A Study on Enhancing Performance of Marker-Based Augmented Reality for Low-end Smartphone

Yanghai Pov **Monioudom Mao** **Lihour Nov**
Department of Computer Science of Cambodia Academy of Digital Technology,
Phnom Penh, Cambodia
Yanghai.Pov@student.cadt.edu.kh

Abstract

Augmented reality (AR) lets people interact with virtual objects in the real world using their smartphones. However, most AR systems need powerful devices to run smoothly, which leaves out many users who have older or low-end phones. This research focuses on making marker-based AR work well on these less powerful smartphones. By using simple and fast algorithms to detect patterns and smart techniques to show 3D models efficiently, the system can recognize images accurately and run smoothly even on budget devices. Tests show that the system keeps good performance and image recognition accuracy, proving that AR can be made more accessible to a wider audience. This work aims to help bring the benefits of AR to more people around the world, regardless of the device they use.

Keywords: *Marker-Based Augmented Reality (AR), Features from Accelerated Segment Test(FAST), Oriented FAST and Rotated BRIEF (ORB)*

1 Introduction

Augmented Reality (AR) is a technology that overlays virtual objects onto the physical world using digital devices such as smartphones or tablets [1], [2]. This integration allows users to interact with both digital and real-world elements simultaneously, creating immersive experiences. It is increasingly applied in various fields such as education, entertainment, healthcare, and retail due to its ability to enhance visual and interactive content. AR systems generally work by capturing the real environment through a camera, detecting visual markers or features, understanding the spatial layout, and rendering 3D content in real time. The rendered virtual objects must align accurately with the real-world environment to ensure a stable and convincing experience. These processes require considerable computational resources, including real-time image processing and graphics rendering, which can be challenging for low-end mobile devices.

There are several types of AR, including Marker-Based AR, Markerless AR (i.e., GPS/Sensor-Based AR), Simultaneous Localization and Mapping (SLAM)-based AR, and Projection-Based AR. Among them, Marker-Based AR is the most lightweight and device-friendly[1]. It uses printed patterns (called markers) to help the system identify specific locations in the camera view, making it easier to place and track virtual content. Marker-Based AR is widely used in mobile applications, especially on devices with limited hardware. However, running AR applications on low-end smartphones remains difficult due to their limited CPU power, smaller RAM, weaker GPUs, and lower-quality cameras. These limitations often cause reduced frame rates, slow marker detection, and unstable rendering [3]. As AR becomes more widely adopted, it is important to develop lightweight and optimized solutions to ensure acceptable performance on these types of devices.

To address this issue, many researchers have explored efficient computer vision algorithms such as Features from Accelerated Segment Test (FAST) and Oriented FAST and Rotated BRIEF (ORB). FAST is a corner detection algorithm that quickly identifies keypoints in an image [4]. ORB builds on FAST by adding rotation-invariant descriptors using BRIEF (Binary Robust Independent Elementary Features) for feature matching [5]. Together, these algorithms provide a fast and effective method for real-time tracking on low-end hardware, without sacrificing too much accuracy. Several studies have shown that combining FAST and ORB improves AR performance. Rublee et al. introduced an optimized Marker-Based AR method using ORB and planar surface recognition [5]. It also used

OpenGL ES 2.0 with vertex rendering optimizations to improve 3D rendering performance. The result showed up to 90% faster recognition and stable frame rates of 16–24 FPS on low-end smartphones [6], [8]. Another study compared Marker-Based AR applications on different mobile devices and found that image resolution and processing speed were key factors affecting performance. Lower resolutions helped improve speed on older phones without significantly affecting detection quality [7], [9].While these studies have successfully improved AR performance on smartphones, most of them focused on mid- to high-end devices or did not evaluate performance differences across hardware levels. Additionally, they often optimized either the feature detection or the rendering process individually, rather than integrating both. As a result, their solutions may still struggle to deliver smooth, real-time AR performance on low-end smartphones with limited CPU and memory capacity.

