

Camera Calibration for Robot Learning

Complete English Tutorial · Theory, Workflows, Code, and Tron2 Deployment

October 15, 2026

Contents

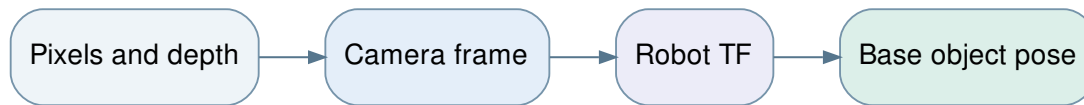
1. Types of Camera Calibration and Their Uses	2
2. Essential Mathematics and Coordinate Frames	2
3. Intrinsic Calibration	3
4. Extrinsic Calibration	5
5. Hand-Eye Calibration	6
6. Specific Considerations for Tron2 Head and Wrist Cameras	6
7. RGB-D and Depth-Camera Calibration	7
8. Complete OpenCV Hand-Eye Example	9
9. Camera Calibration in ROS 2	10
10. Using Calibration Results in ROS TF2 and URDF	11
11. How Calibration Is Used in Robot Learning	12
12. Accuracy Metrics and Calibration Validation	13
13. Common Failure Modes and Troubleshooting	14
14. Recommended Open-Source Tools and References	15
15. Recommended Tron2 Implementation Plan	15
16. Summary and Deployment Checklist	16

Course home · 简体中文 · PDF

Item	Details
Applicability	OpenCV 4.x concepts; ROS 2 Jazzy; device SDK version must be selected separately
Documentation review date	2026-10-18
License	CC BY-NC-SA 4.0 · Xiong Zeng and contributors

1. Types of Camera Calibration and Their Uses

Chapter concept map



In robot learning, *camera calibration* is broader than estimating intrinsics. It also involves the geometric and temporal relationships among cameras, robots, and the workspace.

Type	Main purpose	Typical applications
Intrinsic calibration	Estimate imaging geometry and lens distortion	Projection, PnP, 6D pose
Extrinsic calibration	Estimate a camera pose relative to a reference frame	Grasping, spatial localization
Hand-eye calibration	Estimate a rigid transform between a camera and robot end effector or base	Wrist and fixed cameras
Multi-camera calibration	Estimate relative transforms between cameras	Multi-view perception, fusion
RGB-D/depth calibration	Verify color-depth alignment and depth scale	Point clouds, 3D localization
Temporal calibration	Align image timestamps with robot/sensor measurements	Tracking, feedback control

Closely related topics are **robot kinematic calibration** and **camera-IMU calibration**.

Important distinction: A RealSense device usually comes with factory intrinsics, but installing it on Tron2 still requires determining its pose relative to the robot. If the camera is mounted on a moving head, its transform relative to the base varies with head-joint angles.

2. Essential Mathematics and Coordinate Frames

2.1 Frame definitions

Symbol	Frame	Meaning
W	World	Global or simulation world
B	Robot base	Base frame of the robot
G	Gripper/end effector	Robot terminal link frame
C	Camera optical	Optical coordinate frame
O	Object	Object-centered frame
T	Calibration target	Board or fiducial frame

We use an explicit transform convention throughout:

$${}^A T_B = \begin{bmatrix} {}^A R_B & {}^A t_B \\ 0 & 1 \end{bmatrix}.$$

The transform ${}^A T_B$ maps the coordinates of a point **expressed in frame B into frame A** . Here $R \in SO(3)$, $t \in \mathbb{R}^3$, and $T \in SE(3)$.

In homogeneous coordinates,

$$\bar{p}_A = {}^A T_B \bar{p}_B.$$

Its inverse and composition obey

$${}^B T_A = ({}^A T_B)^{-1}, \quad {}^A T_C = {}^A T_B {}^B T_C.$$

For example, if FoundationPose estimates an object pose ${}^C T_O$ in the camera frame, a base-frame controller needs

$$\boxed{{}^B T_O = {}^B T_C {}^C T_O.}$$

One of the principal objectives of calibration is therefore to obtain an accurate ${}^B T_C$.

2.2 Robot frames versus optical camera frames

The typical ROS REP 103 conventions are:

Axis	Robot body frame	Camera optical frame
X	Forward	Right in the image
Y	Left	Down in the image
Z	Up	Forward along the optical axis

Consequently, `camera_link` and `camera_optical_frame` may be different frames connected by a fixed rotation. Do not interpret optical coordinates returned by OpenCV as robot body-frame coordinates without checking the transform. See [ROS REP 103](#).

