Spatial Model and Coordinate Systems

Phase 5: 3D Reconstruction & Field Reference Frame

This document describes the 3D reconstruction pipeline, coordinate systems, and spatial modeling for the Rugby Vision system.

Table of Contents

1. Overview
2. Coordinate Systems
3. Camera Calibration Model
4. Triangulation Method
5. Rugby Field Reference Frame
6. Spatial Model Pipeline
7. Error Handling
8. Future Improvements

Overview

Phase 5 transforms 2D detections and tracks from Phase 4 into 3D positions in a rugby field coordinate system. The pipeline consists of:

1. **Camera Calibration**: Load intrinsic and extrinsic parameters for each camera
2. **Triangulation**: Reconstruct 3D points from multi-view 2D observations
3. **Field Coordinate Transform**: Map 3D points into rugby field reference frame
4. **Frame State Generation**: Build time series of ball and player positions

Key Components

- `ml/calibration.py` : Camera calibration data structures and loading
- `ml/triangulation.py` : Multi-view triangulation (DLT method)
- `ml/field_coords.py` : Rugby field coordinate system definition
- `ml/spatial_model.py` : Integration and frame state generation

Coordinate Systems

1. Camera Coordinates (OpenCV Convention)

Each camera has its own local coordinate system:

```
      Y (down)
      |
      |
      +------ X (right)
     /
    /
   Z (forward, into scene)
```

- **Origin**: Camera optical center
- **X-axis**: Right (in image plane)
- **Y-axis**: Down (in image plane)
- **Z-axis**: Forward (depth, into the scene)

2. World Coordinates (Rugby Field)

Global 3D coordinate system aligned with the rugby field:

```
                  Z (up)
                  |
                  |
    Try Line 0    |      Try Line 1
        |         |         |
        |    +----+----X (touchline)
        |   /     |
        |  /      |
        | /       |
        |/        |
      Y (field length)
```

- **Origin**: Corner of try line (left corner when viewing from field center)
- **X-axis**: Along touchline (0 to 70m)
- **Y-axis**: Along field length (0 to 100m)
- **Z-axis**: Vertical, upward (0 = ground level)

3. Image/Pixel Coordinates

2D coordinates in camera images:

```
    (0,0) --------- X (width) ---------> (1920, 0)
      |
      |
      Y (height)
      |
      |
      v
    (0,1080) -------------------------> (1920, 1080)
```

- **Origin**: Top-left corner of image
- **X-axis**: Image width (pixels)
- **Y-axis**: Image height (pixels)

Camera Calibration Model

Intrinsic Matrix

The intrinsic matrix **K** (3×3) contains internal camera parameters:

```
K = [[fx,  0, cx],
     [ 0, fy, cy],
     [ 0,  0,  1]]
```

Parameters:
- `fx`, `fy`: Focal lengths in pixels (X and Y directions)
- `cx`, `cy`: Principal point (optical center) in pixels

Typical values for 1920×1080 camera:
- `fx`, `fy`: 1400-1600 (narrower FOV) or 800-1200 (wider FOV)
- `cx`: ~960 (half of image width)
- `cy`: ~540 (half of image height)

Extrinsic Matrix

The extrinsic matrix **[R|t]** (4×4) transforms world points to camera coordinates:

```
[R|t] = [[r11, r12, r13, tx],
         [r21, r22, r23, ty],
         [r31, r32, r33, tz],
         [  0,   0,   0,  1]]
```

Components:
- **R** (3×3): Rotation matrix (orthogonal, det(R) = 1)
- **t** (3×1): Translation vector (camera position in world coords)

Projection Matrix

The full projection matrix **P** (3×4) maps 3D world points to 2D image points:

```
P = K [R|t]

[u]       [X]
[v]  ~ P  [Y]
[1]       [Z]
            [1]
```

Where `~` denotes equality up to scale (homogeneous coordinates).

Validation

The system validates calibration parameters:
- ✅ Intrinsic matrix is 3×3 with bottom row [0, 0, 1]
- ✅ Focal lengths are positive
- ✅ Extrinsic matrix is 4×4 with bottom row [0, 0, 0, 1]
- ✅ Rotation matrix is orthogonal: R × R^T = I

Triangulation Method

Direct Linear Transform (DLT)

We use the **Direct Linear Transform** method for multi-view triangulation.