To fill these gaps, this paper proposes an optimized Marker-Based AR framework called ModAR, which integrates lightweight computer vision and rendering techniques tailored for low-end smartphones. Specifically, the system combines the FAST and ORB algorithms to enhance marker detection speed and stability while minimizing computational load. The proposed method aims to improve real-time performance by maintaining a stable frame rate above 15 FPS, reducing latency, and ensuring smooth operation without requiring additional sensors or external hardware.

2 Methodology

This section will explain how the ModAR system works to recognize a pattern and overlay a 3D model onto it in real time, especially on low-end smartphones. ModAR is based on a marker-based AR (MAR) approach, where a known image called a pattern is used to place virtual objects in the real world through the phone's camera [8]. The system assumes the pattern is a flat, rectangular image. A top-down (nadir) image of this pattern is needed to initiate the AR process. In addition to the pattern image, the 3D model and camera calibration data (i.e., the camera's internal parameters) are required to ensure accurate projection.

The center of the pattern is defined as the origin point $(0, 0, 0)$ in the world coordinate system shows in **Figure 1**. The X and Y axes lie on the surface of the pattern, while the Z axis extends perpendicularly outward from the pattern toward the camera. To simplify calculations and maintain consistency across devices, the four corners of the pattern are normalized relative to the image dimensions. As a result, their X and Y values range from -1 to 1, and their Z values are set to 0. This normalization allows for easy transformation between image coordinates and real-world coordinates [9]. This setup provides a stable spatial reference frame, ensuring that virtual 3D objects are anchored correctly and appear aligned with the physical pattern when viewed through the camera in real time.

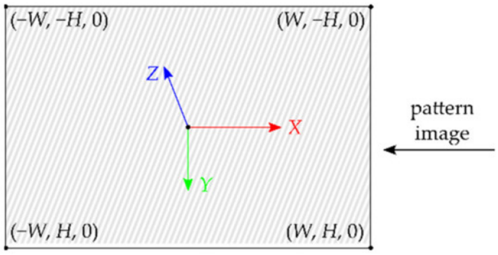

Figure 1: World coordinate system setup for the marker pattern [8].

2.1 Feature Extraction and Image Matching

To recognize the pattern in the real world, ModAR starts by detecting special visual features unique points like corners or edges—in the reference pattern image. During runtime, it looks for those same features in each new frame captured by the phone's camera. To do this efficiently, ModAR uses feature detection and description methods such as ORB, which combines the FAST algorithm for finding keypoints and the BRIEF algorithm for describing them, and BRISK with AGAST, where AGAST detects the keypoints and BRISK describes them in a way that works even when the image is rotated or scaled. After detecting features in both images, the system compares them using Hamming distance, which measures how different the binary descriptors are. To improve accuracy, it performs a cross check only keeping matches that are confirmed in both directions. It then filters out bad matches by removing those with large differences, and applies RANSAC (Random Sample

Consensus) to keep only the best matches that fit a realistic transformation of the image, helping the system determine where and how the pattern appears in the real world [8].

2.2 Pattern Recognition and Camera Pose Estimate

Once RANSAC finds at least 8 good matches, the pattern is considered detected, and a 3D model can be placed on it. RANSAC estimates a homography matrix that describes how the pattern image appears in the real-world camera view. This estimation is refined using the Levenberg–Marquardt algorithm, which improves accuracy. To understand how the camera is positioned and oriented relative to the pattern, the system calculates the camera's 6 degrees of freedom (6-DOF) pose—its position (X, Y, Z) and rotation in 3D space. This is done using a projection transformation matrix, which relates real-world points to their position in the image. The rotation matrix is improved using singular value decomposition (SVD) to ensure it is mathematically valid. Then it's converted into a more compact form using the Rodrigues formula, which helps during optimization [8].