3. Intrinsic Calibration

3.1 Intrinsic matrix and distortion

For an ideal pinhole camera,

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix},$$

where f_x, f_y are focal lengths measured in pixels and c_x, c_y define the principal point. For an undistorted camera-frame point (X, Y, Z) ,

$$u = f_x \frac{X}{Z} + c_x, \quad v = f_y \frac{Y}{Z} + c_y.$$

Real lenses introduce distortion. One common OpenCV model uses

$$D = [k_1, k_2, p_1, p_2, k_3]$$

for radial and tangential distortion; other models are appropriate for fisheye and specialized lenses.

3.2 Why intrinsics matter

Use 1: Projecting 3D points into images. Given a point $p_C = (X, Y, Z)$, camera intrinsics determine its image pixel (u, v) . This is needed for visualization, reprojection, visual servoing, and synthetic-real image alignment.

Use 2: RGB-D back-projection. If an undistorted image pixel (u, v) has a known **optical-axis depth** Z , then

$$p_C = \begin{bmatrix} (u - c_x)Z/f_x \\ (v - c_y)Z/f_y \\ Z \end{bmatrix}.$$

For distorted pixels, undistort first or use a correct camera-model deprojection function. After aligning depth to color, use the camera model appropriate to the aligned representation.

Use 3: 6D object-pose estimation. PnP, FoundationPose, MegaPose, and CAD-model tracking typically use intrinsics. An ordinary RGB pixel by itself determines a ray, not the absolute distance of a point along that ray.

3.3 Choosing a calibration target

Target	Advantages	Limitations	Typical use
Checkerboard	Accurate corners; inexpensive	Corners must be visible	Basic intrinsics
ArUco / AprilTag	Easy identification; markers have IDs	Single-marker corner accuracy varies	PnP and target tracking
ChArUco	Accurate chessboard-like corners plus marker IDs; partial occlusion support	Requires accurate printing and corner indexing	Precise intrinsics and hand-eye calibration

For ordinary robot labs, **ChArUco** is a good starting point, although a checkerboard is also effective. See the [OpenCV ChArUco detection tutorial](#).

3.4 Practical intrinsic-calibration procedure

1. **Prepare a rigid board.** For example, a 7-by-5-square ChArUco board with 30 mm squares and 20 mm tags. These are illustrative dimensions, not a requirement. Print at true scale on a flat backing.
2. **Capture diverse views.** Start with roughly 20–40 sharp, useful images. Move the board across the center, corners, distances, tilt angles, and apparent scales. Keep important camera settings controlled.
3. **Detect corresponding corners.** Use `cv2.findChessboardCornersSB()` or a ChArUco detector. Verify exact 2D–3D point correspondence.
4. **Optimize intrinsics and distortion.** Minimize reprojection residuals across images:

$$\min_{K, D, \{R_j, t_j\}} \sum_{j,i} \|p_{ij} - \pi(K, D, R_j P_i + t_j)\|_2^2.$$

5. **Validate with held-out views.** Inspect RMS reprojection error, residuals across the image, and straight lines after undistortion. An RMS around 0.3–0.5 pixels can be a useful reference for high-quality captures; it is **not** a universal acceptance threshold, nor a measure of millimeter-scale robot accuracy.

4. Extrinsic Calibration

4.1 What an extrinsic transform means

Intrinsics answer “how does a 3D point project into the image?” Extrinsics answer “where is the camera with respect to a reference frame?” For a head-mounted Tron2 camera, a common objective is

$${}^B T_C = \begin{bmatrix} {}^B R_C & {}^B t_C \\ 0 & 1 \end{bmatrix}.$$

4.2 Known target pose: PnP followed by transform composition

Suppose a calibration board has a precisely known base-frame pose ${}^B T_T$. PnP estimates the **target-to-camera** transform ${}^C T_T$ using intrinsics and image corners. Then

