ESP32-S3 Vision (Edge Mask + Template Matching)

This repository contains an ESP-IDF firmware project for an ESP32-S3 camera module plus a Python GUI for live raw video and status. The firmware performs lightweight grayscale vision, supports a train/exec workflow using template IoU, and exposes HTTP endpoints for raw frames and control.

Repository layout

- main/ - ESP-IDF firmware source (single-file application)
- tools/ - Python GUI client and requirements
- managed_components/ - ESP-IDF managed components
- CMakeLists.txt - top-level ESP-IDF project file
- sdkconfig - project configuration

Firmware architecture

Tasks

- vision_task: camera capture + vision pipeline at a fixed interval.
- uart_task: optional serial command input for training and control.

Camera lifecycle

The camera is off by default to avoid allocation failures. It is enabled on demand:

- /cmd?op=cam_on -> initializes the camera and starts vision capture.
- /cmd?op=cam_off -> deinitializes the camera and stops capture.

Internal state:

- s_cam_inited: camera driver initialized state.
- s_cam_enabled: capture enabled state (vision_task checks this).

Vision pipeline (per frame)

1. Capture a grayscale frame (QQVGA 320x240 by default).
2. Compute an edge mask line at bottom EDGE_PERCENT of the frame.
3. Background learning (optional): exponential moving average model.
4. Motion map: decay + boost on per-pixel deltas.
5. Foreground detection per grid block (BLOCK_SIZE).
6. Largest connected blob is selected (grid flood fill).
7. Bounding box is computed and left/center/right classified.
8. Template IoU matching to label as obstacle vs anomaly.
9. Overlay is drawn on the raw buffer for visualization.

Math details (detection core)

This section describes the core math as implemented in main/main.c using the compile-time knobs.

Notation

- $I_t(x, y)$: current grayscale frame at time t.
- $B_t(x, y)$: background model (EMA).
- $M_t(x, y)$: motion energy buffer.
- Block size: $S = BLOCK_SIZE$.
- Foreground decision uses threshold constants: BG_DIFF_THRESH, MOTION_THRESH, MOTION_DECAY, MOTION_BOOST, MOTION_LEVEL.

Region of interest and edge mask

Only the upper part of the frame participates in detection. The "edge" row is:

$$extedge_y = H - \lfloor H \cdot \frac{EDGE_PERCENT}{100} \rfloor$$

The grid height is computed from this top region, so any pixels below edge_y are ignored for blob detection.

The effective ROI height is:

$$H_{roi} = edge_y$$

Grid dimensions are:

$$G_w = \lfloor \frac{W}{S} \rfloor, \quad G_h = \lfloor \frac{H_{roi}}{S} \rfloor$$

Background learning (EMA)

When learning is enabled, the background model updates with a slow EMA:

$$B_t = B_{t-1} + \lfloor \frac{I_t - B_{t-1}}{8} \rfloor$$

This is an integer approximation of an EMA with $\alpha \approx 1/8$.

Motion energy

Let $\Delta_t = \left| I_t - I_{t-1} \right|$. Motion energy is updated per pixel as:

$$M_t = \{ min\left(255, M_{t-1} + MOTION_BOOST\right) \; if \; \Delta_t > MOTION_THRESH \; max\left(0, M_{t-1} - MOTION_DECAY\right)$$

This produces persistence (decay) and burst response (boost) so motion survives a few frames.

Foreground decision

If a background model exists, per pixel foreground is:

$$extfg = \left(\left|I_t - B_t\right| > BG_DIFF_THRESH\right) \ \vee \ \left(M_t > MOTION_LEVEL\right)$$

This combines background subtraction and the motion energy buffer.

Grid downsampling

The frame is partitioned into $S \times S$ blocks. A block is set if the number of foreground pixels in the block exceeds one quarter of its area:

$$extgrid\left(b_x, b_y\right) = 1 \ if \ fg_count > \frac{S^2}{4}$$

Change ratio guard

The total changed ratio is computed as:

$$r = \frac{fg_pixels}{total_pixels}$$

If r exceeds MAX_CHANGED_RATIO, blob detection is skipped to avoid triggering on full-frame changes.

