TIER 1: CRITICAL FILES - ANNOTATED WALKTHROUGHS

MindAR.js Image Recognition Deep Dive

Table of Contents

---

1. IMAGE-LIST.JS - The Foundation

Location: `src/image-target/image-list.js`

Purpose

Creates **image pyramids** at multiple scales. Think of it like having the same photo at different zoom levels.

Why Image Pyramids?

AR needs to recognize markers at different distances from the camera. A poster 1 meter away looks different from 5 meters away. Image pyramids solve this by creating the same image at multiple scales.

Annotated Code:

```
// === buildImageList - For MATCHING (detection) ===
const MIN_IMAGE_PIXEL_SIZE = 100;  // Don't scale images smaller than 100px

const buildImageList = (inputImage) => {
  // Calculate minimum scale: we don't want images smaller than 100px
  // Example: if image is 800px wide, minScale = 100/800 = 0.125 (12.5%)
  const minScale = MIN_IMAGE_PIXEL_SIZE / Math.min(inputImage.width,
inputImage.height);

  // BUILD SCALE LIST
  // Scales grow by 2^(1/3) ≈ 1.26x each step (3 scales per octave)
  const scaleList = [];
  let c = minScale;  // Start from smallest
  while (true) {
    scaleList.push(c);
    c *= Math.pow(2.0, 1.0/3.0);  // Increase by ~26%
    if (c >= 0.95) {  // Stop when we're close to full size
```

```
      c = 1;  // Always include full-size image
      break;
    }
  }
  scaleList.push(c);
  scaleList.reverse();  // Largest first: [1.0, 0.79, 0.63, 0.50, ...]

  // CREATE RESIZED IMAGES
  // Each image is: {data: Uint8Array, width, height, scale}
  const imageList = [];
  for (let i = 0; i < scaleList.length; i++) {
    imageList.push(
      Object.assign(
        resize({image: inputImage, ratio: scaleList[i]}),  // Resize image
        {scale: scaleList[i]}  // Tag with scale factor
      )
    );
  }
  return imageList;  // Array of images at different scales
}

// === buildTrackingImageList - For TRACKING (continuous) ===
// Simpler! Only 2 fixed scales: 256px and 128px
const buildTrackingImageList = (inputImage) => {
  const minDimension = Math.min(inputImage.width, inputImage.height);
  const scaleList = [];

  // Fixed scales: normalize to 256px and 128px
  scaleList.push(256.0 / minDimension);  // Scale to make shortest side = 256px
  scaleList.push(128.0 / minDimension);  // Scale to make shortest side = 128px

  const imageList = [];
  for (let i = 0; i < scaleList.length; i++) {
    imageList.push(
      Object.assign(
        resize({image: inputImage, ratio: scaleList[i]}),
        {scale: scaleList[i]}
      )
    );
  }
  return imageList;  // Always returns 2 images
}
```

Key Takeaways:

- **Matching** uses many scales (3-10 images) for robust detection
- **Tracking** uses only 2 scales (faster, since we already know where the marker is)
- Scale factor example: `0.5` means the image is half the original size

---

2. DETECTOR.JS - The SIFT-like Feature Extractor

Location: `src/image-target/detector/detector.js`

Purpose

Finds **distinctive keypoints** (corners, edges, blobs) in images and computes **FREAK descriptors** (binary fingerprints) for each.

The Big Picture:

This implements a **SIFT-like algorithm**:

1. Build Gaussian pyramid
2. Find Difference-of-Gaussian (DoG) extrema
3. Filter weak/edge responses
4. Compute orientations
5. Extract FREAK descriptors

Key Constants:

```
const PYRAMID_MIN_SIZE = 8;         // Minimum pyramid level size
const PYRAMID_MAX_OCTAVE = 5;       // Maximum 5 octaves

const LAPLACIAN_THRESHOLD = 3.0;    // Minimum feature strength
const EDGE_THRESHOLD = 4.0;         // Reject edge-like features

const NUM_BUCKETS_PER_DIMENSION = 10;  // 10x10 grid
const MAX_FEATURES_PER_BUCKET = 5;     // 5 features per bucket
// Total max features = 10 * 10 * 5 = 500

const ORIENTATION_NUM_BINS = 36;    // 36 orientation bins (10° each)
const FREAK_EXPANSION_FACTOR = 7.0; // Scale FREAK sampling pattern
```