$${}^B T_T = {}^B T_C {}^C T_T,$$

which gives

$$\boxed{{}^B T_C = {}^B T_T ({}^C T_T)^{-1}.}$$

Procedure: Fix the board; measure or touch-probe ${}^B T_T$; acquire a clear image; detect corresponding corners; run PnP; compute ${}^B T_C$; validate at other known target positions.

```
import cv2
import numpy as np

# N-by-3 board corners in meters
object_points = np.asarray(..., dtype=np.float64)
# Corresponding N-by-2 image corners in pixels
image_points = np.asarray(..., dtype=np.float64)
K = np.asarray(..., dtype=np.float64)
dist = np.asarray(..., dtype=np.float64)

ok, rvec, tvec = cv2.solvePnP(
    object_points, image_points, K, dist
)
if not ok:
    raise RuntimeError("PnP failed")

R, _ = cv2.Rodrigues(rvec)
T_C_T = np.eye(4)
T_C_T[:3, :3] = R
T_C_T[:3, 3] = tvec.reshape(3)

# Must come from an independent measurement
T_B_T = np.asarray(..., dtype=np.float64)
T_B_C = T_B_T @ np.linalg.inv(T_C_T)
print(T_B_C)
```

Direction warning: `solvePnP` returns ${}^C T_T$, not its inverse.

Limitation: If the board is placed arbitrarily and its base-frame pose is unknown, one image normally cannot determine ${}^B T_C$. This motivates hand-eye calibration.

5. Hand-Eye Calibration

5.1 The two standard configurations

Property	Eye-in-hand	Eye-to-hand
Camera	Rigidly mounted on the moving end effector	Fixed to the base or workspace
Target board	Usually fixed in the environment	Usually fixed to the end effector
Typical unknown	${}^G T_C$	${}^B T_C$
Example	Wrist camera	Static overhead camera

A Tron2 head camera that moves with yaw/pitch is **not a truly static eye-to-hand camera**. It can only be modeled as static while the head joints remain fixed.

5.2 Eye-in-hand: calibrating the wrist camera

The goal is to estimate the constant ${}^G T_C$. Keep the board stationary while the robot visits several poses. At each pose record:

- End-effector pose from forward kinematics or TF: ${}^B T_{G,i}$.
- Calibration-board pose estimated by PnP: ${}^C T_{T,i}$.

Because the board does not move,

$${}^B T_{G,i} {}^G T_C {}^C T_{T,i} = {}^B T_T.$$

Pairs of observations yield the classical $AX = XB$ problem, with $X = {}^G T_C$. Robot motion must contain **diverse, nonparallel rotation axes**. Pure translations or repeated rotation about a single axis are generally inadequate.

Practical dataset: Begin with about 20–30 valid poses with varied orientations, distances, and image locations. Keep the board visible, ensure image-state alignment, and reject blurred or badly detected frames. Motion diversity is usually more important than simply increasing sample count.

OpenCV implements several solvers, including Tsai, Park, Horaud, Andreff, and Daniilidis. See [calibrateHandEye](#).

5.3 Eye-to-hand: fixed camera with a board on the robot

Here the camera is static, and the board is rigidly mounted on the end effector. Each observation satisfies

$${}^B T_{G,i} {}^G T_T = {}^B T_C {}^C T_{T,i}.$$

Both the base-to-camera transform ${}^B T_C$ and the gripper-to-target transform ${}^G T_T$ are fixed unknowns. Multiple robot poses can determine ${}^B T_C$ without knowing an external board location.

Critical convention: When using OpenCV `cv2.calibrateHandEye()` for this layout, provide ${}^G T_B = ({}^B T_G)^{-1}$ as the *robot* pose input, while continuing to supply ${}^C T_T$ as the target pose. The output is then interpreted as ${}^B T_C$. Reusing the eye-in-hand inputs unchanged can give a plausible-looking but incorrect transform.

6. Specific Considerations for Tron2 Head and Wrist Cameras

6.1 Camera mounted on a moving head

Let H denote the terminal link frame of the head kinematic chain, C_h the head optical frame, and q_h the head-joint vector. The camera pose is

$${}^B T_{C_h}(q_h) = {}^B T_H(q_h) {}^H T_{C_h}.$$

- ${}^B T_H(q_h)$ is a **dynamic** transform from head forward kinematics.
- ${}^H T_{C_h}$ is a **fixed** camera-mount transform to be calibrated or measured.

Suggested procedure: Fix a ChArUco board in the workspace. Capture images across different yaw/pitch poses and record their joint values. Compute ${}^B T_H$ from FK and ${}^{C_h} T_T$ from PnP. Solve for ${}^H T_{C_h}$ using the eye-in-hand formulation, then test on held-out head poses.

What if the yaw and pitch axes do not intersect?