End-to-end decision rule

Let A^* be the cell count of the largest connected component in the grid. The object-present flag is:

$$extobject_present = (r \leq MAX_CHANGED_RATIO) \wedge \left(A^* \geq MIN_BLOB_CELLS\right)$$

If this is false, bbox and label fields are set to “none”.

Largest connected component

The grid is scanned with a 4-neighbor flood fill to find connected components. The largest component (by cell count) is selected as the candidate obstacle region. If its area is below MIN_BLOB_CELLS, the frame is treated as “no object”.

Bounding box and side classification

Let $(minx, miny)$ and $(maxx, maxy)$ be the grid bounds of the largest blob. Pixel-space box is:

$$x_1 = minx \cdot S, \ y_1 = miny \cdot S, \ x_2 = (maxx + 1) \cdot S - 1, \ y_2 = (maxy + 1) \cdot S - 1$$

The box center is compared to a left/right deadband:

$$extleft_bound = \frac{W}{2} - \frac{W}{10}, \ right_bound = \frac{W}{2} + \frac{W}{10}$$

If $c_x < left_bound$ -> left, $c_x > right_bound$ -> right, otherwise center.

Template IoU matching

Templates are binary grids at the same resolution as the current grid. For each template T and grid G:

$$extIoU(G,T) = \frac{\sum_i [G_i \wedge T_i]}{\sum_i [G_i \vee T_i]}$$

If the best IoU across templates meets TEMPLATE_IOU_THRESH, the object is labeled "obstacle"; otherwise it is "anomaly".

Computational cost (order of magnitude)

- Per-pixel operations: $O(W \cdot H)$ for motion + background checks.
- Grid conversion: $O(W \cdot H)$ but with simple integer ops.
- Flood fill: $O(G_w \cdot G_h)$ worst case.
- IoU matching: $O(T \cdot G_w \cdot G_h)$ for T templates.

Mounting angle and ground coverage

Assume the ESP32-S3 Sense lens has diagonal FOV 66.5° and a 4:3 aspect ratio. For a pinhole model:

$$an\left(\frac{H}{2}\right) = \frac{4}{5} tan\left(\frac{D}{2}\right), \quad an\left(\frac{V}{2}\right) = \frac{3}{5} tan\left(\frac{D}{2}\right)$$

With $D = 66.5^\circ$, the approximate fields of view are:

$$H \approx 55.2^\circ, \quad V \approx 43.0^\circ$$

Let the camera height be $h = 17.5$ in $(0.4445$ m) and the downward pitch from horizontal be θ. The ground intersection distances for the top and bottom rays are:

$$D_{far} = \frac{h}{tan\left(\theta - \frac{V}{2}\right)}, \quad D_{near} = \frac{h}{tan\left(\theta + \frac{V}{2}\right)}$$

A good obstacle-avoidance mount for roughly 1 ft to 6 ft of ground coverage is:

$$heta \approx 35^\circ$$

which yields:

$$D_{near} \approx 11\ in, \quad D_{far} \approx 72\ in$$

If you want a specific near/far pair, solve:

$$heta = \frac{1}{2}\left(arctan\left(\frac{h}{D_{near}}\right) + arctan\left(\frac{h}{D_{far}}\right)\right)$$

Approximate object width and side using the bottom-of-frame scale

The robot's front edge visible at the bottom row is $W_{robot} = 18.875$ in. If it spans $w_{robot,px}$ pixels at the bottom of the image, the bottom-row scale is:

$$s_{bottom} = \frac{W_{robot}}{w_{robot,px}} \quad (inches\ per\ pixel)$$

For an object whose lowest visible width is $w_{obj,px}$ pixels near the bottom row, approximate its physical width as:

$$W_{obj} \approx w_{obj,px} \cdot s_{bottom}$$

The lateral offset of the object's base center from the robot centerline is:

$$x_{offset} \approx \left(x_c - \frac{W}{2}\right) \cdot s_{bottom}$$

If $x_{offset} < 0$ the object is left; if $x_{offset} > 0$ it is right.

