it is more expensive in terms of computational complexity. The reason for using this alternative implementation is to recreate an identical system in C/C++ which can be reused in proprietary web technologies such as Adobe Alchemy and Google Native Client (GNC). Both systems were then tested against the JavaScript implementation in order to evaluate the speed differences between JavaScript engines and their proprietary competitors.

The C/C++ code was implemented with the assistance of the Visual Studio IDE and compiled with the GCC/G++ toolchain using the Cygwin runtime-environment to ensure

compatibility with GNC and Alchemy. For that purpose, the dependecies on 3rd-party libraries were kept as small as possible. The core module is only referencing two external sources, first the FAST [RD06b] code for corner detection and second the TooN Matrix library [Dru11] for pose estimation.

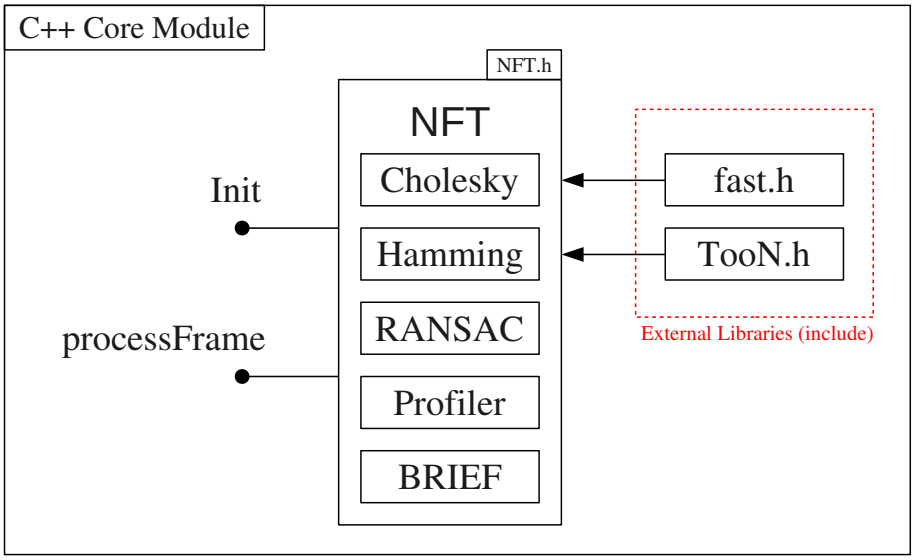


Figure 4.7: C/C++ core module definition (header + include)

As depicted in Figure 4.7 the C/C++ code provides the core components of the detection pipeline, including keypoint detection, description, matching, outlier removal and pose estimation. The image acquisition and rendering tasks are not part of this module owing to the differences present in Alchemy and GNC. Therefore, the input is defined as an array of `unsigned char*` representing a gray-scale image and its output a 4x4 matrix representing the current camera pose. All pre- and post-processing related tasks were implemented in their superior frameworks respectively.

4.2.1 Profiling

Since the C/C++ core module behaves like a closed system, there is no possibility to tap into the pipeline for performance measurements. Hence, a profiler was implemented directly into the module in order to measure the time needed for each stage of the pipeline. The time is measured by using `clock_gettime` [IG08] in Alchemy or `clock()` [Ope97] in GNC. If necessary, the enclosing framework could then request a string presenting the times spent in each of the stages for every processed frame.

4.3 Adobe Alchemy

The implementation was created within FlashDevelop [PE11], an OpenSource IDE targeted at Adobe Flash products. The implementation covers only the detection pipeline whereas the camera access and rendering parts are missing. This was agreed on, as the evaluation only pays attention to the core performance in detecting the target image.

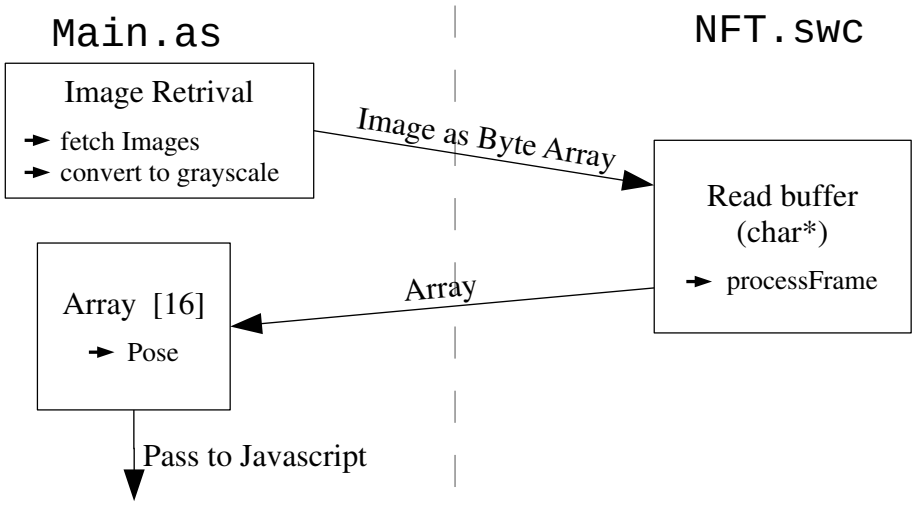


Figure 4.8: Data flow between Flash and C/C++ core module

4.3.1 Structure

Basically, a project based on Adobe Alchemy can be devided into two separate parts. First, a set of C/C++ files which are subject to be compiled to ActionScript code, and second, the resulting library which can then be imported into another ActionScript project, such as Adobe Flash, for further usage. Compiling C/C++ code with Alchemy results in an ActionScript binary library which is not different from any other standard `.swc` file.

For testing purposes, a simple Flash project was created importing the compiled C/C++ core module. In addition, an ActionScript file was implemented for controlling the library and image retrieval. Besides that, helper files required for running the resulting Flash-module were also included.

The project consists of two main parts as illustrated in Figure 4.8. First, the detection pipeline implemented in C/C++ and second an ActionScript file controlling the former. The latter is responsible for fetching images and supplying the detection module with gray-scale variants while the former receives those and returns the camera's pose.

4.3.2 Compilation

Compiling the implemented C/C++ core module (see Section 4.2) is a straightforward process. Before the actual compilation task can be carried out, an additional header file (`AS3.h`) must be included. This file provides constructs to create an interface for the module which is then exposed to the enclosing Adobe Flash project. Once the interfaces are defined, the code is ready to be compiled to `swc`.

4.3.2.1 std::string Bug

Since Adobe Alchemy is still a research project, there exist several drawbacks and bugs in the implementation. Whereas most of the bugs are not very harmful, there is one which causes serious problems. The framework's implementation of `std::string` is erroneous and causes troubles when the copy constructor is used. More on that bug is to be found in [Ado09a] and [Ado09b]. The C/C++ core module implementing the detection pipeline does not rely on `std::string`, but the profiler, which outputs the timings of each run. A solution to this problem was not to use `std::string` at all but `char*` constructs instead.

4.3.3 Web-cam access - image retrieval

Adobe Flash natively supports web-cams allowing Flash programs accessing the user's camera stream. Similar to HTML5, a security prompt asks the user to confirm the usage of the camera. As for evaluation purposes, no camera access is needed, instead a set of static images is used. Before running the detection pipeline, all images are pre-loaded and converted into their gray-scale representation.

4.3.4 Detection phase

Each pre-loaded image is passed over to the detection library by calling the exposed interface function of the module. The module then reads the passed buffer into an `unsigned char*` buffer and processes the frame. As soon as the current pose is found, it returns the pose matrix in form of an array.

4.3.5 Rendering

Rendering 3D content may be achieved using mechanisms described in Section 2.4.1.2. In the course of evaluation, only the detection pipeline was from interest, hence no rendering was implemented in Flash.

4.4 Google Native Client

The AR-pipeline was realized in NaCl with the help of the Google Native Client SDK [Goo11e]. The implementation was created for evaluation purposes only, using the C/C++ core module from Section 4.2 with adaptations in image retrieval and output.

4.4.1 Structure

A typical web-application making use of the Native Client technology consists of three main parts as depicted in Figure 4.9. First, a set of files containing HTML, JavaScript and CSS code acting as presentation layer. Second, the Native Client Module, an executable providing the core functionality in native code. And third, a glue linking together the first and second part, the so-called Pepper library.

Running native compiled code direcly from the web without further inspection raises many security concerns. For this purpose, Google's Native Client technology incorporates mechanisms for checking and validating binary modules as presented in Figure 4.9. Before the first run, the module is investigated and must pass validation in order to start. In case there is a security validation detected during runtime the module is immediately terminated before causing any harm.

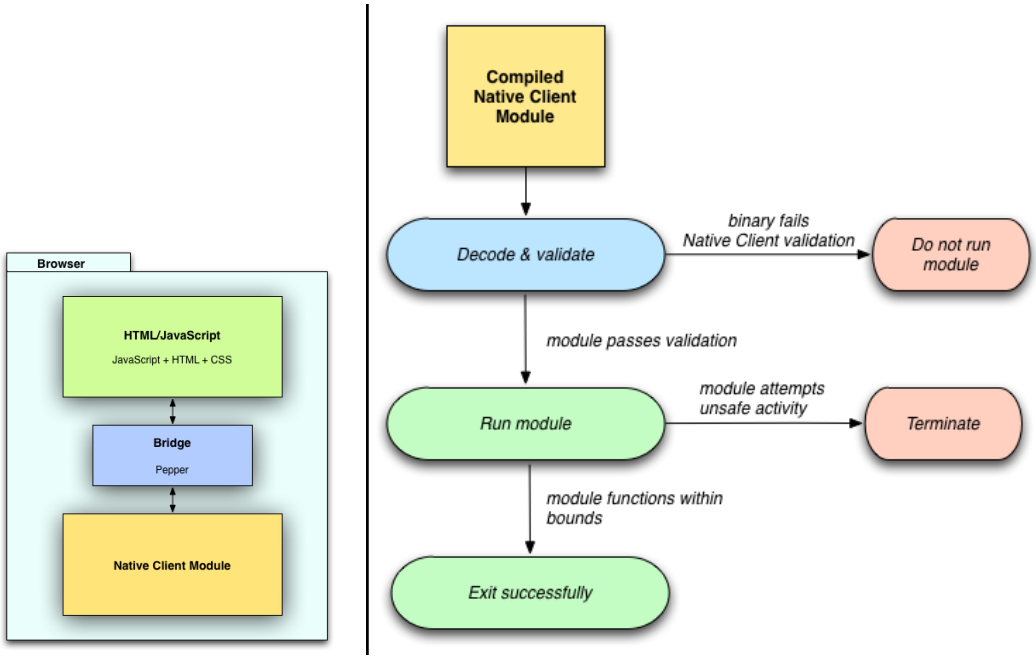
4.4.2 Compilation/Linkage/Interface

The implementation of a NaCl module is similar to Adobe Alchemy's approach and follows two main steps to perform. First, deriving from the provided `Instance` class and implementing the interface method `HandleMessage`. This method handles all incoming messages sent from the browser via `postMessage()`. And second, compiling the project using the provided toolchain to create the final executables.

4.4.3 Messaging system

The communication between JavaScript and the Native Client module is maintained by a messaging system. From within JavaScript, a message can be posted to the NaCl module and vice-verca. This all happens asynchronously, which means that posting messages does not block the further execution on both sides.

These messages are simple strings which may contain JSON-data or other commands using unicode. Its primary purpose is to exchange short messages containing commands or logs rather than sending big chunks of data. This means, ways to exchange binary data,



(a) Structure in Browser (b) Code-security validation

Figure 4.9: Structure of NaCl taken from [Goo11e]

such as images or video data, are not provided. In order to circumvent this limitation, encoding mechanism may be employed, which is described in more detail in Section 4.4.5).

4.4.4 Web-cam access - image retrieval

There is no native camera access available through the native client yet but plans exist to release such an interface in the near future [Wil11].

On the one hand, the native client API offers an interface to write data to an HTML5 `canvas` element, which may be used for advanced visualization tasks. On the other hand, mechanisms for reading the image data are not made available, which means that there is currently no efficient method to get image or video data to the native client module. Therefore, at present the only way of supplying the native client module with moving pictures is to send them frame by frame through the messaging channel. Each frame is aquired the same way as already implemented in the JavaScript version (see Section 4.1.1) where the video is captured by continuously drawing the video onto a canvas element followed by reading the pixels into an array.

4.4.5 Communication

Since the only way of communicating between the native client and the browser is sending messages back and forth, the image data has to be sent through that channel as well. The message exchange is implemented through Simple Remote Procedure Calls (SRPCs) and limited to 128KB per call, including messaging overhead resulting in about 100KB payload. This channel is not supposed to be used for binary data streaming and thus limited to message exchange by strings. To successfully send the current image data to the NaCl module, it has to be encapsulated into a String representation. Hence, Base64 [IET06] encoding was successfully chosen to overcome the limitation and work with binary data. An 8-bit image with a resolution of 320x240 pixels has a size of 76800 Bytes. When encoded to Base64, each triple of Bytes is encoded in 4 Bytes resulting in a total size of 102400 Bytes. Apparently, this is too much for being transferred as a chunk. Therefore, the payload was split into two separate messages. The receiving NaCl module is then responsible for decoding and merging those packets for further use in the detection pipeline.

4.4.5.1 Statemachine

Because the messaging system works asynchronously, there is no blocking during the NaCl module execution. This is a drawback for this particular system since the pipeline works in a sequential manner. A walkaround using a statemachine was therefore applied. The basic schema of the implementation is illustrated in Figure 4.11

In order to initialize the detection pipeline, the target image is sent from the web browser to the NaCl module. For this purpose the NaCl module waits for the first message to arrive, which is the first half of the target image. Afterwards, an acknowledgement is sent back indicating it is ready for the second part. The JavaScript implementation then sends the second part that is also acknowledged by the NaCl module which finishes the initialization.

As soon as the setup process ends, each frame is sequentially sent to the NaCl module, again in two chunks. After assembling the image, the detection pipeline is activated and computes the current pose which is then immediately sent back to the JavaScript message handler. After receiving the camera position, the next frame is sent. This loop continues as long as there are images in the queue.

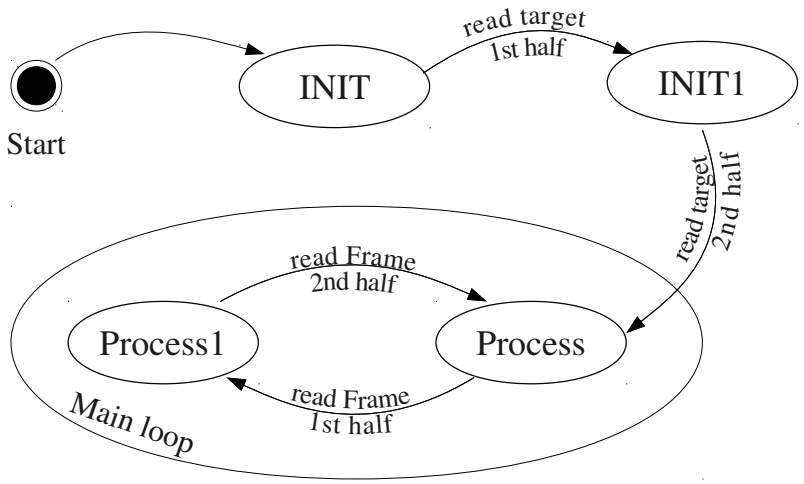
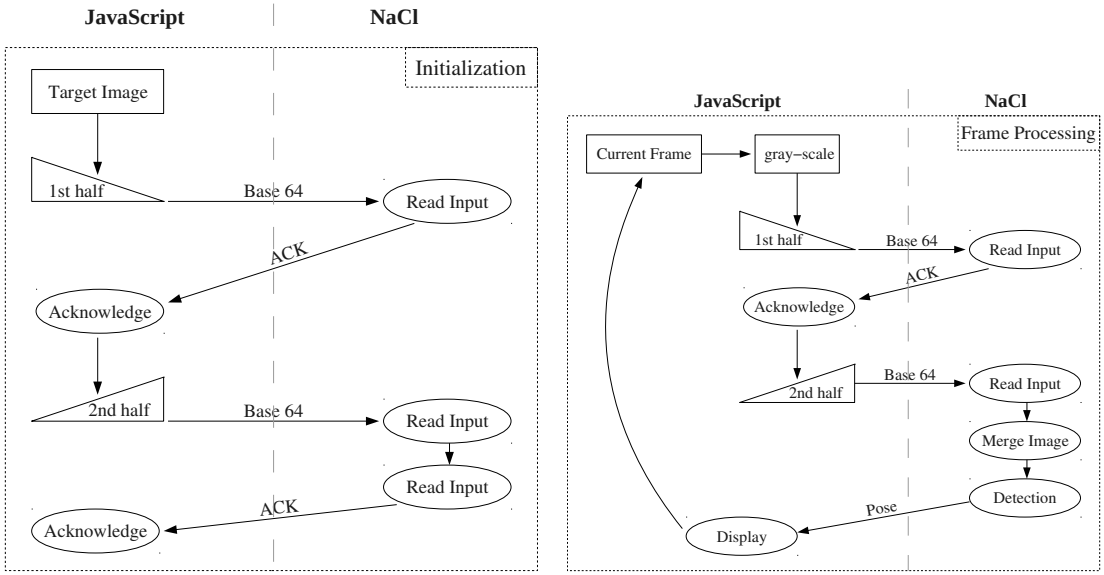


Figure 4.10: Statemachine inside the NaCl module representing the initialization and subsequent detection processes



(a) Data flow when initializing with target image

(b) Data flow when subsequently detecting the current frame

Figure 4.11: Data flow between JavaScript and the NaCl module

4.4.6 Detection pipeline

The detection pipeline is provided by the same C/C++ core module as used in Adobe Alchemy. The detection starts as soon as the two image chunks are received by the NaCl

module. The resulting pose is then sent back to the JavaScript code via the messaging system.

4.4.7 Rendering

For the same reason as in Adobe Alchemy, no rendering stage was implemented in NaCl. If desireable, two different approaches may be realized. First, the direct use of OpenGL within the NaCl module, or second, employing the browser's capability to render WebGL.

4.5 Discussion

This chapter discussed in-depth ways to realize AR solutions in JavaScript, Adobe Alchemy and Google Native Client (GNC). The implementation in JavaScript and HTML5 were the focal points.

Owing to the fact that the incoming video-stream acquired from `getUserMedia` [WHA11b] cannot be intercepted on pixel level, a workaround was established employing the `canvas` [WHA10] element as an image buffer. This allows pixel-per-pixel access required for the subsequent stages of the pipeline.

