IEEE Third International Conference on Data Stream Mining & Processing
August 21-25, 2020, Lviv, Ukraine

Augmented Reality in Web: Results and Challenges

Ruslan Timchenko
It-Jim
Kharkiv, Ukraine
timchenkoruslan97@gmail.com

Oleksiy Grechnyev
It-Jim
Kharkiv, Ukraine
shrike4625@gmail.com

Sergiy Skuratovskyi
It-Jim
Kharkiv, Ukraine
ssnake@it-jim.com

Yurii Chyrka
It-Jim
Kharkiv, Ukraine
yurii.chyrka@it-jim.com

Ievgen Gorovyi
It-Jim
Kharkiv, Ukraine
ceo@it-jim.com

***Abstract*—The paper presents basic concepts of augmented reality applications and challenges in building them in the web. We describe the technical and algorithmic stack required to develop, implement and deploy the augmented reality application. Theoretical concepts behind marker detection and tracking are discussed. Two different pipelines are implemented: server-based with algorithms execution in the cloud and completely front-end solution that runs on a user device. We show advantages and disadvantages of each approach and analyze experimental results as well.**

Keywords—augmented reality, visual tracking, image marker, webAR

I. Introduction

Augmented Reality (AR) is the technology that connects our real life with a digital world. Extending reality is possible by overlying layers with virtual objects over the screen of user devices [1]. Different kind of information can be augmented: text, video, images, audio and both static and dynamic 3d models [2]

Since most of the information comes through the human visual system, AR is becoming more and more widespread. Its strength is already demonstrated in a variety of areas: advertising [3], game industry [1, 4], education [5], medicine [6], entertainment [4, 7-9].

Two important points are taken into consideration during the development of the AR system: technical solutions and hardware platforms used. Several typical computer vision problems are usually solved during the construction of the AR framework. In particular, object detection and recognition (planar markers or arbitrary objects) [2], content-based image retrieval [10] for visual search, simultaneous localization and mapping (SLAM) for 3D object detection [11], markerless tracking and multi-player AR applications [12]. The second point is where to utilize AR technology, i.e. what device to use: mobile phone or tablet, AR glasses, desktop.

There are plenty of different mobile AR systems like ARKit, ARCore, Vuforia, EasyAR, etc. Most of them require installation of the mobile application for AR experience. In the paper, we describe an alternative way to use AR technology: augmented reality in a browser, or web AR [13, 14]. Indeed, the obvious advantage of AR in a mobile browser is instant immersion without need to install any mobile applications.

The paper is structured as follows. The problem of planar marker recognition for AR is discussed in section II. In particular, we consider a typical keypoint-based detection algorithm and analyze various local feature descriptors. In addition, we introduce a hybrid tracking approach which combines sparse optical flow [15] and template-based tracker [2]. The created algorithmic pipeline and its deployment into web AR application are described in section III. We consider two separate architectures: server-based one with algorithm execution in the cloud and pure front-end solution that runs on a user device. We also analyze their strong and weak sides. Finally, experimental results are shown in section IV.

II. Computer Vision Solutions for Marker-Based Augmented Reality

In order to render AR model correctly over the frames from the camera we need to estimate its position. In the case of marker-based AR, the planar marker position in the frame should be known. Hence, we start with the marker detection. Once the marker is found, we track its position in consequent video frames. The marker position in the frame is used to calculate the homography transformation matrix and estimate 6 degrees of freedom (6 DoF) camera position from it.

A. Marker Detection

The estimation of extrinsic camera parameters starts with the detection of the marker position in the scene. By marker we mean not some binary pattern, but a certain image. The fast and robust solution for image marker detection is based on local features. There are plenty techniques for keypoints detection and description. Here we analyzed two of them: ORB [16] and AKAZE [17].