Annotated Code (Main Flow):

```
class Detector {
  constructor(width, height, debugMode = false) {
    this.debugMode = debugMode;
    this.width = width;
    this.height = height;

    // Calculate how many octaves (pyramid levels) we can build
    // Each octave halves the image size
    let numOctaves = 0;
    while (width >= PYRAMID_MIN_SIZE && height >= PYRAMID_MIN_SIZE) {
      width /= 2;
      height /= 2;
      numOctaves++;
      if (numOctaves === PYRAMID_MAX_OCTAVE) break;  // Max 5 octaves
    }
    this.numOctaves = numOctaves;  // Typically 3-5
```

```
  // Caches for tensors and GPU kernels (performance optimization)
  this.tensorCaches = {};
  this.kernelCaches = {};
}

detect(inputImageT) {
  // INPUT: TensorFlow.js tensor of grayscale image
  // OUTPUT: Array of feature points with descriptors

  // === STEP 1: BUILD GAUSSIAN PYRAMID ===
  // Each octave has 2 images: [image1, image2]
  // image2 is more blurred than image1
  const pyramidImagesT = [];
  for (let i = 0; i < this.numOctaves; i++) {
    let image1T;

    if (i === 0) {
      // First octave: blur the input
      image1T = this._applyFilter(inputImageT);
    } else {
      // Higher octaves: downsample the previous octave's most-blurred image
      image1T = this._downsampleBilinear(
        pyramidImagesT[i - 1][pyramidImagesT[i - 1].length - 1]
      );
    }

    // Blur image1 to get image2
    image2T = this._applyFilter(image1T);
    pyramidImagesT.push([image1T, image2T]);
  }
  // Result: pyramidImagesT = [
  //   [octave0_img1, octave0_img2],  // Full resolution
  //   [octave1_img1, octave1_img2],  // 1/2 resolution
  //   [octave2_img1, octave2_img2],  // 1/4 resolution
  //   ...
  // ]

  // === STEP 2: BUILD DoG PYRAMID ===
  // Difference-of-Gaussian = image2 - image1 (finds blobs/edges)
  const dogPyramidImagesT = [];
  for (let i = 0; i < this.numOctaves; i++) {
    let dogImageT = this._differenceImageBinomial(
      pyramidImagesT[i][0],
      pyramidImagesT[i][1]
    );
    dogPyramidImagesT.push(dogImageT);
  }

  // === STEP 3: FIND EXTREMA (local max/min) ===
  // Compare each DoG pixel with 26 neighbors (3x3x3 cube)
  // Only keep if it's a local maximum or minimum
  const extremasResultsT = [];
```

```
for (let i = 1; i < this.numOctaves - 1; i++) {
  // Check octave i against octaves i-1 and i+1
  const extremasResultT = this._buildExtremas(
    dogPyramidImagesT[i - 1],  // Coarser scale
    dogPyramidImagesT[i],      // Current scale
    dogPyramidImagesT[i + 1]   // Finer scale
  );
  extremasResultsT.push(extremasResultT);
}

// === STEP 4: PRUNE EXTREMA ===
// Problem: We might have 10,000+ extrema, too many!
// Solution: Divide image into 10x10 grid (100 buckets)
//           Keep only top 5 strongest extrema per bucket
// Result: Maximum 500 feature points (100 buckets × 5 points)
const prunedExtremasList = this._applyPrune(extremasResultsT);

// === STEP 5: REFINE EXTREMA LOCATIONS ===
// Use sub-pixel localization (quadratic fit) for accuracy
const prunedExtremasT = this._computeLocalization(
  prunedExtremasList,
  dogPyramidImagesT
);

// === STEP 6: COMPUTE ORIENTATIONS ===
// For rotation invariance, find dominant gradient direction around each extrema
// This uses a 36-bin histogram of gradient orientations
const extremaHistogramsT = this._computeOrientationHistograms(
  prunedExtremasT,
  pyramidImagesT
);
const smoothedHistogramsT = this._smoothHistograms(extremaHistogramsT);
const extremaAnglesT = this._computeExtremaAngles(smoothedHistogramsT);
// Result: Each extrema gets an angle (-π to π)

// === STEP 7: EXTRACT FREAK DESCRIPTORS ===
// Sample 37 points around each extrema (6 rings + center)
// These are the "FREAK points" from freak.js
const extremaFreaksT = this._computeExtremaFreak(
  pyramidImagesT,
  prunedExtremasT,
  extremaAnglesT  // Rotate sampling pattern by this angle
);

// === STEP 8: COMPUTE BINARY DESCRIPTORS ===
// Compare all pairs of FREAK points: 37 points = 666 comparisons
// Each comparison = 1 bit (is point A brighter than point B?)
// Result: 666 bits = 83.25 bytes per descriptor
const freakDescriptorsT = this._computeFreakDescriptors(extremaFreaksT);

// === STEP 9: CONVERT TO JAVASCRIPT ARRAYS ===
const prunedExtremasArr = prunedExtremasT.arraySync();
```

```
const extremaAnglesArr = extremaAnglesT.arraySync();
const freakDescriptorsArr = freakDescriptorsT.arraySync();

// === STEP 10: CLEANUP GPU MEMORY ===
// TensorFlow.js requires manual memory management
pyramidImagesT.forEach((ts) => ts.forEach((t) => t.dispose()));
dogPyramidImagesT.forEach((t) => t && t.dispose());
extremasResultsT.forEach((t) => t.dispose());
prunedExtremasT.dispose();
extremaHistogramsT.dispose();
smoothedHistogramsT.dispose();
extremaAnglesT.dispose();
extremaFreaksT.dispose();
freakDescriptorsT.dispose();

// === STEP 11: BUILD OUTPUT ARRAY ===
const featurePoints = [];
for (let i = 0; i < prunedExtremasArr.length; i++) {
  if (prunedExtremasArr[i][0] == 0) continue;  // Skip empty slots

  // Pack 666 bits into 167 integers (4 bits per int)
  const descriptors = [];
  for (let m = 0; m < freakDescriptorsArr[i].length; m += 4) {
    const v1 = freakDescriptorsArr[i][m];
    const v2 = freakDescriptorsArr[i][m + 1];
    const v3 = freakDescriptorsArr[i][m + 2];
    const v4 = freakDescriptorsArr[i][m + 3];
    // Combine 4 bytes into 1 integer
    let combined = v1 * 16777216 + v2 * 65536 + v3 * 256 + v4;
    descriptors.push(combined);
  }

  // Extract position info
  const octave = prunedExtremasArr[i][1];  // Which pyramid level?
  const y = prunedExtremasArr[i][2];       // Row in that level
  const x = prunedExtremasArr[i][3];       // Column in that level

  // Convert back to original image coordinates
  const originalX = x * Math.pow(2, octave) + Math.pow(2, (octave - 1)) - 0.5;
  const originalY = y * Math.pow(2, octave) + Math.pow(2, (octave - 1)) - 0.5;
  const scale = Math.pow(2, octave);

  featurePoints.push({
    maxima: prunedExtremasArr[i][0] > 0,  // Light blob vs dark blob
    x: originalX,        // Pixel x-coordinate
    y: originalY,        // Pixel y-coordinate
    scale: scale,        // Size of feature (pixels)
    angle: extremaAnglesArr[i],  // Orientation (-π to π)
    descriptors: descriptors  // 167-element array of ints
  });
}
```

```
    return { featurePoints, debugExtra };
  }
}
```

Key Implementation Details:

Binomial Filter (Gaussian approximation):

```
_applyFilter(image) {
  // Applies 2D filter: [1,4,6,4,1] × [1,4,6,4,1] / 256
  // This approximates a Gaussian blur
  return tf.engine().runKernel('BinomialFilter', { image });
}
```

Extrema Detection:

```
_buildExtremas(image0, image1, image2) {
  // For each pixel in image1:
  // 1. Check if value > LAPLACIAN_THRESHOLD (strong response)
  // 2. Check if local max/min compared to 26 neighbors
  //    (3×3 in image0, 3×3 in image1, 3×3 in image2)
  // 3. Reject edge responses using Hessian ratio
  // 4. Keep only if passes all tests
  return tf.engine().runKernel('BuildExtremas', { image0, image1, image2 });
}
```

Orientation Computation:

```
_computeOrientationHistograms(prunedExtremasT, pyramidImagesT) {
  // For each extrema:
  // 1. Sample a circular region around it
  // 2. Compute gradient direction at each pixel
  // 3. Build 36-bin histogram weighted by gradient magnitude
  // 4. Find peak bin = dominant orientation
  // This makes the descriptor rotation-invariant!
}
```

Key Takeaways:

- **500 max features** per image (100 buckets × 5)
- **667-bit descriptors** (binary, very fast to compare)
- **Rotation invariant** via orientation normalization
- **Scale invariant** via pyramid processing
- **GPU accelerated** via custom TensorFlow.js kernels

---

3. FREAK.JS - The Descriptor Pattern

Location: `src/image-target/detector/freak.js`

Purpose

Defines the **sampling pattern** for FREAK (Fast Retina Keypoint) descriptors.