The initial step of the detection phase implements the FAST [RD06b] corner detector whose decision tree was generated for JavaScript by a customized script. The succeeding step, the implementation of BRIEF [CLSF10], revealed some flaws inherent in JavaScript when working with binary data. This is due to the fact that JavaScript only offers the `Number` data type to be used for binary operations. Since the `Number` type is a 64bit floating-point number [ECM11] and binary operations are only carried out in a 32bit fixed-point manner, an efficient internal representation in the JavaScript engines is required. Due to certain optimization techniques the respective engines only use 30 to 31bit integers (small integers) [win11a] internally and execute the code much more slowly, when exhausting all 32bits (see Section A.3). Hence, the 128bit BRIEF descriptor was created as an `Array` consisting of 5, not 4, `Number` values. The following RANSAC [FB81] and homography estimation steps make both use of external libraries *glMatrix* [Jon11] and *Sylvester* [Cog11] which provide common matrix and vector operations.

While the detection pipeline primarily operates on binary data (BRIEF description & matching), the tracking pipeline implements an efficient patch-tracker relying on high-performance arrays. The iterative pose refinement step benefits from fixed-size matrix and vector operations which were created as an addon to the *glMatrix* library.

The final rendering step is controlled by SceneJS [Kay11], a JSON based scenegraph featuring a customizable view frustum and model-view matrix required for mapping the real camera's intrinsic and extrinsic parameters.

For evaluation purposes, an implementation of the detection pipeline in C/C++ was carried out. The C/C++ module makes use of two external libraries, the FAST corner detector and the TooN [Dru11] matrix library. This module was then compiled to ActionScript with the help of Adobe Alchemy for further usage in an Adobe Flash AR-pipeline. The same C/C++ module was also incorporated in a Native Client module for the use in Google Native Client (GNC). Mechanisms to exchange binary data between the browser and the NFT module are offered in Adobe Alchemy. In contrast, GNC only allows simple string messages (max 100KByte) to be shared. The capability to pass binary image-data could be achieved by using Base64 [IET06] encoded data-chunks.

The next chapter deals with the comprehensive evaluation of the implemented JavaScript pipeline on the PC and mobile phone also in comparison with alternative approaches using GNC and Adobe Alchemy.

Chapter 5

Evaluation

The test procedure focuses on two different categories, *speed* and *robustness*. The speed evaluation starts with the JavaScript implementation carrying out separate tests for the detection and tracking pipeline. Each phase was tested in a cross-browser domain as well as cross-platform comparing a PC's performance with a mobile phone. Each of the tests carried out is presented in an overview chart along with a more detailed diagram showing the results of performance profiling. In addition to the speed evaluation, the robustness of the tracking and detection pipeline was tested, experimenting under which circumstances the pipeline breaks.

Besides evaluating the JavaScript performance, separate experiments were conducted to compare the alternative implementations in C/C++ (see Section 4.2) to it. This was done in the same manner as for JavaScript described above focusing, however, only on the detection phase of the pipeline. Due to the lack of a Google Native Client runtime engine on mobile devices, these tests were only carried out on the PC.

5.1 Test platform

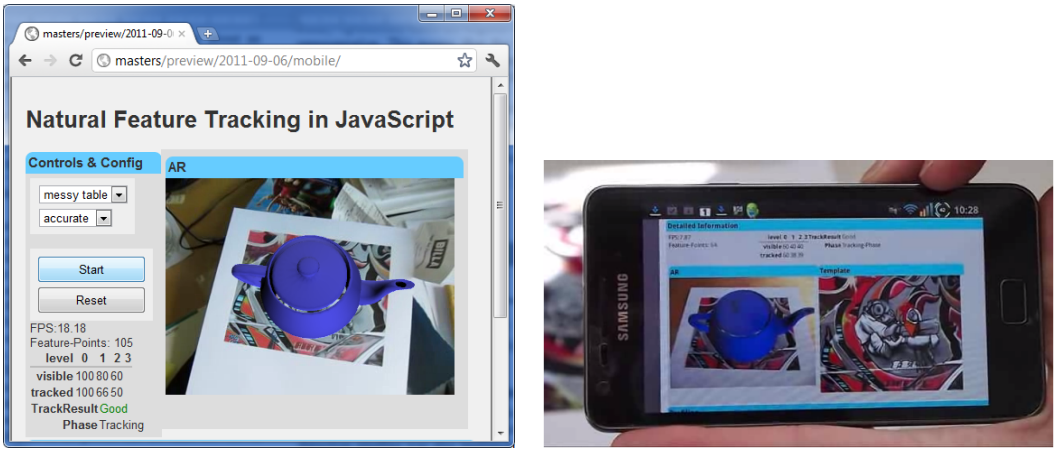
The JavaScript related tests were all carried out on two different platforms, a PC and a mobile phone (see Table 5.1). On one side, a desktop computer was used to represent the performance of x86 platforms. On the mobile side, a Samsung smart-phone of the type Galaxy S II was used for running the experiments. Figure 5.1 shows both devices running the JavaScript implementation.

As for the cross-browser evaluation, the latest stable versions available were installed alongside with some experimental or alpha builds in order to demonstrate the capabilities

of future browser implementations. These releases may not represent the performance of their later final installment.

	Desktop PC	Samsung Galaxy S II
CPU	Core 2 Duo 3GHz	Exynos 1.2 dual core
RAM	4GB	1GB
OS	Windows 7	Android 2.3.5
Browser	Mozilla Firefox 8 Mozilla Firefox 11.0a1 (nightly) Google Chrome 15 Google Chrome Canary 17 (dev) Opera 11.60 Opera Next (12.00 alpha)	Mozilla Firefox 8 Android 2.3.5 WebKit

Table 5.1: Test-systems



(a) Google Chrome

(b) Firefox on the Samsung Galaxy S II smartphone

Figure 5.1: Both test platforms running the JavaScript implementation

5.2 JavaScript implementation

Both stages, the detection as well as the tracking, were evaluated comprehensively regarding speed and robustness. Because profiling tools were not available as developer extensions in all browsers, a programmatic approach was taken to measure the time for

each major component in the implementation. Although a HTML5 proposal exists for performance measurements named *Navigation Timing* [WW11] it is designed to time page load events and other navigation related tasks. No additional functionality is provided to support the developer during JavaScript performance evaluation. Therefore, a profiler was implemented for timing the individual stages with the help of the `Date` structure present in JavaScript and its ability to measure in millisecond (`ms`) accuracy. Because of this limited accuracy, parts which are finished within a fraction of a `ms` cannot be measured. Since the interest lies in testing whether real-time performance ($\approx$ 50 ms per frame) can be achieved, timings lower than the accuracy are not considered important.

5.2.1 Detection pipeline

The evaluation of the detection pipeline is split into three parts. First, a cross-browser speed comparison on the PC, followed by performance measurements on the mobile phone and experiments concerning robustness.

In pursuance of detecting bottlenecks and problem areas in certain modules, the measurement of the detection pipeline is divided into six subsequent stages. These stages are named *integral*, *fast*, *descriptor*, *match*, *ransac* and *pose*. The *integral* part holds the time spent for constructing the integral image out of the incoming video frame. During the *corner detection* stage, the time detecting the FAST corners as well as the following non-max suppression are measured. For the *descriptor*, the performance of creating the BRIEF descriptor for all keypoints is timed. The time consumed in the *matching* stage covers brute-force matching of all keypoints including all template image-levels. The *outlier removal* stage incorporates the time spent running RANSAC. Finally, during *pose estimation*, the homography is computed followed by decomposing and recovering the camera's extrinsic parameters.

The evaluation does neither include the input (camera access & frame reading) nor the output (3D rendering) part. This is because the pipeline is later compared to the alternative implementations where those parts are not present. A seperate investigation concerning these parts is conducted in the appendix (see Section A.4).

5.2.1.1 Settings

The detailed configuration of the detection pipeline is shown in Table 5.2. These settings are used throughout all tests, including the mobile browser as well as the alternative implementations.

Setting	Value
Image size (pixel):	320x240
FAST corner detector:	250 corners per frame
BRIEF: descriptor length:	128bit
BRIEF: image-levels (template):	5
RANSAC: min inlier:	30
RANSAC: max iterations:	300
RANSAC: error measure:	max 16px squared

Table 5.2: Configuration of the detection phase for evaluation

5.2.1.2 Test-set & Test-method

The test-set consisted of 60 images extracted from a video stream in a time-interval of 1/3 of a second (3 fps). This approach was used in order to make the results comparable, whereas using a video might yield to a completely different outcome. The only component which relies on random values is the RANSAC stage, for which a deterministic alternative to `Math.random` was used during the experiments based on Robert Jenkins' hash-function [Wan07]. The combination of static images and a deterministic random value generator allows the creation of repeatable results for later comparison.

A single test-run covered the detection of the target image in the image-stream and the computation of the initial camera pose. The values were averaged from five test-runs.

5.2.1.3 PC cross-browser

To point out the strength and weaknesses of different JavaScript-engine implementations, a cross-browser test was conducted.

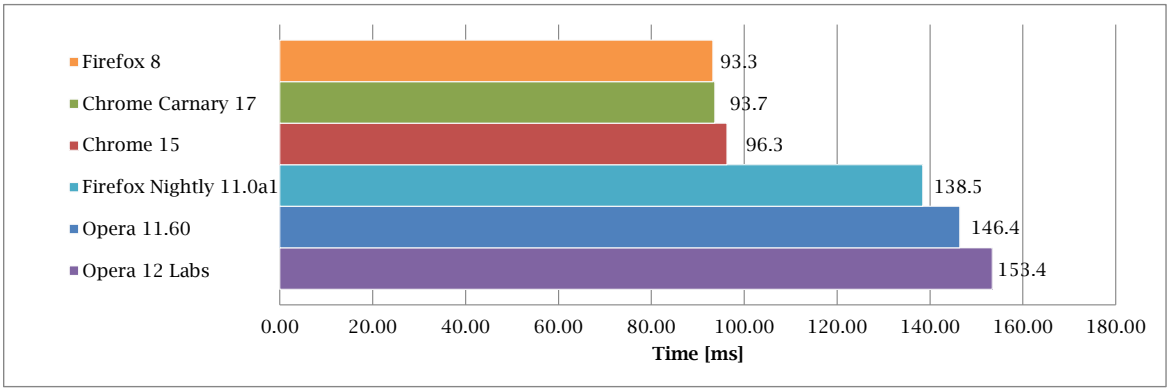


Figure 5.2: Detection Pipeline: overall PC cross-browser performance

Overall The ranking in Figure 5.2 summarizes the overall performance of the evaluated desktop browsers. Firefox in its latest stable version (FF8) takes the lead in the detection phase with an average time of 93.2 ms for detection. Google Chrome spends only 3 ms more on that task and scores second place with 96 ms. The Opera browser comes last with an average of 146 ms which is over 50% slower than Firefox and Chrome.

As for the developer preview versions, the newest Chromium snapshot scores best among them being just 2% ahead of the final release. Surprisingly, Firefox Nightly performs much worse than the latest stable (FF8), yielding in a slow-down of almost 50%. Not only the new Firefox version suffers from low maturity, the Opera 12 Lab version runs 5% slower than its final counterpart.

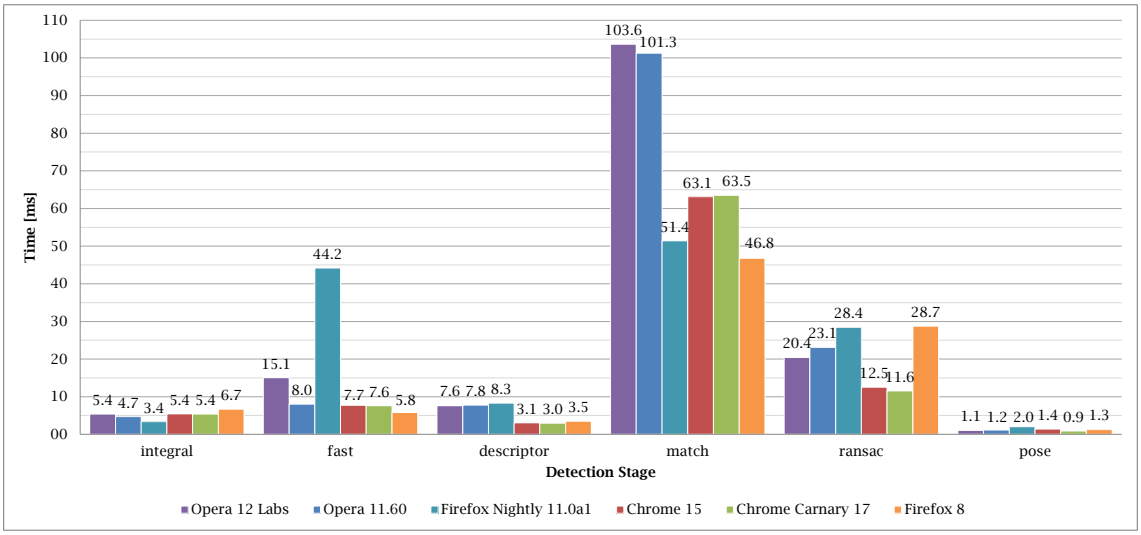


Figure 5.3: Detection Pipeline: PC cross-browser profiling

Profiling Figure 5.3 illustrates the breakdown of the overall timings for each stage of the detection pipeline.

During computation of the integral image, significant differences can be observed, ranging from 3 to 7 ms. While the Opera browser takes the lead for the group of final versions, Firefox Nightly outperforms all other competitors ranging from 40% (Opera) to almost 100% (FF8).

FF8 shows top performance at the task extracting FAST corners. FF(finishes this task in an average of 5.8 ms which is about 34% faster than all other competitors. Taking a look at the dev versions, Firefox and Opera Labs break ranks. While the latter doubles the time in comparison with its stable version, the former runs more than 7x slower.

The task of creating the BRIEF descriptor is finished by FF8 and Chrome in 3 ms,

while Opera seems to have problems performing binary operations completing this task in 8 ms. There is no obvious difference between the Chrome and Opera dev builds with their final counterparts, Firefox 10 instead struggles and describes the keypoints 2.4x slower.

Until this stage, the runtime of the individual parts was between 3 and 8 ms. The following matching stage is the most time expensive part of the pipeline ranging from 47 ms (FF8) up to 104 ms (Opera). FF8 is an average of 35% faster than Google and over 50% compared to Opera. During matching, no significant differences between the dev. builds and the final releases could be observed. This is valid for the outlier removal as well.

While FF8 is ahead of Chrome in the previous 3 stages, *outlier removal* is about 2.5x slower. Opera performs similar to Firefox, performing just slightly better.

No significant differences could be observed during the pose estimation stage, where all browsers processed that step in 1 to 2 ms.

Discussion The detection phase is processed at interactive framerates (above 10 fps) in half of the browsers, while the other half can only maintain 6-7 fps. The former group consists of FF8 and both Chrome versions, whereas Firefox Nightly and both Opera versions are part of the second group. Due to the fact that the tracker does not remain in the detection phase if a sufficiently stable pose is found, the speed is not as important for AR experience as for the tracking phase.

As the dissection of the timings of the pipeline shows, each JavaScript engine possesses certain strength and weaknesses. While most of the browsers show a similar pattern, Firefox Nightly breaks ranks. The only stage where it outperforms its stable counterpart is the computation of the integral image. The most significant slow-down is seen in the FAST corner detection. The problem may lie within the new JavaScript engine called IonMonkey [Moz11j], which is supposed to execute the code much faster than the previous version. Due to the early stage of the development, problems induced by the extension Type Inference (TI) [Hac11] may be ironed out in the final release. It might even be the case that the JIT compilation breaks and the engine falls back to interpreting the bytecode. A more concise investigation of that problem is to be found in the appendix (see Section A.1).

Apart from Firefox Nightly's problems, the Opera browser shows weaknesses in the matching stage while spending twice the time in this stage compared to FF8. The computation of the Hamming-distance is not as efficient as in other JavaScript engines, infering that the integer handling mechanism in Carakan is not yet fully optimized. The prob-

lems caused by different integer implementations in the respective JavaScript engines are described in the appendix (see Section A.3).

5.2.1.4 PC vs. Mobile

This section focuses in the comparison between the desktop's and the mobile's device performance. In order to give them equal chances, the same Firefox versions were installed on both systems. In addition to the Firefox browser on Android, the built-in WebKit browser was included in the experiments as well.

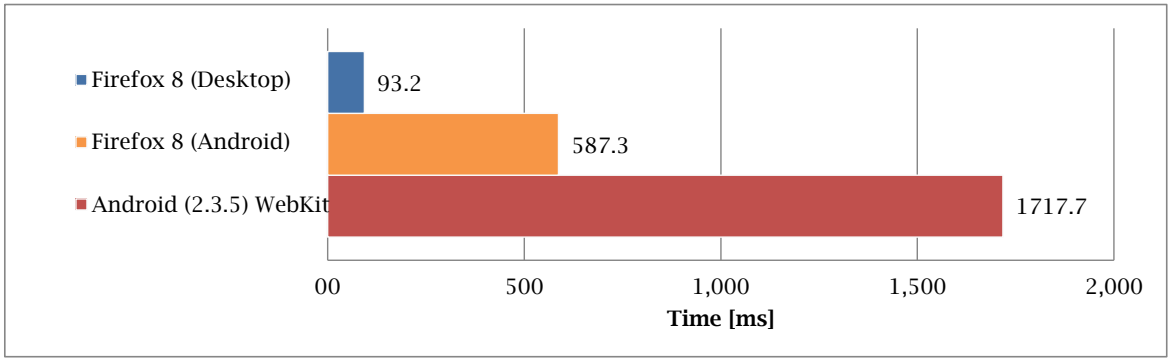
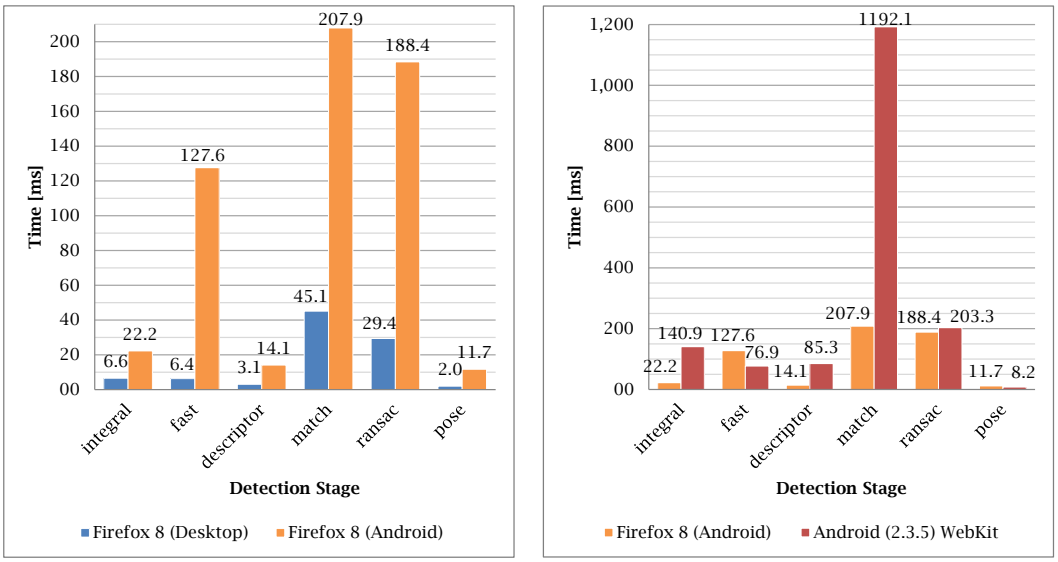


Figure 5.4: Detection Pipeline: Overall PC vs. mobile performance

Overall As seen in Figure 5.4 carrying out the detection task on the Samsung Galaxy S II takes an average of 587 ms, which is about 6x slower than on a mid-range desktop computer. In comparison to the built-in WebKit browser, the Firefox browser for Android outperforms the former by over 400%.