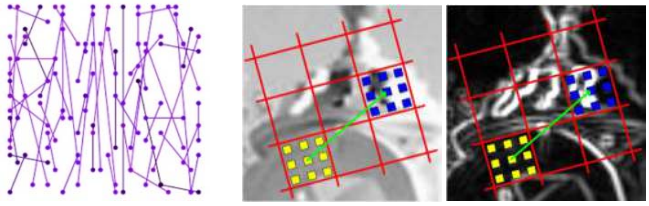
The use of ORB features provides the fastest marker detection procedure comparing to analogues. For the keypoint detection, the oriented FAST [18] is applied in combination with the image pyramid. The local patch around the keypoint is characterized by binary ORB descriptor (Fig. 1, a). It is based on the steered version of BRIEF [19] with binary tests analyzed for correlation to provide more distinctive features.

AKAZE for now is the optimal combination between speed and accuracy. It incorporates fast explicit diffusion, which provides more efficient scale space forming than in SIFT and KAZE. The modified version of LDB descriptor

978-1-7281-3214-3/20/$31.00 ©2020 IEEE

[20] (Fig. 1, b) provides a rotation invariance and preserves computational efficiency without the use of integral images. Scale-dependent sub-sampling (small squares in Fig. 1, b) is used instead of calculating the average value of each area (large squares).

To compare the efficiency of matching ORB and AKAZE features, we used our company business card as a marker (Fig. 2). The keypoint locations were the same in both cases, thus only descriptors and their matching affected the result. The 50 best matches are represented in Fig. 2.



a b

Fig. 1. ORB (a) and M-LDB (b) descriptors.

Fig. 2. Keypoint matching with ORB (upper) and AKAZE (lower).

It is clear that AKAZE provides more reliable matches. Still, most of ORB matches are also good, so random sample consensus (RANSAC) or a similar technique provides a good estimation of the homography matrix in this case, too. And as computation time is crucial for AR in general and WebAR in particular, ORB is our tool of choice.

B. Tracking

Marker detection is a time-consuming operation. Moreover, typical keypoint descriptors handle quite limited view angles [2], and detection fails, which results in a bad AR experience of a user.

To solve both problems, marker tracking algorithms are typically used [2, 15,21]. A common way to track inter frame changes is to use sparse optical flow (OF) methods [2, 23]. This is accomplished by matching adjacent frames instead of straightforward comparison of the reference marker image with the camera frame image, which is not efficient.

A common problem of OF methods in AR applications is the drift of estimated pixel locations. This leads to incorrect camera pose estimation and, as a result, incorrect AR model augmentation. To improve the robustness of our planar tracker, we utilized a template-based tracking called the sum of conditional variances (SCV) [22]. It is an iterative approach, which is used to refine the residual error in homography estimation after OF application.

The proposed fusion of two different trackers leads to a robust tracking.

Firstly, OF is applied to estimate initial homography between adjacent frames. Secondly, SCV refinement is used to receive more accurate warping between the frames (Fig. 3). Such combination allows to compensate OF drift and handles extremal view angles.

Fig. 3. Two-step homography estimation.

C. Camera Pose Estimation

Camera pose estimation is the primary element of AR system that affects the correct rendering of models. Thus, it should be precisely retrieved from the camera frames [2].

Let's consider the common projective geometry for the pinhole camera model [13]. The camera projects points from the 3D world *(x,y,z)* into a 2D pixel in the image plane *(u, v, w)*. Here *w* is a scale parameter for homogeneous coordinates. This transformation can be written as follows:

$$\begin{bmatrix} uw \\ vw \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (1)$$

where c_x, c_y denote the origin of image coordinates, in our application they are equal to zero; f_x, f_y are focal lengths of the camera, responsible for the scaling (zooming). The first matrix is the intrinsic one, which is independent on the scene. It is specific for the particular camera device and can be found once during the camera calibration [14]. The next matrix is the extrinsic one. It contains the camera pose, describing transformation from the world coordinates to the camera coordinate system. The camera pose consists of a translational displacement of the camera (t-vector) and its orientation (r-elements), which represents the transformation (translation and rotation) between the world and the camera coordinate systems.

