

Vidhatu Patel

Project Portfolio

github.com/vidhatu

BigTrack

github.com/vidhatu/bigtrack

Multi-Camera 3D Object Tracking with Browser-Based Computer Vision

Team project with Thomas Fasan | Server-Side Development, final project | May 2026

My role	Client-side lead (browser CV pipeline, ChArUco calibration, 3D camera-placement UI); co-designed the full system architecture and estimation math with Thomas; contributed to the Rust server.
Languages	Rust, C++, JavaScript
Key tools	OpenCV, ONNX Runtime, WebGPU, GTSAM, Tokio, Axum, Rerun, SQLite, Docker

Project Description

Reconstructing the 3D path of a moving object from several camera views is a well-studied problem, but practical systems usually need dedicated hardware, offline calibration, or computer vision (CV) expertise. BigTrack removes those barriers: any device with a webcam and a modern browser can join a shared tracking session. Each device detects the object in its own video feed, and a central server fuses those 2D detections into a 3D trajectory while also correcting errors in where each camera was assumed to be. Thomas and I designed the architecture and estimation math together; I led the browser client, Thomas led the Rust server, and we each contributed code to both.

- **In-browser detection.** OpenCV runs as WebAssembly (WASM, a binary format browsers run at near-native speed), or a YOLO (You Only Look Once) detector runs on ONNX (Open Neural Network Exchange) Runtime with WebGPU, both in a background Web Worker thread.
- **Per-device calibration.** Each client measures its camera's *intrinsics* (focal length, optical center, lens distortion) from a printed ChArUco board, a chessboard with embedded ArUco fiducial markers.
- **Coarse camera placement.** Users roughly place each camera in an interactive 3D view. These *extrinsics* (camera position and orientation) only need to be approximate.
- **Joint optimization.** The server refines the trajectory and all camera poses together in a GTSAM (Georgia Tech Smoothing and Mapping) factor graph: live with the iSAM2 solver, then more accurately after recording.
- **Physical constraints.** Optional ground-plane and parabolic-flight priors narrow the solution space. Observations that strongly violate them are filtered out.
- **Live visualization.** Camera frustums, detections, and the 3D track stream to a Rerun web viewer, and finished sessions are saved to SQLite.

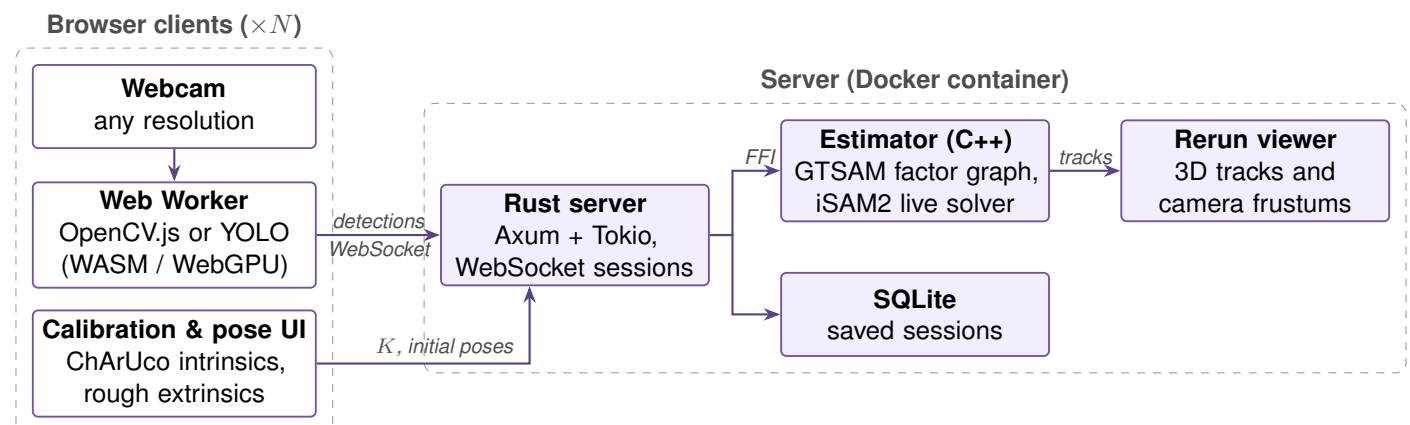


Figure 1: System architecture. FFI (Foreign Function Interface) lets the Rust server call the C++ estimator.