Profiling For compatibility reasons, only the Firefox versions of both systems are compared to each other. On the mobile phone, the average time to detect the target takes about 587 ms per frame. This is about 6 times slower than on the PC. Interestingly, mostly the ratio is between 3.4 (integral image) to 6.4 (outlier removal), except for corner detection, which is more than 20 times faster on the PC.

When comparing the built-in WebKit browser in Android with Firefox for Android, the overall difference is significant. While the former is 4 times slower than the latter, the slow-down is not evenly distributed over the entire pipeline. Calculating the integral image (6x), keypoint description (5x) and the matching stage (9x) are substantially slower in WebKit. On the other hand, the FAST keypoint detection (1.7x) and pose estimation (1.5x) are faster than in Firefox. In the RANSAC stage, no differences could be observed.



(a) PC vs. Mobile

(b) Firefox 8 mobile vs. WebKit

Figure 5.5: Detection Pipeline: PC vs. mobile profiling

Discussion Firefox mobile running on the Samsung Galaxy S II smart-phone processes the pipeline about 6 times slower than a mid-range PC, yielding in a framerate (2 fps) far from interactive (above 10 fps). This is not an issue as long as a robust initial pose is found and handed over to the tracking phase. The Android WebKit browser is not as efficient as Firefox spending over 2 seconds on a single frame to detect the target.

While the runtime of the individual detection phases was about 6 times faster on the PC compared to the smartphone the FAST corner detection was even 20 times slower on the mobile. When focusing on the WebKit browser it finishes the FAST stage by a factor of 1.7 faster than Firefox mobile. This is very interesting, since the WebKit browser is an average, taking the whole pipeline into consideration, 4x slower than the Firefox mobile. The Android WebKit browser is based upon Google's V8 JavaScript engine [Bru09] [Kra10] and delivers much worse results in comparison with the desktop version incorporated in Chrome. On the PC, Firefox outperforms Chrome by a factor of 1.3 during corner detection, while the ranking is upside-down when running on Android. The SpiderMonkey JavaScript Engine might not optimize the deeply-nested if-else branches, as applied in FAST, when running on ARM devices.

The weaknesses of the WebKit browser are the keypoint matching and description

stages, where former is 9x and latter 5x slower compared to Firefox mobile. It may be concluded that the handling of binary operations and integer types is not yet fully optimized in V8 for running on mobile devices.

5.2.1.5 Robustness

The detection pipeline is not covariant to rotation and perspective distortion to a large degree [CLSF10]. Robustness to scale is dependent on the number of image levels used, which is set five levels allowing a certain degree of scale covariance. Figure 5.6 shows two scenarios, one rotation and one tilt, representing the bounds of feasibility.

Quality measure In order to get a simple quality-measure of the inlier set, the number of inliers is taken into consideration. If the number is below a certain threshold, in this case 80, then the detection is considered not successfully.

Test setup & Test method Both robustness tests started off with the same initial position where the camera was positioned with its image plane parallel to the target image aligned in the center occupying 50% of the view.

The robustness to rotation was then tested by rotating the target around its z-axis (perpedicular to the image plane) in steps of 5 degrees. This was repeated until the pipeline was not able to detect the target any more.

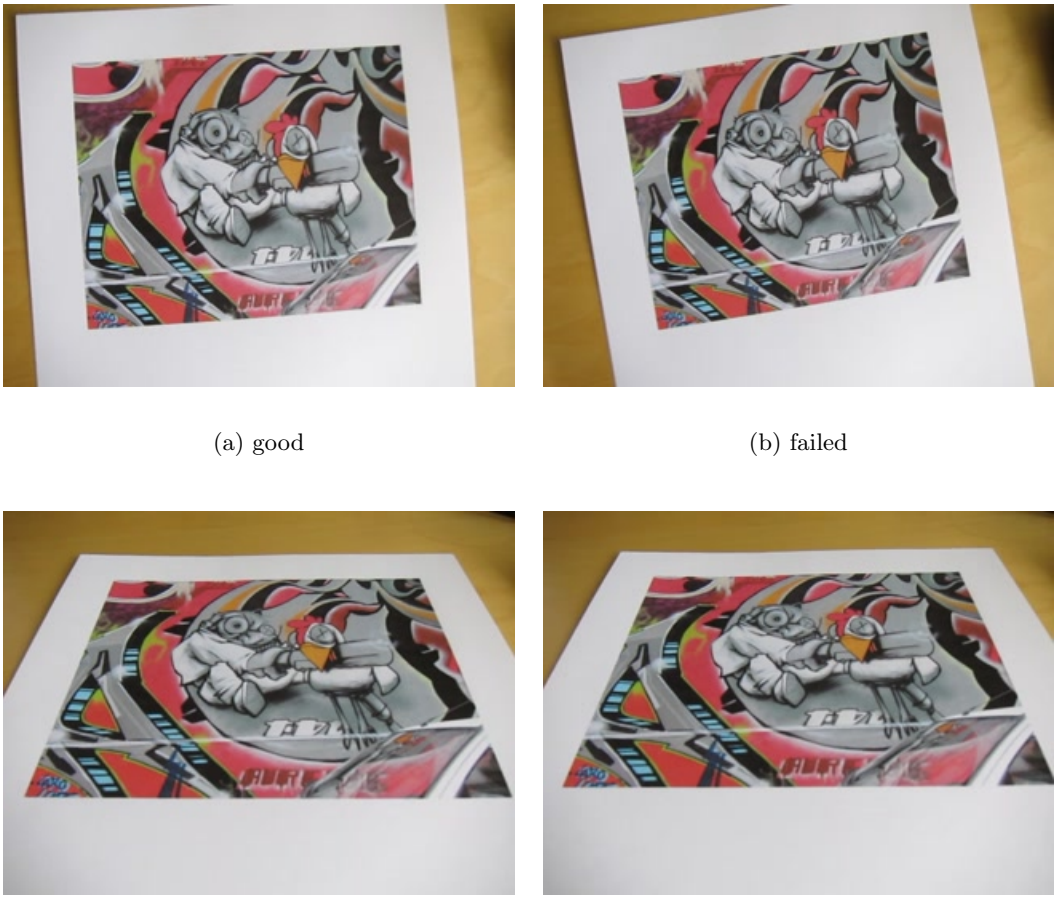
The test concerning the robustness to tilt started with the same initial position as described above. This time, instead of rotating the camera in the direction of the z-axis, a rotation around the x-axis was applied. The rotation started off with 0 degrees, meaning the image plane is parallel to the target, and was then increased by 5 degrees in each step continuing until the detetor fails.

Results Figures 5.6(a) and 5.6(b) show two subsequent frames recorded during the rotation robustness test. The frame on the left hand side was still detected (5 degrees) while the increase to 10 degrees could not be detected any more.

The second scenario, illustrated in Figures 5.6(c) and 5.6(d) demonstrates the robustness to tilt. While a tilt of 55 degrees (5.6(c)) could be detected successfully, an increase to 60 degrees (5.6(d)) led to a fail of the detector.

Discussion As already pointed out above, the detector's ability to detect the target under rotation is rather limited (5 to 10 degrees). This is not considered a problem since

the detection phase is only required during initialization and after tracking loss. The second test scenario, which concerned with perspective distortion in case of camera tilt, revealed that a rather high level (55 - 60 degrees) of robustness is reached.



(a) good (b) failed

(c) good (d) failed

Figure 5.6: Robustness of the Detection-Stage under various viewpoints

5.2.2 Tracking pipeline

As for the detection part, the evaluation of the tracking pipeline is divided into three sections. Starting with a cross-browser speed comparison carried out on the desktop PC. After that, different browsers are tested on the smartphone in addition to a comparison between the two platforms. The robustness of the tracking pipeline is the focal point in the third section.

5.2.2.1 Settings

The detailed configuration of the tracking-phase is shown in Table 5.3. The separation into mobile and PC is necessary due to the lower processing power of mobile devices. The configuration for PCs is more robust and can maintain a more stable pose where enough processing power is available. While mobile devices operate 5 to 10 times slower than most modern PCs (see Section 5.2.1.4) the configuration is therefore kept simplier and less robust.

	PC	mobile
Image size (pixel):	320x240	320x240
Patch size (pixel):	8x8	8x8
NCC-threshold:	0.5	0.5
Pose-refinement (max. iterations):	5	5
Levels (num):	3	2
Keypoints per level:	120, 100, 70	60, 40
Search-radius in px per level:	2, 2, 4	2, 3
Typed arrays	disabled	disabled

Table 5.3: Patch-tracker settings for evaluation

5.2.2.2 Test-set & Test-method

In order to achieve comparable results and carry out a fair competition, not a video but a static series of subsequent frames was used in the tracking phase. This ensures that the track does not get lost, even if the browser would be too slow to maintain in tracking phase when operating on a video.

The testset contained 100 still images which were extracted from a pre-recorded video clip. In order to test larger frame-to-frame changes, only every third frame was included into the testset. This reduced the original framerate from 30 fps to 10 fps.

Apart from an overall speed comparison, the pipeline was also profiled in order to compare separate steps to find possible strength and weaknesses of each individual contestant. For that purpose, the pipeline was split up into three main parts *keyframe* creation, keypoint *tracking* and *pose* refinement. The *keyframe* creation takes the gray-scale image and performs half-sampling depending on the number of levels (see Table 5.3). The two latter parts, keypoint *tracking* and *pose* refinement are performed on each image level. Hence, depending on the number of image levels, the tracking and pose refinement steps appear more often in the chart. Keypoint tracking starts with a set of keypoints which

are searched within the current frame and results in an updated set of correspondences. This represents the input to the pose refinement stage, where the current camera pose is iteratively refined as described in Section 3.7.2.2.

5.2.2.3 PC: Cross Browser

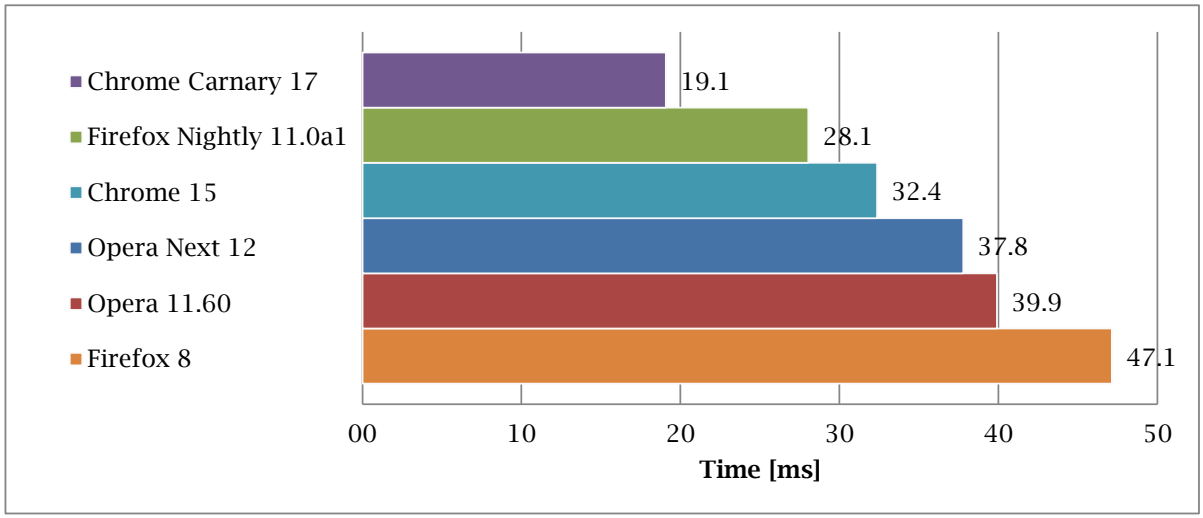


Figure 5.7: Tracking Pipeline: Overall PC cross-browser performance

Overall The ranking illustrated in Figure 5.7 gives an overall picture of the web browsers' performance. The big difference (28 ms) between the fastest (Canary 17) and the slowest browser (FF8) is very interesting. While Canary 17 processes one single frame in an average of 19.1 ms, FF8 is 2.5 times slower (47.1 ms).

Interestingly, the newest development build of Google Chrome (Canary 17) is more than 40% faster compared to its latest stable counterpart (32.4 ms). Firefox Nightly ranks second (28.1 ms) in between the two Chrome browsers, outperforming its predecessor (47.1 ms) also by 40%.

The Opera browser in version 11.52 spends an average of 48 ms on processing a single frame, scoring slightly better than the last one (FF8). Compared to the first rank, the slowdown is more than 100%. In respect to the best stable browser (Chrome 15) the performance penalty is reduced to 30%.

Profiling The break-down of the overall timing into the individual stages is presented in Figure 5.8. Generally speaking, the creation of the keyframe is finished in between 1

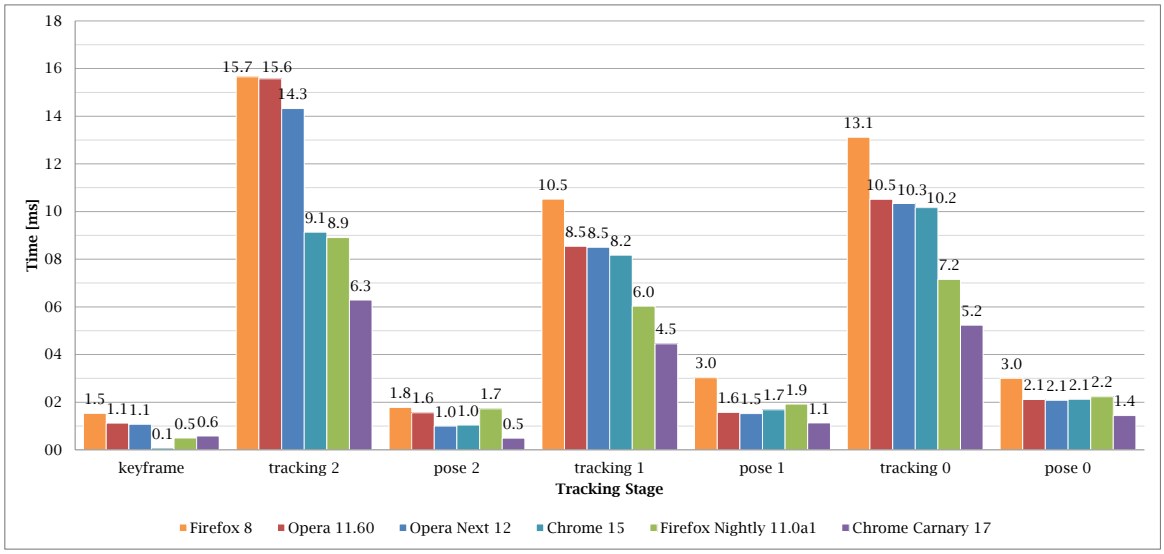


Figure 5.8: Tracking Pipeline: PC cross-browser profileing

and 2 ms among all browsers. Due to the resolution of 1 ms for measuring timings, values below 1.0 ms must be treated very cautiously, because of rounding errors. Consequently the performance in the first stage cannot be compared based on absolute numbers but only the ranking can be taken into consideration. Among the stable browsers, Chrome 15 is ahead of Opera, while FF8 is the slowest. The developer builds of Chrome and Firefox are performing best when creating the keyframe.

A similar result is shown during keypoint tracking, where FF8 and Opera bring up the rear with an average of 16 ms while Chrome 15 and Firefox Nightly spend 9 ms on that task. Chrome Canary takes the lead being 20% ahead of its competitors finishing the keypoint tracking in 6.3 ms. In this stage, a substantial speed-up of almost a factor of 2 can be observed between the FF8 and Firefox Nightly. A similar observation can be done when comparing the latest stable version of Chrome with the development build (Canary 17), where latter is more than 30% faster than the former. The keypoint tracking in the subsequent levels are performing in the same nature and are therefore not subject for further discussion.

The last step in the pipeline, the pose refinement, does not reveal much new information. Both Firefox versions including Opera are for the most part slower by a factor of 1.2 to 1.8 than the competitors developed by Google. Generally, the pose optimization task takes between 1 and 3 ms to yield in satisfactory results.

Discussion All web browsers are capable of processing the tracking pipeline fast enough to achieve real-time framerates (above 20 fps). The best performing browser in this pipeline is Chrome Canary 17 (52 fps), far ahead of Firefox Nightly (35 fps) and Chrome 15 (28 fps). The V8 JavaScript engine incorporated in Google's browser outperforms TraceMonkey present in FF8 (20 fps) and Carakan available in Opera (21 fps). Interestingly, the successor of the TraceMonkey JavaScript engine called IonMonkey (see Section 2.2.4.1) present in Firefox Nightly does not only overtake Firefox 8, but also all other competitors except Canary. While the new adaptations to the engine caused troubles in the detection stage (see Section 5.2.1.3), the tracking pipeline greatly benefits from those optimizations. Since this test heavily relies on array-access speed and infering the correct type of the used values, the newly introduced Type Inference (TI) [Hac11] [Moz11j] extension produces the desired effect.

The profiling shows, that most of the time in the tracking pipeline is spend during keypoint tracking (84%) whereas the pose refinement (14%) and keyframe creation (2%) are much less time consuming. The overall result is reflected in the profiling, where the ranking of the browsers does not change significantly. The only exception is Chrome, being slower during keypoint tracking than Firefox Nightly, but faster in the pose refinement phases.