What is FREAK?

FREAK mimics the human retina: denser sampling in the center, sparser at the edges.

Annotated Code:

```
// 37 sampling points arranged in 6 concentric rings + 1 center
// Each point has: [sigma, x, y]
// - sigma: Gaussian smoothing width (larger = more blurred)
// - x, y: Normalized coordinates relative to keypoint

const FREAK_RINGS = [
  // RING 5 (outermost, radius ≈ 1.0)
  {
    sigma: 0.550000,  // Heavy smoothing for outer ring
    points: [
      [-1.000000, 0.000000],   // Left
      [-0.500000, -0.866025],  // Upper-left (60° spacing)
      [0.500000, -0.866025],   // Upper-right
      [1.000000, -0.000000],   // Right
      [0.500000, 0.866025],    // Lower-right
      [-0.500000, 0.866025]    // Lower-left
    ]  // 6 points evenly spaced in a circle
  },

  // RING 4 (radius ≈ 0.93)
  {
    sigma: 0.475000,
    points: [
      [0.000000, 0.930969],
      [-0.806243, 0.465485],
      [-0.806243, -0.465485],
      [-0.000000, -0.930969],
      [0.806243, -0.465485],
      [0.806243, 0.465485]
    ]
  },

  // RING 3 (radius ≈ 0.85)
  {
    sigma: 0.400000,
    points: [
      [0.847306, -0.000000],
      [0.423653, 0.733789],
      [-0.423653, 0.733789],
      [-0.847306, 0.000000],
      [-0.423653, -0.733789],
      [0.423653, -0.733789]
```

```
    ]
  },

  // RING 2 (radius ≈ 0.74)
  {
    sigma: 0.325000,
    points: [
      [-0.000000, -0.741094],
      [0.641806, -0.370547],
      [0.641806, 0.370547],
      [0.000000, 0.741094],
      [-0.641806, 0.370547],
      [-0.641806, -0.370547]
    ]
  },

  // RING 1 (radius ≈ 0.60)
  {
    sigma: 0.250000,
    points: [
      [-0.595502, 0.000000],
      [-0.297751, -0.515720],
      [0.297751, -0.515720],
      [0.595502, -0.000000],
      [0.297751, 0.515720],
      [-0.297751, 0.515720]
    ]
  },

  // RING 0 (innermost, radius ≈ 0.36)
  {
    sigma: 0.175000,  // Light smoothing for inner ring
    points: [
      [0.000000, 0.362783],
      [-0.314179, 0.181391],
      [-0.314179, -0.181391],
      [-0.000000, -0.362783],
      [0.314179, -0.181391],
      [0.314179, 0.181391]
    ]
  },

  // CENTER POINT
  {
    sigma: 0.100000,  // Minimal smoothing
    points: [[0, 0]]  // Exactly at the keypoint
  }
];

// Flatten into single array: 37 points total
const FREAKPOINTS = [];
for (let r = 0; r < FREAK_RINGS.length; r++) {
```

```
    const sigma = FREAK_RINGS[r].sigma;
    for (let i = 0; i < FREAK_RINGS[r].points.length; i++) {
      const point = FREAK_RINGS[r].points[i];
      FREAKPOINTS.push([sigma, point[0], point[1]]);
    }
  }
  // Result: FREAKPOINTS.length = 37
```

How FREAK Descriptors Work:

1. **Sample 37 points** around keypoint (rotated by orientation angle)
2. **Compare all pairs** of points: Is point A brighter than point B?
3. **Number of comparisons:** 37 × 36 / 2 = **666 pairwise comparisons**
4. **Encode as binary:** Each comparison = 1 bit → 666 bits total
5. **Pack into integers:** 666 bits ÷ 32 bits/int ≈ 21 integers

Visual Representation:

```
       Ring 5: ● ● ● ● ● ●  (outermost, sigma=0.55)
     Ring 4: ●  ●  ●  ●  ●  ●  (sigma=0.475)
    Ring 3:  ●   ●   ●   ●   ●   ●  (sigma=0.40)
   Ring 2:   ●    ●    ●    ●    ●    ●  (sigma=0.325)
  Ring 1:     ●     ●     ●     ●     ●     ●  (sigma=0.25)
 Ring 0:       ●      ●      ●      ●      ●      ●  (sigma=0.175)
Center:                 ★  (keypoint, sigma=0.10)
```

Example Comparison:

```
// For each pair (i, j) where i < j:
if (intensity[point_i] < intensity[point_j]) {
  bit = 1;
} else {
  bit = 0;
}

// Example: Compare point 0 vs point 1, point 0 vs point 2, etc.
// Total: (37 * 36) / 2 = 666 comparisons = 666 bits
```

Key Takeaways:

- **37 sample points** in retina-like pattern
- **666 binary comparisons** = descriptor
- **Fast matching** via Hamming distance (count differing bits)
- **Rotation handled** by rotating sampling pattern by orientation angle

---

4. MATCHING.JS - Feature Correspondence

Location: `src/image-target/matching/matching.js`

Purpose

Finds which **query features** (from video frame) match which **keyframe features** (from compiled target).