Tools and Methods

Layer	Tools	How we used them
Client vision	OpenCV.js (built for WASM with Emscripten), ONNX Runtime Web, Web Workers	Color-based and YOLO detection plus ChArUco calibration, off the UI (user interface) thread.
Acceleration	WebGPU, falling back to WebGL, then plain WASM	Runs neural inference on whatever a device offers, from laptops to phones.
Server	Rust, Tokio (async runtime), Axum (web framework)	Accepts detection streams from any number of clients per session.
Estimation	C++, GTSAM, iSAM2	Jointly estimates the trajectory and camera poses, live and after recording.
Interop	rust-bindgen	Generates Rust bindings to the C++ estimation code.
Visualization	Rerun (rerun.io)	One Rerun server, with one RecordingStream per session.
Storage, deployment	SQLite via sqlx, Docker, build scripts	Persists sessions; packages GTSAM, the Emscripten toolchain and Rerun into one reproducible container.

Method: tracking and calibration as one factor graph

A *factor graph* links unknown variables (circles) through measurements and priors (squares). Each camera c has an unknown pose $T_c \in SE(3)$, the set of 3D rotations plus translations, and known intrinsics K_c . The object has an unknown position $\mathbf{p}_t \in \mathbb{R}^3$ at each timestep t . Every pixel detection $\mathbf{z}_{c,t}$ adds a reprojection factor, every user-entered pose $\bar{T}_c$ adds a weak prior, and each enabled constraint adds a residual $\mathbf{r}_k$. The estimate solves the nonlinear least-squares problem

$$\min_{\{T_c\}, \{\mathbf{p}_t\}} \sum_{c,t} \|\pi_{K_c}(T_c^{-1}\mathbf{p}_t) - \mathbf{z}_{c,t}\|_{\Sigma_z}^2 + \sum_c \|\text{Log}(\bar{T}_c^{-1}T_c)\|_{\Sigma_T}^2 + \sum_k \|\mathbf{r}_k\|_{\Sigma_k}^2$$

where π_{K_c} projects a camera-frame point to pixels and Log turns a pose error into a 6-vector. A parabolic prior, for example, enforces $\mathbf{p}_t = \mathbf{p}_0 + \mathbf{v}_0\Delta t + \frac{1}{2}\mathbf{g}\Delta t^2$. iSAM2 (incremental Smoothing and Mapping 2) updates only the affected part of the solution as new factors arrive, which makes live estimation feasible.

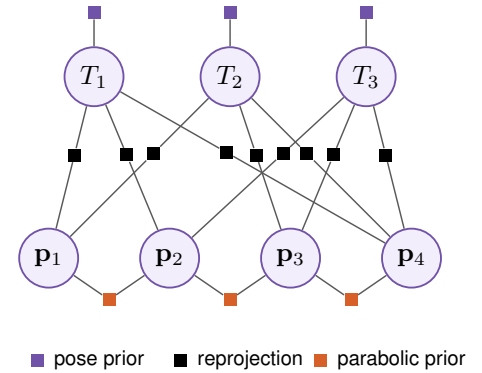


Figure 2: Simplified factor graph: 3 cameras, 4 timesteps.

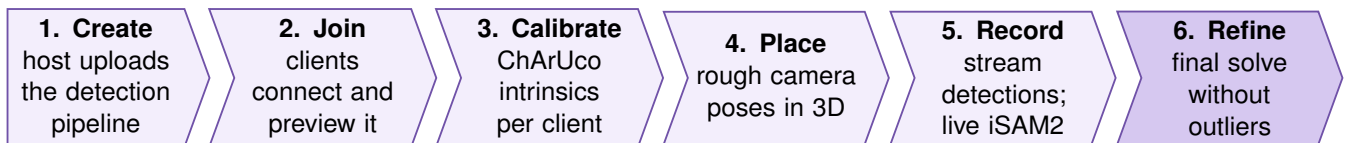


Figure 3: Lifecycle of a tracking session, from setup to the final estimate.