2.3 3D Render

The 3D model used in ModAR is stored in the `.obj` file format, which includes information about the model's shape and its textures. To display the model, the system first sends the model data to the GPU using a vertex buffer. For every frame, it applies a set of transformation matrices to place and display the model correctly. These transformations are combined using (1).

$$\text{Final Position} = \text{Projection} \times \text{View} \times \text{Model} \times \text{Vertex} \quad (1)$$

To make the rendering more efficient, the system compresses texture files using techniques like ETC1 and PVRTC, which reduce memory usage. It also uses a method called frustum culling, which ensures only the parts of the model that are visible in the camera view are drawn—saving processing power. Additionally, when the same model needs to appear multiple times or is animated, the system uses geometry instancing. This allows it to draw many copies of the same model with just one command, which helps improve performance[9]. The entire rendering process runs on a lightweight graphics engine that uses OpenGL ES 2.0, making it suitable even for mobile devices with limited hardware[10].

2.4 AR Integration

For AR to work properly, the virtual model must align correctly with the real world. This is called registration. The system ensures the 3D model sits at the center of the pattern, facing the camera along the Z-axis. Key elements needed for AR include the estimated camera position and orientation, the camera calibration matrix, the pattern image, and the 3D model and rendering functions. To coordinate rendering and detection processes, semaphores (synchronization tools) are used. These help avoid overlapping memory usage and wasted computing time. When a pattern is detected, the AR system calculates a model matrix based on the camera's pose and applies the appropriate projection matrix to render the model in real-time. The projection matrix is adapted to match OpenGL requirements, ensuring the 3D model appears correctly on screen. Clipping planes, field of view, and aspect ratio are also considered to avoid rendering errors due to depth miscalculations. Finally, shaders are used to define how the model looks (lighting, color, texture), and the rendering engine builds the final AR scene by combining all these components[10].

3 Implementation

The ModAR prototype uses OpenCV (C++ via Android NDK) for computer vision and OpenGL ES 2.0 for graphics through the Android SDK. Java communicates with the native C++ code via JNI, with the C++ compiled into a shared library using CMake. The 3D engine handles vertices, models, textures, and shaders[8].

Pattern detection and 3D rendering run simultaneously on separate threads, sharing camera position, orientation, and recognition status. Semaphores ensure this shared data is safely managed.

For each camera frame, the app runs cameraPoseEstimation, which detects features, matches them to the pattern, and calculates the camera's position and orientation. To reduce flickering from motion blur or noise, the model is displayed only after two consecutive frames have enough matches [9].

The 3D rendering runs on its own thread and is initialized when the graphics engine starts. The renderer manages things like depth testing (to correctly show which objects are in front), culling back faces (to improve performance), and setting up the viewport when the screen changes. It loads 3D models, textures, and sets up the camera using various helper classes. Lighting and model-view matrices are also prepared to display the scene properly[10].

The Java side of the app includes classes like JavaCameraView, which handle camera operations getting the camera's projection matrix, intrinsic parameters, and capturing frames to send to the pose estimation process. The AndroidCameraView class converts camera frames into a format usable by the native code. Finally, the MainARActivity manages permissions, sets up views, and controls the overall AR experience.

Table 1: Specifications of low-end and mid-range test devices.

Specification	Low-End	Mid-Range
Model	Galaxy Ace 2	ZTE Blade A5
Year	2012	2019
CPU	800 MHz dual-core	1.6 GHz octa-core
Chipset	ARM Cortex-A9	Spreadtrum SC9863A
GPU	ARM Mali 400	IMG8322
Storage	4 GB	16 GB
Memory	768 MB RAM	2 GB RAM
Camera	5 MP	13 MP
Video	720×1280, 30 FPS	1080×1920, 30 FPS
Screen Res.	480×800 px	720×1440 px

4 Experimental Results and Discussion

4.1 Experimental Setup

To evaluate ModAR's performance, a series of experiments were conducted using two Android smartphones: one older low-end device and one affordable mid-range device. The aim was to understand how hardware specifications impact the system's ability to detect patterns and render 3D models in real time. Both devices supported OpenGL ES 2.0 and ran at least Android 4.4 KitKat (API level 19). The specifications of these devices are presented in **Table 1**. The testing used six pattern images, whose dimensions and resolutions are presented in **Table 2** and four 3D models included with the ModAR prototype, presented in **Table 3**. The pattern images were everyday objects such as book covers, a board game box, a framed wall painting, and a graffiti mural, with resolutions ranging from 0.07 to 0.25 megapixels. The camera resolution was 0.92 MP (720×1280 pixels) on the low-end device and 2.07 MP (1080×1920 pixels) on the mid-range device, both running at 30 frames per second. The 3D models came from various sources like image-based photogrammetry and laser scanning, and they varied in size and complexity. "POTTERY" and "BUST" were small models, "STATUE" was a medium-sized model with vertex color instead of texture, and "CHURCH" was a large, highly detailed model. This setup allowed for a detailed performance assessment of ModAR's detection and rendering processes across different device capabilities[8].

