Natural Feature Tracking in JavaScript

Christoph Oberhofer, Jens Grubert, Gerhard Reitmayr

Institute for Computer Graphics and Vision, University of Technology Graz

ABSTRACT

We present an efficient natural feature tracking pipeline solely implemented in JavaScript. It is embedded in a web technology-based Augmented Reality system running plugin-free in web browsers. The evaluation shows that real-time framerates on desktop computers and while on smartphones interactive framerates are achieved.

KEYWORDS: Augmented Reality, Natural Feature Tracking, JavaScript, HTML5, WebGL, Web Technology.

INDEX TERMS: H.5.1 [Multimedia Information Systems]: Artificial, augmented, and virtual realities; H.3.5 [Online Information Services]: Web-based services

1 INTRODUCTION

With ubiquitous internet access and an ever increasing number of smartphone and tablet users web applications promise to be a viable alternative for traditional PC and native mobile applications [2]. However, performance demanding applications are still widely developed in native code resulting in high developing costs due to the variety of platforms to support [2]. In particular computer vision algorithms needed for realizing markerless Augmented Reality (AR) applications have been deemed too computational intensive for implementing them directly in the web technology stack. While the marker based AR system ARToolkit has been demonstrated to work in web browsers [3], natural feature tracking (NFT) algorithms are usually implemented using plugin technologies like Adobe Flash. We are the first to demonstrate a plugin-free NFT AR pipeline in the web browser. It is based on a JavaScript implementation of state-of-the art computer vision techniques. This NFT pipeline is combined with real-time 3D rendering in WebGL, video playback, and basic sensor access [5].

2 WEB TECHNOLOGY BASED AUGMENTED REALITY PIPELINE

In order to achieve a plugin-free AR experience in a web browser, all stages of the pipeline (video access, detection and tracking, rendering) have to be representable in HTML5 [5], JavaScript and WebGL. Our pipeline builds on the getUserMedia interface of the MediaStream API [5] for live camera access, respectively on the video element for video playback, combined with the canvas element for pixel access, followed by pose detection and tracking, and finally rendering of the augmentation in WebGL.

At the core of the web technology-based AR pipeline is a two-stage detection and tracking approach, similar to the work by Wagner et al.[6]. The detection pipeline starts with FAST interest point detection at multiple image levels[4]. A 128-bit version of the BRIEF descriptor is used for keypoint description [1].

ch.oberhofer@gmail.com, [grubert|reitmayr]@icg.tugraz.at

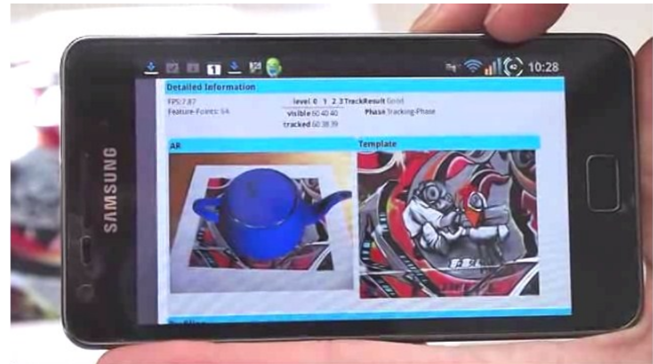

Figure 1. AR pipeline inside Firefox on a Samsung Galaxy SII.

It is internally represented as four 64 bit doubles that are converted to integers for binary operations by the JavaScript engine.

We use brute-force nearest neighbour search using the Hamming distance as distance measure for matching, followed by RANSAC-based outlier removal. The homography is directly computed from four point correspondences. The camera pose is estimated from the homography and the camera intrinsics.

The tracking phase employs a patch-tracker using Normalized Cross Correlation. We implemented linear algebra algorithms for fixed sized rather than dynamically sized matrices, resulting in significant speedups. Finally, the graphics pipeline renders the objects according to the current pose. Examples running on a web browser on a mobile phone and on the PC can be seen in Figure 1.

3 EVALUATION

We evaluated the natural feature tracking pipeline on a desktop PC (Core 2 Duo 3 GHZ, 4GB RAM) and on a Samsung Galaxy SII smartphone. We tested Mozilla Firefox 6, Chromium 15, Opera 11.51 on the desktop. Mozilla Firefox 6 and Android 2.3.3 web browser based on Webkit were evaluated on the smartphone. The test set consists of static image data (320x240 px) as frame dropping might occur during access to the live camera stream.

For the detection phase, timings were measured on seven different parts: integral image (used to speedup BRIEF computation), FAST corner detection (250 keypoints), descriptor computation (128 bit, 5 image levels in the template), initial descriptor matching, outlier removal (30 inlier required, max. 300 iterations), and pose estimation. Results for the PC can be seen in Figure 2 top (averaged from 60 iterations). On the PC Firefox 6 was fastest (93 ms per frame), Opera slowest (336 ms). The matching stage is the most expensive and consumes between 50% (Firefox) and 87% (Opera) of the entire processing time. All other stages are evenly distributed among the tested browsers. 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. For all but the corner detection phase the ratio is between 3.4 (integral image) to 6.4 (outlier removal). FAST corner detection is more than 20 times faster on the PC.