Key Constants:

```
const INLIER_THRESHOLD = 3;         // Max 3 pixels reprojection error
const MIN_NUM_INLIERS = 6;          // Need at least 6 matches
const CLUSTER_MAX_POP = 8;          // K-D tree backtracking limit
const HAMMING_THRESHOLD = 0.7;      // Lowe's ratio test threshold
```

The Matching Pipeline:

1. K-D tree search for candidate matches
2. Hamming distance filtering
3. Hough voting for geometric consistency
4. RANSAC homography estimation
5. Inlier filtering
6. Second pass with homography guidance

Annotated Code:

```
const match = ({keyframe, querypoints, querywidth, queryheight, debugMode}) => {
  // INPUT:
  // - keyframe: Compiled target data with pre-built features
  // - querypoints: Features detected in current video frame
  // - querywidth/height: Video frame dimensions

  // OUTPUT:
  // - H: 3×3 homography matrix (or null if no match)
  // - matches: Array of {querypoint, keypoint} pairs

  // === STEP 1: INITIAL MATCHING ===
  const matches = [];

  for (let j = 0; j < querypoints.length; j++) {
    const querypoint = querypoints[j];

    // Separate maxima (bright blobs) from minima (dark blobs)
    const keypoints = querypoint.maxima
      ? keyframe.maximaPoints
      : keyframe.minimaPoints;

    if (keypoints.length === 0) continue;

    // Get K-D tree root for fast nearest-neighbor search
    const rootNode = querypoint.maxima
      ? keyframe.maximaPointsCluster.rootNode
      : keyframe.minimaPointsCluster.rootNode;

    // === K-D TREE SEARCH ===
    // Find candidate keypoints that might match
    const keypointIndexes = [];
```

```
  const queue = new TinyQueue([], (a1, a2) => {return a1.d - a2.d});
  _query({
    node: rootNode,
    keypoints,
    querypoint,
    queue,
    keypointIndexes,
    numPop: 0
  });

  // === HAMMING DISTANCE MATCHING ===
  // Find best and second-best matches
  let bestIndex = -1;
  let bestD1 = Number.MAX_SAFE_INTEGER;  // Best distance
  let bestD2 = Number.MAX_SAFE_INTEGER;  // Second-best distance

  for (let k = 0; k < keypointIndexes.length; k++) {
    const keypoint = keypoints[keypointIndexes[k]];

    // Compute Hamming distance (count differing bits)
    const d = hammingCompute({
      v1: keypoint.descriptors,
      v2: querypoint.descriptors
    });

    if (d < bestD1) {
      bestD2 = bestD1;
      bestD1 = d;
      bestIndex = keypointIndexes[k];
    } else if (d < bestD2) {
      bestD2 = d;
    }
  }

  // === LOWE'S RATIO TEST ===
  // Only accept match if best is significantly better than second-best
  // This filters ambiguous matches
  if (bestIndex !== -1 &&
      (bestD2 === Number.MAX_SAFE_INTEGER ||
       (bestD1 / bestD2) < HAMMING_THRESHOLD)) {
    matches.push({querypoint, keypoint: keypoints[bestIndex]});
  }
}

// Need at least 6 matches to proceed
if (matches.length < MIN_NUM_INLIERS) return {debugExtra};

// === STEP 2: HOUGH VOTING ===
// Filter matches using geometric consistency
// Groups matches by similar scale/rotation/position
const houghMatches = computeHoughMatches({
  keywidth: keyframe.width,
```

```
  keyheight: keyframe.height,
  querywidth,
  queryheight,
  matches,
});

// === STEP 3: RANSAC HOMOGRAPHY ===
// Estimate 3×3 homography matrix
// Maps keyframe coordinates → query coordinates
const H = computeHomography({
  srcPoints: houghMatches.map((m) => [m.keypoint.x, m.keypoint.y]),
  dstPoints: houghMatches.map((m) => [m.querypoint.x, m.querypoint.y]),
  keyframe,
});

if (H === null) return {debugExtra};

// === STEP 4: INLIER FILTERING ===
// Keep only matches that agree with homography
const inlierMatches = _findInlierMatches({
  H,
  matches: houghMatches,
  threshold: INLIER_THRESHOLD  // 3 pixels
});

if (inlierMatches.length < MIN_NUM_INLIERS) return {debugExtra};

// === STEP 5: SECOND MATCHING PASS ===
// Now that we have a homography, search more exhaustively
const HInv = matrixInverse33(H, 0.00001);  // Invert homography
const dThreshold2 = 10 * 10;  // 10-pixel search radius
const matches2 = [];

for (let j = 0; j < querypoints.length; j++) {
  const querypoint = querypoints[j];

  // Warp query point to keyframe space
  const mapquerypoint = multiplyPointHomographyInhomogenous(
    [querypoint.x, querypoint.y],
    HInv
  );

  let bestIndex = -1;
  let bestD1 = Number.MAX_SAFE_INTEGER;
  let bestD2 = Number.MAX_SAFE_INTEGER;

  const keypoints = querypoint.maxima
    ? keyframe.maximaPoints
    : keyframe.minimaPoints;

  for (let k = 0; k < keypoints.length; k++) {
    const keypoint = keypoints[k];
```

```
    // Only consider keypoints within 10 pixels of warped position
    const d2 = (keypoint.x - mapquerypoint[0]) ** 2 +
               (keypoint.y - mapquerypoint[1]) ** 2;
    if (d2 > dThreshold2) continue;

    // Compute descriptor distance
    const d = hammingCompute({
      v1: keypoint.descriptors,
      v2: querypoint.descriptors
    });

    if (d < bestD1) {
      bestD2 = bestD1;
      bestD1 = d;
      bestIndex = k;
    } else if (d < bestD2) {
      bestD2 = d;
    }
  }

  // Lowe's ratio test again
  if (bestIndex !== -1 &&
      (bestD2 === Number.MAX_SAFE_INTEGER ||
       (bestD1 / bestD2) < HAMMING_THRESHOLD)) {
    matches2.push({querypoint, keypoint: keypoints[bestIndex]});
  }
}

// === STEP 6: REFINED HOMOGRAPHY ===
const houghMatches2 = computeHoughMatches({
  keywidth: keyframe.width,
  keyheight: keyframe.height,
  querywidth,
  queryheight,
  matches: matches2,
});

const H2 = computeHomography({
  srcPoints: houghMatches2.map((m) => [m.keypoint.x, m.keypoint.y]),
  dstPoints: houghMatches2.map((m) => [m.querypoint.x, m.querypoint.y]),
  keyframe,
});

if (H2 === null) return {debugExtra};

const inlierMatches2 = _findInlierMatches({
  H: H2,
  matches: houghMatches2,
  threshold: INLIER_THRESHOLD
});
```

```
  // Return refined results
  return {H: H2, matches: inlierMatches2, debugExtra};
};

// === K-D TREE TRAVERSAL ===
const _query = ({node, keypoints, querypoint, queue, keypointIndexes, numPop}) => {
  // Leaf node: add all points
  if (node.leaf) {
    for (let i = 0; i < node.pointIndexes.length; i++) {
      keypointIndexes.push(node.pointIndexes[i]);
    }
    return;
  }

  // Compute distances to all children
  const distances = [];
  for (let i = 0; i < node.children.length; i++) {
    const childNode = node.children[i];
    const centerPointIndex = childNode.centerPointIndex;
    const d = hammingCompute({
      v1: keypoints[centerPointIndex].descriptors,
      v2: querypoint.descriptors
    });
    distances.push(d);
  }

  // Find minimum distance child
  let minD = Number.MAX_SAFE_INTEGER;
  for (let i = 0; i < node.children.length; i++) {
    minD = Math.min(minD, distances[i]);
  }

  // Queue non-minimum children for potential exploration
  for (let i = 0; i < node.children.length; i++) {
    if (distances[i] !== minD) {
      queue.push({node: node.children[i], d: distances[i]});
    }
  }

  // Recurse into minimum child(ren)
  for (let i = 0; i < node.children.length; i++) {
    if (distances[i] === minD) {
      _query({node: node.children[i], keypoints, querypoint, queue, keypointIndexes,
numPop});
    }
  }

  // Pop up to 8 queued nodes (backtracking)
  if (numPop < CLUSTER_MAX_POP && queue.length > 0) {
    const {node, d} = queue.pop();
    numPop += 1;
    _query({node, keypoints, querypoint, queue, keypointIndexes, numPop});
```

```
  }
};

// === INLIER DETECTION ===
const _findInlierMatches = (options) => {
  const {H, matches, threshold} = options;
  const threshold2 = threshold * threshold;

  const goodMatches = [];
  for (let i = 0; i < matches.length; i++) {
    const querypoint = matches[i].querypoint;
    const keypoint = matches[i].keypoint;

    // Warp keypoint using homography
    const mp = multiplyPointHomographyInhomogenous(
      [keypoint.x, keypoint.y],
      H
    );

    // Check reprojection error
    const d2 = (mp[0] - querypoint.x) ** 2 +
               (mp[1] - querypoint.y) ** 2;

    if (d2 <= threshold2) {
      goodMatches.push(matches[i]);
    }
  }
  return goodMatches;
}
```