Let's consider how the marker location in the frame is used to retrieve the camera pose. It is known that the transformation between the planar object locations in two images can be described with the homography matrix [2]:

$$M'(u', v') = H \times M(u, v) = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} M(u, v) \quad (2)$$

Here, M and M' are the original marker and the warped marker on the frame, their pixel coordinates are represented as *(u,v)* and *(u`,v`)* respectively.

The matrix elements are found from matching keypoints in two pictures. Some matches may be wrong, so we used

RANSAC to drop out the outliers and get the right homography.

Homography is a more generic operation compared to the 6 DoF of camera pose (3x1 translation vector and 3x1 rotation vector). So, it is possible to convert homography into a camera pose. The corners of marker are linked with the homography projected ones onto the frame. This set of points is used to estimate rotations and translation vectors basing on Levenberg-Marquardt optimization [24, 25]. The method iteratively minimizes the pixel reprojection error points, represented as the sum of squared distances between the corresponding images.

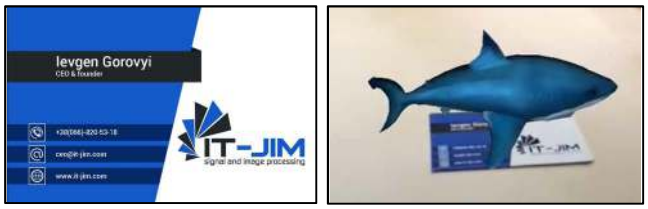


Fig. 4. Example of 3D model augmentation: image marker (left), 3D model over the detected marker (right)

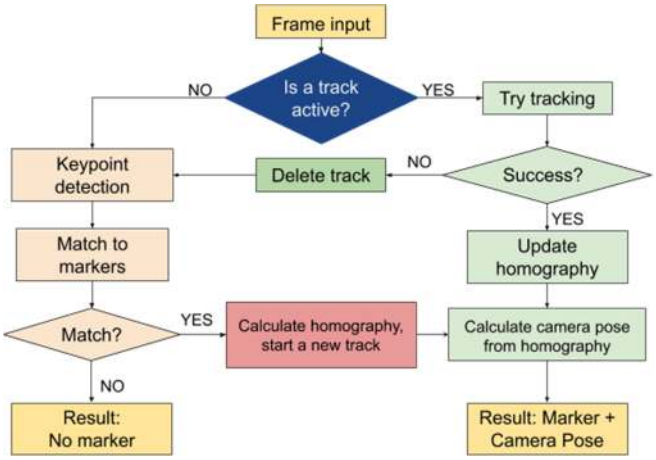


Fig. 5. The main algorithm pipeline.

As a result, 6 DoF camera pose is constantly updated based on the applied planar marker detection and tracking procedures. The estimated camera pose is used to render the AR content. An example of image and 3D model rendering is shown in the Fig. 4.

III. Created Pipeline and Its Implementations

In this section, we consider practical aspects of integration of developed computer vision algorithms and discuss important details of web AR system architecture.

A. High-level Scheme of Camera Frames Processing

The described technical solutions were combined into the following pipeline (Fig. 5).

For a given camera frame, either marker detection or tracking algorithms are used. The homography matrix between marker and camera frame is constantly updated. Finally, the camera pose is estimated in real-time providing the information for AR models rendering in the browser window.

We have created two conceptually different deployments: a front-end-back-end with algorithms running in the cloud and a purely front-end pipeline. The next sub-section describes those two ways of deployment in details.

B. Frontend-Backend Approach

At the front end, we have an outgoing and incoming data streams. Therefore, we divided front end operations into two independent asynchronous threads: camera rendering and model rendering (Fig. 6). Unlike the single sequential thread, when we show video frames after the server response, in this case 3D model is overlaid over the live video stream without any delays and freezes. This approach looks better for a user and provide more opportunities.

The first thread takes a frame from the device camera and immediately displays it on the web page. To use a video stream from the camera, the browser must ensure that the requested web page is safe, thus only HTTPS pages with a SSL/TSL certificate are allowed to access the camera. The captured frame is downsized and converted to JPEG format. At the end, the processed frame is sent to the back end via a secure websocket (wss).