Problem Formulation

Given:
- N ≥ 2 calibrated cameras
- 2D observations (u_i, v_i) in each camera i

Find:
- 3D point X = [X, Y, Z]^T in world coordinates

Algorithm

1. **Build Linear System**: For each observation (u, v) in camera with projection matrix P:

```
u × P[2,:] - P[0,:] = 0
v × P[2,:] - P[1,:] = 0
```

This gives us matrix equation: **A × X_h = 0** where X_h = [X, Y, Z, 1]^T

1. **Solve via SVD**: The solution is the right singular vector corresponding to the smallest singular value:

```
A = U Σ V^T
X_h = V[:, -1]  (last column of V)
```

1. **Convert to 3D**: Divide homogeneous coordinates by the 4th component:

```
X = X_h[0:3] / X_h[3]
```

Reprojection Error

To validate triangulation quality, we compute reprojection error:

```
error_i = sqrt((u_i - u_proj_i)^2 + (v_i - v_proj_i)^2)
```

where (u_proj_i, v_proj_i) is the projection of X back into camera i.

Threshold: We reject reconstructions with max error > 100 pixels.

Edge Cases

The triangulation function handles:
- ❌ Fewer than 2 observations → ValueError
- ❌ Degenerate geometry (rank(A) < 3) → return None
- ❌ Point at infinity (X_h[3] ≈ 0) → return None
- ❌ High reprojection error → return None

Rugby Field Reference Frame

Field Model

The `FieldModel` dataclass defines rugby field dimensions and the coordinate transform:

```python
@dataclass
class FieldModel:
    field_length: float = 100.0  # Try line to try line (m)
    field_width: float = 70.0    # Touchline to touchline (m)
    try_line_0: float = 0.0      # First try line Y-coord
    try_line_1: float = 100.0    # Second try line Y-coord
    origin_offset: np.ndarray    # World origin to field origin
    rotation_matrix: np.ndarray  # 3×3 rotation to field coords
```

Standard Dimensions

Rugby field dimensions follow World Rugby regulations:

```
              70m (width)
        |<-------------------->|

    ═════╪═════════════════════╪═════ Try Line 0 (Y=0)
         |                    |
         |                    |
         |     PLAYING FIELD  |      100m
         |                    |      (length)
         |                    |
    ═════╪═════════════════════╪═════ Try Line 1 (Y=100m)

        X=0                  X=70m
```

Coordinate Transform

To transform a 3D point from world to field coordinates:

```
field_point = R @ (world_point - offset)
```

Where:
- `R`: Rotation matrix (3×3)
- `offset`: Translation vector (3×1)

Validation

Field coordinate validation:
- ✅ **Z ≥ 0**: Points must be at or above ground level
- ✅ **Bounds checking**: Points should be within field dimensions (with tolerance)
- ✅ **Reasonable height**: Z < 50m (sanity check)

Tolerance: Default 5m for out-of-bounds plays (e.g., throw-ins)

Spatial Model Pipeline

FrameState Data Structure

Each frame’s spatial state is captured in:

```python
@dataclass
class FrameState:
    timestamp: float                        # Frame timestamp (seconds)
    frame_number: int                       # Frame index
    ball_pos_3d: Optional[np.ndarray]       # Ball position [X, Y, Z] or None
    players_pos_3d: Dict[int, np.ndarray]   # track_id -> [X, Y, Z]
    ball_detected: bool                     # Whether ball was detected
    n_players_tracked: int                  # Number of players in 3D
```

Pipeline Steps

```
┌─────────────────────────────────────────────────────────┐
│ Phase 4: Multi-Camera Detection & Tracking              │
│  - 2D bounding boxes for ball and players               │
│  - Track IDs maintained across frames                   │
└─────────────────┬───────────────────────────────────────┘
                  │
                  v
┌─────────────────────────────────────────────────────────┐
│ Phase 5: 3D Reconstruction                               │
└─────────────────────────────────────────────────────────┘
                  │
     ┌────────────┼────────────┐
     │            │            │
     v            v            v
┌──────────┐ ┌──────────┐ ┌───────────┐
│ Camera   │ │ Camera   │ │ Camera    │
│ Calib.   │ │ Calib.   │ │ Calib.    │
└────┬─────┘ └────┬─────┘ └─────┬─────┘
     │            │             │
     └────────────┼─────────────┘
                  │
                  v
         ┌───────────────┐
         │ Triangulation │  ← Multi-view DLT
         │   (per frame) │
         └───────┬───────┘
                  │
                  v
         ┌───────────────┐
         │ Transform to  │  ← Field coord system
         │ Field Coords  │
         └───────┬───────┘
                  │
                  v
         ┌───────────────┐
         │  FrameState   │  ← 3D positions + metadata
         └───────────────┘
```

Ball Position Reconstruction

For each frame:

1. **Collect observations**: Find ball detections across all cameras
2. **Select best detection**: Use highest confidence if multiple balls detected per camera
3. **Get pixel coordinates**: Center of bounding box (x_center, y_center)
4. **Triangulate**: Apply DLT with ≥2 observations
5. **Transform**: Map to field coordinates
6. **Validate**: Check $Z \geq 0$ and reasonable bounds

Player Position Reconstruction

For each tracked player (per track_id):

1. **Match tracks**: Find same track_id across cameras
2. **Get pixel coordinates**: Bottom-center of bounding box (feet position)
3. **Triangulate**: Apply DLT with ≥2 observations
4. **Transform**: Map to field coordinates
5. **Validate**: Check $Z \geq 0$ and field bounds

Note: Players use bottom-center of bounding box (feet) while ball uses center.

