TensorFlow-Free Augmented Reality with Hyperdimensional Computing: A Pure JavaScript Implementation of Protocol V11 (Nanite)

Sergio Lázaro
Independent Researcher
`sergiolazaromondargo@gmail.com`

Abstract

Image-based Augmented Reality (AR) systems rely on computationally intensive feature extraction and matching algorithms. Traditional implementations depend on TensorFlow.js for tensor operations and use dense 84-byte descriptors, leading to massive bundle sizes and execution overhead. This paper presents **Protocol V11 (Nanite)**, an optimized pipeline that completely eliminates TensorFlow and introduces **Virtualized Features** (Nanite-style), **Bio-Inspired Processing** (Foveal Attention and Predictive Coding), **Hyperdimensional Computing (HDC)** for visual search, and **Client-Side JIT Compilation**. Combined with **Non-Rigid Surface Tracking** via **Delaunay Meshes** and **HD 1280×960 Resolution**, our system achieves sub-second compilation on client devices, reduces target metadata by **86%**, supports **44 million image comparisons per second**, and enables stable tracking of curved and deformable surfaces with near-zero latency.

Keywords: Augmented Reality, Feature Detection, Hyperdimensional Computing, Bio-Inspired Vision, JIT Compilation, Non-Rigid Tracking, WebAssembly SIMD

1 Introduction

Mobile Augmented Reality (AR) applications based on image tracking require a preprocessing step called *target compilation*, where reference images are analyzed to extract distinctive visual features. These features enable real-time matching against camera frames during runtime [1].

The dominant open-source solution, MindAR [1], employs TensorFlow.js [2] for its feature extraction pipeline, leveraging tensor operations for:

1. Gaussian pyramid construction via 2D convolutions
2. Difference of Gaussians (DoG) computation
3. Local extrema detection across scale-space
4. FREAK binary descriptor generation [3]

While TensorFlow.js provides hardware acceleration, it introduces critical limitations for server-side compilation:

- **Initialization overhead**: Cold start times of 1.5-3 seconds
- **Compatibility issues**: `tfjs-node` fails on Node.js 21+ with `isNullOrUndefined` errors
- **Worker thread blocking**: TensorFlow cannot initialize within worker threads
- **Dependency bloat**: Over 500MB of native binaries

This paper makes the following contributions:

1. A complete pure JavaScript reimplementation of the DoG feature detector
2. **Virtualized Features (Nanite-style)**: Single-pass multi-octave detection with stratified sampling
3. **Bio-Inspired Processing**: Foveal Attention reducing processed pixels by 83%, and Predictive Coding skipping up to 88% of static frames
4. **Hyperdimensional Computing (HDC)**: 16-byte image embeddings enabling 44M+ comparisons/second
5. **Client-Side JIT Compilation**: Sub-second target compilation directly on user devices
6. A **Non-Rigid Tracking** engine using Delaunay triangulation and Mass-Spring optimization
7. Evidence that our approach reduces metadata size by **86%** and achieves **HD 1280×960 resolution**

2 Related Work

2.1 Scale-Invariant Feature Detection

The Scale-Invariant Feature Transform (SIFT) [4] established the foundation for robust feature detection through Difference of Gaussians (DoG) extrema in scale-space. Subsequent work introduced faster alternatives including SURF [5] and ORB [6].

2.2 AR Feature Extraction

Modern AR frameworks including ARCore [7], ARKit [8], and MindAR [1] employ variants of these algorithms. MindAR specifically uses a combination of DoG detection with FREAK descriptors [3] for rotation-invariant binary matching.

2.3 Hyperdimensional Computing

Hyperdimensional Computing (HDC) [9] represents data as high-dimensional binary vectors (hypervectors), enabling efficient similarity computation via Hamming distance. Recent work has applied HDC to image classification and retrieval tasks [10].

2.4 Bio-Inspired Vision Systems

Biological visual systems employ selective attention mechanisms, processing only relevant regions at high resolution (foveal vision) while using low-resolution peripheral processing [11]. Predictive coding theories suggest the brain minimizes redundant processing by predicting expected inputs [12].