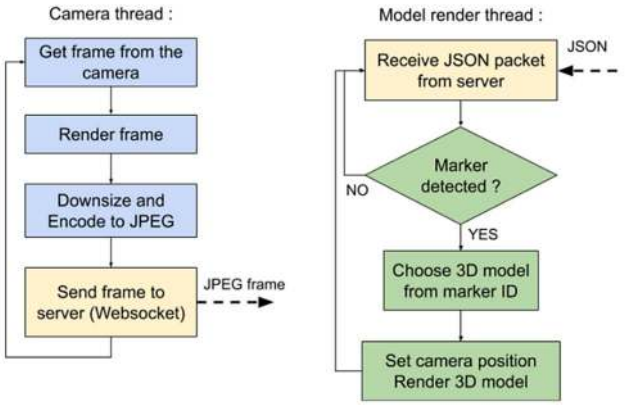


Fig. 6. Frontend architecture. Camera and rendererasynchronical threads.

The second thread, named ‘model render thread’, receives JSON packets from the server. They contain an ID of the identified marker (if there is no marker in the frame, it is equal to -1) and the camera parameters. The latter depend on the render library. In our case, ThreeJS library [26] was used, which creates a camera object by three vectors: ‘position’, ‘lookAt’ and ‘up’.

The calculation of Three JS camera pose has a few important moments. The first is the freedom to choose a render formalism. That means the choice between ‘moving the model’ and ‘moving the camera’. We prefer to move camera rather than the model, because this way is more similar to the real camera moving. Thus, the camera position t_{cam} is calculated as $t_{cam} = -R^{-1}t$, where t is the camera translation for the stationary model described above in the camera pose (1).

The second moment is the usage of left-handed system in contrast with the right-handed one in OpenCV library that was used in C++ pipeline implementation. So, we have to change the sign of z-component of the camera position.

The ‘lookAt’ vector indicates the point the camera is facing at, the ‘up’ vector determines the rotation of the camera view. Both vectors can be found from inverse perspective projection transformation $(u,v) \rightarrow (x,y,z)$. There is an infinite number of solutions, so for simplicity we assume that marker is located in xy0 plane. Considering F: $(u,v) \rightarrow (x,y,0)$, we then project $p_0 = F(0, 0), p_1 = F(0, 1)$. p_0 is the viewport center and

represents the ‘lookAt’ vector, while $up = p_1 - p_0$ determines the ‘up’ vector.

The back-end is implemented using Java and Spring Boot. Its architecture is shown in the Fig. 7. It communicates with the front-end via secure web sockets only. For each web session, a separate engine object is created (a wrapper to C++ algorithms), thus the server can process multiple sessions correctly. The engine performs all steps of frame processing, including detection, tracking and calculation of camera pose. At the end, it sends information to the front-end.

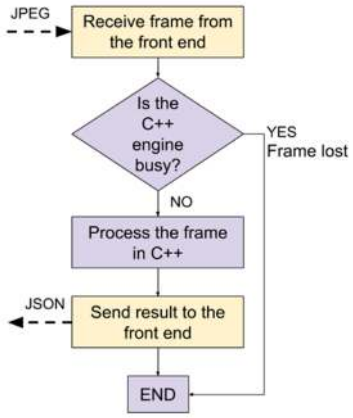


Fig. 7. Server structure.