Key Concepts:

Hamming Distance:

```
// Counts how many bits differ between two binary descriptors
// Fast: Just XOR and count 1s
const d = hammingCompute({v1: desc1, v2: desc2});
// Result: 0 = identical, 666 = completely different
```

Lowe's Ratio Test:

```
// Reject ambiguous matches
if ((bestD1 / bestD2) < 0.7) {
  // bestD1 is significantly better than bestD2 → good match
} else {
  // Too close → ambiguous → reject
}
```

Homography:

A 3×3 matrix that warps one image plane to another:

```
[x']   [h11 h12 h13] [x]
[y'] = [h21 h22 h23] [y]
[1 ]   [h31 h32 h33] [1]
```

Key Takeaways:

- **Two-pass matching** for better accuracy
- **K-D tree** for fast search (O(log n) instead of O(n))
- **Lowe's ratio test** filters ambiguous matches
- **Hough voting** ensures geometric consistency
- **RANSAC** handles outliers robustly

---

5. MATCHER.JS - High-Level Matching Interface

Location: `src/image-target/matching/matcher.js`

Purpose

Wraps `matching.js` and converts results to 3D coordinates for pose estimation.

Annotated Code:

```
class Matcher {
  constructor(queryWidth, queryHeight, debugMode = false) {
    this.queryWidth = queryWidth;   // Video frame width
    this.queryHeight = queryHeight; // Video frame height
    this.debugMode = debugMode;
  }

  matchDetection(keyframes, featurePoints) {
    // INPUT:
    // - keyframes: Array of compiled target scales (from .mind file)
    // - featurePoints: Detected features from current video frame

    // OUTPUT:
    // - screenCoords: 2D pixel locations in video
    // - worldCoords: 3D positions on target plane (z=0)
    // - keyframeIndex: Which scale matched best

    let debugExtra = {frames: []};
    let bestResult = null;

    // Try matching against all keyframe scales
    for (let i = 0; i < keyframes.length; i++) {
      const {H, matches, debugExtra: frameDebugExtra} = match({
        keyframe: keyframes[i],
        querypoints: featurePoints,
        querywidth: this.queryWidth,
        queryheight: this.queryHeight,
```

```
        debugMode: this.debugMode
      });
      debugExtra.frames.push(frameDebugExtra);

      // Keep best result (most matches)
      if (H) {
        if (bestResult === null || bestResult.matches.length < matches.length) {
          bestResult = {keyframeIndex: i, H, matches};
        }
      }
    }

    if (bestResult === null) {
      return {keyframeIndex: -1, debugExtra};
    }

    // === CONVERT MATCHES TO 3D COORDINATES ===
    const screenCoords = [];  // 2D in video
    const worldCoords = [];   // 3D on target plane
    const keyframe = keyframes[bestResult.keyframeIndex];

    for (let i = 0; i < bestResult.matches.length; i++) {
      const querypoint = bestResult.matches[i].querypoint;
      const keypoint = bestResult.matches[i].keypoint;

      // Screen coordinates (pixels in video)
      screenCoords.push({
        x: querypoint.x,
        y: querypoint.y,
      });

      // World coordinates (normalized units, assuming target is 1 unit wide/tall)
      // The +0.5 and /scale convert from downsampled coordinates to original
      worldCoords.push({
        x: (keypoint.x + 0.5) / keyframe.scale,
        y: (keypoint.y + 0.5) / keyframe.scale,
        z: 0,  // Target is a flat plane
      });
    }

    return {screenCoords, worldCoords, keyframeIndex: bestResult.keyframeIndex,
debugExtra};
  }
}
```

Coordinate Systems:

```
SCREEN COORDS (pixels):         WORLD COORDS (normalized):
  (0,0) -------- (640,0)          (0,0) -------- (1.0,0)
    |              |                |               |
    |   Video      |                |   Target      |
```

```
    |   Frame      |               |   Image       |
    |              |               |               |
 (0,480) ---- (640,480)         (0,1.0) ---- (1.0,1.0)
```

Key Takeaways:

- **Multi-scale matching**: Tries all keyframe scales, keeps best
- **Coordinate conversion**: Pixel → Normalized world coordinates
- **Z = 0**: Assumes target is a flat plane

---

6. ESTIMATOR.JS - 3D Pose Estimation

Location: `src/image-target/estimation/estimator.js`

Purpose

Converts 2D-3D point correspondences into a 6-DOF camera pose (position + rotation).