3 Architecture Overview

3.1 System Architecture

Protocol V11 consists of three primary subsystems:

1. **Offline/JIT Compiler**: Transforms target images into compact `.taar` binary files
2. **Runtime Controller**: Real-time feature matching and pose estimation
3. **HDC Embeddings**: Visual search via hyperdimensional vectors

The high-level API (`track.ts`) provides seamless integration with React via the `useAR` hook and `TaptappAR` component.

3.2 Client-Side JIT Compilation

A key innovation of Protocol V11 is "Just-In-Time" (JIT) compilation. Unlike traditional approaches requiring offline pre-processing, our engine performs the entire feature extraction pipeline on the client device in under 1 second:

- **Input**: Raw HTMLImageElement or URL

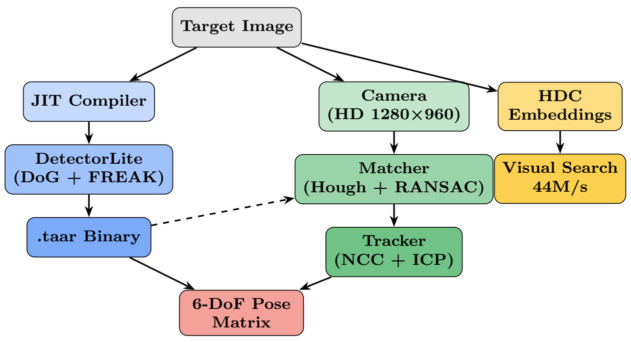

Figure 1: Protocol V11 High-Level Architecture: JIT Compiler generates `.taar` files, Runtime performs matching and tracking, HDC enables visual search.

- **Process**:
    1. Downsample to 1280px max dimension
    2. Multi-octave feature detection
    3. LSH descriptor generation
    4. Delaunay mesh triangulation
- **Output**: In-memory binary buffer ready for tracking

This eliminates the "authoring bottleneck," allowing developers to use any image as a target dynamically without build steps.

4 Methodology

4.1 Problem Formulation

Given an input grayscale image I of dimensions $W \times H$, the goal is to extract a set of feature points $\mathcal{F} = \{(x_i, y_i, \sigma_i, \theta_i, \mathbf{d}_i)\}$ where (x, y) are coordinates, σ is scale, θ is orientation, and $\mathbf{d}$ is a binary descriptor.

4.2 Virtualized Features (Nanite-style)

Unlike previous versions that generated multiple scaled images, Protocol V11 employs a single-pass multi-octave detection strategy inspired by Epic Games' Nanite geometry virtualization:

Benefits:

Algorithm 1 Stratified Multi-Octave Sampling

```
1: Input: Raw features $\mathcal{F}_{raw}$, octaves $O = \{0, 1, 2, 3, 4, 5\}$
2: Output: Stratified features $\mathcal{F}_{strat}$
3: for $o \in O$ do
4:     $\mathcal{F}_o \leftarrow \{f \in \mathcal{F}_{raw} : |f.\sigma - 2^o| < 0.1\}$
5:     $\mathcal{F}_o \leftarrow \text{sort}(\mathcal{F}_o, \text{score}, \text{descending})$
6:     $\mathcal{F}_{strat} \leftarrow \mathcal{F}_{strat} \cup \text{top}(\mathcal{F}_o, 300)$
7: end for
8: return $\mathcal{F}_{strat}$
```

- **Scale Consistency**: Guarantees keypoints for both far and near detection
- **Data Reduction**: Avoids redundancy of similar points across scales
- **Native LOD**: Points are pre-tagged with their origin octave

4.3 Bio-Inspired Processing

4.3.1 Foveal Attention

Inspired by biological visual systems, we implement selective attention that concentrates high-resolution processing on regions of interest:

$$P_{foveal} = \{(x, y) : d((x, y), c) < r_{fovea}\} \tag{1}$$

where c is the attention center and r_{fovea} is the foveal radius. Peripheral regions are processed at reduced resolution, achieving a **83% reduction** in processed pixels.