For the tracking phase, the times are split up into keyframe creation, in which an image pyramid is calculated, tracking of an

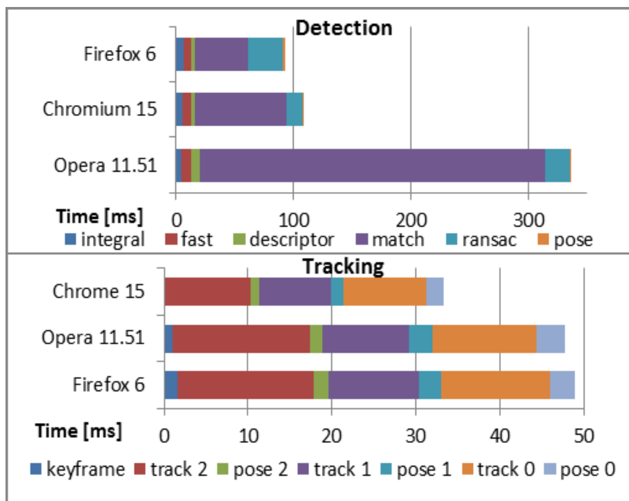

Figure 2. Detection (top) and tracking (bottom) timings on a PC.

image level (3 levels: 120, 100, 70 keypoints, 8x8 px patch size, search radius: 2, 2, 4 px), and pose optimization on that level. The timings overview can be seen in Figure 2, bottom (averaged from 100 iterations). Chromium browser was fastest, processing a frame in an average time of 33 ms, followed by Opera and Firefox which both fall behind 15 ms. The CPU time is mostly spent (80%) in the tracking parts. In the tracking-phase on the mobile (2 levels: 60, 40 keypoints, 8x8 px patch size, search radius: 2, 3 px) Firefox 6 runs at 90 ms, which is about five times slower than on the Desktop. Each part of the profile has the same slowdown ratio compared to the PC.

Furthermore, we compared the JavaScript implementation of the detection part (in Firefox 6) with Adobe Alchemy and Google Native Client. Both frameworks accept C/C++ code as input to be compiled to proprietary binaries. The evaluation was carried out using the same configuration as described in Section 3. The implementation in Google Native Client shows a 4x speedup over JavaScript and a 2x speedup over Adobe Alchemy. While the matching time for Alchemy and JavaScript are almost the same JavaScript is slower overall. There is a major difference in the outlier removal stage (29 ms for JavaScript, less than 1 ms for Alchemy and Native Client).

4 Discussion

Once a target is detected it can be tracked in real-time on the PC (around 30 fps) and at interactive framerates (10 fps) inside a mobile web browser. However, the detection phase is not yet real-time capable due to the high costs of matching the BRIEF descriptors. JavaScript does not offer a construct representing binary data so we had to use the Number data type, the only data type on which binary operations can be executed. Furthermore an explicit differentiation between fixed and floating point numbers is not possible, adding another potential bottleneck. We see that benefits of using binary operations in native code do not show up in JavaScript. This leaves the question whether employing more robust descriptors in addition to acceleration structures for matching might result in better performance in JavaScript. 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 20 times slower on the mobile. We assume that this is due to the large complexity of an automatically generated decision tree with over 8000 lines of code and we plan to investigate a simpler version of the detector.

In contrast to the initialization phase, when most of the processing time is spent in binary operations, the performance during the tracking phase heavily depends on array-access speed. JavaScript arrays can be sparse or dense and are not type-bound. Hence the speed is not always predictable. Typed arrays, introduced along with WebGL, promise faster access than the usual JavaScript array as they are natively implemented for different number-types like float or int32. However, during evaluations not presented here we encountered a slow-down of almost 50% in the tracking phase using typed-arrays. This might indicate that they still have the potential of being optimized in JavaScript engines or that algorithms have to be more carefully adapted to employ them more efficiently.

Another area that could improve the overall performance is the access to the video stream itself. Because there is no interface available to directly capture the data-stream yet all pixels from the video element have to be copied to the canvas element introducing another slowdown at the beginning of the pipeline.

The power of JavaScript engines, while not reaching native code level, is steadily increasing as they are one new front line in the browser wars [3]. We believe that the gap between JavaScript and native code can be further lowered if support for low-level data types is further enhanced – as already intended with typed arrays. Alternatively WebCL might become a viable alternative for performing computationally intensive tasks directly in the web browser without the need for plugins.

5 Conclusion

We presented a natural feature tracking pipeline implemented solely in JavaScript that integrates well with other technologies into a plugin-free web technology based Augmented Reality pipeline. The system is capable of running at 30 fps on desktop browsers and 10 fps in web browsers on modern mobile phones. Even though the performance is still significantly lower when compared to other approaches like Alchemy or Google Native Client we believe that delivering cross-platform Augmented Reality solutions through a combination of web technologies like JavaScript, HTML5 and WebGL can be a viable alternative to plugin-based, and in the future maybe to native implementations.

However, for this vision to become a reality, device access must be standardized and feature segmentation must be limited across browsers.

References

[1] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. BRIEF: Binary Robust Independent Elementary Features. In *ECCV 2010*, pages 778-792.
[2] A. Charland and B. Leroux. Mobile application development: web vs. native. In *Commun. ACM*, 54, pages 49-53, May 2011.
[3] I. Heikkine. Remixing Reality. https://demos.mozilla.org/en/#remixingreality. Last visited on 13.09.2011.
[4] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *ECCV 2006*, pages 430-443.
[5] W3C. HTML5 W3C Working Draft. http://www.w3.org/TR/html5/. Last visited on 13.09.2011.
[6] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Real-Time Detection and Tracking for Augmented Reality on Mobile Phones. In *IEEE TVCG*, 99(1), pages 355-368, November 2010.