Table 2: Dimensions and resolution of the pattern images used in the experiments.

Pattern Image	Dimensions (px)	Resolution (MP)
BOTERO	348 × 717	0.25
SCYTHE	300 × 245	0.07
SURVEYING	310 × 450	0.14
JEFFERS	250 × 294	0.07
PAINTING	336 × 252	0.08
GRAFFITI	400 × 347	0.14

Table 3: Size, faces, and vertices of the 3D models used in the experiments.

3D Model	Size (KB)	Faces / Vertices
POTTERY	167	3296 / 1661
BUST	903	8767 / 4555
STATUE	1690	21,453 / 42,712
CHURCH	87	14,645,744 / 7,335,148

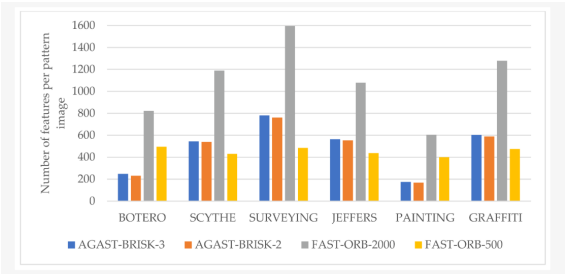

Figure 2: Number of feature points for each pattern image used in the experiments extracted by the tested detectors with different parameterization [8].

Table 4: Parameterization evaluated in the experiments performed using the AGAST detector and the BRISK descriptor.

Experiment	**AGAST-BRISK-3**	**AGAST-BRISK-2**
AGAST detection threshold score	30	30
Number of octaves	3	2
Scale applied to the pattern used for sampling the neighborhood of each feature	1	1

Table 5: Average computational time and percentage of CPU usage for ModAR prototypes with low-end device.

Task	**AGAST-BRISK-3**		**AGAST-BRISK-2**		**FAST-ORB-2000**		**FAST-ORB-500**	
	Time (s)	CPU (%)	Time (s)	CPU (%)	Time (s)	CPU (%)	Time (s)	CPU (%)
Feature detection	2.425	49.47	2.296	47.73	0.086	12.51	0.069	12.02
Feature description	2.337	45.91	2.229	46.69	0.230	32.09	0.189	34.29
Image matching	0.029	0.53	0.026	0.43	0.020	2.82	0.016	3.20
Homography estimation	0.089	1.49	0.063	1.31	0.098	19.37	0.035	6.30

4.2 Image Matching and Pattern Recognition

Two main types of algorithms for detecting features in images such as AGAST-BRISK and FAST-ORB were tested. There are four different setups, including AGAST-BRISK-2, AGAST-BRISK-3, FAST-ORB-500, and FAST-ORB-2000. Among these, FAST-ORB-2000 performed the best overall. It offered a good balance between speed and accuracy, worked well on both low-end and mid-range phones, and was able to recognize patterns even if they were rotated or smaller. FAST-ORB-2000 took a bit longer—around 0.4 seconds per frame on the low-end device and 0.35 seconds on the mid-range but gave more reliable and consistent pattern detection[8]. The AGAST-BRISK parameterization evaluated in the experiments is shown in **Table 4**. The number of feature points extracted for each pattern by the different detectors is illustrated in **Figure 2**.