Error Handling

Guard Clauses

Following project coding standards, we use guard clauses for validation:

```
# Insufficient cameras
if len(observations) < 2:
    raise ValueError("Need at least 2 observations for triangulation")

# Degenerate geometry
if np.linalg.matrix_rank(A) < 3:
    return None  # Cannot solve

# Point below ground
if field_point[2] < -0.1:
    return None  # Invalid position
```

Missing Data Handling

Scenario	Handling
Ball not detected in frame	`ball_pos_3d = None`, `ball_detected = False`
Ball detected in <2 cameras	Cannot triangulate → `ball_pos_3d = None`
Player track in <2 cameras	Skip this player for this frame
High reprojection error	Reject triangulation → return None
Invalid field coordinates	Return None, skip this detection

Logging

Key events are logged:
- ✅ Calibration file loaded successfully
- ⚠️ Triangulation failed (degenerate geometry)
- ⚠️ Point rejected (high reprojection error)
- ⚠️ Point rejected (invalid field coordinates)

Future Improvements

Phase 6: Temporal Filtering

- **Kalman filtering**: Smooth 3D trajectories over time
- **Velocity estimation**: Compute ball/player velocities
- **Occlusion handling**: Predict positions during missing detections
- **Outlier rejection**: Use temporal consistency to filter bad reconstructions

Phase 7: Advanced Calibration

- **Online calibration refinement**: Use field lines for continuous calibration
- **Rolling shutter correction**: Account for camera sensor effects
- **Lens distortion**: Model and correct radial/tangential distortion
- **Auto-calibration**: Recover calibration from video alone (structure-from-motion)

Phase 8: Decision Engine

- **Forward pass detection**: Use ball 3D trajectory + player positions
- **Pass direction computation**: Analyze ball velocity and player orientations
- **Rule violations**: Detect offside, knock-ons, etc.
- **Confidence scoring**: Probabilistic decision making

Performance Optimization

- **Batch triangulation**: Vectorize operations for multiple points
- **GPU acceleration**: Use CUDA for projection matrix operations
- **Multi-threading**: Parallelize per-frame reconstruction
- **Caching**: Store projection matrices to avoid recomputation

Robustness Improvements

- **RANSAC triangulation**: Reject outlier observations
- **Bundle adjustment**: Jointly optimize 3D points and camera poses
- **Multi-hypothesis tracking**: Handle track ID ambiguities
- **Uncertainty quantification**: Estimate 3D position covariance

---

References

1. **Hartley, R. & Zisserman, A.** (2004). Multiple View Geometry in Computer Vision. Cambridge University Press.
   - Chapter 12: Structure Computation (DLT triangulation)

2. **OpenCV Documentation**. Camera Calibration and 3D Reconstruction.
   - https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html

3. **World Rugby**. Laws of the Game - Rugby Union.
   - Field dimensions and regulations

Appendix: Code Examples

Loading Calibration

```
from ml.calibration import load_calibration_from_file

calibrations = load_calibration_from_file('config/camera_calibration_example.json')
# Returns: Dict[camera_id, CameraCalibration]
```

Triangulating a Point

```
from ml.triangulation import triangulate_point

observations = [
    (calibrations['cam_0'], (640.5, 480.2)),
    (calibrations['cam_1'], (1024.1, 512.7)),
    (calibrations['cam_2'], (800.0, 600.0)),
]

point_3d = triangulate_point(observations)  # Returns [X, Y, Z] or None
```

Transforming to Field Coordinates

```
from ml.field_coords import transform_to_field_coords, get_standard_rugby_field

field_model = get_standard_rugby_field()
field_point = transform_to_field_coords(point_3d, field_model)
# Returns [X_field, Y_field, Z_field] or None
```

Building Frame States

```python
from ml.spatial_model import build_frame_states

frame_states = build_frame_states(
    tracking_results=tracking_results,  # Dict[camera_id, DetectionTrackingResult]
    calibrations=calibrations,          # Dict[camera_id, CameraCalibration]
    field_model=field_model,            # FieldModel
    fps=30.0                            # Frames per second
)

# Returns List[FrameState]
```

Document Version: 1.0
Last Updated: Phase 5 Implementation
Author: Rugby Vision Team