5.2.2.4 PC vs. Mobile

As for the detection phase, the tracking stage was also evaluated on the mobile phone in comparison with the desktop PC. For the purpose of fair competition, the same version of Firefox was installed on both systems. Besides Firefox on the smartphone, the Android WebKit browser was used in the cross-browser competition. The configuration of the tracker is shown in Table 5.3 whereas for this test, the settings present for the mobile device were also applied to the desktop PC.

Overall The chart presented in Figure 5.9 compares the two Android web browsers, Firefox 8 and WebKit, with the Firefox equivalent running on a PC. A clear advantage of the PC version of Firefox can be observed (16 ms), being 5.5x faster than the Android (91 ms) version whereas the WebKit (159 ms) browser slows down by almost a factor of 10.

A comparison amongst the mobile versions reveals a clear winner in the form of Firefox, which is an average of 70% faster than WebKit. The advantage in the tracking pipeline is not as obvious as it was in the detection stage (400% see Section 5.2.1.4).

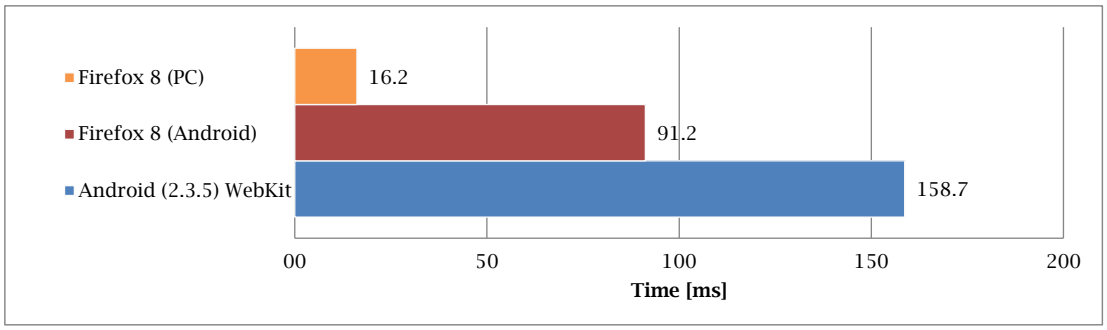


Figure 5.9: Tracking Pipeline: Overview PC vs. Mobile performance

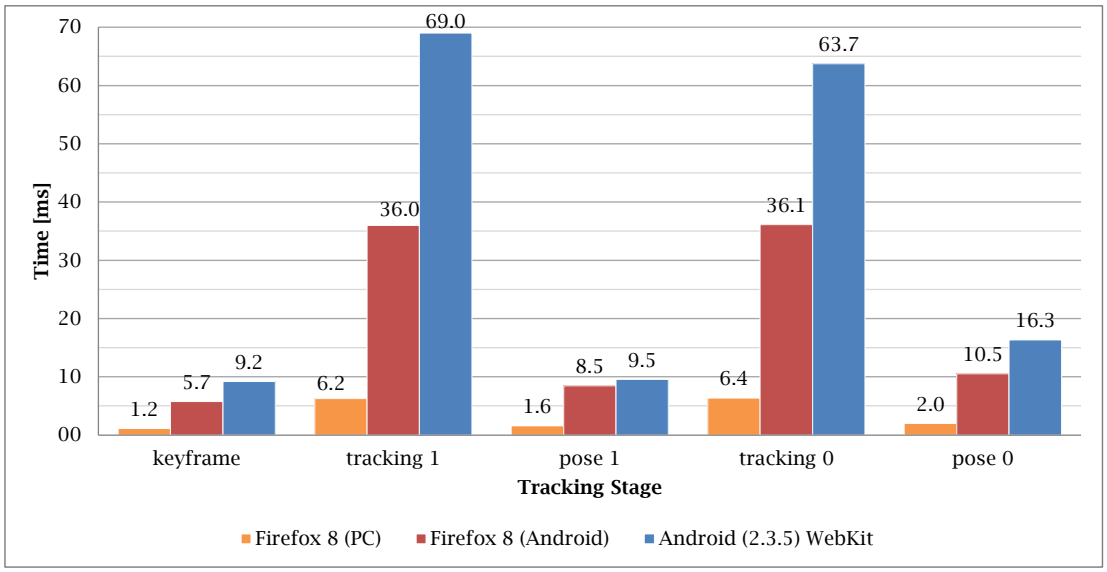


Figure 5.10: Tracking Pipeline: PC vs. Mobile profiling

Profiling The separation of the timings measured in the tracking pipeline is illustrated in Figure 5.10. When comparing the PC version of Firefox with the version for Android, the slow-down ratio is maintained between 5 (keyframe creation) and 6 (keypoint tracking).

When contrasting the WebKit browser and the Firefox Android, the difference is not evenly distributed among all the stages. While pose refinement is slowed down by a factor 1.1 to 1.5, the keypoint tracking takes almost twice as long (36 vs. 69 ms).

Discussion Summing up, an interactive framerate (above 10 fps) can be achieved when using the Firefox browser (11 fps) on Android. The WebKit browser achieves framerates only around 6 to 7 ms. The WebKit browser seems to have problems when operating on big arrays, since the array intensive tasks such as keyframe creation as well as the keypoint

tracking are much slower than the pose refinement step.

In comparison with the PC, both mobile browsers cannot compete on the same level. Despite the fact that the processing power is much lower on mobile phones compared to the PC, the efficiency of the JavaScript engines already reached a high level.

5.2.2.5 Robustness

The patch-tracker allows for more extreme angles, compared to the detection phase (see Section 5.2.1.5), due to the incremental tracking, predicting the distortion in the image. Two scenarios are presented in Figure 5.11 testing once the robustness to camera tilt and second the tracker's ability to follow the target when partially out of view.

Quality measure The quality of the patch-tracker is determined by the ratio of tracked points to visible points. If this ratio falls below a certain threshold (1/3 in this case) then it is assumed the tracking has failed. This information is supplied to the pipeline which then remains in the tracking phase or falls back into initialization.

Test setup & Test method The test concerning the robustness to perspective distorition, caused by camera tilt, is initialized with the camera's image plane positioned in parallel (0 degrees) to the target image. The target is aligned in the camera's image center occupying 50% of the view. Then a rotation around the x-axis was applied in steps of 5 degrees until the tracker failed.

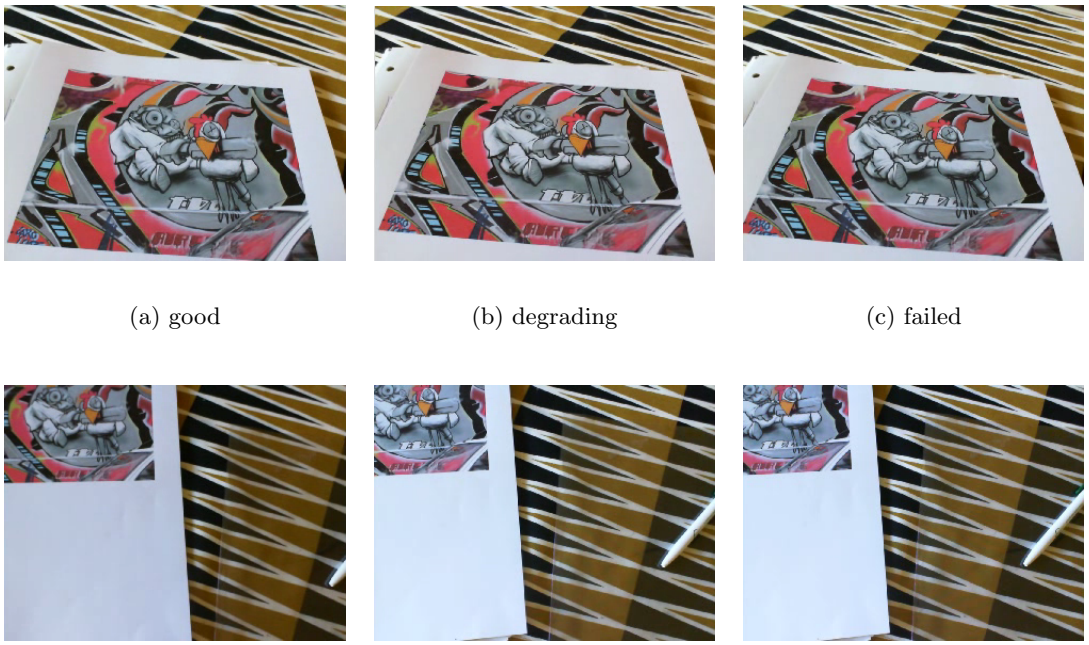
In the second scenario, testing the tracker's ability of handling occlusion, the camera is initialized in the same orientation but further away (30% of the view). The target was then slowly moved in the -xy direction leaving the view in the top-left corner. The movement was carried out by increasing the occlusion by 5% each step until the tracker could not longer follow.

Results The first row in Figure 5.11 shows the behavior of the tracking pipeline when tilting the camera. While the tracking quality is still good at an angle of 45 degrees (5.11(a)) the quality quickly degrades when incrementing to 50 degrees (5.11(b)) and finally fails in the next step (55 degrees) as seen in 5.11(c).

The figures in the row below illustrate the subsequent frames which were recorded during the test concering partial occlusion. The first frame (5.11(d)) shows 70% the target's area which results in good tracking quality while the successive increase of occlusion

(50%) degraded the quality (5.11(e)). Finally, the tracker was not able (5.11(f)) to follow the target any more with less than 45% of the target's area remaining in view.

Discussion The patch-tracker is robust to perspective distortion in high levels (up to 50 degrees) which is considered sufficient for many tracking applications. Also the ability to handle occlusion up to 50% makes the tracker robust in many situations.



(a) good (b) degrading (c) failed

(d) good (e) degrading (f) failed

Figure 5.11: Robustness of the Tracking-Pipeline under various conditions

5.3 JavaScript vs. C/C++

This section is dedicated to the evaluation of the implemented C/C++ detection pipeline which is being used in Google Native Client (GNC) and Adobe Alchemy. Both approaches are then ultimately compared with the JavaScript implementation.

For the profiling task of the pipeline, exactly the same parts were measured, namely creating an *integral image*, *FAST corner detection*, Keypoint *description* and *matching*, *outlier removal* with RANSAC and the final *pose estimation*. These stages are described in-depth in the implementation part (see Section 4.1.2 (JavaScript) and 4.2 (C/C++)).

5.3.1 Settings

The configuration of the detection pipeline is equivalent to the one used for the JavaScript evaluation (see Section 5.2.1.1).

5.3.2 Test-set & Test-method

Due to reasons for efficient comparison, the same test-set was used throughout all tests with regards to the detection pipeline. Again, 60 still images extracted from a video were used for that purpose. More details are to be found in Section 5.2.1.2.

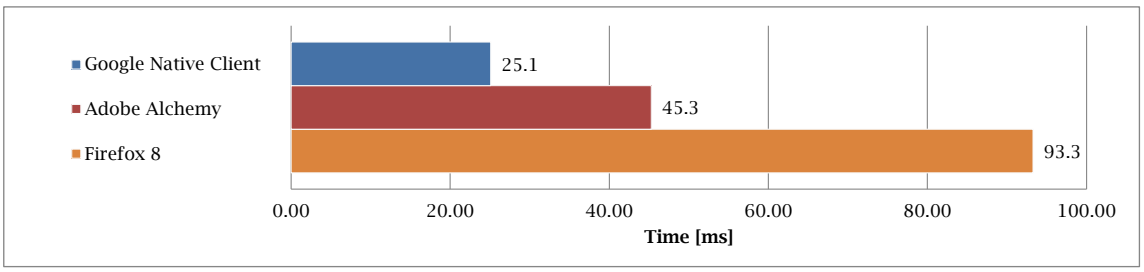


Figure 5.12: Detection Pipeline: Overall JavaScript vs. C/C++ performance

Overall Speaking in absolute numbers, Google Native Client performed best with an average of 25 ms per frame to detect the target whereas Adobe Alchemy spent almost twice the time resulting in 45.3 ms. The JavaScript implementation running in Firefox 8 finished last with an average of 93,2 ms per a frame. When taking JavaScript as the basis of the overall evaluation, the implementation in Google's Native Client shows a 4x speed-up whereas Adobe Alchemy processes each frame twice as fast.

Profiling When taking apart the individual timings of each different implementation, a more detailed analysis can be conducted, which is presented in Figure 5.13.

The performance ratio between JavaScript and Alchemy is very unevenly distributed throughout the entire pipeline ranging from 1.2x (matching stage) to 30x (RANSAC). When comparing the JavaScript implementation with Google's Native Client, similar results can be observed as the speed-up ranges from 0.75x (descriptor creation) to 30x (RANSAC).

The creation of the integral image takes about 6.6 ms in Firefox 8, while Adobe's Alchemy is almost 5x faster (1.4 ms) and NaCl even faster (below 1 ms). Almost the same results can be observed in the corner detection stage, where GNC (1 ms) is ahead of the

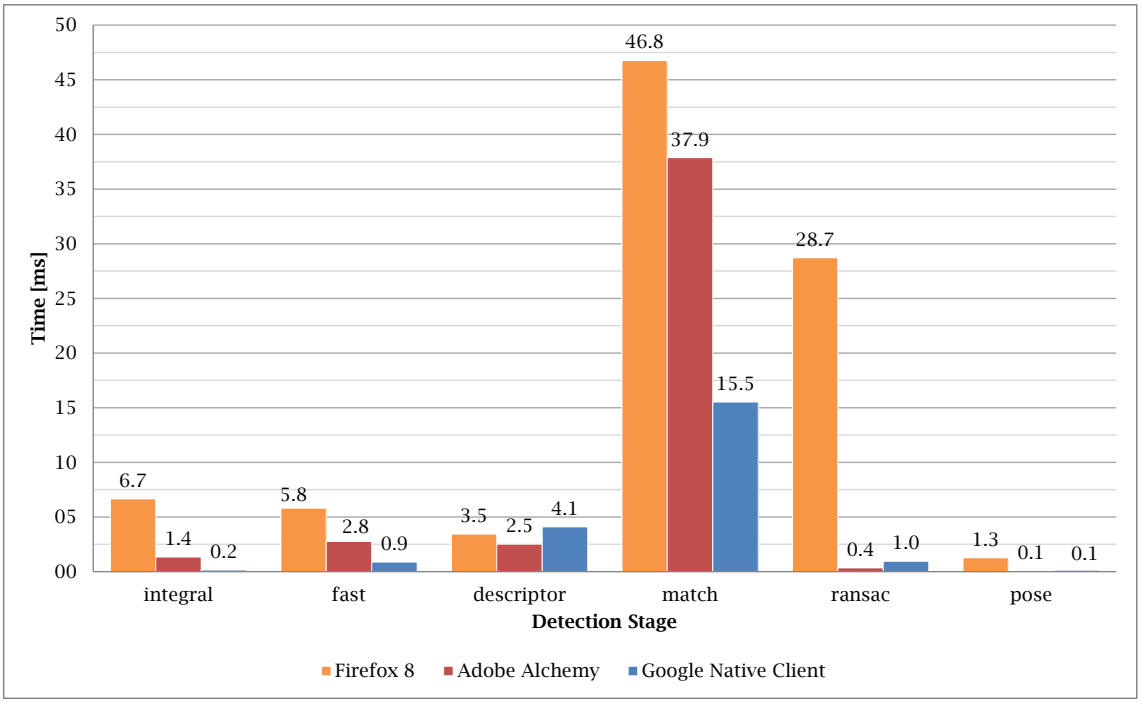


Figure 5.13: Detection Pipeline: JavaScript vs. C/C++ profiling

other contestants being 3x faster than Alchemy (2.8 ms) and 6x faster than JavaScript (6.4 ms). The descriptor creating stage is the most interesting one since all three competitors are almost equally fast. While Alchemy is finished in 2.5 ms, the JavaScript implementation ranks second (3.1 ms) with one `ms` ahead of GNC with the slowest (4.1 ms) performance in this step. During keypoint matching, the initial ranking is restored. Where the JavaScript implementation takes 45 ms to process, the code running inside Adobe Flash Player performs 16% better, and the NaCl module finishes the stage in even 15.5 ms, reducing the time by 65% compared to JavaScript. A completely different picture emerges in the RANSAC stage, where the JavaScript implementation (29.4 ms) is more than 30 times slower than both of the contestants ($\approx 1ms$). The last step, consisting of homography computation and pose estimation, both C/C++ participants deliver times which cannot be accurately measured, therefore considered being non-existent.

Discussion The overall observed difference between the JavaScript implementation and the C/C++ based counterparts remains mostly on the same level throughout all the stages in the detection pipeline. The drawback in JavaScript is strikingly present in memory-access intensive tasks such as integral image computation and outlier removal.

Surprisingly, working with binary data does not always perform better in the alternative implementation, since the descriptor creation was faster in JavaScript than in GNC. The most time consuming part of the pipeline is the matching stage in which JavaScript is only 18% slower than Adobe Alchemy. The ActionScript bytecode running inside the Flash Player is also executed by a virtual machine, but having type information available which is supposed to be a major advantage. This type information does not significantly speed up the excessively used binary operations in the matching stage. In contrary, the natively compiled version present in GNC finishes the matching task in 1/3 of the time.

5.4 Discussion

This chapter presented the results of extensive evaluations and tests carried out on the PC and mobile phone concerning speed and robustness of the JavaScript implementation.

Tests conducted on the PC revealed that the detection pipeline is executed at interactive framerates (>10 fps) whereas the tracking pipeline runs at real-time speed (>20 fps).

The cross-browser comparison of the detection pipeline showed that JavaScript engines are not yet optimized for handling binary data as seen in the matching stage, the most time-consuming one accounting for over 50% of the time. In this stage, Firefox 8 (FF8) is more than 2 times faster than Opera. When focusing on the FAST [RD06b] stage, interesting results could be observed where the newer JavaScript engine IonMonkey [Moz11j] (FF11) is 7 times slower than JägerMonkey [Bli10] (FF8), its predecessor. Generally speaking FF8 was the fastest during the detection phase being able to process a single frame in 94 ms, followed by Chrome (96 ms) and Opera (146 ms).

When concentrating on the tracking pipeline, the overall ranking is completely different than during detection. The new developer build of Google Chrome outperforms all other contestants by at least 50%. Not only the newest Chrome is preview faster than its stable predecessor (32 vs. 19 ms), but also Firefox Nightly shows significant speed improvements (28 vs. 47 ms) compared to the latest final release (FF8). Opera, however, does not show much difference between their latest final and development releases, ranking just ahead of FF8 with 38 ms per frame.

This indicates that the browsers currently under development (Chrome 17 & Firefox 11) are far ahead of their stable counterparts, except during the detection phase, where FF11 shows shortcomings in the FAST corner detection.