The CPU and time usage results showed that the AGAST-BRISK methods used nearly 95% of the CPU, which is very high and not efficient for mobile devices. In contrast, the FAST-ORB methods only used about 50% of the CPU, making them much more efficient for real-time use. The observed efficiency of FAST-ORB compared to AGAST-BRISK can be explained by its algorithmic design. FAST detects corners using simple intensity comparisons around a pixel, avoiding the complex multi-scale keypoint detection required by AGAST. ORB then uses the lightweight BRIEF descriptor, which encodes features as binary strings for fast Hamming-distance matching. In addition, FAST-ORB does not require the extensive sampling of multiple scales and orientations like AGAST-BRISK, which significantly reduces computation. In contrast, AGAST-BRISK involves more computationally intensive operations for both detection and description, increasing CPU load. This combination makes FAST-ORB significantly lighter and more suitable for low-end devices. The pattern recognition was almost perfect, reaching close to 100% accuracy, as long as the pattern appeared in the camera frame at least half as large as its original size.This information is presented in **Table 5** for the low-end device and in **Table 6** for the mid-range device. This means the system works best when the pattern was reasonably visible and not too small[8].

The best method for pattern recognition in ModAR is FAST-ORB-2000 because it is fast, stable, and highly accurate. The AR system performs well with small and medium-sized 3D models, even on older smartphones. However, when using large 3D models, the performance decreases due to limitations in the GPU and memory. To ensure smooth operation on low-end devices, optimization techniques such as instancing and grouping Vertex Buffer Objects (VBOs) are crucial. These optimizations help the system run efficiently and provide a better AR experience on less powerful phones.

The AR system ran at 16 to 24 frames per second (FPS), which is generally acceptable for real-time use. Smaller and medium-sized models, like “POTTERY” and “STATUE,” were ren-

Table 6: Average computational time and percentage of CPU usage for ModAR prototype with mid-range device.

Method	AGAST-BRISK-3		AGAST-BRISK-2		FAST-ORB-2000		FAST-ORB-500	
	Time (s)	CPU (%)	Time (s)	CPU (%)	Time (s)	CPU (%)	Time (s)	CPU (%)
Feature detection	1.926	49.52	1.851	47.70	0.069	14.21	0.057	10.33
Feature description	1.795	45.50	1.656	46.51	0.194	37.46	0.155	35.25
Image matching	0.027	0.53	0.023	0.41	0.070	13.94	0.012	3.40
Homography estimation	0.080	1.50	0.050	1.28	0.023	3.89	0.026	5.30

dered quickly and smoothly on both devices. However, the large "CHURCH" model caused noticeable slowdowns, especially on the low-end phone. To improve performance, optimization techniques such as instancing which allows drawing many copies of the same object in a single call and organizing the models into a single memory buffer called a Vertex Buffer Object (VBO). These optimizations improved rendering performance by up to 3.7 times on low-end devices[8].

GPU performance and CPU profiling results are presented in **Figure 3**, and memory profiling for the large 3D model "CHURCH" is shown in **Figure 4**. These results highlight that while CPU usage was significant, the main challenge lay in GPU memory and rendering. By applying optimization strategies, it became possible to achieve smoother frame rates and reduce resource consumption, even on constrained hardware. This demonstrates the importance of efficient GPU memory management for making marker-based AR systems practical on low-end smartphones. Furthermore,these findings confirm that targeted optimizations can significantly extend the usability of AR applications on budget devices without requiring additional hardware support.

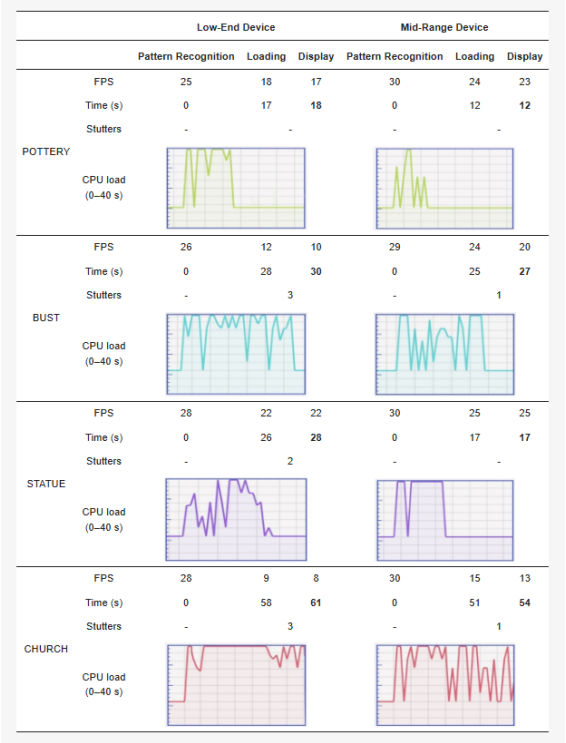

Figure 3: GPU performance and CPU profiling results during the graphics computations of four AR sessions [8].