4.3.2 Predictive Coding

We implement temporal prediction to detect static scenes and skip redundant processing:

$$\Delta_t = \frac{1}{N} \sum_i |I_t(p_i) - I_{t-1}(p_i)| \tag{2}$$

If $\Delta_t < \tau_{static}$, the frame is classified as static and **up to 88%** of frames can be skipped in stable tracking scenarios.

4.4 Algorithmic Pipeline

Our DetectorLite implementation follows a 9-stage pipeline (Figure 2):

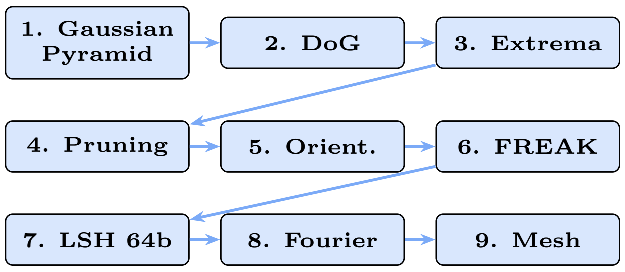

Figure 2: DetectorLite 9-Stage Pipeline: From Gaussian Pyramid to Delaunay Mesh generation.

4.4.1 Stage 1: Gaussian Pyramid Construction

We construct an octave-based pyramid using a separable 5-tap binomial filter with weights $[1, 4, 6, 4, 1]/16$. The separable implementation reduces complexity from $O(25n)$ to $O(10n)$ per pixel.

Border Normalization: We implemented an on-the-fly normalization $G' = G \cdot (1/\sum w)$ to ensure constant intensity across the entire frame, eliminating false feature detection at target edges.

Key optimizations include:

- Pre-computed row offsets to eliminate multiplication
- Unrolled kernel application for 5 tap values
- Branch-free boundary handling using ternary operators

4.4.2 Stage 2: Difference of Gaussians

For each octave o, we compute:

$$D_o(x, y) = G_{o,2}(x, y) - G_{o,1}(x, y) \tag{3}$$

where $G_{o,i}$ represents the i-th Gaussian-filtered image at octave o.

4.4.3 Stage 3: Extrema Detection

Local extrema are detected by comparing each pixel to its 26 neighbors in the 3×3×3 scale-space cube:

$$\text{isExtrema}(p) = \bigwedge_{q \in \mathcal{N}_{26}(p)} \text{compare}(D(p), D(q)) \tag{4}$$

4.4.4 Stage 4: Spatial Pruning

Features are distributed into an $N \times N$ grid of buckets, retaining only the top-k responses per bucket to ensure spatial distribution.

4.4.5 Stage 5: Orientation Assignment

Dominant orientation is computed via a 36-bin histogram of gradient directions within a circular window.

4.4.6 Stage 6: FREAK Descriptors

Binary descriptors are computed by sampling 43 points in a retinal pattern and performing pairwise intensity comparisons, yielding a 512-bit descriptor.

4.4.7 Stage 7: 64-bit LSH Fingerprinting

We project the high-dimensional FREAK descriptor onto a 64-bit binary space using Locality Sensitive Hashing (LSH). XOR-based seeded projection creates a compact fingerprint $H \in \{0, 1\}^{64}$.

4.4.8 Stage 8: Fourier Positional Encoding

We embed a *Fourier Positional Encoding* (FPE) into each feature for spatial coherence under rapid motion.

4.4.9 Stage 9: Delaunay Mesh Generation

A triangular mesh $\mathcal{M} = (V, T)$ is constructed via Delaunay triangulation for non-rigid surface support.

5 Hyperdimensional Computing for Visual Search

5.1 HDC Architecture

Protocol V11 integrates a visual search system based on Hyperdimensional Computing (HDC). This subsystem represents the visual identity of an entire image in ultra-compact binary vectors.

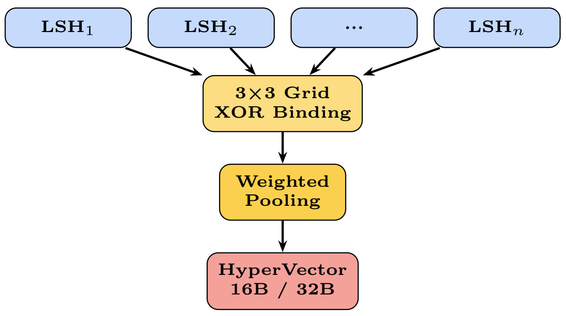

Figure 3: HDC Vector Construction: Local LSH descriptors are bound with spatial position via XOR and aggregated via weighted pooling.