Depth-aware version (if you want better accuracy)

For a base pixel row y, convert to a vertical angle from the optical axis:

$$\phi(y) = arctan\left(\left(\frac{2y}{H} - 1\right)tan\left(\frac{V}{2}\right)\right)$$

Then estimate ground distance:

$$D(y) = \frac{h}{tan(\theta + \phi(y))}$$

Horizontal span at that distance is:

$$W_{span}(y) = 2D(y)tan\left(\frac{H}{2}\right)$$

and inches-per-pixel is $s(y) = W_{span}(y)/W$. Use $s(y)$ in place of s_{bottom} for width and side estimates.

Robot front edge calibration (metric + pixels)

The robot front edge is assumed to appear as a dark line near the bottom of the frame:

- Edge physical width: 15.5 in (0.3937 m)
- Edge vertical drop from camera: 8.0 in (0.2032 m)
- Camera height: 17.5 in (0.4445 m)

- Camera pitch: 30° down from horizontal

There are two bumps (E-stop switches) on the edge, separated by:

- 13. 0 in (0. 3302 m)

Calibration run

1) Turn on the camera and make sure the edge line and bumps are visible.
2) Call `/cmd?op=calib` (or send `calib` over UART).
3) The device logs the detected edge line row, edge width in pixels, and the bump pixel locations.

The firmware stores these values in a calibration struct for use in side classification and scaling. If you want to hardcode the values later, copy the logged pixel positions into constants.

Side approximation using bumps

After calibration, left/right boundaries are defined using the bump pixel positions on the edge. If the object center is left of the left bump it is “left”; if right of the right bump it is “right”; otherwise it is “center”.

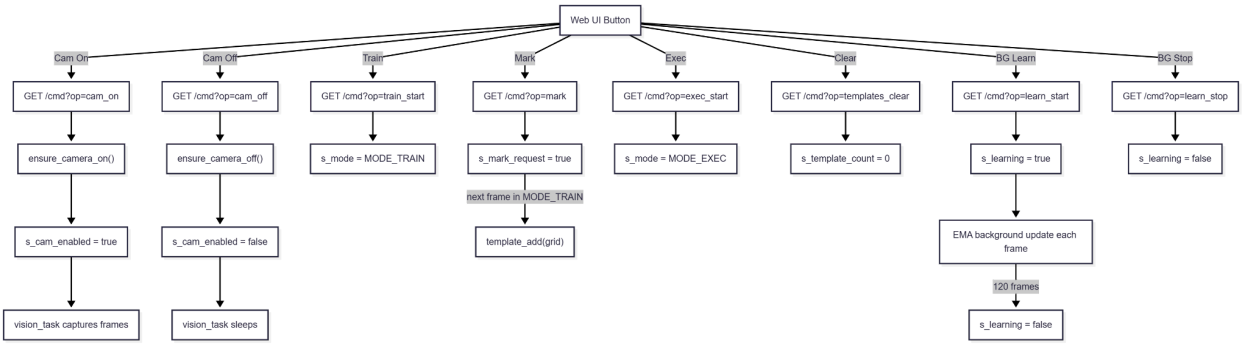
Ground grid overlay

The raw frame includes a projected ground grid for visualization. The grid uses the camera height, pitch, and FOV to render a warped grid that approximates the ground plane in the image.

Design intent and tradeoffs

- Grayscale + block grid keeps compute and memory low on ESP32-S3 while preserving coarse shape.
- Combining background subtraction with motion energy improves robustness when the background drifts.
- The change ratio guard avoids false positives when exposure changes or the camera is obstructed.
- Template IoU is a cheap shape matcher suitable for small template sets.
- Raw /raw streaming avoids JPEG heap pressure at the expense of bandwidth.

Web UI button flowchart



Web UI controls (training-only)

The HTML server is a temporary training and diagnostics surface. Final builds will not include the web UI; use UART commands for training and control instead.