Annotated Code:

```
class Estimator {
  constructor(projectionTransform) {
    // Projection matrix: converts 3D world → 2D screen
    // This encodes camera intrinsics (focal length, principal point)
    this.projectionTransform = projectionTransform;
  }

  // === INITIAL POSE ESTIMATION ===
  estimate({screenCoords, worldCoords}) {
    // INPUT:
    // - screenCoords: [{x, y}, ...] in pixels
    // - worldCoords: [{x, y, z}, ...] in world units

    // OUTPUT:
    // - modelViewTransform: 4×4 matrix (rotation + translation)

    // Uses DLT (Direct Linear Transformation) + homography decomposition
    const modelViewTransform = estimate({
      screenCoords,
      worldCoords,
      projectionTransform: this.projectionTransform
    });

    return modelViewTransform;
  }

  // === POSE REFINEMENT ===
  refineEstimate({initialModelViewTransform, worldCoords, screenCoords}) {
    // INPUT:
    // - initialModelViewTransform: Initial guess from estimate()
    // - New screenCoords/worldCoords pairs
```

```
    // OUTPUT:
    // - updatedModelViewTransform: Refined 4×4 matrix

    // Uses ICP (Iterative Closest Point) to minimize reprojection error
    const updatedModelViewTransform = refineEstimate({
      initialModelViewTransform,
      worldCoords,
      screenCoords,
      projectionTransform: this.projectionTransform
    });

    return updatedModelViewTransform;
  }
}
```

What These Functions Do:

estimate() - DLT Algorithm:

1. Compute homography H from 2D-2D correspondences
2. Decompose H into rotation R and translation t
3. Construct 4×4 modelView matrix:

```
[r11 r12 r13 tx]
[r21 r22 r23 ty]
[r31 r32 r33 tz]
[0   0   0   1 ]
```

refineEstimate() - ICP:

1. Project 3D world points using current pose
2. Compute reprojection errors
3. Adjust pose to minimize errors
4. Iterate until convergence

The Camera Pipeline:

```
3D World Point → ModelView → Camera Space → Projection → 2D Screen
    [X,Y,Z]         (R,t)      [x,y,z]         (P)       [u,v]
```

ModelView Matrix Decomposition:

```
// A 4×4 matrix encoding both rotation and translation:
//
// [ R11  R12  R13  Tx ]   ← First 3 columns = rotation
// [ R21  R22  R23  Ty ]   ← Last column = translation
// [ R31  R32  R33  Tz ]
// [  0    0    0   1  ]   ← Homogeneous coordinate
```

Key Takeaways:

- **Initial estimate**: Fast but approximate (DLT)
- **Refinement**: Slower but accurate (ICP)
- **Projection matrix**: Camera intrinsics (from calibration or assumed)

---

7. COMPILER-BASE.JS - The Compilation Orchestrator

Location: `src/image-target/compiler-base.js`

Purpose

Converts user-provided marker images into optimized `.mind` files for runtime matching.

Annotated Code:

```
const CURRENT_VERSION = 2;  // .mind file format version

class CompilerBase {
  constructor() {
    this.data = null;  // Will hold compiled data
  }

  compileImageTargets(images, progressCallback) {
    return new Promise(async (resolve, reject) => {

      // === STEP 1: CONVERT TO GRAYSCALE ===
      const targetImages = [];
      for (let i = 0; i < images.length; i++) {
        const img = images[i];

        // Create canvas (browser-specific, subclass implements)
        const processCanvas = this.createProcessCanvas(img);
        const processContext = processCanvas.getContext('2d');
        processContext.drawImage(img, 0, 0, img.width, img.height);
        const processData = processContext.getImageData(0, 0, img.width,
img.height);

        // Convert RGBA to grayscale (simple average)
        const greyImageData = new Uint8Array(img.width * img.height);
        for (let i = 0; i < greyImageData.length; i++) {
          const offset = i * 4;
          greyImageData[i] = Math.floor(
            (processData.data[offset] +     // R
             processData.data[offset + 1] + // G
             processData.data[offset + 2])  // B
            / 3
          );
        }

        targetImages.push({
          data: greyImageData,
          height: img.height,
```

```
        width: img.width
      });
    }

    // === STEP 2: EXTRACT MATCHING DATA (50% progress) ===
    const percentPerImage = 50.0 / targetImages.length;
    let percent = 0.0;
    this.data = [];

    for (let i = 0; i < targetImages.length; i++) {
      const targetImage = targetImages[i];

      // Build multi-scale pyramid
      const imageList = buildImageList(targetImage);

      const percentPerAction = percentPerImage / imageList.length;

      // Extract SIFT-like features + FREAK descriptors for each scale
      const matchingData = await _extractMatchingFeatures(imageList, () => {
        percent += percentPerAction;
        progressCallback(percent);
      });

      this.data.push({
        targetImage: targetImage,
        imageList: imageList,
        matchingData: matchingData  // Array of keyframes with features
      });
    }

    // === STEP 3: BUILD TRACKING IMAGE LISTS ===
    for (let i = 0; i < targetImages.length; i++) {
      const trackingImageList = buildTrackingImageList(targetImages[i]);
      this.data[i].trackingImageList = trackingImageList;
    }

    // === STEP 4: EXTRACT TRACKING DATA (remaining 50%) ===
    // This is delegated to subclass (uses worker in browser)
    const trackingDataList = await this.compileTrack({
      progressCallback,
      targetImages,
      basePercent: 50
    });

    for (let i = 0; i < targetImages.length; i++) {
      this.data[i].trackingData = trackingDataList[i];
    }

    resolve(this.data);
  });
}
```

```
// === EXPORT TO BINARY FORMAT ===
exportData() {
  const dataList = [];
  for (let i = 0; i < this.data.length; i++) {
    dataList.push({
      targetImage: {
        width: this.data[i].targetImage.width,
        height: this.data[i].targetImage.height,
        // Note: pixel data NOT exported (too large)
        // It's rebuilt at runtime from target dimensions
      },
      trackingData: this.data[i].trackingData,
      matchingData: this.data[i].matchingData
    });
  }

  // Serialize using MessagePack (compact binary format)
  const buffer = msgpack.encode({
    v: CURRENT_VERSION,  // Version 2
    dataList
  });

  return buffer;  // Uint8Array ready to save as .mind file
}

// === IMPORT FROM BINARY FORMAT ===
importData(buffer) {
  const content = msgpack.decode(new Uint8Array(buffer));

  // Version check
  if (!content.v || content.v !== CURRENT_VERSION) {
    console.error("Your compiled .mind might be outdated. Please recompile");
    return [];
  }

  const { dataList } = content;
  this.data = [];
  for (let i = 0; i < dataList.length; i++) {
    this.data.push({
      targetImage: dataList[i].targetImage,
      trackingData: dataList[i].trackingData,
      matchingData: dataList[i].matchingData
    });
  }

  return this.data;
}

// Subclass implements (browser vs Node.js)
createProcessCanvas(img) {
  console.warn("missing createProcessCanvas implementation");
}
```

```
  compileTrack({progressCallback, targetImages, basePercent}) {
    console.warn("missing compileTrack implementation");
  }
}

// === FEATURE EXTRACTION HELPER ===
const _extractMatchingFeatures = async (imageList, doneCallback) => {
  const keyframes = [];

  for (let i = 0; i < imageList.length; i++) {
    const image = imageList[i];

    // Create detector for this scale
    const detector = new Detector(image.width, image.height);

    await tf.nextFrame();  // Yield to event loop (prevent freezing)

    tf.tidy(() => {  // Auto-cleanup GPU memory
      // Convert to tensor
      const inputT = tf.tensor(image.data, [image.data.length], 'float32')
                       .reshape([image.height, image.width]);

      // Detect features
      const { featurePoints: ps } = detector.detect(inputT);

      // Separate maxima and minima
      const maximaPoints = ps.filter((p) => p.maxima);
      const minimaPoints = ps.filter((p) => !p.maxima);

      // Build K-D trees for fast searching
      const maximaPointsCluster = hierarchicalClusteringBuild({
        points: maximaPoints
      });
      const minimaPointsCluster = hierarchicalClusteringBuild({
        points: minimaPoints
      });

      // Store keyframe data
      keyframes.push({
        maximaPoints,
        minimaPoints,
        maximaPointsCluster,
        minimaPointsCluster,
        width: image.width,
        height: image.height,
        scale: image.scale
      });

      doneCallback(i);
    });
  }
```

```
  return keyframes;  // Array of keyframes, one per scale
}
```