A naive expression $R_z(q_y)R_y(q_p)$ omits the offset between rotation axes. Use the full kinematic chain:

$${}^B T_{C_h} = {}^B T_Y T_Y(q_y) {}^Y T_P T_P(q_p) {}^P T_{C_h}.$$

The fixed transform ${}^Y T_P$ contains both rotation and translation between yaw and pitch frames. First inspect the `origin` and `axis` entries in the URDF. If the actual axis geometry differs from the kinematic model, correcting only the camera-mount extrinsic will **not necessarily eliminate** posture-dependent errors.

For an initial screwdriver-grasping experiment with a fixed head viewpoint, calibrating a single base-to-camera pose is a reasonable starting point. Add full dynamic head modeling when active vision is needed.

6.2 Wrist-mounted camera

Let C_w be the wrist camera and q_a the arm configuration:

$${}^B T_{C_w}(q_a) = {}^B T_G(q_a) {}^G T_{C_w}.$$

The first term is obtained from FK; the second is estimated through eye-in-hand calibration. If the camera is mounted on `wrist_link` rather than the precise terminal frame used for FK, consistently compose the corresponding link transforms.

6.3 Relative transform between head and wrist cameras

Once each camera's base-frame pose is known,

$${}^{C_h} T_{C_w} = ({}^B T_{C_h})^{-1} {}^B T_{C_w}.$$

A separate two-camera calibration is not always necessary, although observing the same target from both cameras is an excellent consistency check. Since the head and arm move, the relative camera-to-camera transform is generally **time-varying**.

7. RGB-D and Depth-Camera Calibration

7.1 Internal camera frames

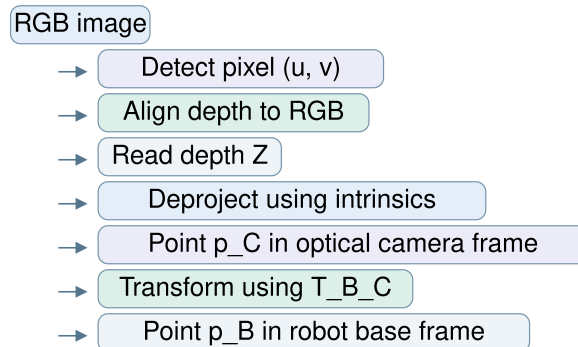
Devices such as RealSense D435 and D455 often expose color, infrared, and depth camera frames. Their optical centers usually differ, requiring a known rigid transform such as

$${}^{C_{\text{rgb}}} T_{C_{\text{depth}}}.$$

The RealSense SDK can normally provide factory intrinsics, extrinsics, and depth scale. Begin by **reading and validating** these values before attempting new device-level calibration.

7.2 From an RGB pixel to a point in the robot base frame

The key pipeline is:



A minimal RealSense example is:

```

import pyrealsense2 as rs
import numpy as np

pipeline = rs.pipeline()
config = rs.config()
config.enable_stream(rs.stream.depth)
config.enable_stream(rs.stream.color)
pipeline.start(config)

align = rs.align(rs.stream.color)
try:
    frames = pipeline.wait_for_frames()
    aligned = align.process(frames)

    depth = aligned.get_depth_frame()
    color = aligned.get_color_frame()
    if not depth or not color:
        raise RuntimeError("Missing RGB-D frame")

    u, v = 320, 240 # Example: check image bounds
    z = depth.get_distance(u, v) # Meters
    if z <= 0:
        raise RuntimeError("Invalid depth")

    intr = depth.profile.as_video_stream_profile().get_intrinsics()
    point_C = rs.rs2_deproject_pixel_to_point(
        intr, [float(u), float(v)], z
    )
    print("Point in camera optical frame:", point_C)
finally:
    pipeline.stop()

```

Given a separately calibrated camera-to-base transform,

```

point_C_h = np.r_[np.asarray(point_C), 1.0]
point_B_h = T_B_C @ point_C_h
print("Point in base frame:", point_B_h[:3])

```

This implements “RGB pixel → 3D camera point → 3D robot-base point.”

7.3 Should factory depth calibration be redone?

Not automatically. First verify device calibration at the intended operating range:

Item	Purpose
RGB intrinsics	Accurate color-image projection
Depth intrinsics	Accurate depth deprojection
RGB-depth extrinsics	Correct spatial alignment
Depth scale	Correct metric units
Depth accuracy near the target	Reliable working-distance estimation
Camera-to-robot extrinsics	Correct conversion into the robot base

Thin and reflective objects, including screwdrivers, can cause missing depth, incorrect boundaries, and reflection-related failures. Such errors are not always fixed by re-estimating RGB intrinsics. See the [RealSense SDK repository](#).