The C/C++ implemented detection pipeline running inside the ActionScript VM

(Adobe Alchemy) and natively on the CPU (Google Native Client) speed up the runtime compared to JavaScript by a factor of 2 and 4 respectively. This concludes, that JavaScript is still far from competing with native code, but already fast enough to power the tracking pipeline in real-time.

The same tests carried out on the smartphone performed 5 to 6 times slower compared to FF8 on the PC. When contrasting the built-in Android WebKit browser and FF8 mobile, the former performs 1.7 (tracking) to 4 (detection) times slower. While the detection phase runs with an average of 2 fps, the tracking pipeline can maintain interactive framerates (11 fps) in FF8. Difficulties inherent in FF8 could be observed during the detection phase where FAST was 1.7x slower than on the WebKit browser, while the latter performed 9x slower during the matching stage.

To sum up, interactive framerates may be possible when employing FF8, although deeply nested if-else constructs (FAST) are causing problems during detection.

Tests concerning robustness showed that the detection stage consisting of FAST and BRIEF is not very robust to rotation and viewpoint changes. Due to the multi-scale FAST approach, scale-invariance can be achieved on a certain level. In contrast, the tracking phase benefits from a-priori information and clever prediction of the patch-transformations, the robustness to scale and rotation is ensured.

Chapter 6

Conclusions & Future Work

The main goals of this thesis were the development and evaluation of a natural feature tracking Augmented Reality pipeline solely implemented in HTML5 and JavaScript. The use of standardized web-technologies enables the AR pipeline to run in any modern web browser regardless of the operating system (Win vs. Mac vs. Linux) or platform (desktop vs. tablet vs. smart-phone) used.

The work started off with an extensive research on already existing approaches, which are entirely based on proprietary web-technologies such as ActiveX or Adobe Flash. The only notable implementation avoiding proprietary browser addons is a port of the popular ARToolKit to JavaScript, which is only capable of tracking fiducial markers. While designing the detection and tracking pipeline problems occured due to the living HTML5 standard and draft specifications, it was uncertain if an entire pipeline could be built. As a result of the implementation, the framework *augmentedJS* was created. Optimizations were required, especially during keypoint-matching, because of JavaScript's limited functionality when working on binary data. The final evaluation of the JavaScript implementation consisted of cross-browser comparisons both on desktop and smart-phone platforms in addition to alternative implementations carried out in Adobe Alchemy and Google Native Client. The quick iteration cycle of browser vendors made the evaluation process rather difficult when trying to reflect the latest developments.

Overall, *augmentedJS* is capable of running in real-time on moderate PC configurations and delivering interactive experience to smart-phones and thus successfully achieving the thesis goal. In comparison with the alternative approaches based on Adobe Alchemy and Google Native Client, JavaScript is still not as powerful as its competitors but mechanisms such as just-in-time compilation and type inference are gradually reducing the gap. Apart

from the performance's perspective, the support for recently introduced technologies in the respective web browsers varies greatly, especially when focusing on the access to the user's web-cam via `getUserMedia` and real-time 3D rendering with *WebGL*.

An AR NF-tracking pipeline, available in HTML5, opens new opportunities to provide potential end-users with a consistent AR experience throughout a variety of devices running HTML5-enabled web browsers. Since no installation or separate download is necessary, the user's acceptance is likely to increase. These findings point out that HTML5, in combination with JavaScript, forms a powerful platform for the development of high-performance applications and tools needed for data processing and 3D rendering. Furthermore, many developers can benefit from writing Software in HTML5, as this reduces the development costs when support for different devices and platforms is required.

6.1 Future Work

The next step to be taken is the publication of the *augmentedJS* framework, and making it available as an open source resource to the developer community. Interested developers and research facilities may benefit from the available code and appreciate the chance to contribute in order to increase the maturity and compatibilty.

During the design stage, ideas arose to use hybrid tracking modalities by making use of the built-in inertial sensors available in today's smart-phones and tablet computers such as: compass, GPS and accelerometer. The HTML5 standard proposes the access to these sensors via the Geolocation API (GPS) in addition to the DeviceOrientation API (compass, accelerometer and orientation). This hybrid approach may be used during the initialization phase of the AR pipeline in a greater context.

Although great effort was put into optimizing the performance, additional steps may be taken into consideration to increase the responsiveness and thus the user experience of the JavaScript implementation. The framework could benefit from the more efficient implementation of typed arrays over native JavaScript arrays. This improvement may only work in specific cases and requires careful data-flow analysis in order to avoid pitfalls caused by a browser's internal optimization techniques.

Since JavaScript is executed in a single-threaded manner, and supports only limited multi-threading via the Web workers API, an approach utilizing the entire computing power of multi-core CPUs and GPUs is considered. The ability to mobilize all computing resources available on individual devices makes alternative AR tracking techniques, such as *parallel tracking and mapping*, feasible on the web. One way of achieving this may be the

use of *WebCL* - the *OpenCL* binding for JavaScript - allowing massive parallelism of code executing on the CPU and GPU. Another technique benefitting from parallel computing is provided by Intel's *River Trail* JavaScript engine leveraging multiple CPU cores and vector instructions to greatly increase the performance.

Appendix A

Appendix

A.1 Harris vs. FAST

This section investigates the performance of different corner-detection approaches such as FAST and Harris-corners. When it comes to the FAST implementation six variations (FAST-7 to FAST-12) were tested.

A.1.1 Testset & Testmethod

A sequence of static image files (84 pcs. 320x240px) extracted from a video clip were contained in the testset. The individual timings of detecting the corners in the images were then averaged.

A.1.2 Results

The diagram presented in Figure A.1 shows the time in [`ms`] for one single frame being processed by the use of different detectors.

When focusing on the different FAST-versions, a decline of time can be observed when increasing the length of the arc. While Opera and Chrome produce almost the same performance throughout all FAST versions, the Firefox browsers show weaknesses in the first three tests. FF8 spends between 38 ms (FAST-8) up to 90 ms (FAST-7) on detecting the keypoints when focusing on FAST-7 to FAST-9. As for the following tests (FAST-10 to FAST-12) FF8 is on a par with Opera forming the lead with 7 to 9 ms. As already pointed out in Section 5.2.1.3 the FAST stage causes troubles when using the IonMonkey JavaScript engine present in Firefox beginning with version 9.0. Even in the latter tests

(FAST-10 to FAST-12) the average time spent on detecting the corners is 5 (FAST-12) to 10 (FAST-12) times higher compared to FF8.

The Harris corner detector, as shown on the right hand side, performs worse compared to FAST. While the Chrome browser was the second slowest in FAST ranking behind FF8 and Opera, it outperforms all its competitors when applying the Harris detector finishing in 52 ms. The newest Firefox development build is slightly slower (76 ms) but far ahead of its predecessor FF8 (133 ms). The Opera browser ranks last as it is almost 6 times slower than Chrome.

The chart in Figure A.2 illustrates the stability of different browsers throughout the whole testset when applying FAST-10. When focusing on the first 5 frames a similar behavior of all JavaScript engines can be observed. While the generated native code is not yet optimized in the first frame/s, the initial performance is much slower compared to the subsequent frames whose performance is almost evenly distributed (except for FF12). This is true for Chrome, FF8 and Opera but not for FF12. When concentrating on the lowest processing time (5 ms) it is far ahead of all competitors, but due to code optimization problems the timings oscillate between 100 ms and 5 ms.

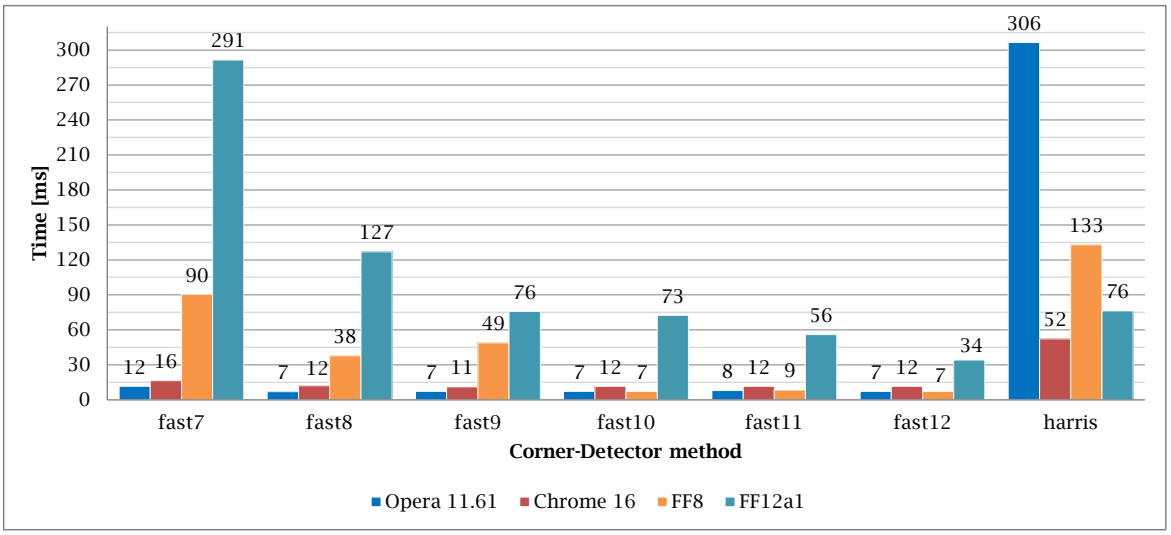


Figure A.1: Harris vs. FAST

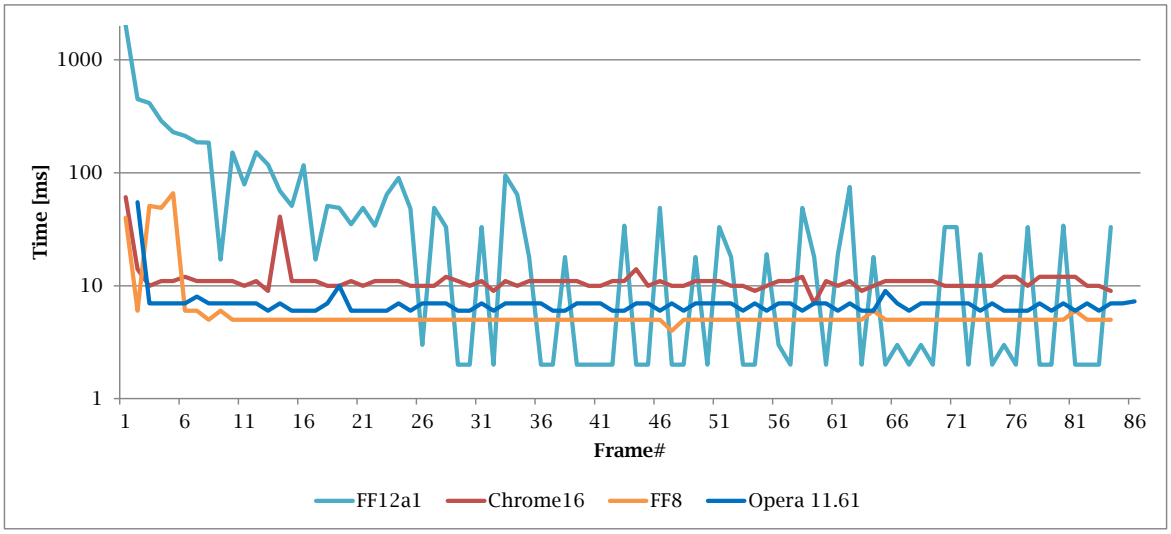


Figure A.2: FAST runtime-stability

A.1.3 Discussion

When comparing the FAST-10 corner detector with Harris a difference by a factor of 10 can be observed with most browsers. The only exception is FF12 which seems to have great difficulties with its newly introduced IonMonkey JavaScript engine.

A.2 Web worker

Nowadays the single-core performance has not increased much for nine years when the first CPU hit the 3GHz mark []. Since then Moores' Law could only be proven valid with the integration of more than one core per chip.

JavaScript is typically interpreted in a single-threaded manner and does not provide parallel programming. Hence, the presence of multi-core hardware does not affect the speed of execution. To overcome the lack of multithreaded programming, the HTML5 specification proposes a WebWorker [WHA11c] interface allowing to execute the code in another thread than the UI-one resulting in a much more responsive system. The communication between the main thread and the worker relies on messages sent between those two. The messages are based on JSON and are passed by value, which means that no shared data is hold between threads. This fact makes the overhead of passing data in and out heavily dependent on the amount of data being sent across.

A.2.1 Evaluating Web worker employment in the tracking pipeline

In order to determine the possibilities of Web workers to increase the performance of the tracker, different scenarios are tested. Firstly the idea of outsourcing the entire patch-tracker, and secondly the execution of the patch-finder in different Web workers.

Note: All tests are carried out with an implementation of a "ping"-operation within each worker, meaning no work is done at all. The results simply show the overhead of using web-workers in different scenarios.

A.2.1.1 Posting entire image to worker

In order to outsource the work of the patch-tracker to other threads, the current image has to be copied to the corresponding workers. Depending on the number of workers, sending the image-array to the worker is very expensive, yet needs to be done only once per frame and worker. Inside the workers, the most expensive operations *transform* & *find* can be executed.

Testset & Testmethod For this particular test one single Web worker was used as a receiver of messages. To imitate an entire gray-scale image an array consisting of 76,800 entries (320 x 240px) was created in advance. This array is used as the "payload" within the message sent to the worker. Only one message is posted for each single frame.

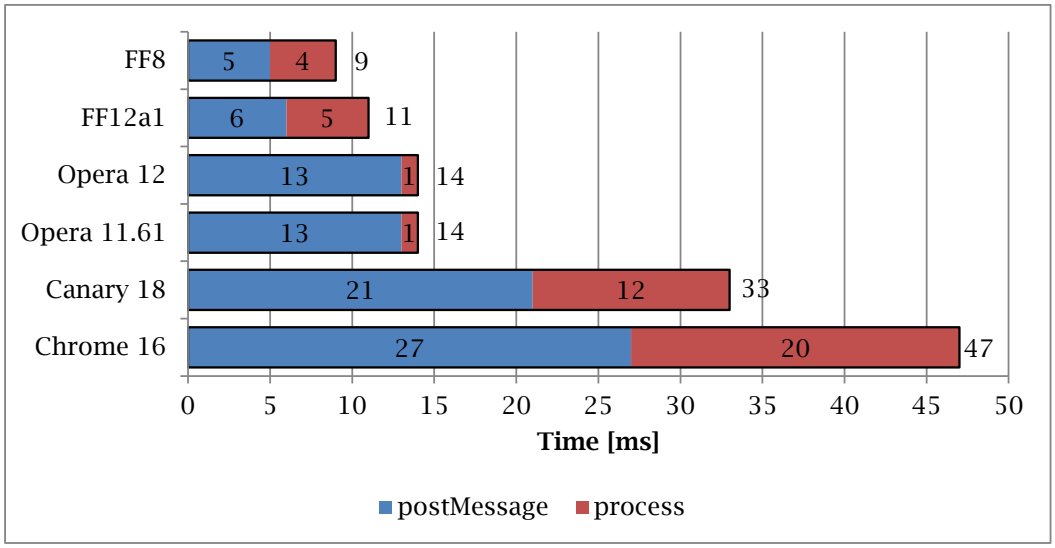


Figure A.3: Web worker performance when using entire image

Results The chart presented in Figure A.3 shows the individual timings (*postMessage*) for posting the single message from the UI-thread to the worker and the time (*process*) needed to execute the empty callback indicating that the message has been received.

The overall result ranges from 9 ms (FF8) to 47 ms (Chrome 16) indicating a big difference between the individual implementations. While both Firefox versions handle Web workers very efficiently, Google has not yet fully optimized its Web worker implementation, although improvements can be observed (Canary 18 vs. Chrome 16).

A.2.1.2 Posting patch & search-window for PatchFinder

In order to avoid sending the entire video-frame only the patch and the search-window are sent to the workers instead.

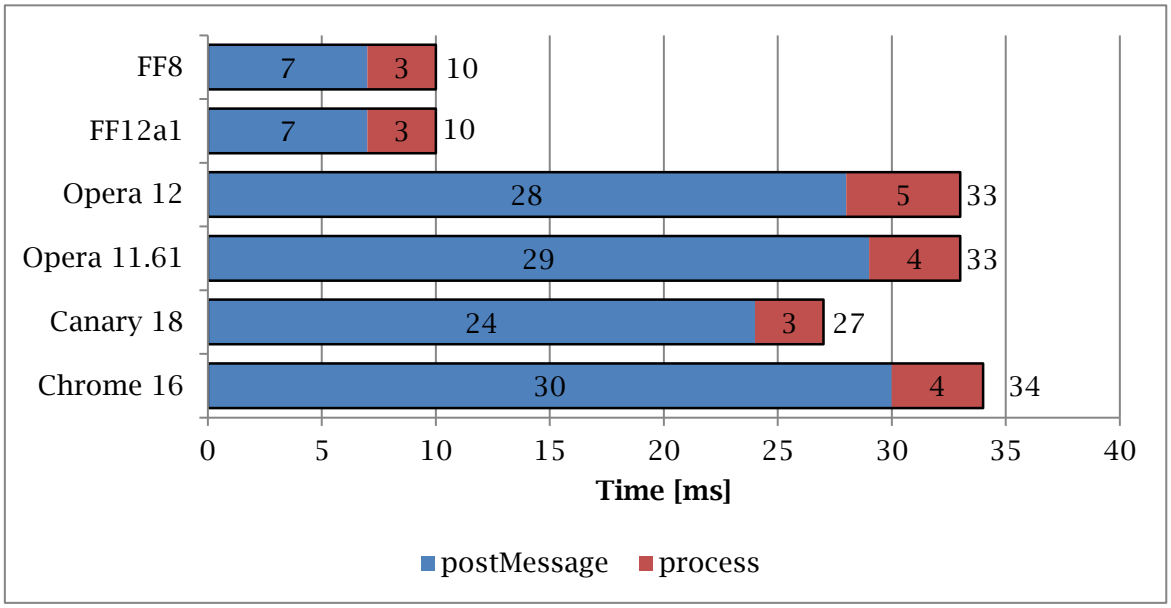


Figure A.4: Web worker performance when posting the search windows and patches 240 times with a payload of 280px

Testset & Testmethod For this test one single Web worker was created. Simulating the patch-tracker searching on three different levels (100 & 80 & 60 KPs) with appropriate radi (2 & 3 & 4 px) an average *payload* of 280px per search window sent 240 times is considered suitable.

Results Figure A.4 presents the individual timings for posting 240 messages from the UI-thread, which are each attached with a payload of an array having 280 entries.