Data Structure:

```
// Compiled .mind file structure:
{
  v: 2,  // Version
  dataList: [
    {
      targetImage: {width, height},
      matchingData: [  // Multi-scale keyframes
        {
          maximaPoints: [{x, y, scale, angle, descriptors}, ...],
          minimaPoints: [{x, y, scale, angle, descriptors}, ...],
          maximaPointsCluster: {rootNode, ...},  // K-D tree
          minimaPointsCluster: {rootNode, ...},
          width, height, scale
        },
        // ... more scales
      ],
      trackingData: [...]  // Tracking-specific features
    },
    // ... more targets
  ]
}
```

Progress Distribution:

```
0% ──────────────────────────────────────────────── 100%
│                                                    │
├────────────────────┼───────────────────────────────┤
   Matching (50%)           Tracking (50%)
```

Key Takeaways:

- **Matching data**: Multi-scale SIFT-like features with K-D trees
- **Tracking data**: Simpler features for continuous tracking
- **Binary format**: MessagePack for compact storage
- **Version control**: Prevents loading incompatible files

---

8. COMPILER.JS - Browser-Specific Compiler

Location: `src/image-target/compiler.js`

Purpose

Browser implementation that uses HTML5 Canvas and Web Workers.

Annotated Code:

```
class Compiler extends CompilerBase {

  // === BROWSER-SPECIFIC CANVAS CREATION ===
  createProcessCanvas(img) {
    // Creates an HTML5 Canvas element
    const processCanvas = document.createElement('canvas');
    processCanvas.width = img.width;
    processCanvas.height = img.height;
    return processCanvas;
  }

  // === OFFLOAD TRACKING COMPILATION TO WORKER ===
  compileTrack({progressCallback, targetImages, basePercent}) {
    return new Promise((resolve, reject) => {

      // Spawn Web Worker (runs in separate thread)
      const worker = new CompilerWorker();

      // Handle messages from worker
      worker.onmessage = (e) => {
        if (e.data.type === 'progress') {
          // Worker reports progress
          progressCallback(basePercent + e.data.percent * basePercent/100);
        }
        else if (e.data.type === 'compileDone') {
          // Worker finished
          resolve(e.data.list);  // Array of tracking data
        }
      };

      // Send task to worker
      worker.postMessage({
        type: 'compile',
        targetImages  // Grayscale image data
      });
    });
  }
}
```

Why Use a Web Worker?

```
Without Worker:                    With Worker:
  Main Thread                        Main Thread        Worker Thread
  ============                       ===========        =============
  Compile (5s) → UI FROZEN           Start Worker  →    Compile (5s)
                                     UI Responsive!
                                     ← Progress updates
                                     ← Results
```

Worker Communication:

```
// Main thread → Worker
worker.postMessage({
  type: 'compile',
  targetImages: [{data, width, height}, ...]
});

// Worker → Main thread
postMessage({type: 'progress', percent: 25});
postMessage({type: 'compileDone', list: [trackingData1, trackingData2, ...]});
```

Key Takeaways:

- **Canvas**: Browser-specific image processing
- **Web Worker**: Prevents UI freezing during compilation
- **Message passing**: Progress updates and results
- **Asynchronous**: Returns a Promise

Summary: The Complete Flow

Compilation (Offline):

```
User Images
    ↓
[Compiler] Convert to grayscale
    ↓
[buildImageList] Create multi-scale pyramid (3-10 scales)
    ↓
[Detector] Extract SIFT-like features + FREAK descriptors
    │
    ├─ Build Gaussian pyramid
    ├─ Compute DoG (Difference of Gaussian)
    ├─ Find extrema (local max/min)
    ├─ Prune to 500 features max
    ├─ Compute orientations
    └─ Extract 666-bit FREAK descriptors
    ↓
[hierarchicalClustering] Build K-D trees for fast search
    ↓
[exportData] Serialize to .mind file (MessagePack)
```