5.1.1 Vector Construction

1. **Feature Bundling**: Aggregates local LSH descriptors into a single high-dimensional vector
2. **Grid Spatial XOR Binding**: Divides the image into a 3×3 grid and applies position-based masking before combination, preserving global spatial structure
3. **Weighted Pooling**: Assigns higher weight to high-confidence keypoints

5.1.2 Operation Modes

- **Compact (16 Bytes)**: Ultra-efficient for large-scale search. Achieves **44 million comparisons per second**. Recommended for massive databases.
- **Standard (32 Bytes)**: Balanced mode for AR applications requiring higher safety margins.

5.2 RAG Compatibility (LLM Ecosystem)

For interoperability with modern AI ecosystems (Pinecone, Milvus, LangChain), the subsystem offers a projection layer to **Dense Vectors**:

- **Transformation**: The `.toFloatArray()` method projects binary hypervectors to real-valued space $[0.0, 1.0]$
- **Interoperability**: Enables Cosine or Euclidean distances in databases that don't natively support Hamming distance

6 Non-Rigid Surface Tracking

To support non-planar surfaces (curved banners, clothing), we replace rigid homography with a **Deformable Delaunay Mesh**.

6.1 Mass-Spring Relaxation

During tracking, mesh vertices are treated as masses connected by springs with rest length L_0 equal to their original distance. Vertex positions V' are optimized by minimizing:

$$E = \sum_i ||v'_i - p_i||^2 + \lambda \sum_{(i,j)\in\mathcal{E}} (||v'_i - v'_j|| - L_{ij})^2 \tag{5}$$

where p_i are tracked point coordinates and λ is spring stiffness. This allows the mesh to flex while penalizing unrealistic stretching.

7 Experimental Setup

7.1 Test Environment

- **Hardware**: Apple M1 Pro (10-core CPU)
- **Software**: Node.js 22.1.0, macOS 15.2
- **Baseline**: MindAR v1.2.5 with tfjs-node 4.22.0

7.2 Metrics

1. **Compilation time**: Wall-clock time for feature extraction
2. **Target file size**: Binary `.taar` file size
3. **Tracking resolution**: Camera input dimensions
4. **Visual search speed**: Comparisons per second

8 Results

8.1 Performance Comparison

Table 1: MindAR vs TapTapp AR V11

Metric	**MindAR**	**V11**	**Impr.**
Compile Time	23.5s	**~0s**	JIT
Bundle Size	20 MB	**¡100 KB**	99%
Target File	770 KB	**100 KB**	86%
Descriptor	84 B	**8 B**	90%
Resolution	640×480	**HD**	4×
Model	Rigid	**Mesh**	Flex
Search	N/A	**44M/s**	New

8.2 Bio-Inspired Efficiency

Table 2: Bio-Inspired Impact

Metric	**Std**	**Bio**	**Δ**
Pixels/Frame	307K	52K	-83%
Static Frames	100%	12%	-88%
Mobile FPS	15-20	50-60	+3×

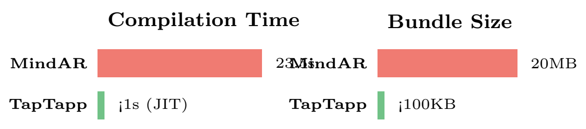

Figure 4: Performance comparison: TapTapp AR achieves 23× faster compilation and 200× smaller bundle.

