Detection and Tracking Overview - Phase 4

Executive Summary

This document describes the detection and tracking subsystem implemented in Phase 4 of the Rugby Vision project. This layer detects players and ball in video frames and maintains consistent track IDs across frames for each camera.

Current Status: Phase 4 implementation with mock/stub detector for development and testing.

Production Readiness: The architecture supports swapping to real YOLO models with minimal code changes.

Architecture

Component Overview

```
┌─────────────────────────────────────────────────────────────┐
│                    Video Frames (per camera)                │
└──────────────────────────┬──────────────────────────────────┘
                           │
                           ▼
┌─────────────────────────────────────────────────────────────┐
│                    Detector (detector.py)                   │
│  - Input: Frame, camera_id, frame_id                        │
│  - Output: List[Detection]                                  │
│  - Current: Mock detector with synthetic detections         │
│  - Future: YOLO-based detector (YOLOv8, YOLOv9, etc.)       │
└──────────────────────────┬──────────────────────────────────┘
                           │
                           ▼
┌─────────────────────────────────────────────────────────────┐
│                    Tracker (tracker.py)                     │
│  - Input: List[Detection] (per frame)                       │
│  - Output: List[Track]                                      │
│  - Method: IOU-based tracking (simplified ByteTrack-style)  │
│  - Maintains: Track IDs per camera across frames            │
└──────────────────────────┬──────────────────────────────────┘
                           │
                           ▼
┌─────────────────────────────────────────────────────────────┐
│          Orchestration (detection_tracking_api.py)          │
│  - API: run_detection_and_tracking(clip_definition)         │
│  - Coordinates: Detector + Tracker for all cameras          │
│  - Returns: Structured detections and tracks                │
└─────────────────────────────────────────────────────────────┘
```

Data Structures

Detection

Represents a single object detection in a frame.

```
@dataclass
class Detection:
    camera_id: str                              # Camera identifier
    frame_id: int                               # Frame number
    bbox: Tuple[float, float, float, float]    # (x, y, width, height) in pixels
    class_name: str                             # 'player' or 'ball'
    confidence: float                           # 0.0 to 1.0
```

Properties:
- `center`: Returns `(center_x, center_y)` of bounding box
- `area`: Returns area in square pixels

Validation:
- `class_name` must be 'player' or 'ball'
- `confidence` must be in range [0.0, 1.0]
- Bounding box dimensions must be positive

Track

Represents a tracked object across multiple frames.

```
@dataclass
class Track:
    track_id: int                # Unique identifier
    class_name: str              # 'player' or 'ball'
    detections: List[Detection]  # Chronological list
    last_update_frame: int       # Most recent frame ID
    is_active: bool              # Whether still being tracked
```

Properties:
- `length`: Number of detections in track
- `latest_detection`: Most recent Detection object
- `latest_bbox`: Bounding box of latest detection

Methods:
- `add_detection(detection)`: Add a detection to this track

Detection Approach

Current Implementation: Mock Detector

For Phase 4, we use a **mock/stub detector** that generates synthetic detections:

- **Purpose**: Enable end-to-end development and testing without requiring trained models
- **Detections**:
- 5-8 players per frame
- 1 ball per frame (with variable confidence)

- Realistic movement patterns over time
- Proper bounding box sizes

Example Usage:

```
detector = Detector(use_mock=True)
detections = detector.detect(frame, camera_id="cam1", frame_id=42)
```

Mock Detection Characteristics:
- Players: 75-95% confidence, distributed across field
- Ball: 50-85% confidence (lower due to smaller size)
- Movement: Objects move consistently across frames
- Reproducibility: Seeded by camera_id + frame_id for deterministic results

Future: Real YOLO Models

The architecture is designed to support real object detection models with minimal changes.