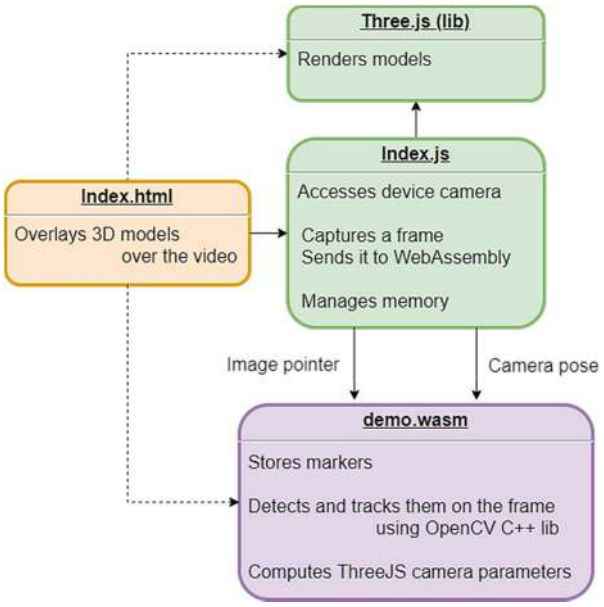


Fig. 8. The architecture of the pure front-end solution.

C. Frontend-only Architecture

To avoid the main problem of the previous architecture, the lag introduced by the network latency, we made a front-end only solution (Fig. 8).

In this way, there is no lag (i.e. transferring an encoded frame to the server and an array of numbers back). On the other hand, all files (including bulky 3D models) have to be downloaded before the web application starts. The required time depends mostly on the user’s connection speed and can be quite long.

This application consists of separate JS modules and WebAssembly files [27]. Here the backend server is absolutely replaced by the wasm file, which was compiled from C++ project with the main algorithm pipeline [28].

WebAssembly is an open standard that defines a portable binary-code to run natively in browsers. In order to compile C++ project, we used Emscripten SDK [29]. It is a suitable instrument to call C++ functions from JavaScript side, and often the speed of procedures is higher than of pure JavaScript.

Fortunately, our architecture has not changed much and it keeps the primary logic of the previous architecture unchanged. The only difference is that a user now downloads all files and does all the calculations in the browser.

D. Architecture Summary

A brief summary with the strong and weak sides of each solution are presented in the Table 1. On its basis, one can select any of the solutions for the particular case that is most suitable for the specified conditions.

The frontend-backend solution is the best when we need complicated processing (potentially including some neural networks) or when we want to cover as much devices as possible, regardless of their hardware computational performance. However, it requires a stable connection during an AR session for unceasing data stream. Moreover, it can be very costly in two scenarios: either we want to run our application worldwide (then we need to deploy it over multiple servers over the globe to minimize the network latency between users and the nearest servers)or we want to serve a lot of users simultaneously(then we need to run a powerful backend server).

TABLE I. PROS AND CONS OF ARCHITECTURE TYPES

	Pros	Cons
Frontend-backend	- Provides better performance - Allows to run heavy algorithms - Covers weak devices	- Network latency - Requires a reliable connection - Costly in multiuser and worldwide usage scenarios
Frontend-only	- No network lags - Runs independently from the network - Easier implementation	- Requires a powerful hardware - Downloads heavy models

The frontend-only solution is easier to implement and has a rather weak dependency on the network connection. Once the user gets all the necessary data from the server, the network can be turned off and the web page with AR application will still operate offline in the browser. To enjoy AR experience, the user has to run it on a powerful modern device with a few gigabytes of RAM (it depends on the weight of the models) and mid to high level system-on-chip (SoC).

IV. EXPERIMENTAL RESULTS

This section contains some relevant information about benchmarking of utilized computer vision solutions as well as performance tests of the AR system.

A. Benchmarks for Marker Detectors

To compare different detectors and find their optimal parameters, we have conducted a number of experiments changing detector parameters. We tuned the input image size, number of pyramid levels (octaves) and looked at the dependence on the number of features. As our target is mobile devices, we need to be sure that their computational power is sufficient to use our application in real time and balance between the detection quality and the algorithm speed.

Considering the results, presented in Fig.9 – Fig.11, we have chosen ORB detector with 1000 keypoints and 8 scale levels for 730x410 resolution.

We tested the performance on preliminary recorded videos with markers in various positions. The quality metrics was defined as the ratio of frames with detected markers to the total number of frames in the video sample. The successful detection is registered if the average distance of 20 best matches is below a certain threshold (32.0 in our model).