8. Complete OpenCV Hand-Eye Example

8.1 Data conventions

For each of N sampled configurations, store two time-matched 4-by-4 transforms:

```
base_T_gripper.npy   shape = (N, 4, 4)
camera_T_target.npy  shape = (N, 4, 4)
```

These are ${}^B T_{G,i}$ and ${}^C T_{T,i}$. Use meters for translations. Rotation blocks must be orthonormal with determinants near one, and each last row must be $[0, 0, 0, 1]$.

8.2 Code supporting both configurations

```
import cv2
import numpy as np

# ^B T_G : gripper -> robot base
T_B_G = np.load("base_T_gripper.npy")
# ^C T_T : calibration target -> camera optical frame
T_C_T = np.load("camera_T_target.npy")

assert T_B_G.shape == T_C_T.shape
assert T_B_G.ndim == 3 and T_B_G.shape[1:] == (4, 4)
assert len(T_B_G) >= 3

mode = "eye_in_hand" # or "eye_to_hand"

if mode == "eye_in_hand":
    robot_poses = T_B_G
elif mode == "eye_to_hand":
    robot_poses = np.linalg.inv(T_B_G)
else:
    raise ValueError("Unknown calibration mode")

R_robot = [T[:3, :3] for T in robot_poses]
t_robot = [T[:3, 3] for T in robot_poses]
R_target = [T[:3, :3] for T in T_C_T]
t_target = [T[:3, 3] for T in T_C_T]

R, t = cv2.calibrateHandEye(
    R_robot, t_robot, R_target, t_target,
    method=cv2.CALIB_HAND_EYE_PARK
```

```

)

X = np.eye(4)
X[:3, :3] = R
X[:3, 3] = t.reshape(3)

if mode == "eye_in_hand":
    print("^G T_C (gripper <- camera):")
else:
    print("^B T_C (base <- camera):")
print(X)
np.save("handeye_result.npy", X)

```

The example illustrates correct API conventions. For real deployment, reject outliers, ensure sufficiently rich rotational motion, and validate on held-out poses. Confirm that `camera_T_target` refers to the **optical** camera frame, not an ambiguous `camera_link`.

8.3 Validating the output

For eye-in-hand, compute for each observation

$$\hat{T}_{BT,i} = {}^B T_{G,i} {}^G T_C {}^C T_{T,i}.$$

The estimated board pose must remain nearly constant across robot motions.

For eye-to-hand, compute

$$\hat{T}_{GT,i} = ({}^B T_{G,i})^{-1} {}^B T_C {}^C T_{T,i}.$$

This estimated board-to-gripper transform should likewise be constant. Report translational and angular residuals separately, and test with robot poses excluded from calibration.

9. Camera Calibration in ROS 2

9.1 Useful tools

Tool	Main use
camera_calibration	Checkerboard-based intrinsic calibration
image_pipeline	Image undistortion and rectification
tf2	Coordinate-transform management
robot_state_publisher	Publish robot frames from URDF and joint states
easy_handeye2	ROS 2 hand-eye workflow
MoveIt Calibration	GUI hand-eye workflow
Kalibr	Multi-camera and camera-IMU spatiotemporal calibration

9.2 Intrinsic calibration in ROS 2

On a compatible ROS 2 distribution:

```

sudo apt install \
  ros-${ROS_DISTRO}-camera-calibration \
  ros-${ROS_DISTRO}-image-pipeline

ros2 topic list

```

If images and camera information are published as `/camera/image_raw` and `/camera/camera_info`, a checkerboard workflow can be launched with:

```
ros2 run camera_calibration cameracalibrator \
  --size 8x6 \
  --square 0.025 \
  image:=/camera/image_raw \
  camera:=/camera
```

8x6 is the number of **internal corners**; 0.025 is the physical square width in meters. Both must match the board. Actual topics depend on the camera driver. See [ROS image_pipeline](#).

9.3 Hand-eye calibration in ROS 2

Useful projects include [easy_handeye2](#) and [MoveIt Calibration](#). A typical workflow is:

1. Publish end-effector poses in TF.
2. Detect the board and publish its observed pose.
3. Collect observations across diverse robot poses.
4. Select eye-in-hand or eye-to-hand.
5. Compute, validate, and save the estimated transform.
6. Integrate the result into URDF or the TF tree.

A short Python/OpenCV script often suffices for a one-off experiment. ROS tooling becomes valuable when the team needs a repeatable, standardized calibration procedure.