Each button is a thin wrapper around a `/cmd?op=...` request and maps directly to internal state in the vision pipeline:

Train

- HTTP op: `train_start`.
- Effect: sets `s_mode = MODE_TRAIN` (see `vision_set_mode`).
- Pipeline behavior: while in TRAIN mode, the next `Mark` toggles `s_mark_request` and causes the current binary grid (`s_grid`) to be stored as a template on the next frame.
- When to use: before capturing new templates for a scene. This does not change the background model; it only changes how the template buffer is populated.

Mark

- HTTP op: `mark`.
- Effect: sets `s_mark_request = true`.
- Pipeline behavior: on the next frame *in TRAIN mode*, the grid generated by the foreground pipeline is copied into the template store (`template_add`). The template is a binary grid at the current grid resolution (`G_w x G_h`) and represents the object's coarse shape.
- When to use: with the target object present and stable in view. Repeat to capture multiple templates covering view angles or shapes. If `MODE_EXEC` is active, `Mark` is ignored (no template is added).

Exec

- HTTP op: `exec_start`.
- Effect: sets `s_mode = MODE_EXEC`.
- Pipeline behavior: each frame is labeled by template IoU. If the best IoU across stored templates meets `TEMPLATE_IOU_THRESH`, the object is labeled `obstacle`; otherwise it is labeled `anomaly`. If no templates exist, all detections are `anomaly`.
- When to use: after training to enable runtime classification.

Clear

- HTTP op: `templates_clear`.
- Effect: clears the stored template buffer and resets the template count to zero.
- Pipeline behavior: template IoU matching is effectively disabled until new templates are added.
- When to use: to reset training, or when switching to a new environment/object set.

BG Learn

- HTTP op: `learn_start`.
- Effect: sets `s_learning = true` and resets the background model if needed.
- Pipeline behavior: background is updated each frame via an integer EMA:

$$B_t = B_{t-1} + \lfloor \frac{I_t - B_{t-1}}{8} \rfloor$$

This slowly converges to the static scene. A counter (`s_learn_frames`) auto-stops learning after 120 frames to avoid drift. - When to use: in a stable scene with no objects in the ROI. Run for a few seconds, then stop.

BG Stop

- HTTP op: `learn_stop`.
- Effect: sets `s_learning = false` immediately.
- Pipeline behavior: freezes the background model so foreground detection uses a stable reference.
- When to use: after a short BG Learn window so moving objects do not get absorbed into the background model.

Template matching

Templates are stored as binary grids at the same grid resolution as the current frame. A match is accepted if IoU >= TEMPLATE_IOU_THRESH. Up to MAX_TEMPLATES templates are kept; older ones are evicted when full.

Memory model

Large buffers are allocated in PSRAM when available, then internal RAM as fallback:

- s_prev, s_motion, s_bg: full-frame buffers.
- s_raw_buf: latest raw grayscale frame exposed via HTTP.
- s_templates: template grid buffers.

HTTP API

The HTTP server runs on port 80. Endpoints:

- GET /
  - Simple web UI with raw canvas stream and buttons.
- GET /raw

    - Raw grayscale bytes (w*h).
    - Response headers: X-Width, X-Height.
- GET /status
    - JSON detection summary:
        - object_present, kind, side, bbox, edge_y, frame_id, mode, templates, cam_enabled
        - edge_line_y, edge_width_px, edge_scale_m
        - obj_width_px, obj_width_m, obj_offset_m
        - calib_valid, bump_left_x, bump_right_x, bump_sep_px
- GET /cmd?op=…
    - Control operations:
        - cam_on, cam_off
        - train_start, train_stop
        - exec_start
        - mark
        - templates_clear, templates_list
        - learn_start, learn_stop, bg_clear

Optional (disabled by default):

- GET /jpg
- GET /mjpeg

These are guarded by ENABLE_JPEG_STREAM (set to 0 by default to avoid malloc failures on JPEG conversion).