B. *Performance Tests for Frontend-Backend Architecture*

We have tested the influence of the backend server's computational power on the execution time. The results for two types of AWS instances are shown in Table 2. FPS is the time it takes a frame to complete the entire pipeline with rendering. When a marker is present in the scene, FPS is determined mostly by tracking. If there is no marker, tracking is impossible, and only detection affects speed.

Although FPS is quite high, there is a significant lag between the camera and the render threads. The main problem here is not C++ algorithms, but quality and stability of the internet connection between the browser and the back-end server. The lag mainly stems from the network latency, which varies for different conditions. A delay of a few frames (0.1-0.2s) was very common in our case. In addition, some frames may be lost on the server.

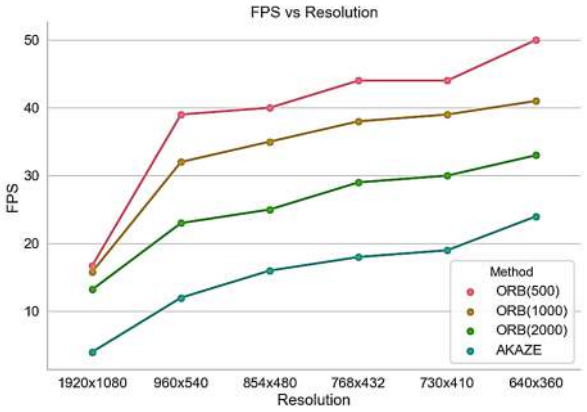


Fig. 9. FPS dependence on the image resolution and a detector type.

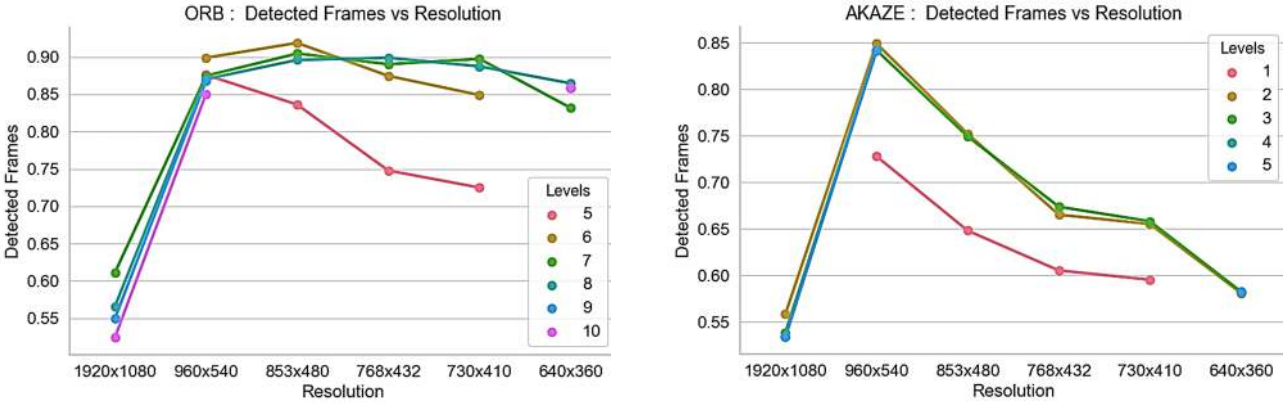


Fig. 10. Accuracy provided by ORB and AKAZE dependion on input image resolution and a number of levels.

TABLE II. TIMING ON DIFFERENT AWS INSTANCES

Functions	T2 small	T2 large
Detection (marker)	120 ms	77 ms
Detection (no marker)	100 ms	46 ms
Tracking	5 ms	4 ms
Decode JPEG	1.2 ms	1.0 ms
Camera pose estimation	0.3 ms	0.2 ms
FPS (marker)	~20	~20
FPS (no marker)	6-7	>10

Unfortunately, network latency can be reduced only by decreasing the distance between the geographical positions of user and server.