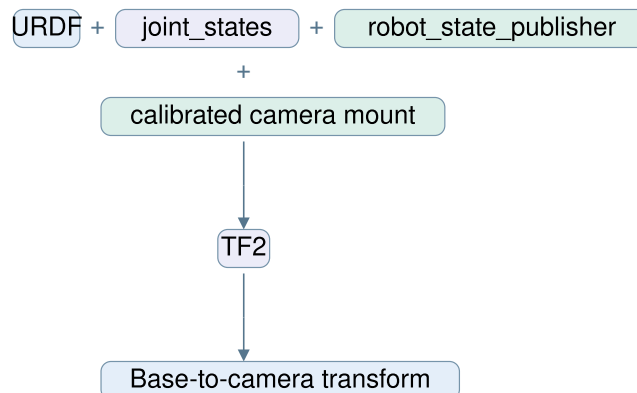
10. Using Calibration Results in ROS TF2 and URDF

10.1 Recommended TF structure

The following is an illustration; replace link names with the actual Tron2 URDF definitions:

```
base_link
├── arm_link_1
│   ├── ...
│   └── wrist_link
│       ├── wrist_camera_link
│       └── wrist_camera_optical_frame
├── head_yaw_link
│   ├── head_pitch_link
│   ├── head_camera_link
│   └── head_camera_optical_frame
```

Separate fixed camera-mount transforms from dynamic robot kinematics:



10.2 Static extrinsic publishers

The following translation and rotation values are **syntax examples**, not actual Tron2 calibration parameters:

```
ros2 run tf2_ros static_transform_publisher \
  --x 0.03 --y 0.00 --z 0.08 \
  --qx 0 --qy 0 --qz 0 --qw 1 \
  --frame-id wrist_link \
  --child-frame-id wrist_camera_optical_frame
```

Warning: The rotation into a real optical frame is often **not** the identity quaternion. If the same parent-child transform is already published through URDF and `robot_state_publisher`, do not publish it again.

10.3 Dynamic transform checks

```
ros2 run tf2_ros tf2_echo \
  base_link wrist_camera_optical_frame

ros2 run tf2_ros tf2_echo \
  base_link head_camera_optical_frame
```

As arm and head joints move, these base-to-camera transforms should update. For recorded images, query TF at the **image acquisition timestamp**, not blindly at the current robot-state timestamp.

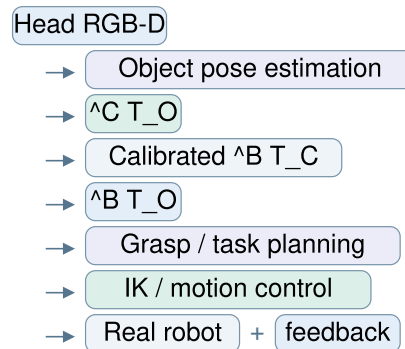
11. How Calibration Is Used in Robot Learning

11.1 6D pose estimation and visually guided grasping

Suppose FoundationPose estimates a screwdriver pose ${}^{C_h}T_O$ from the head RGB-D camera. Convert it to the base frame using

$${}^B T_O = {}^B T_{C_h}(q_h) {}^{C_h} T_O.$$

A common pipeline is:



Distinguish **pose-estimation error** from **camera-extrinsic error**. Even a very accurate 6D pose estimator produces systematically wrong robot-frame positions if ${}^B T_C$ is incorrect.

11.2 Imitation learning and VLAs

This section only establishes **when explicit extrinsics are needed**. BC/ACT and demonstrations belong in [Data and Imitation Learning](#); SAM/Foundation models, temporal estimation and reacquisition belong in [Visual State Estimation](#).

Actions may be represented as TCP increments $a_t = \Delta T_{\text{TCP}}$ or future joint sequences $a_t = q_{t:t+H}$. Whether explicit extrinsics are required depends on the architecture and task:

Task	Explicit extrinsics?
End-to-end behavior cloning from RGB to joint commands	Not necessarily
End-to-end RGB + proprioception policy	Not necessarily
Recovering object 3D positions from depth and pixels	Yes
Retargeting video trajectories into robot SE(3) motion	Usually
Multi-view 3D sensor fusion	Yes
Matching camera geometry across simulation and hardware	Yes
FoundationPose coupled with IK	Yes

End-to-end VLA models can learn an implicit visual-to-action mapping. Nevertheless, explicit, consistent camera geometry is often valuable for transfer across cameras, robots, and workspaces.