UART commands

UART input mirrors the HTTP commands (one per line):

- learn_start, learn_stop, bg_clear
- train_start, train_stop, exec_start
- mark
- templates_clear, templates_list
- status

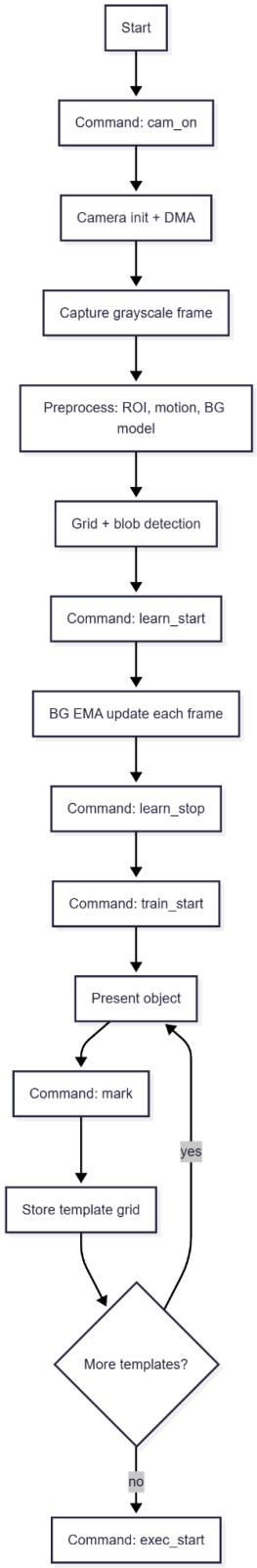
Implementation evolution

The vision pipeline evolved across three implementation stacks. This table summarizes the technical progression and why each transition happened.

Stage	Toolchain / SDK	Language / runtime	Vision approach	Data path / I/O	Concurrency model	Memory strategy	Limits observed	Reason for transition

Arduino	Arduino IDE + arduino-esp32	C++ on Arduino core	Edge detection + smoothing/threshold filters (early prototypes)	Basic camera driver with frame copies into CPU buffers	Single loop() with blocking processing	Minimal buffers in DRAM	Limited headroom for heavier algorithms and tuning; harder to modularize	Needed tighter control over compute and memory, plus better tooling
PlatformIO	PlatformIO + Arduino core	C++ on Arduino core	Expanded filter stack, parameter sweeps, refactoring of stages	Same camera driver; improved build/test workflow	Still largely single-threaded	Similar DRAM usage, limited multi-buffering	Arduino-core constraints remained; integration of advanced libs harder	Moved to native ESP-IDF for DMA + FreeRTOS and deeper control
ESP-IDF	ESP-IDF v6.x + esp32-camera	C (native)	Grayscale ROI + BG EMA + motion energy + grid/blob + template IoU	DMA camera input; raw stream buffer; UART + HTTP training control	FreeRTOS tasks (vision_task, uart_task)	Full-frame buffers + template grids; PSRAM preferred, internal fallback	More complex pipeline but stable; training surface is temporary	Enables heavier algorithms, predictable timing, and direct GPIO control