Both Firefox versions take the lead with an overall of 10 ms for sending and returning to the UI-thread outperforming all other competitors by a factor of almost 3. While Opera and Chrome spend 33 ms, Canary shows slight speed improvements compared to its predecessor (27 ms).

A.2.2 Discussion

The results of the two test scenarios show that the overhead (10 to 47 ms) is significant when sending messages to Web workers without even executing any work. Tracking key-points on all levels takes between 16 and 38 ms (see Section 5.2.2.3) which is almost the same time needed for just posting the image data from one thread to another.

Web workers may be suitable for longer-running computations with less data to be shared, but not for the use inside the tracker. As a result, employing Web workers is not considered efficient enough to gain any benefit in executing the patch-tracker.

A.3 Integer-Performance evaluation

Because of the design decisions of many JavaScript engines, a number used as an integer may not always be represented as an integer internally. This is crucial to know when working with numbers for representing binary data. This section investigates the influence of using the `Number` implementation in different scenarios. The testcase is adapted to represent a part of the detection pipeline (matching) where binary data is excessively used. Herein the Hamming-Distance is calculated in order to measure the similarity of two arbitrary points. More on that topic can be found in Section 4.1.2.3

The `Number` class in JavaScript provides the only way of representing numbers. It is internally stored as a 64bit double floating point number [ECM11], but may appear as *integer* or *float* to the outside. Another problem worth mentioning is that working with a 32bit integer is not trivial in JavaScript. The difficulty does not lie in programming, but in the performance penalty when using 31 or 32 bit integers.

A.3.1 Testset & Testmethod

The testset consisted of two arrays each containing 1,000 entries of descriptors in the form used and described in Section 4.1.2.2. Each of the four numbers in the descriptor was only

filled with limited bits depending on the current test. The tests were carried out starting with 1 bit throughout to full 32 bits.

In order to represent a real-world scenario each of the descriptors contained in the first array was compared to the one in the second array which yielded to 1,000,000 calculations per test.

A.3.2 Results

As illustrated in Figure A.5 the performance of all browsers is constant until 30bits are reached. The Internet Explorer and Google Chrome are the first (31bit) to indicate problems, followed by Opera and Firefox in the 32bit case. While both Firefox versions show only little deviation the competitors increase the runtime by a factor of 2 (Chrome & IE9) to 4 (Opera).

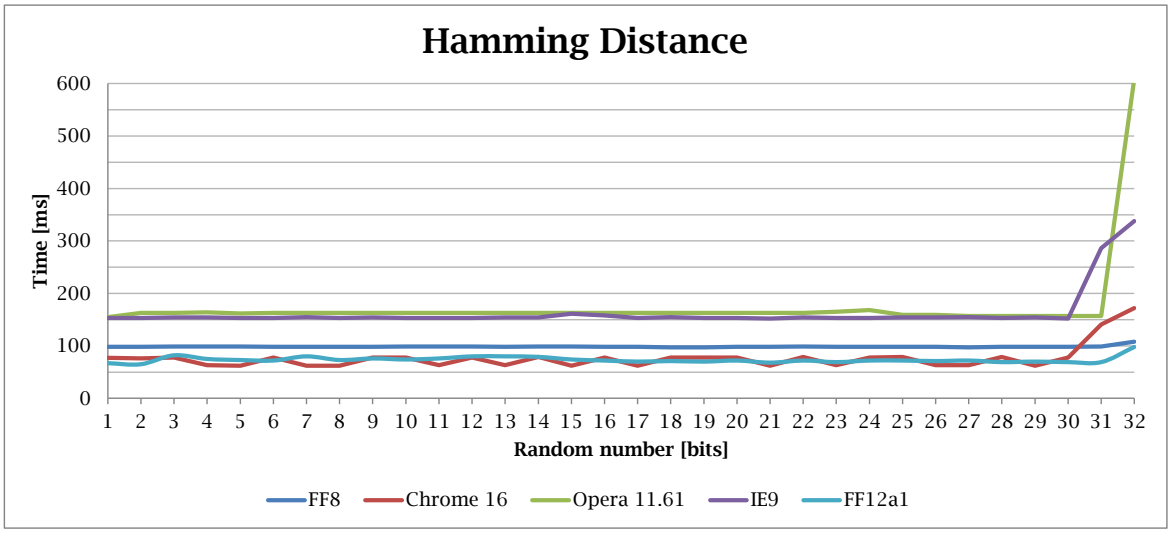


Figure A.5: Times executing 1,000,000 hamming-distance computations in 128bit

A.3.3 Discussion

The Opera browser performs very constantly including the test on 31bits. When operating on 32bit numbers the JavaScript engine struggles and consumes more than 400% CPU-time. The JavaScript engine Kraken may internally use a signed 31bit integer number where the MSB is reserved for special purposes.

The V8 JavaScript engine, incorporated in Chrome, uses internally ”small integers”

when operating on fixed-point numbers ranging from 0 to 2^{30-1}. These "small integers", called *smi*, are basically unsigned 31bit integers [win11a] [fsc11] [Goo11g] causing a slowdown when using 31 and 32bits.

When fast binary operations are needed in a cross-browser application, it is advisable to rethink the implementation and use a maximum of 30bits to be on the safe side. The code may become more complex, but may benefit from the limitations and design-decisions implemented in the JavaScript engines. An alternative test using only 25 & 26 bits is conducted in the next section.

A.3.4 Comparing 4 vs. 5 numbers for representing a 128bit binary descriptor

In order to circumvent the problems stated above another test was carried out using five lower resolution numbers instead of four. The new approach stores 2 x 25bit and 3 x 26bits in an array to represent the 128bit descriptor. The benchmark was adapted to the new scheme.

A.3.4.1 Results

The measurements of matching 1,000,000 128bit descriptors using four numbers (32bit each) and five numbers (25 or 26bit) are presented in Figure A.6.

All browsers show improvements using 5 numbers instead of 4 increasing the performance by a factor of 1.3 (FF8) up to 3.2 (Opera).

A.3.4.2 Discussion

Using the construct consisting of 5 numbers instead of 4 during matching dramatically decreases the CPU-time. Due to these results the implementation of the keypoint descriptor was adapted in order to avoid the implications caused by the different integer representations in the JavaScript engines.

A.4 Canvas drawing - image retrieval

Investigating the performance of grabbing frames from a video-clip and processing for internal representation. The AR-pipeline implemented in JavaScript makes use of the following flow: video → canvas → grayscale. First the video-clip attached to the `video` element is drawn on to the `canvas` element which acts as an image buffer. This buffer

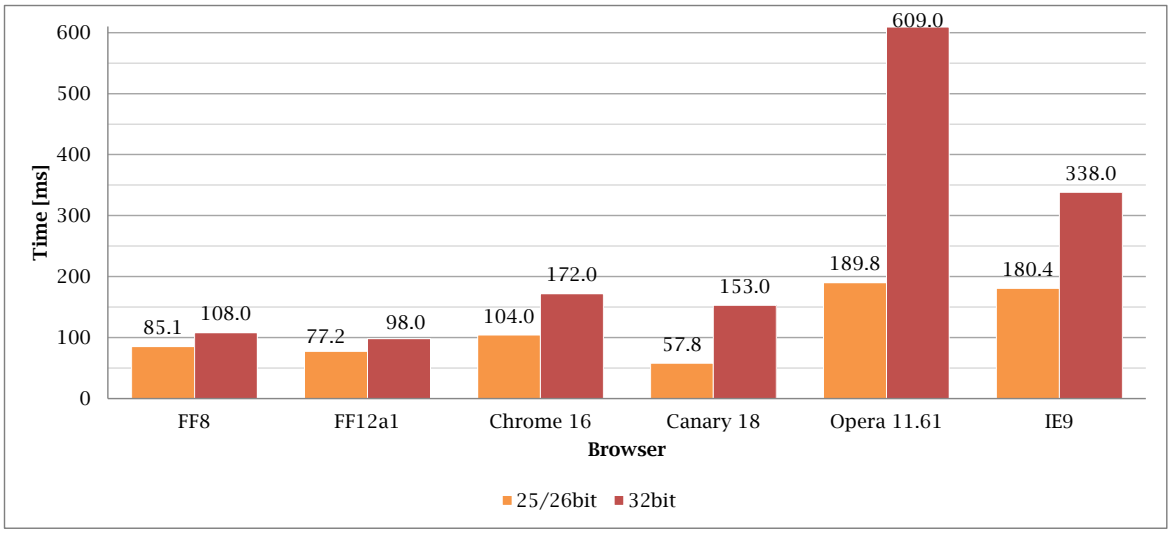


Figure A.6: Hamming-Distance computation comparison once using a maximum of 26bit per number and once using all 32bit.

is then read out and converted to gray-scale in the same step. More information can be found in Section 4.1.1.

Apart from measuring the performance of the detection and tracking phase, the input delay is equally important for real-time interaction. In this context the input is defined as the sequence of drawing the current video frame on a `canvas` and computing a gray-scale representation from the image buffer.

A.4.1 Testset & Testmethod

A pre-recorded video-clip was used as a source for the `video` element. The measurement of the performance starts with invoking the `drawImage` method of the canvas 2D context drawing the current video-frame on to the `canvas` element. The measurement ends when the resulting image-array is filled with gray-scale image data. This sequence was repeated until 200 frames were captured.

A.4.2 Results

The chart in Figure A.7 presents the time in `ms` for each frame to be drawn, read and converted to gray-scale. In this test, the Chrome browser ranks first with spending an average of 1.9 ms on that task. FF12 scores second (2.7 ms) and is thus slightly ahead of

both Opera browsers. FF8 performs worst, requiring 5.5 ms to process that task.

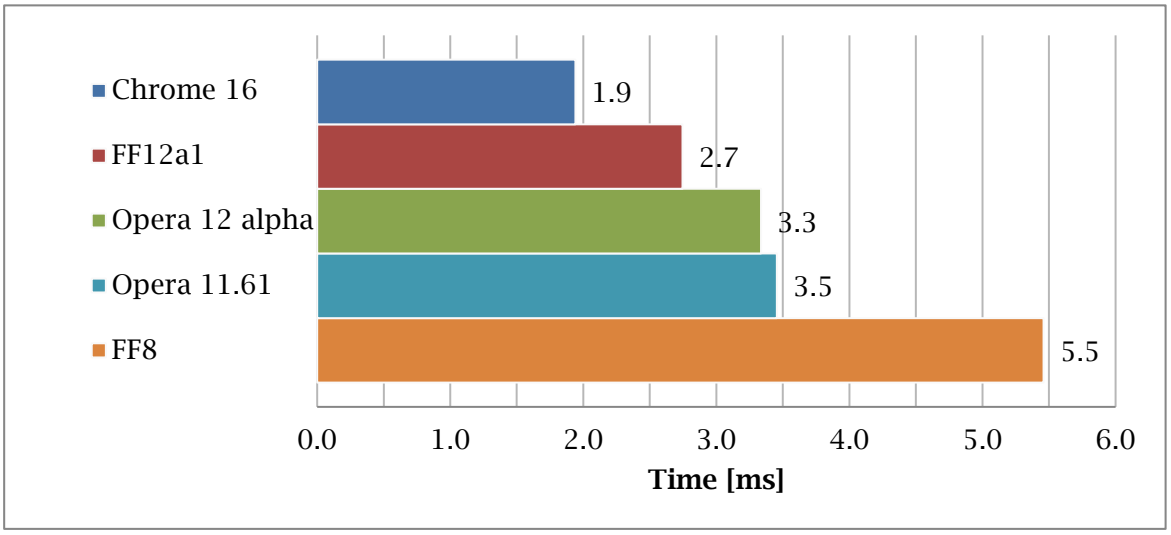


Figure A.7: Time needed for drawing a single video-frame to canvas, reading buffer and converting to gray-scale representation

A.4.3 Discussion

The times to acquire a single frame from the video-feed lie in an acceptable range (2 to 5.5 ms) causing no significant impact on real-time performance.

A.4.4 Real vs. virtual frame-rates on mobile phones

When performing the tests on mobile phones no actual measurements can be carried out because the canvas update may not always redraw the screen. The programmatically measured update rates may be much higher compared to the actual screen refresh. More information on that issue can be found in [Chr11].

A.5 Typed vs. Non-Typed

A.5.1 Testset & Testmethod

Two operations were extracted from the AR-tracking pipeline, namely *dot product* and *image moments*. The former operation computes the dot product of two images while the latter calculates the moments of a given image. Both functions rely on excessive array read-access and test the efficiency of different array implementations. The *dot product*

operates on two randomly generated 160x120px images resulting in 2 x 19,200 array access operations. The second test calculating the *image moments* operates on a 200x200px image yielding in 40,000 array read operations.

Three different types (`Uint8Array`, `Int32Array` and the native JavaScript `Array`) of arrays were tested in both functions (*dot product* and *image moments*) yielding in 6 individual test cases. Each test case was executed 1,000,000 times and finally condensed into measurements of 100 executions.

A.5.2 Results

The chart in Figure A.8 presents the results of the conducted tests.

Both Chrome browsers show their strengths when on operating on arrays regardless of the type used. FF12 is the only browser capable of competing with Google's products showing similar results when working on JavaScript arrays.

Firefox' and Opera's implementation of Int32Array is more than six times slower than the native JavaScript version. While FF8 does not show any differences when typed or native arrays are used, FF12 benefits from significant improvements when favoring Uint8Array over native JavaScript arrays.

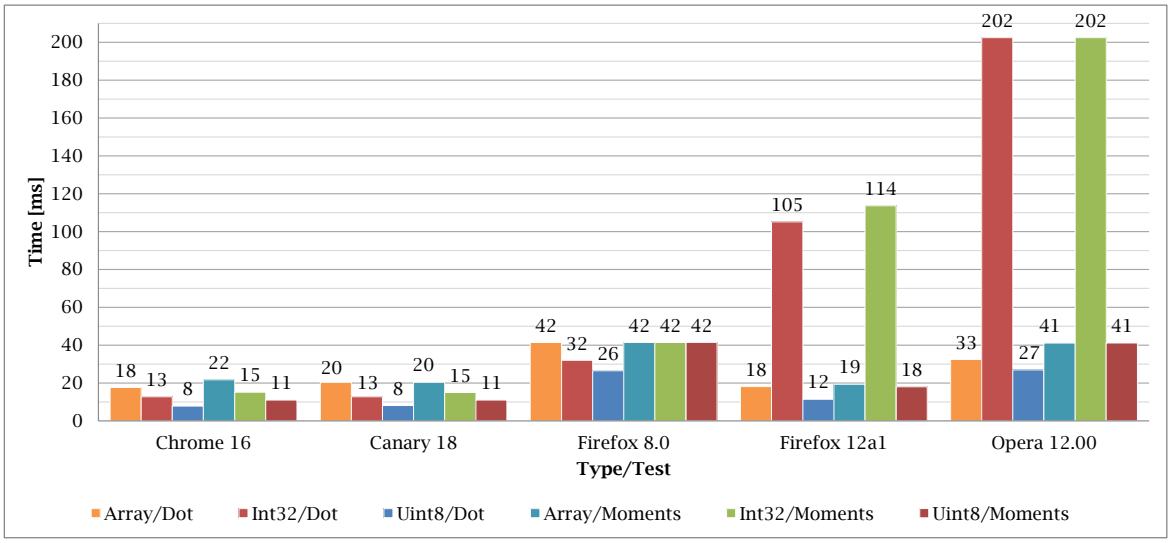


Figure A.8: Performance of different Array-Types used for computing the dot-product and image-moments. The results in `ms` show the time needed to execute the test-case 100 times.

A.5.3 Discussion

The anticipated performance improvement is not seen in all the tested browsers. While Google Chrome's implementation of typed arrays outperforms the native JavaScript one in all tests, FF12 and Opera show weaknesses in their 32bit array type (Int32Array). FF8 shows improvements when using typed arrays in particular cases (computing dot-product), but performs worst compared to all other competitors (except Int32Arrays).

Using typed arrays may speed up the code in specific cases (browser/code combination) but on the other hand, it is just possible that the execution might also be slowed down. Tests conducted on the AR-pipeline revealed that no speed-up is recognized, rather the opposite was the case.

To sum up, working with typed arrays requires well-designed data-usage and continuous testing in order to gain benefits.

Bibliography

[Ado08] Adobe Labs. Alchemy:libraries. http://labs.adobe.com/wiki/index.php/Alchemy:Libraries, 2008. [Online; accessed 19-November-2011].

[Ado09a] Adobe Forums. Bug: std::string serious problems. http://forums.adobe.com/thread/487570, 2009. [Online; accessed 19-November-2011].

[Ado09b] Adobe Forums. std::vector and memory allocation. http://forums.adobe.com/message/2227018, 2009. [Online; accessed 19-November-2011].

[Ado09c] Adobe Labs. Alchemy. http://labs.adobe.com/technologies/alchemy/, 2009. [Online; accessed 06-November-2011].

[Ado11a] Adobe Systems Inc. Adobe air. http://www.adobe.com/products/air.html, 2011. [Online; accessed 28-November-2011].

[Ado11b] Adobe Systems Inc. Pc penetration — statistics — adobe flash platform runtimes. http://www.adobe.com/products/flashplatformruntimes/statistics.html, 2011. [Online; accessed 06-November-2011].

[Ado11c] Adobe Systems Inc. Stage 3d — adobe developer connection. http://www.adobe.com/devnet/flashplayer/stage3d.html, 2011. [Online; accessed 06-November-2011].

[AN11a] Tomi Aarnio and Jari Nikara. Webcl - frequently asked questions. http://webcl.nokiaresearch.com/faq.html, 2011. [Online; accessed 05-December-2011].

[AN11b] Tomi Aarnio and Jari Nikara. Webcl - gpu computing on the web. http://webcl.nokiaresearch.com/, 2011. [Online; accessed 05-December-2011].

[App11a] Apple Inc. About canvas. http://developer.apple.com/library/safari/#documentation/AudioVideo/Conceptual/HTML-canvas-guide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40010542, 2011. [Online; accessed 06-November-2011].

[App11b] Apple Inc. About the safari 3.1 update. http://support.apple.com/kb/TA25197, 2011. [Online; accessed 06-November-2011].

[App11c] Apple Inc. Safari 5.1/webkit2. http://trac.webkit.org/browser/releases/Apple/Safari%205.1/WebKit2?rev=91373, 2011. [Online; accessed 28-November-2011].

[AWW01] B Danette Allen, G Welch, and G Welch. Tracking: Beyond 15 minutes of thought. *SIGGRAPH Course Pack*, pages 27599–3175, 2001.

[Azu95] R. Azuma. A survey of augmented reality, 1995.