11.3 Reinforcement learning and sim-to-real

Consider an image-based control policy

$$a_t = \pi(I_t, q_t).$$

If camera poses, viewing angles, focal lengths, crop windows, or resizing differ between simulation and hardware, the observation distributions shift. Match at least

$$K_{\text{sim}} \approx K_{\text{real}}, \quad {}^B T_{C,\text{sim}} \approx {}^B T_{C,\text{real}}.$$

Then randomize residual camera-calibration uncertainty in training, for instance

$$\delta t \sim \mathcal{N}(0, \Sigma_t), \quad \delta \theta \sim \mathcal{N}(0, \Sigma_R).$$

Camera domain randomization improves robustness but should not replace accurate initial calibration.

11.4 Visual feedback and synchronization

For object target pose T_O^* and current pose $T_{O,t}$, an $SE(3)$ -valued tracking error may be constructed as

$$e_t = \text{Log}((T_O^*)^{-1} T_{O,t}).$$

Accurate visual control requires temporal alignment as well as spatial calibration. If the image was acquired at t_{image} , use the robot joint state and

$${}^B T_C(t_{\text{image}})$$

at that time, rather than pairing an older image with the latest arm pose. This matters especially for moving wrist and head cameras.

12. Accuracy Metrics and Calibration Validation

Evaluate the system at **image, geometry, and task** levels:

Level	Metric	Purpose
Image	RMS reprojection error and spatial residuals	Validate intrinsics and corner detection
Geometry	Translation and rotation consistency	Validate extrinsic and hand-eye constraints
Task	3D localization error and grasp success	Validate full robot-system performance

12.1 Reprojection error

For observed image corner p_i and predicted corner $\hat{p}_i$,

$$e_i = \|p_i - \hat{p}_i\|_2$$

with RMS

$$e_{\text{RMS}} = \sqrt{\frac{1}{N} \sum_{i=1}^N e_i^2}.$$

An image-space error measured in pixels is not directly a millimeter-level robot-endpoint error.

12.2 3D localization error

Given a target point with known base-frame position p_B^* , compare it to the visually estimated $\hat{p}_B$:

$$e_p = \|\hat{p}_B - p_B^*\|_2.$$

Test the workspace center and boundaries, several heights and viewing angles, and multiple yaw/pitch configurations. If head motion causes a substantial error increase, examine head kinematics, axis offsets, timestamps, and camera-mount calibration.

12.3 Grasp-level validation

One useful metric is TCP position error,

$$e_{\text{grasp}} = \|p_{\text{TCP,actual}} - p_{\text{TCP,target}}\|_2.$$

Also evaluate screwdriver grasp success, repeatability, orientation error, and disturbance recovery. These depend on FK, IK, actuation, TCP geometry, and object perception as well as camera calibration. A grasp failure does **not** prove the camera was calibrated incorrectly.

13. Common Failure Modes and Troubleshooting

Symptom	Likely cause	Check
Consistent spatial offset	Extrinsic translation bias	Inspect ${}^B T_C$
Error grows with viewpoint changes	Rotation calibration or kinematic error	Test many joint poses
Localization drifts after head motion	Incorrect yaw/pitch axes or offsets	Compare URDF to hardware
Unstable wrist hand-eye estimates	Weak rotational excitation / outliers	Use multi-axis poses, reject bad frames
Incorrect object distance	Depth scale, reflections, depth noise	Measure known distances

Symptom	Likely cause	Check
RGB-depth pixel mismatch	Alignment or camera-model mismatch	Check alignment and intrinsics
Vision axes disagree with control axes	Incorrect optical-frame mapping	Inspect TF frame conventions
Static tests succeed but moving tests fail	Image/joint timestamp mismatch	Query TF at exposure time
Simulation succeeds; robot fails	Camera, dynamics, or control sim-to-real gap	Diagnose perception and actuation separately

A conservative data-collection sequence is: **move into pose** → **wait until stable** → **acquire image and corresponding robot state** → **save sample**. This reduces motion blur and time-alignment error.

14. Recommended Open-Source Tools and References

Tool	Main function	Recommended use
OpenCV	Intrinsics, PnP, hand-eye	Learn first
ROS image_pipeline	Intrinsics and image processing in ROS	ROS deployments
easy_handeye2	ROS 2 hand-eye workflow	Repeatable calibration
MoveIt Calibration	GUI calibration tools	Interactive setup
Kalibr	Multi-camera, camera-IMU, temporal calibration	Complex sensor suites
RealSense SDK	Device parameters, alignment, deprojection	D435/D455 and related devices
ROS REP 103	ROS frame conventions	Optical-frame verification

For this two-camera system, prioritize TF, PnP, and hand-eye calibration.