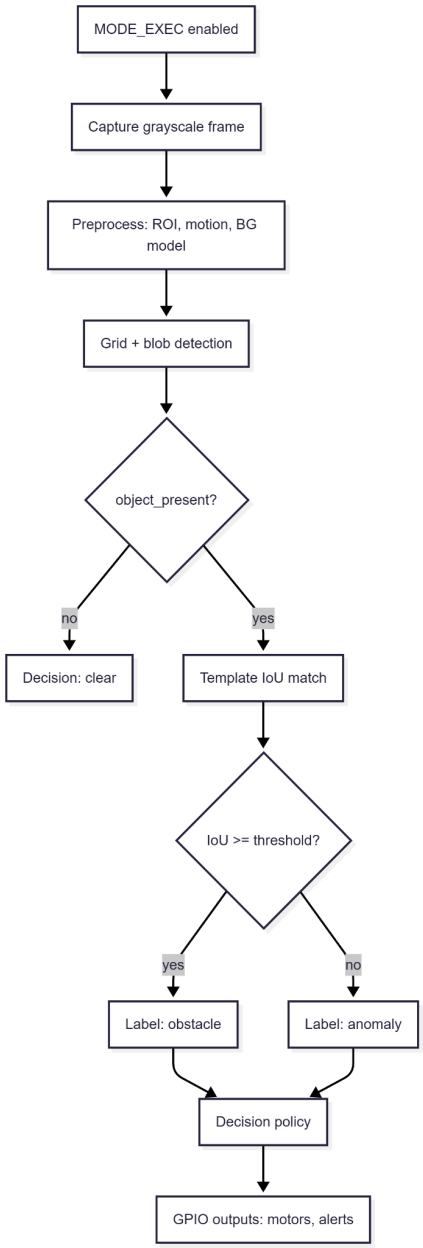
Training workflow



Typical training flow:

1. cam_on
2. train_start
3. Place an obstacle and press mark (repeat for multiple templates).
4. exec_start

In exec mode, detections are labeled as obstacles if they match a template; otherwise anomaly.



Configuration knobs (compile-time)

These are defined in main/main.c and can be tuned as needed:

- FRAME_WIDTH, FRAME_HEIGHT
- EDGE_PERCENT
- BLOCK_SIZE
- BG_DIFF_THRESH, MOTION_THRESH, MOTION_DECAY, MOTION_BOOST, MOTION_LEVEL
- MIN_BLOB_CELLS, MAX_CHANGED_RATIO
- JPEG_QUALITY, JPEG_INTERVAL, ENABLE_JPEG_STREAM
- WIFI_AP_SSID, WIFI_AP_PASS, WIFI_STA_SSID, WIFI_STA_PASS
- WIFI_STA_EAP_ENABLED and EAP identity/username/password

Note: WPA2-Enterprise requires esp_eap_client.h to be present in the ESP-IDF installation.

Testing and evaluation

This section captures observed behavior from current field tests and provides a structured format for ongoing evaluation. Rows labeled “Synthetic” are illustrative placeholders (not measured) to show how rigorous reporting should look.

Observed behavior (user report, May 2026)

- Trained 10 objects, each presented ~3 times for ~3 seconds, alternating.
- In EXEC, 9 objects were recognized immediately; the 10th required additional time and failed twice before recognition.
- False positives occurred for ~10-15 seconds over ~5 minutes of operation with no objects in view.

Run ID	Objects trained	Presentation protocol	Immediate recog.	Delayed recog.	Misses	False-positive time	Notes
R-2026-05-04-A	10	3x for ~3s each, alternating	9	1	2	10-15 s / 300 s	10th object required extra time; 2 misses before recognition.
R-SYN-01	12	3x for 2-3s each, alternating	11	1	0	0-3 s / 300 s	Illustrative baseline with stable lighting.

R-SYN-02	10	3x for ~3s each, alternating	8	2	1	5-8 s / 300 s	Illustrative run with background motion.
R-SYN-03	8	4x for ~2s each, alternating	8	0	0	0 s / 240 s	Illustrative best-case after calibration.

Build and flash (ESP-IDF)

From the project root:

```
idf.py set-target esp32s3
idf.py build
idf.py flash
idf.py monitor
```

The serial port can be set with `-p COMx` on Windows.

Python GUI client

The GUI reads /raw and /status and optionally writes commands over serial.

Install requirements:

```
python -m pip install -r tools/requirements.txt
```

Run:

```
python tools/vision_gui.py
```

Set the HTTP base URL to your device IP (AP default is http://192.168.4.1).

Notes and limitations

- The camera starts disabled; call cam_on before streaming.
- /raw returns raw grayscale without compression; bandwidth is higher but avoids JPEG memory pressure.
- The frame size is fixed at QQVGA (320x240) unless changed in code.
- AP+STA mode is enabled when WIFI_STA_SSID is non-empty.