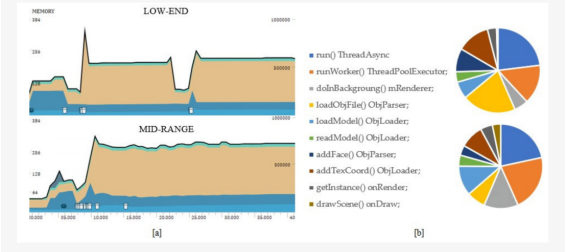

Figure 4: Memory profiling of the "CHURCH" 3D Model: [a] 40s timeline upon pattern recognition with the memory allocation of the main memory types; [b] graphics objects and classes with the most heap count [8].

5 Conclusion

In conclusion, this review demonstrates that marker-based augmented reality can work effectively on low-end smartphones despite their limited processing power and hardware. Efficient algorithms such as FAST, ORB, OpenGL ES 2.0, geometry instancing, and frustum culling enable AR systems to achieve good accuracy and smooth performance. The research also highlights the importance of designing AR systems that account for device diversity and user needs. Focusing on lightweight methods and performance improvements can help marker-based AR reach everyone, bridging the digital divide and

expanding opportunities for users globally.

References

[1] R. Azuma, "A survey of augmented reality," *Presence: Teleoperators Virtual Environ.*, vol. 6, no. 4, pp. 355–385, Aug. 1997, doi: 10.1162/pres.1997.6.4.355.

[2] S. R. Fan and S. Y. Lin, "Augmented reality for mobile devices: A survey," *IEEE Access*, vol. 8, pp. 136–153, 2020, doi: 10.1109/ACCESS.2020.2965539.

[3] M. Billinghurst, A. Clark, and G. Lee, "A survey of augmented reality," *Found. Trends Hum. Comput. Interact.*, vol. 8, no. 2–3, pp. 73–272, 2015, doi: 10.1561/1100000049.

[4] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2006, pp. 430–443, doi: 10.1007/11744023_34.

[5] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2011, pp. 2564–2571, doi: 10.1109/ICCV.2011.6126544.

[6] X. Xu, D. Chen, J. Ren, and B. Dang, "Research on augmented reality method based on improved ORB algorithm," *J. Phys. Conf. Ser.*, vol. 1453, no. 1, p. 012024, Jan. 2020, doi: 10.1088/1742-6596/1453/1/012024.

[7] S. Irawan, M. Suhartono, R. R. Suryanegara, and D. Suryani, "Performance evaluation of marker-based augmented reality applications on low-end and high-end mobile devices," in *Proc. Int. Conf. ICT Smart Soc. (ICISS)*, 2019, pp. 1–6, doi: 10.1109/ICISS48059.2019.8969765.

[8] S. Verykokou, C. Ioannidis, and G. Kambourakis, "Mobile augmented reality for low-end devices based on planar surface recognition and optimized vertex rendering," *J. Phys. Conf. Ser.*, vol. 1453, no. 1, p. 012024, Jan. 2020, doi: 10.1088/1742-6596/1453/1/012024.

[9] M. Fernández, R. Martínez, L. Pérez, A. León, and D. Díaz, "Performance evaluation of marker-based augmented reality applications on mobile devices," in *Proc. 16th Int. Conf. Ubiquitous Comput. Commun. (IUCC)*, Granada, Spain, Dec. 2017, pp. 623–630, doi: 10.1109/IUCC.2017.00117.

[10] A. Kaehler and G. Bradski, *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*, Newton, MA, USA: O'Reilly Media Inc., 2017.