[Bau07] Valentin Bauer. Die neuen pforten der wahrnehmung. http://www.metaio.com/press/press-release/2007/die-neuen-pforten-der-wahrnehmung/, 2007. [Online; accessed 06-November-2011].

[Bea78] P.R. Beaudet. Rotationally invariant image operators. In *International Joint Conference on Pattern Recognition*, pages 579–583, 1978.

[Beb11] Michael Bebenita. Broadway - a javascript h.264 decoder. https://github.com/mbebenita/Broadway, 2011. [Online; accessed 11-December-2011].

[Ben10] Marco Di Benedetto. Spidergl - 3d graphics for next-generation www. http://spidergl.org/, 2010. [Online; accessed 29-November-2011].

[BETG08] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (SURF). *Computer Vision and Image Understanding*, 110(3):346 – 359, 2008. Similarity Matching in Computer Vision and Multimedia.

[Bev11] Peter Beverloo. Flexbox, web sockets, inclusion of webrtc and smooth scrolling. http://peter.sh/2011/06/flexbox-web-sockets-inclusion-of-webrtc-and-smooth-scrolling/, 2011. [Online; accessed 06-November-2011].

[Bha93] Devesh K Bhatnagar. Position trackers for head mounted display systems: A survey. Technical report, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1993.

[Bio11] BioDigital Systems. Biodigital human: Explore the body in 3d. http://www.biodigitalhuman.com/, 2011. [Online; accessed 19-November-2011].

[Bit10] Bitmanagement Software GmbH. Bs sdk - tutorials. http://www.bitmanagement.de/developer/contact/sdk-prev/doc/tutorials/index.html, 2010. [Online; accessed 06-November-2011].

[Bli10] Christopher Blizzard. a quick note on javascript engine components. http://hacks.mozilla.org/2010/03/a-quick-note-on-javascript-engine-components/, 2010. [Online; accessed 06-November-2011].

[Bov11] Andreas Bovens. Opera 11.50 released: Speed dial extensions, improved standards support, and more. http://my.opera.com/ODIN/blog/2011/06/28/opera-11-50-released-speed-dial-extensions-improved-standards-support, 2011. [Online; accessed 29-November-2011].

[Bra11] Tim Bray. Web vs. native. http://www.tbray.org/ongoing/When/201x/2011/06/14/Native-vs-Web, 2011. [Online; accessed 28-November-2011].

[Bru09] Matt Brubeck. Android 2.0 uses v8 javascript engine. http://limpet.net/mbrubeck/2009/11/06/android-v8.html, 2009. [Online; accessed 29-November-2011].

[Bus11] BusinessWire. Smartphone market grows 79.7% year over year in first quarter of 2011, according to idc. http://www.businesswire.com/news/home/20110505007011/en/Smartphone-Market-Grows-79.7-Year-Year-Quarter, 2011. [Online; accessed 04-November-2011].

[Cab11] Ricardo Cabello. Javascript 3d engine. https://github.com/mrdoob/three.js, 2011. [Online; accessed 29-November-2011].

[Cav97] Rick Cavallaro. The foxtrax hockey puck tracking system. *IEEE Computer Graphics and Applications*, 17:6–12, 1997.

[CC09] Giandomenico Caruso and Umberto Cugini. Augmented reality video see-through hmd oriented to product design assessment. In Randall Shumaker, editor, *Virtual and Mixed Reality*, volume 5622 of *Lecture Notes in Computer Science*, pages 532–541. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-02771-0_59.

[Chr08] Chromium Projects, The. Multi-process architecture. http://www.chromium.org/developers/design-documents/multi-process-architecture, 2008. [Online; accessed 29-November-2011].

[Chr10] Chromium Project. A new crankshaft for v8. http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html, 2010. [Online; accessed 06-November-2011].

[Chr11] Sean Christmann. Guimark 3 - mobile showdown. http://www.craftymind.com/guimark3/, 2011. [Online; accessed 30-December-2011].

[CLSF10] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. BRIEF: Binary Robust Independent Elementary Features. In *European Conference on Computer Vision*, September 2010.

[Cog11] James Coglan. Sylvester - vector and matrix math for javascript. http://sylvester.jcoglan.com/, 2011. [Online; accessed 16-November-2011].

[Cro01] Douglas Crockford. Javascript: The world's most misunderstood programming language. http://javascript.crockford.com/javascript.html, 2001. [Online; accessed 05-November-2011].

[Cro11] Douglas Crockford. The application/json media type for javascript object notation (json). http://tools.ietf.org/html/rfc4627, 2011. [Online; accessed 18-November-2011].

[CWHT09] D. Cheng, Y. Wang, H. Hua, and M. M. Talha. Design of an optical see-through head-mounted display with a low f-number and large field of view using a freeform prism. *ao*, 48:2655, May 2009.

[CZ10] Kevin Carle and Chris Zacharias. Introducing youtube html5 supported videos. http://youtube-global.blogspot.com/2010/01/introducing-youtube-html5-supported.html, 2010. [Online; accessed 06-November-2011].

[DB81] D.H. and Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111 – 122, 1981.

[Øde11] Ruarí Ødegaard. New opera labs release, with getusermedia and opera reader. http://my.opera.com/desktopteam/blog/2011/10/19/

new-opera-labs-release-with-getusermedia-and-opera-reader, 2011. [Online; accessed 14-November-2011].

[Doj11] Dojo Foundation, The. Unbeatable javascript tools - the dojo toolkit. http://dojotoolkit.org/, 2011. [Online; accessed 06-November-2011].

[Dou10] Brad Dougherty. Try our new html5 player! on vimeo staff blog. http://vimeo.com/blog:268, 2010. [Online; accessed 06-November-2011].

[Dru11] Tom Drummond. Toon: Tom's object-oriented numerics library. http://www.edwardrosten.com/cvd/toon.html, 2011. [Online; accessed 19-November-2011].

[EC11] Vyacheslav Egorov and Erik Corry. A game changer for interactive performance. http://blog.chromium.org/2011/11/game-changer-for-interactive.html, 2011. [Online; accessed 29-November-2011].

[ECM11] ECMA International. Ecmascript language specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, 2011. [Online; accessed 06-November-2011].

[Eic11] Brendan Eich. New javascript engine module owner. http://brendaneich.com/2011/06/new-javascript-engine-module-owner/, 2011. [Online; accessed 05-November-2011].

[Eri11a] Ericsson Labs. Beyond html5: Experiment with real-time communication in a browser. https://labs.ericsson.com/developer-community/blog/beyond-html5-experiment-real-time-communication-browser, 2011. [Online; accessed 06-November-2011].

[Eri11b] Ericsson Labs. Web real-time communication. https://labs.ericsson.com/apis/web-real-time-communication/, 2011. [Online; accessed 06-November-2011].

[Far05] Dirk Farin. *Automatic Video Segmentation Employing Object/Camera Modeling Techniques*. PhD thesis, Technische Universiteit Eindhoven, 2005.

[FB81] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24:381–395, June 1981.

[FHP98] Eric Foxlin, Michael Harrington, and George Pfeifer. Constellation: a wide-range wireless motion-tracking system for augmented reality and virtual set applications. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 371–378, New York, NY, USA, 1998. ACM.

[Fox96] Eric Foxlin. Inertial head-tracker sensor fusion by a complimentary separate-bias kalman filter. In *Proceedings of the 1996 Virtual Reality Annual International Symposium (VRAIS 96)*, VRAIS '96, pages 185–, Washington, DC, USA, 1996. IEEE Computer Society.

[fsc11] fschneider@chromium.org. Performance penalty when using integers greater 30bit. `http://code.google.com/p/v8/issues/detail?id=1758`, 2011. [Online; accessed 30-December-2011].

[Ful11] Scott M. Fulton. Build 2011: What is winrt, and is silverlight dead? `http://www.readwriteweb.com/hack/2011/09/build-2011-what-is-winrt-and-i.php`, 2011. [Online; accessed 27-November-2011].

[Gar08] Geoffrey Garen. Announcing squirrelfish. `http://www.webkit.org/blog/189/announcing-squirrelfish/`, 2008. [Online; accessed 06-November-2011].

[Gol11] Golem.de. Video: Mozilla-cto brendan eich über javascript. `http://video.golem.de/internet/5996/mozilla-cto-brendan-eich-ueber-javascript.html`, 2011. [Online; accessed 06-November-2011].

[Goo09] Google Inc. Chromium os. `http://www.chromium.org/chromium-os`, 2009. [Online; accessed 04-November-2011].

[Goo11a] Google Inc. Adobe flash player 11 - android market. `https://market.android.com/details?id=com.adobe.flashplayer&hl=en`, 2011. [Online; accessed 06-November-2011].

[Goo11b] Google Inc. Design elements - v8 javascript engine. `http://code.google.com/apis/v8/design.html`, 2011. [Online; accessed 06-November-2011].

[Goo11c] Google Inc. Google docs. `https://docs.google.com/`, 2011. [Online; accessed 27-November-2011].

[Goo11d] Google Inc. Google mail. https://mail.google.com/, 2011. [Online; accessed 27-November-2011].

[Goo11e] Google Inc. Native client sdk - google code. http://code.google.com/intl/de-DE/chrome/nativeclient/, 2011. [Online; accessed 21-November-2011].

[Goo11f] Google Inc. V8 javascript engine. http://code.google.com/p/v8/, 2011. [Online; accessed 06-November-2011].

[Goo11g] Google Inc. V8 javascript engine - smi-ops.js. http://code.google.com/p/v8/source/browse/branches/2.4/test/mjsunit/smi-ops.js?r=5659, 2011. [Online; accessed 30-December-2011].

[Goo11h] Google Inc. Webrtc. http://www.webrtc.org, 2011. [Online; accessed 06-November-2011].

[Goo11i] Google Inc. What's inside google chrome? http://www.google.com/chrome/intl/en/webmasters-faq.html#insidechrome, 2011. [Online; accessed 28-November-2011].

[Hac11] Brian Hackett. Fast and precise hybrid type inference for javascript. http://people.mozilla.org/~lmesa/ti-draft.pdf, 2011. [Online; accessed 29-November-2011].

[Ham50] Richard W. Hamming. Error detecting and error correcting codes. In *Bell System Technical Journal*, volume 29, pages 147–160, 1950.

[Ham08] Naomi Hamilton. The a-z of programming languages: Javascript. http://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/, 2008. [Online; accessed 05-November-2011].

[HBO05] Anders Henrysson, Mark Billinghurst, and Mark Ollila. Face to face collaborative ar on mobile phones. In *Proceedings of the 4th IEEE/ACM International Symposium on Mixed and Augmented Reality*, ISMAR '05, pages 80–89, Washington, DC, USA, 2005. IEEE Computer Society.

[Hei11] Ilmari Heikkinen. Jsartoolkit. https://github.com/kig/JSARToolKit/, 2011. [Online; accessed 04-November-2011].

[HF09] Steven J. Henderson and Steven Feiner. Evaluating the benefits of augmented reality for task localization in maintenance of an armored personnel carrier turret. In *Proceedings of the 2009 8th IEEE International Symposium on Mixed and Augmented Reality*, ISMAR '09, pages 135–144, Washington, DC, USA, 2009. IEEE Computer Society.

[Hic04] Ian Hickson. [whatwg] what open mailing list announcement. `http://lists.whatwg.org/htdig.cgi/whatwg-whatwg.org/2004-June/000005.html`, 2004. [Online; accessed 06-November-2011].

[Hix04] Ian Hixie. Extending html. `http://ln.hixie.ch/?start=1089635050&count=1`, 2004. [Online; accessed 06-November-2011].

[Hol10] Thom Holwerda. Internet explorer 9 is not cheating on sunspider benchmark. `http://www.osnews.com/story/24044/Internet_Explorer_9_Is_Not_Cheating_on_SunSpider_Benchmark/`, 2010. [Online; accessed 06-November-2011].

[HS88] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988.

[htm11] html5.org. Html5 tracker. `http://html5.org/tools/web-apps-tracker?from=5944&to=5945`, 2011. [Online; accessed 06-November-2011].

[HZ04] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[IET06] IETF. The base16, base32, and base64 data encodings. `http://tools.ietf.org/html/rfc4648`, 2006. [Online; accessed 07-December-2011].

[IG08] IEEE and "The Open Group". The open group base specifications issue 7 - clock_getres. `http://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_gettime.html`, 2008. [Online; accessed 04-December-2011].

[Ima11] Imagination Computer Services GesmbH. Augmented reality for flash. `http://www.imagination.at/en/?Products:Augmented_Reality_for_Flash`, 2011. [Online; accessed 06-November-2011].

[IN211] IN2AR. In2ar - adobe flash based augmented reality. `http://www.in2ar.com/`, 2011. [Online; accessed 06-November-2011].

[Jam07] Sudha (Su) Jamthe. Sv web builders event - world premier of opera with builtin video support. http://coolastory.blogspot.com/2007/03/sv-web-builders-event-world-premier-of.html, 2007. [Online; accessed 06-November-2011].

[jav11] java.net. Java 3d. http://java3d.java.net/, 2011. [Online; accessed 06-November-2011].

[Job10] Steve Jobs. Thoughts on flash. http://www.apple.com/hotnews/thoughts-on-flash/, 2010. [Online; accessed 06-November-2011].

[Joh07] Tim Johansson. Taking the canvas to another dimension. http://my.opera.com/timjoh/blog/2007/11/13/taking-the-canvas-to-another-dimension, 2007. [Online; accessed 06-November-2011].

[Jon11] Brandon Jones. Javascript matrix library for high performance webgl apps. https://github.com/toji/gl-matrix, 2011. [Online; accessed 16-November-2011].

[Joy11a] Joyent Inc. joyent/node. https://github.com/joyent/node, 2011. [Online; accessed 06-November-2011].

[Joy11b] Joyent Inc. node.js - evented i/o for v8 javascript. http://nodejs.org, 2011. [Online; accessed 06-November-2011].

[jQu10] jQuery Project, The. jquery: The write less, do more, javascript library. http://jquery.com/, 2010. [Online; accessed 06-November-2011].

[Kat04] Hirokazu Kato. Artoolkit. http://www.hitl.washington.edu/artoolkit/, 2004. [Online; accessed 04-November-2011].

[Kay11] Lindsay Kay. Scenejs - webgl scene graph library. http://scenejs.org/, 2011. [Online; accessed 18-November-2011].

[KB99] Hirokazu Kato and Mark Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality*, pages 85–, Washington, DC, USA, 1999. IEEE Computer Society.

[Khr09a] Khronos Group. Khronos details webgl initiative to bring hardware-accelerated 3d graphics to the internet. http://www.khronos.org/news/press/2009/08, 2009. [Online; accessed 06-November-2011].

[Khr09b] Khronos Group. The opengl es shading language. http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf, 2009. [Online; accessed 05-December-2011].

[Khr11a] Khronos Group. Collada - 3d asset exchange schema. http://www.khronos.org/collada/, 2011. [Online; accessed 19-November-2011].

[Khr11b] Khronos Group. Differences between webgl and opengl es 2.0. http://www.khronos.org/registry/webgl/specs/latest/#6, 2011. [Online; accessed 06-November-2011].

[Khr11c] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, 2011. [Online; accessed 05-December-2011].

[Khr11d] Khronos Group. User contributions - webgl public wiki. http://www.khronos.org/webgl/wiki/User_Contributions#Frameworks, 2011. [Online; accessed 18-November-2011].

[Khr11e] Khronos Group. Webcl - heterogeneous parallel computing in html5 web browsers. http://www.khronos.org/webcl/, 2011. [Online; accessed 05-December-2011].

[Khr11f] Khronos Group. Webgl specification - version 1.0, 10 february 2011. https://www.khronos.org/registry/webgl/specs/1.0/, 2011. [Online; accessed 06-November-2011].

[KI07] Anne van Kesteren and Opera Inc. [whatwg] ¡video¿ element proposal. http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2007-February/009702.html, 2007. [Online; accessed 06-November-2011].

[KM07] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, ISMAR '07, pages 1–10, Washington, DC, USA, 2007. IEEE Computer Society.

[KP11] Anne van Kesteren and Simon Pieters. Html5 differences from html4. http://dev.w3.org/html5/html4-differences/, 2011. [Online; accessed 06-November-2011].

[Kra10] Tom Krazit. Google i/o keynote day 2: Android day (live blog). http://news.cnet.com/8301-30684_3-20005447-265.html, 2010. [Online; accessed 29-November-2011].

[Kri08] Paul Krill. Javascript creator ponders past, future. http://www.infoworld.com/d/developer-world/javascript-creator-ponders-past-future-704, 2008. [Online; accessed 05-November-2011].

[Lak07] Pratap Lakshman. Jscript deviations from es3. http://wiki.ecmascript.org/lib/exe/fetch.php?id=resources:resources&cache=cache&media=resources:jscriptdeviationsfromes3.pdf, 2007. [Online; accessed 29-November-2011].

[Lat11] Chris Lattner. Llvm and clang: Advancing compiler technology. http://llvm.org/pubs/2011-02-FOSDEM-LLVMAndClang.pdf, 2011. [Online; accessed 28-November-2011].

[Law09] Eric Lawrence. Q&a: 64-bit internet explorer. http://blogs.msdn.com/b/ieinternals/archive/2009/05/29/q-a-64-bit-internet-explorer.aspx, 2009. [Online; accessed 28-November-2011].

[lay11] layar. Augmented reality browser: Layar. http://www.layar.com/, 2011. [Online; accessed 27-November-2011].

[LCS11] Stefan Leutenegger, Margarita Chli, and Roland Siegwart. Brisk: Binary robust invariant scalable keypoints. In *Proceedings of the IEEE International Conference on Computer Vision*, ICCV '11, 2011.

[Lew95] J. P. Lewis. Fast normalized cross-correlation. In *Vision Interface*, volume 1, pages 120–123. Citeseer, 1995.

[LF06] Vincent Lepetit and Pascal Fua. Keypoint recognition using randomized trees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28:1465–1479, September 2006.

[LHM00] Chien-Ping Lu, Gregory D. Hager, and Eric Mjolsness. Fast and globally convergent pose estimation from video images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22:610–622, June 2000.

[Lin98] Tony Lindeberg. Feature detection with automatic scale selection. *International Journal of Computer Vision*, 30:79–116, 1998.

[Lin09] Jens Lindström. Carakan. `http://my.opera.com/core/blog/2009/02/04/carakan`, 2009. [Online; accessed 06-November-2011].

[Low99] David G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, pages 1150–, Washington, DC, USA, 1999. IEEE Computer Society.

[Low04] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[Man09] David Mandelin. an overview of tracemonkey. `http://hacks.mozilla.org/2009/07/tracemonkey-overview/`, 2009. [Online; accessed 06-November-2011].