C. *Performance Tests for Serverless Architecture*

In the serverless solution, the performance is determined by the device (e.g. cell phone) capabilities and does not depend on any server where the web page is located.

For testing, we used a laptop with Intel Core i5-8250U CPU at 1.60GHz. The results are demonstrated in Table 3. It is about as powerful as modern high-end mobile devices. The performance of the majority of devices is expected to be worse.

TABLE III. TIMING ON THE USER'S LAPTOP

Functions	Time
Detection (1 marker)	150-250 ms
Tracking	15-25 ms
Rendering	10-30 ms
Memory management	1-2 ms
FPS (marker)	~16
FPS (no marker)	3-5

As the resulting timing depends on the current load of the user's CPU, we give the time ranges. WebAssembly reduces the speed of detection and tracking compared to the original C++ code.

V. CONCLUSIONS

We described the important elements required to build a web-based AR application. The applied computer vision solutions were analyzed and discussed. Two different architectures for

the AR system were proposed. The developed AR pipeline is flexible and extendable. We are planning to integrate a new group of computer vision and machine learning algorithms for face tracking, text recognition, arbitrary 2D/3D object recognition in the near future. In addition, additional work will be done for the algorithm optimization, which is crucial for the application running in a mobile browser.

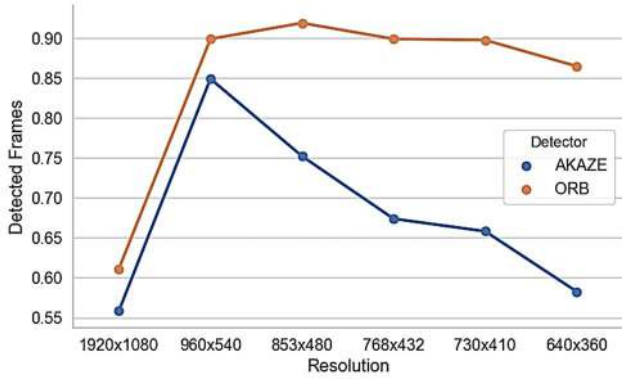


Fig. 11. Accuracy of detectors depending on the Image Resolution. For each resolution was chosen the best properties provided the large number of detected frames.

REFERENCES

[1] V. Vovk et al., "Light-Weight Tracker for Sports Applications" Proceedings of Signal Processing Symposium, 17-19 September, Krakow, Poland, pp.255-259, 2019.

[2] I. Gorovyi et al., "Advanced Image Tracking Approach for Augmented Reality Applications" Proceedings of Signal Processing Symposium, 12-14 September, Jachranka, Poland, pp.266-270, 2017.

[3] Y.-G. Kim and W.-J. Kim, "Implementation of augmented reality system for smartphone advertisements," Int. J. Multimedia Ubiquitous Eng., vol. 9, no. 2, pp. 385–392, 2014.

[4] (Jan. 2017). Augmented/Virtual Reality Report 2017. Accessed: Feb. 17, 2018. [Online]. Available: https://www.digi-capital.com/reports

[5] B. Perry, "Gamifying French language learning: A case study examining a quest-based, augmented reality mobile learning-tool," Procedia-Social Behav. Sci., vol. 174, pp. 2308–2315, 2015

[6] Ha, Ho-Gun & Hong, Jaesung. (2016). Augmented Reality in Medicine. Hanyang Medical Reviews. 36. 242. 10.7599/hmr.2016.36.4.242.

[7] Pokémon GO Report. Accessed: Mar. 5, 2018. [Online]. Available: https://pokemongolive.com/en/post/headsup

[8] Z. Shi, H. Wang, W. Wei, X. Zheng, M. Zhao, and J. Zhao, "A novel individual location recommendation system based on mobile augmented reality," in Proc. IEEE Int. Conf. Identificat., Inf., Knowl. Internet Things (IIKI), Oct. 2015, pp. 215–218.

[9] N. Gavish et al., "Evaluating virtual reality and augmented reality training for industrial maintenance and assembly tasks," Interact. Learn. Environ., vol. 23, no. 6, pp. 778–798, 2015.