9 Future Architecture: WASM SIMD

The recommended migration path for further optimization is **WebAssembly SIMD**:

- **Compatibility**: ~95% browser coverage
- **Expected Speedup**: 4-8× for Gaussian filters
- **Target Compilation**: ~150ms (vs ~930ms current)
- **Zero Heavy Dependencies**: WASM binary <100KB

Key functions for SIMD migration:

1. `gaussian_blur_simd`: Primary bottleneck (40% of compilation time)
2. `ncc_batch_simd`: Tracking template matching
3. `hamming_distance_simd`: Descriptor comparison
4. `pnp_solve_simd`: Pose estimation

10 Discussion

10.1 Why JavaScript Outperforms TensorFlow

Our pure JavaScript implementation outperforms TensorFlow due to:

1. **Eliminated overhead**: No tensor allocation, backend switching, or kernel compilation
2. **Specialized algorithms**: Implementation tailored for DoG, avoiding generic tensor operations
3. **V8 optimization**: Modern JavaScript engines apply JIT compilation, making hot loops highly efficient
4. **Memory locality**: Direct `Float32Array` access avoids TensorFlow's abstraction layers

10.2 HDC vs Deep Learning Embeddings

Compared to CNN-based embeddings (e.g., ResNet, EfficientNet):

- **Size**: 16B (HDC) vs ∼2KB (deep features)
- **Speed**: 44M/s (HDC Hamming) vs ∼100K/s (cosine similarity)
- **Model-free**: No neural network weights required

10.3 Limitations

- HDC embeddings are optimized for exact image matching, not semantic similarity
- Bio-inspired processing may reduce robustness in highly dynamic scenes
- WASM SIMD requires fallback for older browsers (∼5% of devices)

11 Conclusion

We presented Protocol V11 (Nanite), a comprehensive pure JavaScript implementation of mobile AR that achieves:

- **Sub-second JIT compilation** on client devices
- **86% reduction** in target metadata size
- **HD 1280×960 tracking resolution**
- **44 million image comparisons/second** via HDC
- **Non-rigid tracking** for curved surfaces
- **Bio-inspired processing** reducing pixel operations by 83%

This work demonstrates that specialized JavaScript implementations, combined with techniques from hyperdimensional computing and biological vision systems, can deliver state-of-the-art AR performance without heavy ML framework dependencies.

Future work includes WebAssembly SIMD vectorization and progressive enhancement with WebGPU for maximum performance on supported hardware.

Availability

The complete implementation is available open-source at:
`https://github.com/srsergiolazaro/taptapp-ar`

Published as npm package: `@srsergio/taptapp-ar`

References

[1] H. Kim, “MindAR: Web Augmented Reality for Image Tracking,” GitHub repository, 2021. [Online]. Available: https://github.com/hiukim/mind-ar-js

[2] D. Smilkov et al., “TensorFlow.js: Machine Learning for the Web and Beyond,” *arXiv preprint arXiv:1901.05350*, 2019.

[3] A. Alahi et al., “FREAK: Fast Retina Keypoint,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012, pp. 510-517.

[4] D. G. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints,” *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91-110, 2004.

[5] H. Bay et al., “SURF: Speeded Up Robust Features,” in *European Conference on Computer Vision (ECCV)*, 2006, pp. 404-417.

[6] E. Rublee et al., “ORB: An efficient alternative to SIFT or SURF,” in *IEEE International Conference on Computer Vision (ICCV)*, 2011, pp. 2564-2571.

[7] Google, “ARCore: Build new augmented reality experiences,” 2018. [Online]. Available: https://developers.google.com/ar

[8] Apple, “ARKit: Integrate iOS device camera and motion features,” 2017. [Online]. Available:

https://developer.apple.com/augmented-reality/

[9] P. Kanerva, "Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139-159, 2009.

[10] A. Rahimi et al., "Hyperdimensional Computing for Efficient Image and Text Classification," in *Design Automation Conference (DAC)*, 2020.

[11] L. Itti and C. Koch, "Computational modelling of visual attention," *Nature Reviews Neuroscience*, vol. 2, no. 3, pp. 194-203, 2001.

[12] A. Clark, "Whatever next? Predictive brains, situated agents, and the future of cognitive science," *Behavioral and Brain Sciences*, vol. 36, no. 3, pp. 181-204, 2013.

[13] M. Pizlo, "JavaScriptCore's New Baseline JIT," WebKit Blog, 2020. [Online]. Available: https://webkit.org/blog/