Recommended Models

1. **YOLOv8** (Ultralytics)
   - State-of-the-art accuracy
   - Fast inference (30+ FPS on GPU)
   - Easy fine-tuning
   - Pre-trained on COCO dataset

2. **YOLOv9** (Latest)
   - Improved accuracy and speed
   - Better handling of small objects (good for ball detection)

3. **Custom Fine-Tuned Models**
   - Train on rugby-specific dataset
   - Optimize for player and ball detection
   - Handle rugby-specific challenges (scrums, rucks, occlusions)

Integration Steps

To swap in a real YOLO model:

1. Install Dependencies

```
pip install ultralytics
```

2. Update Detector Initialization

```
from ultralytics import YOLO

class Detector:
    def __init__(self, use_mock=False, model_path="yolov8n.pt"):
        if not use_mock:
            self.model = YOLO(model_path)
        # ... rest of init
```

3. Implement Real Detection Method

```python
def _detect_with_model(self, frame, camera_id, frame_id):
    results = self.model(frame, conf=self.confidence_threshold)
    detections = []

    for result in results:
        for box in result.boxes:
            # Map YOLO class IDs to 'player' or 'ball'
            class_id = int(box.cls[0])
            class_name = self._map_class_id(class_id)

            if class_name not in ['player', 'ball']:
                continue

            # Extract bbox in (x, y, w, h) format
            x1, y1, x2, y2 = box.xyxy[0].cpu().numpy()
            bbox = (float(x1), float(y1), float(x2-x1), float(y2-y1))

            detection = Detection(
                camera_id=camera_id,
                frame_id=frame_id,
                bbox=bbox,
                class_name=class_name,
                confidence=float(box.conf[0])
            )
            detections.append(detection)

    return detections
```

4. Add Class ID Mapping

```python
def _map_class_id(self, class_id: int) -> str:
    # COCO dataset: person = 0
    # Custom model: define your own mapping
    if class_id == 0:
        return 'player'
    if class_id == 37:  # sports ball in COCO
        return 'ball'
    return 'unknown'
```

5. Update Main Code

```python
# Use real detector
detector = Detector(use_mock=False, model_path="yolov8n.pt")
```

Tracking Methodology

IOU-Based Tracking

We use **Intersection over Union (IOU)** for matching detections to existing tracks:

1. **For each detection**: Compute IOU with all active tracks of the same class
2. **Best Match**: Assign detection to track with highest IOU (above threshold)
3. **Unmatched Detections**: Create new tracks
4. **Track Aging**: Deactivate tracks not updated for `max_age` frames

IOU Computation:

```
IOU = Intersection Area / Union Area

Where:
- Intersection: Overlap between two bounding boxes
- Union: Total area covered by both bounding boxes
```

Tracking Parameters

```
tracker = Tracker(
    iou_threshold=0.3,   # Minimum IOU for matching (0.0 to 1.0)
    max_age=30,          # Max frames without update before deletion
    min_hits=3           # Min detections before track is confirmed
)
```

Parameter Tuning:
- **iou_threshold**: Lower = more lenient matching (good for fast motion)
- **max_age**: Higher = tracks survive longer gaps (good for occlusions)
- **min_hits**: Higher = fewer false positive tracks (good for noisy detections)

Per-Camera Tracking

Each camera maintains **separate track IDs**:
- Avoids cross-camera ID conflicts
- Simplifies tracking logic
- Allows independent tracking quality per camera
- Future: Can add cross-camera re-identification in later phases

API Usage

Basic Workflow

```python
from ml.detector import Detector
from ml.tracker import Tracker
from ml.detection_tracking_api import (
    ClipDefinition,
    run_detection_and_tracking,
    get_detections_summary
)

# 1. Define clip
clip = ClipDefinition(
    clip_id="match_123_pass_45",
    camera_ids=["cam1", "cam2", "cam3"],
    frames_per_camera={
        "cam1": [frame1_cam1, frame2_cam1, ...],
        "cam2": [frame1_cam2, frame2_cam2, ...],
        "cam3": [frame1_cam3, frame2_cam3, ...],
    },
    start_frame=0,
    end_frame=100
)

# 2. Run detection and tracking
result = run_detection_and_tracking(clip)

# 3. Access results
print(f"Processed {result.frame_count} frames")
print(f"Total detections: {result.detection_count}")
print(f"Total tracks: {result.track_count}")

# 4. Get summary
summary = get_detections_summary(result)
print(f"Player detections: {summary['player_detections']}")
print(f"Ball detections: {summary['ball_detections']}")

# 5. Access per-camera data
for camera_id, tracks in result.tracks_per_camera.items():
    print(f"{camera_id}: {len(tracks)} tracks")
    for track in tracks:
        print(f"  Track {track.track_id}: {track.class_name}, "
              f"{track.length} detections")
```