[MC07] Eric Miraglia and Douglas Crockford. Yui theater: Douglas crockford, the javascript programming language. `http://www.yuiblog.com/blog/2007/01/24/video-crockford-tjpl/`, 2007. [Online; accessed 05-November-2011].

[met11a] metaio GmbH. By 2014: Augmented reality will be on every smartphone. `http://www.metaio.com/press/press-release/2011/by-2014-augmented-reality-will-be-on-every-smartphone/`, 2011. [Online; accessed 27-November-2011].

[met11b] metaio GmbH. metaio — mobile sdk — augmented reality 3d. `http://www.metaio.com/software/mobile-sdk/`, 2011. [Online; accessed 04-November-2011].

[Met11c] Metaio Inc. Metaio viewer. `http://www.metaio.com/software/viewer/`, 2011. [Online; accessed 04-November-2011].

[Mic96a] Microsoft Corp. Creating activex components in c++. `http://msdn.microsoft.com/en-us/library/ms974283.aspx`, 1996. [Online; accessed 06-November-2011].

[Mic96b] Microsoft Corp. Microsoft announces activex technologies. `http://www.microsoft.com/presspass/press/1996/mar96/activxpr.mspx`, 1996. [Online; accessed 06-November-2011].

[Mic10a] Microsoft Corp. Announcing silverlight for symbian - rtm. http://blogs.msdn.com/b/slsymbian/archive/2010/07/06/announcing-silverlight-for-symbian-rtm.aspx, 2010. [Online; accessed 06-November-2011].

[Mic10b] Microsoft Corp. The browser choice screen for europe: What to expect, when to expect it. http://blogs.technet.com/b/microsoft_on_the_issues/archive/2010/02/19/the-browser-choice-screen-for-europe-what-to-expect-when-to-expect-it.aspx, 2010. [Online; accessed 06-November-2011].

[Mic10c] Microsoft Corp. The new javascript engine in internet explorer 9. http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx, 2010. [Online; accessed 06-November-2011].

[Mic11a] Microsoft Corp. 3-d graphics overview. http://msdn.microsoft.com/en-us/library/gg197424%28v=XNAGameStudio.35%29.aspx, 2011. [Online; accessed 06-November-2011].

[Mic11b] Microsoft Corp. Authenticode. http://technet.microsoft.com/en-us/library/cc750035.aspx, 2011. [Online; accessed 06-November-2011].

[Mic11c] Microsoft Corp. A history of internet explorer. http://windows.microsoft.com/en-US/internet-explorer/products/history, 2011. [Online; accessed 05-November-2011].

[Mic11d] Microsoft Corp. Internet explorer 6 countdown — death to ie 6. http://www.ie6countdown.com/, 2011. [Online; accessed 28-November-2011].

[Mic11e] Microsoft Corp. Internet explorer 9 guide for developers. http://msdn.microsoft.com/en-us/ie/hh410106#_HTML5_canvas, 2011. [Online; accessed 06-November-2011].

[Mic11f] Microsoft Corp. Jscript (ecmascript3). http://msdn.microsoft.com/en-us/library/hbxc2t98%28VS.85%29.aspx, 2011. [Online; accessed 06-November-2011].

[Mic11g] Microsoft Corp. Microsoft office web apps. http://office.microsoft.com/en-us/web-apps/, 2011. [Online; accessed 27-November-2011].

[Mic11h] Microsoft Corp. Silverlight overview. http://msdn.microsoft.com/en-us/library/bb404700%28v=vs.95%29.aspx, 2011. [Online; accessed 06-November-2011].

[Mic11i] Microsoft Corp. Webgl considered harmful. http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx, 2011. [Online; accessed 06-November-2011].

[Mic11j] Microsoft MSDN. Understanding user-agent strings. http://msdn.microsoft.com/en-us/library/ms537503%28v=vs.85%29.aspx, 2011. [Online; accessed 29-November-2011].

[MK94] Paul Milgram and Fumio Kishino. A Taxonomy of Mixed Reality Visual Displays. *IEICE Transactions on Information Systems*, E77-D(12), December 1994.

[Möl11a] Erik Möller. Emberwind - github. https://github.com/operasoftware/Emberwind, 2011. [Online; accessed 06-November-2011].

[Möl11b] Erik Möller. Emberwind reborn in html5. http://my.opera.com/emoller/blog/index.dml/tag/emberwind, 2011. [Online; accessed 06-November-2011].

[Mon11] Gautier de Montmollin. The transparent language popularity index. http://lang-index.sourceforge.net/, 2011. [Online; accessed 05-November-2011].

[Moz11a] Mozilla Developer Network. Gecko. https://developer.mozilla.org/en/Gecko, 2011. [Online; accessed 29-November-2011].

[Moz11b] Mozilla Developer Network. Javascript. https://developer.mozilla.org/en/JavaScript, 2011. [Online; accessed 29-November-2011].

[Moz11c] Mozilla Developer Network. mozilla-central: js/src/jsarray.cpp. https://hg.mozilla.org/mozilla-central/file/4b71b1e9cc0c/js/src/jsarray.cpp, 2011. [Online; accessed 13-December-2011].

[Moz11d] Mozilla Developer Network. Nanojit. https://developer.mozilla.org/En/Nanojit, 2011. [Online; accessed 06-November-2011].

[Moz11e] Mozilla Developer Network. Remixing reality — demo studio — mozilla developer network. https://developer.mozilla.org/en-US/demos/detail/remixing-reality, 2011. [Online; accessed 06-November-2011].

[Moz11f] Mozilla Developer Network. Spidermonkey - mdn. https://developer.mozilla.org/en/SpiderMonkey, 2011. [Online; accessed 28-November-2011].

[Moz11g] Mozilla Developer Network. Spidermonkey internals - mdn. https://developer.mozilla.org/En/SpiderMonkey/Internals, 2011. [Online; accessed 13-December-2011].

[Moz11h] Mozilla Developer Network. Tamarin - mdn. https://developer.mozilla.org/en/Tamarin, 2011. [Online; accessed 28-November-2011].

[Moz11i] Mozilla Foundation. Boot to gecko. https://wiki.mozilla.org/B2G, 2011. [Online; accessed 04-November-2011].

[Moz11j] Mozilla Foundation. Modern jit compiler for javascript (ionmonkey). https://wiki.mozilla.org/Platform/Features/IonMonkey, 2011. [Online; accessed 06-November-2011].

[Moz11k] Mozilla Foundation. mozilla/pdf.js. https://github.com/mozilla/pdf.js, 2011. [Online; accessed 06-November-2011].

[Moz11l] MozillaWiki. Pdf.js - mozillawiki. https://wiki.mozilla.org/PDF.js, 2011. [Online; accessed 06-November-2011].

[MS02] Krystian Mikolajczyk and Cordelia Schmid. An affine invariant interest point detector. In Anders Heyden, Gunnar Sparr, Mads Nielsen, and Peter Johansen, editors, *Computer Vision - ECCV 2002*, volume 2350 of *Lecture Notes in Computer Science*, pages 128–142. Springer, Berlin / Heidelberg, 2002.

[Nar11] Anant Narayanan. What's next: Rainbow and webrtc. https://mozillalabs.com/rainbow/2011/08/04/whats-next-rainbow-and-webrtc/, 2011. [Online; accessed 06-December-2011].

[Net95] Netscape Inc. Netscape and sun announce javascript, the open, cross-platform object scripting language for enterprise networks and the internet. http://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html, 1995. [Online; accessed 05-November-2011].

[Ngu11] Tom Nguyen. Updates from the lab. http://blogs.adobe.com/flashplayer/2011/09/updates-from-the-lab.html, 2011. [Online; accessed 19-November-2011].

[NY99] Ulrich Neumann and Suya You. Natural feature tracking for augmented reality. *IEEE Transactions on Multimedia*, 1(1):53–64, 1999.

[nya11] nyatla. Nyartoolkit - artoolkit class library for java. `http://nyatla.jp/nyartoolkit/wiki/index.php?NyARToolkit%20for%20Java.en`, 2011. [Online; accessed 06-November-2011].

[Ope97] Open Group, The. The single unix specification, version 2 - time.h - time types. `http://pubs.opengroup.org/onlinepubs/7908799/xsh/time.h.html`, 1997. [Online; accessed 04-December-2011].

[Ope06] Opera Software ASA. Changelog for opera 9.0 for windows. `http://www.opera.com/docs/changelogs/windows/900/`, 2006. [Online; accessed 06-November-2011].

[Ope11] Opera Software ASA. Ecmascript support in opera presto 2.8. `http://www.opera.com/docs/specs/js/`, 2011. [Online; accessed 06-November-2011].

[Ora10] Oracle Corp. Code samples and apps applets. `http://java.sun.com/applets/`, 2010. [Online; accessed 06-November-2011].

[Ora11] Oracle Corp. Java se desktop technologies. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-140239.html`, 2011. [Online; accessed 06-November-2011].

[PE11] Mika Palmu and Philippe Elsass. Flashdevelop is a free and open source (mit license) source code editor. `http://www.flashdevelop.org`, 2011. [Online; accessed 19-November-2011].

[Pit11] Chris Pitchin. Overview of pdf.js guts. `http://blog.mozilla.com/cjones/2011/06/15/overview-of-pdf-js-guts/`, 2011. [Online; accessed 06-November-2011].

[Pro10] Prototype Core Team. Prototype javascript framework. `http://prototypejs.org/`, 2010. [Online; accessed 06-November-2011].

[Qua11a] Qualcomm Austria Research Center GmbH. Ar sdk — qualcomm augmented reality. `https://ar.qualcomm.at/qdevnet/`, 2011. [Online; accessed 04-November-2011].

[Qua11b] Qualcomm Incorporated. Augmented reality sdk. `https://developer.qualcomm.com/develop/mobile-technologies/augmented-reality`, 2011. [Online; accessed 27-November-2011].

[RBBS06] Bernhard Reitinger, Alexander Bornik, Reinhard Beichel, and Dieter Schmalstieg. Liver surgery planning using virtual reality. *IEEE Comput. Graph. Appl.*, 26:36–47, November 2006.

[RD06a] Gerhard Reitmayr and Tom W. Drummond. Going out: Robust modelbased tracking for outdoor augmented reality. In *Proceedings of 5th IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR*, pages 109–118, 2006.

[RD06b] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European Conference on Computer Vision*, volume 1, pages 430–443, May 2006.

[RF00] Jannick P. Rolland and Henry Fuchs. Optical versus video see-through head-mounted displays in medical visualization. *Presence: Teleoper. Virtual Environ.*, 9:287–309, June 2000.

[Ric11] Tibbett Rich. Native webcam support and orientation events - technology preview. `http://my.opera.com/core/blog/2011/03/23/webcam-orientation-preview`, 2011. [Online; accessed 06-November-2011].

[Riv09] Samuel Asher Rivello. Augmented reality using a webcam and flash. `http://www.adobe.com/devnet/flash/articles/augmented_reality.html`, 2009. [Online; accessed 06-November-2011].

[Ros11] Edward Rosten. Fast corner detection. `http://www.edwardrosten.com/work/fast.html`, 2011. [Online; accessed 14-November-2011].

[Rud05] Ruderman Ruderman. What's new in firefox 1.5 (comprehensive). `http://www.squarefree.com/burningedge/releases/1.5-comprehensive.html`, 2005. [Online; accessed 06-November-2011].

[Sam11] Samsung Electronics Corp. Samsung's webcl prototype for webkit. `http://code.google.com/p/webcl/`, 2011. [Online; accessed 05-December-2011].

[Saq11] Saqoosha. Flartoolkit. http://www.libspark.org/wiki/saqoosha/FLARToolKit/en, 2011. [Online; accessed 06-November-2011].

[SB02] Gilles Simon and Marie-Odile Berger. Pose estimation for planar structures. *IEEE Comput. Graph. Appl.*, 22:46–53, November 2002.

[Sch10] René Schulte. Slartoolkit - silverlight and windows phone augmented. http://slartoolkit.codeplex.com/, 2010. [Online; accessed 06-November-2011].

[Sta08] Maciej Stachowiak. Introducing squirrelfish extreme. http://www.webkit.org/blog/214/introducing-squirrelfish-extreme/, 2008. [Online; accessed 06-November-2011].

[Sta11] StatOWL. Rich internet application market share. http://www.statowl.com/custom_ria_market_penetration.php, 2011. [Online; accessed 06-November-2011].

[Sun96] Sun Microsystems. Javasoft ships java 1.0. http://web.archive.org/web/20080205101616/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml, 1996. [Online; accessed 06-November-2011].

[Syn11] Dionysios G. Synodinos. Ionmonkey: Mozilla's new javascript jit compiler. http://www.infoq.com/news/2011/05/ionmonkey, 2011. [Online; accessed 06-November-2011].

[Tec10] TechCrunch. Sergey brin: Native apps and web apps will converge in the not-too-distant future. http://techcrunch.com/2010/05/19/chrome-os-versus-android/, 2010. [Online; accessed 28-November-2011].

[TLF11] Tomasz Trzcinski, Vincent Lepetit, and Pascal Fua. Thick boundaries in binary space and their influence on nearest-neighbor search. Technical report, Swiss Federal Institute of Technology, Lausanne (EPFL), 2011.

[Tom10] Predrag Tomasevic. Versatile webcam c# library. http://www.codeproject.com/KB/miscctrl/webcam_c_sharp.aspx, 2010. [Online; accessed 06-November-2011].

[Tot11a] Total Immersion. Free augmented reality software : D'fusion studio by total immersion. http://www.t-immersion.com/products/dfusion-suite/dfusion-studio, 2011. [Online; accessed 27-November-2011].

[Tot11b] Serkan Toto. Moverio: Epson announces world's first see-through 3d head-mounted display. http://techcrunch.com/2011/11/09/moverio-epson-announces-worlds-first-see-through-3d-head-mounted-display/, 2011. [Online; accessed 28-November-2011].

[Tri11] Andrew Trice. Why cross platform mobile development? http://www.tricedesigns.com/2011/10/20/why-cross-platform-mobile-development/, 2011. [Online; accessed 28-November-2011].

[Vid11] VideoLAN Organization. Video lan - vlc - free multimedia solutions for all os! http://www.videolan.org/, 2011. [Online; accessed 14-November-2011].

[Vuk07] Vladimir Vukicevic. Canvas 3d: Gl power, web-style. http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/, 2007. [Online; accessed 06-November-2011].

[W3C99] W3C. Html 4.01 specification. http://www.w3.org/TR/html4/, 1999. [Online; accessed 06-November-2011].

[W3C11a] W3C. Html5 - w3c working draft. http://www.w3.org/TR/html5/, 2011. [Online; accessed 27-November-2011].

[W3C11b] W3C. Pubstatus - audio wg wiki. http://www.w3.org/2011/audio/wiki/PubStatus, 2011. [Online; accessed 06-November-2011].

[Wan07] Thomas Wang. Integer hash function. http://www.concentric.net/~ttwang/tech/inthash.htm, 2007. [Online; accessed 29-December-2011].

[Way10] Peter Wayner. 7 programming languages on the rise. http://www.infoworld.com/d/developer-world/7-programming-languages-the-rise-620?page=0,2, 2010. [Online; accessed 05-November-2011].

[Web11a] WebKit Project. Javascript. http://www.webkit.org/projects/javascript/index.html, 2011. [Online; accessed 29-November-2011].

[Web11b] WebKit Project. Sunspider javascript benchmark. http://www.webkit.org/perf/sunspider/sunspider.html, 2011. [Online; accessed 06-November-2011].

[WF02] G Welch and E Foxlin. Motion tracking: no silver bullet, but a respectable arsenal. *IEEE Computer Graphics and Applications*, 22(6):24–38, 2002.

[Wha05] Wharton School of the University of Pennsylvania. Browser wars: Will firefox burn explorer? `http://knowledge.wharton.upenn.edu/article.cfm?articleid=1162`, 2005. [Online; accessed 29-November-2011].

[WHA10] WHATWG. The canvas element - html standard. `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html`, 2010. [Online; accessed 15-November-2011].

[WHA11a] WHATWG. The video element. `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-video-element.html#the-video-element`, 2011. [Online; accessed 06-November-2011].

[WHA11b] WHATWG. Web real-time communication apis. `http://www.whatwg.org/specs/web-apps/current-work/webrtc.html#obtaining-local-multimedia-content`, 2011. [Online; accessed 06-November-2011].

[WHA11c] WHATWG. Web workers - html standard. `http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html`, 2011. [Online; accessed 31-December-2011].

[Wik11] Wikitude GmbH. Wikitude world browser. `http://www.wikitude.com`, 2011. [Online; accessed 27-November-2011].

[Wil11] Brett Wilson. Reading canvas-data - native-client-discuss. `https://groups.google.com/group/native-client-discuss/msg/70f23fbe356b6d7d`, 2011. [Online; accessed 21-November-2011].

[win11a] wingolog. value representation in javascript implementations. `http://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations`, 2011. [Online; accessed 13-December-2011].

[Win11b] Danny Winokur. Flash to focus on pc browsing and mobile apps; adobe to more aggressively contribute to html5. `http://blogs.adobe.com/conversations/2011/11/flash-focus.html`, 2011. [Online; accessed 29-November-2011].

[WN11a] Amos Wenger and Jens Nockert. jsmad - github. `https://github.com/nddrylliog/jsmad`, 2011. [Online; accessed 06-November-2011].

[WN11b] Amos Wenger and Jens Nockert. Jsmad - javascript mpeg audio decoder. `http://jsmad.org/`, 2011. [Online; accessed 06-November-2011].

[WR01] Luke Wroblewski and Esa M. Rantanen. Design considerations for webbased applications. In *In: Proceedings of the human factors and ergonomics society, 45th annual meeting*, 2001.

[WRM+10] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Real-time detection and tracking for augmented reality on mobile phones. *Visualization and Computer Graphics, IEEE Transactions on*, 16(3):355 –368, may-june 2010.

[WS07] D Wagner and D Schmalstieg. *ARToolKitPlus for Pose Tracking on Mobile Devices ARToolKit*, pages 139–146. Citeseer, 2007.

[WW11] W3C and Zhiheng Wang. Navigation timing. `http://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html`, 2011. [Online; accessed 02-December-2011].

[XKM09] Changhai Xu, Benjamin Kuipers, and Aniket Murarka. 3d pose estimation for planes. In *ICCV Workshop on 3D Representation for Recognition (3dRR-09)*, 2009.

[YNA99] Suya You, Ulrich Neumann, and Ronald Azuma. Hybrid inertial and vision tracking for augmented reality registration. In *Proceedings of the IEEE Virtual Reality*, VR '99, pages 260–, Washington, DC, USA, 1999. IEEE Computer Society.