15. Recommended Tron2 Implementation Plan

A four-phase rollout helps isolate different error sources instead of combining them from the start.

Phase 1: Intrinsics and depth verification

- Read RealSense RGB/depth intrinsics and distortion coefficients.
- Verify or recalibrate wrist RGB intrinsics when necessary.
- Inspect RGB-depth alignment, depth scale, and error at typical grasping distances.
- Ensure actual image resolution, crop, and resizing match the camera model used by the perception pipeline.

This phase generally does not require robot movement.

Phase 2: Head camera calibration

Goal: obtain ${}^B T_{C_h}(q_h)$.

1. Inspect the Tron2 yaw/pitch axes, fixed offsets, and URDF definitions.
2. Fix a ChArUco board and capture multiple head configurations.
3. Record $q_{h,i}$ and use FK for ${}^B T_H(q_{h,i})$.
4. Run PnP to get ${}^{C_h} T_{T,i}$.
5. Calibrate the fixed mount ${}^H T_{C_h}$.
6. Verify 3D localization on held-out yaw/pitch poses.

If experiments currently use only one head orientation, start by locking the head and calibrating the corresponding base-camera transform.

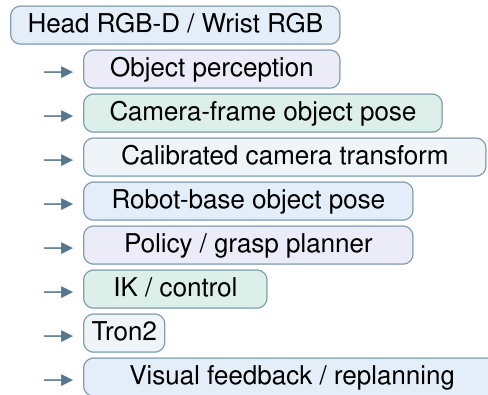
Phase 3: Wrist camera and tool center point

Goals: obtain ${}^G T_{C_w}$ and ${}^G T_{TCP}$.

1. Fix the target board and capture 20–30 diverse robot poses.
2. Save the matched end-effector FK and target pose observations.
3. Run `calibrateHandEye()`.
4. Validate board-pose constancy on unseen robot poses.

Phase 4: Visual control and learning deployment

Suggested unified data flow:



Prioritize three transforms:

$${}^B T_{C_h}(q_h), \quad {}^G T_{C_w}, \quad {}^G T_{TCP}.$$

Combined with FK, TF2, and synchronized timestamps, these connect head perception, wrist vision, and grasp control in one consistent coordinate system.

16. Summary and Deployment Checklist

Core question	Main technique	Role in robot learning
How does an image encode 3D geometry?	Intrinsics, distortion, PnP, RGB-D	Back-projection and pose estimation
Where is the camera relative to the robot?	Extrinsics and hand-eye	Robot-frame grasp planning
Do transforms stay valid during motion?	FK, URDF, TF2, synchronization	Active vision and closed-loop control

Use the following checklist before deployment:

- ☐ Image resolution and intrinsics match, and the image's distortion/rectification status is known.
- ☐ Each transform has a clearly documented direction (for example, ${}^B T_C$).
- ☐ `camera_link` and `camera_optical_frame` are not confused.
- ☐ FK, PnP, and hand-eye measurements use consistent units and link definitions.
- ☐ Head yaw/pitch axes, fixed offsets, and wrist mount extrinsics are verified.
- ☐ RGB-D data are aligned, and depth definitions/scales are known.
- ☐ Extrinsics are evaluated with independent poses, not just the fit data.
- ☐ Camera exposure times are aligned with joint-state and TF timestamps.
- ☐ Final quality is tested using real 3D points and actual grasps.

For a Tron2 manipulation stack, OpenCV PnP, hand-eye calibration, robot forward kinematics, and ROS TF2 are the four most valuable foundations. Together, they enable head/wrist vision without requiring an external Vicon system.

Note: Board dimensions, sample counts, paths, frame names, and some command arguments in this tutorial are examples or engineering recommendations, not verified Tron2 manufacturer specifications or measured calibration results. Real deployments must be validated against the actual device, URDF, logs, and physical tests.