Custom Detector and Tracker

```
# Use custom parameters
detector = Detector(
    use_mock=True,
    confidence_threshold=0.6  # Higher threshold for fewer false positives
)

trackers = {
    "cam1": Tracker(iou_threshold=0.3, max_age=20),
    "cam2": Tracker(iou_threshold=0.3, max_age=20),
}

result = run_detection_and_tracking(
    clip,
    detector=detector,
    tracker_per_camera=trackers
)
```

Performance Considerations

Current Performance (Mock Detector)

- **Detection**: ~50 FPS on CPU (negligible overhead)
- **Tracking**: ~1000 FPS on CPU (very fast)
- **Bottleneck**: Frame I/O and preprocessing

Expected Performance (Real YOLO)

Model	Hardware	FPS	Notes
YOLOv8n	CPU	10	Nano model, fast inference
YOLOv8n	GPU	100+	Excellent for real-time
YOLOv8m	CPU	5	Medium model, better accuracy
YOLOv8m	GPU	50+	Good balance
YOLOv8x	GPU	20+	Best accuracy, slower

Optimization Strategies:
1. **Model Quantization**: Use FP16 or INT8 for faster inference
2. **Batch Processing**: Process multiple frames simultaneously
3. **Frame Skipping**: Process every Nth frame for near-real-time
4. **GPU Acceleration**: Use CUDA/TensorRT for maximum performance

---

Limitations and Future Improvements

Current Limitations

1. **Mock Detections**: Not real, limited testing value for ML accuracy
2. **IOU Tracking**: Simple approach, struggles with:
   - Fast motion
   - Heavy occlusions
   - Identity switches
3. **No Cross-Camera Association**: Each camera tracks independently
4. **No Re-identification**: Lost tracks cannot be recovered

Phase 5+ Improvements

1. **Real Object Detection**:
   - Train YOLOv8 on rugby-specific dataset
   - Fine-tune for ball detection (small object challenge)
   - Handle occlusions in scrums/rucks

2. **Advanced Tracking**:
   - Implement DeepSORT or ByteTrack for appearance-based tracking
   - Add Kalman filter for motion prediction
   - Handle occlusions better

3. **Cross-Camera Tracking**:
   - Associate tracks across cameras using 3D position
   - Maintain global track IDs
   - Handle camera handoffs

4. **Re-identification**:
   - Use jersey numbers for player identification
   - Re-associate lost tracks

5. **Performance Optimization**:
   - TensorRT acceleration
   - Multi-GPU support
   - Asynchronous frame processing

Testing Strategy

Unit Tests

All core functionality has comprehensive unit tests:

- **test_detector.py**: Detection dataclass, mock detection generation, validation
- **test_tracker.py**: Track dataclass, IOU computation, track updates, aging

Test Data

Uses synthetic data generated by `mock_data_generator.py`:
- Known ground truth
- Deterministic results
- Fast test execution

Integration Tests

Backend integration tests validate full pipeline:
- Video ingestion → Detection → Tracking → 3D reconstruction

Dependencies

Python Packages

Current (Mock Detector):

```
numpy>=1.24.0
opencv-python>=4.8.0
```

Future (Real Detector):

```
ultralytics>=8.0.0  # YOLOv8
torch>=2.0.0        # PyTorch backend
```

Model Files

Pre-trained Models (when using real detection):
- Download from Ultralytics: `yolov8n.pt`, `yolov8s.pt`, etc.
- Store in `/ml/models/` directory
- Custom models: Train and export to ONNX or TorchScript

Conclusion

Phase 4 implements a complete detection and tracking subsystem with:
- ✅ Clean data structures (Detection, Track)
- ✅ Modular architecture (Detector, Tracker, API)
- ✅ Mock implementation for development
- ✅ Production-ready architecture for real models
- ✅ Comprehensive unit tests
- ✅ Clear documentation

Next Steps:
- Phase 5: Use detections for 3D reconstruction
- Phase 6: Integrate with decision engine
- Later phases: Swap to real YOLO models and fine-tune

For questions or issues, refer to:
- Code: `/ml/detector.py`, `/ml/tracker.py`, `/ml/detection_tracking_api.py`
- Tests: `/ml/tests/test_detector.py`, `/ml/tests/test_tracker.py`
- Architecture: `ARCHITECTURE_OVERVIEW.md`
- Contributing: `CONTRIBUTING.md`