[10] M. A. Tahoun, et al., "A robust content-based image retrieval system using multiple features representations," Proceedings. 2005 IEEE Networking, Sensing and Control, 2005., Tucson, AZ, 2005, pp. 116-122.

[11] H. Durrant, T. Bailey. "Simultaneous localization and mapping: Part I", IEEE Robotics & Automation, (2006), pp. 99-108.

[12] Nowacki, Paweł & Woda, Marek. (2020). "Capabilities of ARCore and ARKit Platforms for AR/VR Applications". 10.1007/978-3-030-19501-4_36.

[13] Qiao, Xiuquan & Pei, Ren & Dustdar, Schahram & Liu, Ling & Ma, Huadong & Junliang, Chen. (2019). Web AR: A Promising Future for Mobile Augmented Reality - State of the Art, Challenges, and Insights. Proceedings of the IEEE. 107. 1-16. 10.1109/JPROC.2019.2895105.

[14] A. Charland and B. Leroux, "Mobile application development: Web vs. native," Commun. ACM, vol. 54, no. 5, pp. 49–53, May 2011

[15] Senst, Tobias & Eiselein, Volker & Sikora, Thomas. (2012). Robust Local Optical Flow for Feature Tracking. IEEE Transactions on Circuits and Systems for Video Technology. 22. 10.1109/TCSVT.2012.2202070.

[16] 9. E. Rublee, V. Rabaud, K. Konolige, G. Bradski, Gary, ORB: an efficient alternative to SIFT or SURF. Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2011, pp. 2564-2571. doi 10.1109/ICCV.2011.6126544

[17] P. F. Alcantarilla, J. Nuevo, A. Bartoli, Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces. 2013, British Machine Vision Conference (BMVC). doi 10.5244/C.27.13.

[18] E. Rosten, T. Drummond. Machine learning for high-speed corner detection. European Conference on Computer Vision (ECCV), 2006, pp. 430-443.

[19] M. Calonder, V. Lepetit, C. Strecha, P. Fua, Brief: Binary robust independent elementary features. European Conference on Computer Vision (ECCV), 2010, pp. 778-792.

[20] X. Yang, K. T. Cheng, LDB: An ultra-fast feature for scalable augmented reality. IEEE and ACM Intl. Sym. on Mixed and Augmented Reality (ISMAR), 2012. doi 10.1109/ISMAR.2012.6402537

[21] Bertrand Delabarre, Eric Marchand. Visual Servoing using the Sum of Conditional Variance. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS'12, 2012, Vilamoura, Portugal. pp.1689-1694. ffhal-00750602f

[22] Richa, Rogerio & Sznitman, Raphael & Taylor, Russell & Hager, Gregory. (2011). Visual tracking using the sum of conditional variance. IEEE International Conference on Intelligent Robots and Systems. 2953-2958. 10.1109/IROS.2011.6094650.

[23] OpenCV. Open Source Computer Vision Library. 2015.

[24] Levenberg, K.: A method for the solution of certain non-linear problems in least-squares. Quarterly of Applied Mathematics 2 (1944) 164–168

[25] Marquardt, D.: An algorithm for the least-squares estimation for non-linear parameters. SIAM J. Applied Mathematics 11 (1963) 431–441

[26] Dirksen, Jos (2013). Learning Three.js: The JavaScript 3D Library for WebGL. UK: Packt Publishing. ISBN 9781782166283.

[27] A. Haas et al., "Bringing the web up to speed with WebAssembly," in Proc. 38th ACM SIGPLAN Conf. Program. Language Design Implement., 2017, pp. 185–200.

[28] A. Møller, "Technical perspective: WebAssembly: A quiet revolution of the Web," Commun. ACM, vol. 61, no. 12, p. 106, 2018.

[29] Alon Zakai. "Emscripten: an LLVM-to-JavaScript compiler. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion". Association for Computing Machinery, New York, NY, USA, 301–312. 2011