Runtime Detection (Online):

```
Video Frame
    ↓
[InputLoader] Grayscale conversion (WebGL)
    ↓
[Detector] Extract features from frame
```

```
    │
    └─ Same process as compilation
    ↓
[Matcher] Match against compiled keyframes
    │
    ├─ K-D tree search (fast candidate selection)
    ├─ Hamming distance (count differing bits)
    ├─ Lowe's ratio test (reject ambiguous matches)
    ├─ Hough voting (geometric consistency)
    └─ RANSAC homography (robust estimation)
    ↓
[Estimator] DLT pose estimation
    │
    ├─ Compute homography from 2D-2D matches
    ├─ Decompose into rotation + translation
    └─ Return 4×4 modelView matrix
    ↓
[Tracker] Template matching (next frames)
    │
    └─ More efficient than full detection
    ↓
[Estimator] ICP refinement
    │
    ├─ Project 3D points using current pose
    ├─ Compute reprojection errors
    └─ Minimize errors iteratively
    ↓
6-DOF Pose (rotation + translation)
    │
    └─ Used to render 3D objects on target
```

Key Data Structures:

Feature Point:

```
{
  maxima: true,              // Light blob (vs dark)
  x: 245.5,                  // Pixel x-coordinate
  y: 123.8,                  // Pixel y-coordinate
  scale: 4.0,                // Feature size
  angle: 1.57,               // Orientation (radians)
  descriptors: [int, ...]    // 167 integers (666 bits)
}
```

Keyframe (Compiled):

```
{
  maximaPoints: [feature, ...],      // Light features
  minimaPoints: [feature, ...],      // Dark features
  maximaPointsCluster: {rootNode},   // K-D tree
  minimaPointsCluster: {rootNode},   // K-D tree
```

```
  width: 640,
  height: 480,
  scale: 0.5                          // Downsampling factor
}
```

Match:

```
{
  querypoint: {x, y, descriptors},   // From video frame
  keypoint: {x, y, descriptors}      // From compiled target
}
```

Homography Matrix (3×3):

```
[
  [h11, h12, h13],   // Maps target → video
  [h21, h22, h23],
  [h31, h32, h33]
]
```

ModelView Matrix (4×4):

```
[
  [r11, r12, r13, tx],   // Rotation + translation
  [r21, r22, r23, ty],
  [r31, r32, r33, tz],
  [ 0,   0,   0,   1]
]
```

Performance Optimizations:

1. **Image Pyramids**: Multi-scale processing for scale invariance
2. **K-D Trees**: O(log n) search instead of O(n)
3. **GPU Kernels**: WebGL shaders + TensorFlow.js
4. **Web Workers**: Parallel processing without blocking UI
5. **Bucket Pruning**: Limit to 500 features max
6. **Binary Descriptors**: Fast Hamming distance (XOR + popcount)
7. **Two-Pass Matching**: Homography-guided refinement
8. **Template Tracking**: Avoid full detection every frame

Algorithm Summary:

Component	Algorithm	Purpose
Detection	SIFT-like DoG	Find keypoints
Descriptor	FREAK (666-bit)	Binary fingerprint
Matching	K-D tree + Hamming	Find correspondences

Filtering	Lowe's ratio	Reject ambiguous
Consistency	Hough voting	Geometric check
Homography	RANSAC	Robust estimation
Pose	DLT + ICP	6-DOF transform

Typical Performance:

- **Compilation**: 2-10 seconds per target (one-time)
- **Detection**: 30-60 FPS (first detection)
- **Tracking**: 60+ FPS (continuous)
- **Memory**: ~5-20 MB per compiled target
- **Features**: ~100-500 per image

Glossary

DoG (Difference-of-Gaussian): Subtraction of two blurred images; approximates Laplacian for blob detection

Extrema: Local maxima or minima in DoG space; candidate keypoints

FREAK: Fast Retina Keypoint; binary descriptor using retina-like sampling pattern

Hamming Distance: Count of differing bits between two binary strings

Hough Voting: Geometric consistency check using scale/rotation/position bins

Homography: 3×3 matrix warping one planar surface to another

ICP: Iterative Closest Point; refines pose by minimizing reprojection error

K-D Tree: Space-partitioning data structure for fast nearest-neighbor search

Lowe's Ratio Test: Reject match if best/second-best ratio > threshold

ModelView Matrix: 4×4 transform from world coordinates to camera coordinates

Octave: Doubling of scale in pyramid (image size halves)

Projection Matrix: 3×4 or 4×4 matrix converting 3D camera coords to 2D screen

RANSAC: Random Sample Consensus; robust estimation ignoring outliers

SIFT: Scale-Invariant Feature Transform; classic keypoint detector

---

File Dependencies

```
compiler.js
    ↓ extends
compiler-base.js
    ↓ uses
image-list.js
    ↓ uses
```

```
detector/detector.js
    ↓ uses
detector/freak.js

matching/matcher.js
    ↓ uses
matching/matching.js
    ↓ uses
matching/hierarchical-clustering.js
matching/hamming-distance.js
matching/hough.js
matching/ransacHomography.js

estimation/estimator.js
    ↓ uses
estimation/estimate.js
estimation/refine-estimate.js
```

Next Steps

To achieve full control of the image recognition system:

Tier 2 (Supporting Systems):

- **tracker/tracker.js** - Template-based continuous tracking
- **input-loader.js** - WebGL video frame preprocessing
- **controller.worker.js** - Worker thread orchestration

Tier 3 (Deep Dive):

- **matching/hierarchical-clustering.js** - K-D tree implementation
- **estimation/estimate.js** - Homography decomposition math
- **estimation/refine-estimate.js** - ICP algorithm details
- **detector/kernels/** - CPU vs GPU implementations

Tier 4 (Advanced):

- **matching/hough.js** - Geometric voting implementation
- **matching/ransacHomography.js** - Robust homography estimation
- **utils/geometry.js** - Matrix math utilities

Document generated for MindAR.js v1.2.5 Focus: Image Target Recognition System