Challenges and Solutions

Challenge	How we addressed it
Sharing user-supplied code (JavaScript or ONNX models) across clients.	Only the host can upload; clients can preview the JavaScript first; all of it runs in a sandboxed Web Worker. Published ONNX CVEs (Common Vulnerabilities and Exposures) involve file-system or Python APIs (application programming interfaces) that browser JavaScript cannot reach.
Varied client GPU (graphics processing unit) support.	ONNX Runtime prefers WebGPU and falls back to WebGL, then WASM; pipelines stay light enough for the weakest device.
Inaccurate hand-placed cameras.	User-entered poses act as weak priors, so the optimizer refines them alongside the trajectory as observations accumulate.
I/O-bound server, CPU-heavy solver.	Tokio handles all networking; each session gets a dedicated thread with its own GTSAM instance, and live estimation can be disabled on slow servers. This was our first async Rust project.
Rust-C++ interop and heavy builds.	Bindings generated with rust-bindgen; the whole server ships as a Docker container; Rerun is pinned because its Rust query API is not yet stable.
Outlier detections.	Observations that violate active constraints are filtered live, and users can exclude more before the final full-horizon solve.

Results and Outcomes

We tested the system with three laptop webcams with overlapping views of a dorm room, tracking a thrown volleyball. Its distinctive color suits the color-based detector, and its near-parabolic flight tests the parabolic prior.

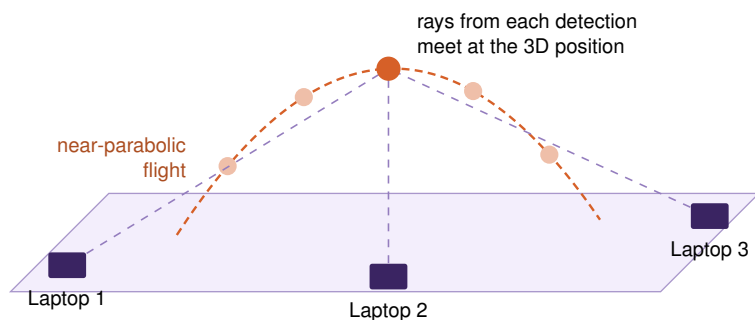


Figure 4: Schematic of the dorm-room test (not to scale). Each laptop's 2D detection defines a ray in 3D. The estimator finds the ball position where the rays best agree, while also refining where each camera is.

What worked

- ▶ The full pipeline ran end to end: three browser clients streamed detections to the Rust server, which ran the GTSAM estimate and sent the result to Rerun.
- ▶ Client-side detection held at least 30 FPS (frames per second) on every laptop, reaching up to 60 FPS.
- ▶ One throw was reconstructed accurately in 3D.

What fell short

- ▶ We could not reproduce an accurate reconstruction beyond that single throw.
- ▶ Likely causes: the small room limited camera spacing and flight length; consumer webcams aren't hardware-synchronized, so a fast ball shifts between "simultaneous" frames; and starting camera poses were rough.

Outcome. The project earned an A as a proof of concept: every component worked, but tracking was not yet reliable across repeated throws. We believe a more rigorous setup would help: a larger space with wider camera spacing, synchronized timestamps, and tape-measured camera positions to separate setup error from estimation error.

Skills Learned

- ▶ Asynchronous Rust with Tokio and Axum
- ▶ Rust-C++ interoperability through an FFI
- ▶ Factor-graph optimization and incremental smoothing
- ▶ Multi-view geometry and camera calibration
- ▶ In-browser machine learning (ML) with ONNX Runtime
- ▶ Threat modeling for shared, user-supplied code
- ▶ Containerizing complex C++ toolchains with Docker
- ▶ Diagnosing why a prototype falls short