Video Ingest Design - Multi-Camera Synchronization

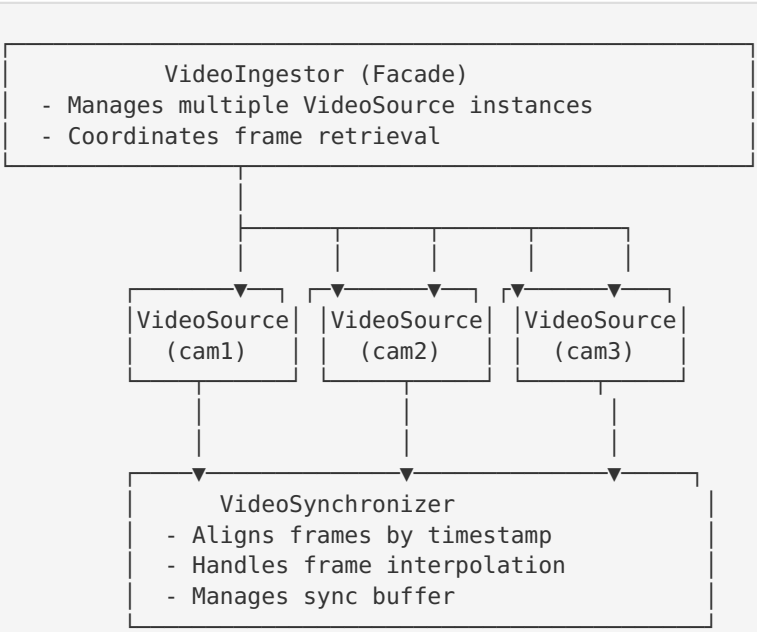
Overview

The video ingestion and synchronization subsystem is responsible for loading video from multiple camera sources, aligning frames temporally, and providing synchronized frame batches to downstream processing stages. This is a critical foundation for accurate 3D reconstruction and decision making.

Design Goals

1. **Temporal Accuracy**: Synchronize frames across cameras within ±33ms (1 frame @ 30fps)
2. **Flexibility**: Support both pre-recorded files and live streams (Phase 2+)
3. **Robustness**: Handle missing frames, jitter, and variable frame rates gracefully
4. **Performance**: Minimize latency and memory footprint
5. **Simplicity**: Clean API for downstream consumers

Architecture



Core Components

1. VideoSource

Represents a single camera video source (file or stream).

Responsibilities:
- Open and read video file/stream
- Extract frames sequentially

- Provide frame metadata (timestamp, frame number, resolution)
- Handle format conversions

Key Methods:

```
class VideoSource:
    def __init__(self, source_path: str, camera_id: str)
    def open(self) -> bool
    def read_frame(self) -> Optional[Frame]
    def seek(self, timestamp: float) -> bool
    def get_metadata(self) -> VideoMetadata
    def release(self) -> None
```

Frame Extraction:
- Uses OpenCV `cv2.VideoCapture` for reading
- Computes timestamp from frame number and FPS
- Handles codec-specific quirks

2. VideoIngestor

Main facade for managing multiple video sources.

Responsibilities:
- Initialize multiple VideoSource instances
- Coordinate frame reading across sources
- Pass frames to VideoSynchronizer
- Provide high-level API for clip analysis

Key Methods:

```
class VideoIngestor:
    def __init__(self, sources: List[VideoSourceConfig])
    def load_sources(self) -> None
    def get_frame_at_time(self, timestamp: float) -> Dict[str, Frame]
    def get_frame_batch(self, start_time: float, end_time: float) -> List[Dict[str,
Frame]]
    def release(self) -> None
```

Usage Example:

```
sources = [
    VideoSourceConfig(path="/path/cam1.mp4", camera_id="cam1"),
    VideoSourceConfig(path="/path/cam2.mp4", camera_id="cam2"),
    VideoSourceConfig(path="/path/cam3.mp4", camera_id="cam3"),
]

ingestor = VideoIngestor(sources)
ingestor.load_sources()

# Get synchronized frames at specific time
frames = ingestor.get_frame_at_time(timestamp=10.5)  # {camera_id -> Frame}

# Get batch of frames over time window
batch = ingestor.get_frame_batch(start_time=10.0, end_time=15.0)  # List of frame
dicts
```

3. VideoSynchronizer

Core synchronization logic.

Responsibilities:
- Align frames from different cameras to common timeline
- Handle frame rate differences
- Interpolate or skip frames as needed
- Manage sync buffer for efficient access

Key Methods:

```
class VideoSynchronizer:
    def __init__(self, sync_tolerance: float = 0.033)
    def synchronize_frames(self, frames: Dict[str, List[Frame]], target_time: float) -
> Dict[str, Frame]
    def build_sync_buffer(self, frames: Dict[str, List[Frame]]) -> SyncBuffer
    def get_closest_frames(self, buffer: SyncBuffer, timestamp: float) -> Dict[str,
Frame]
```

Timestamp Alignment Strategy

Problem Statement

Different cameras may have:
1. **Different start times**: Camera A starts at t=0, Camera B starts at t=2.3s
2. **Different frame rates**: Camera A at 30fps, Camera B at 25fps, Camera C at 60fps
3. **Clock drift**: Internal clocks may not be perfectly synchronized
4. **Jitter**: Frame timestamps may have small variations

Solution: Master Timeline Approach

1. Establish Master Timeline

- Choose one camera as reference (typically widest angle or most stable)
- OR use external timecode if available (broadcast scenario)
- All timestamps normalized to master timeline

2. Offset Correction

For each camera, compute initial time offset:

```
def compute_time_offset(reference_camera: VideoSource, target_camera: VideoSource) ->
float:
    """
    Compute time offset between target camera and reference.
    Uses first frame timestamps.
    """
    ref_first_timestamp = reference_camera.get_first_frame_timestamp()
    target_first_timestamp = target_camera.get_first_frame_timestamp()
    return target_first_timestamp - ref_first_timestamp
```

3. Frame Alignment

Given target timestamp `t` on master timeline:

For each camera:
1. Adjust for offset: `t_camera = t - offset`
2. Find closest frame: `frame = find_closest_frame(camera, t_camera)`
3. Check tolerance: `if abs(frame.timestamp - t_camera) > tolerance: mark as missing`

4. Tolerance Threshold

- Default tolerance: **±33ms** (1 frame at 30fps)
- If no frame within tolerance, mark camera as `missing` for that timestamp
- Decision: skip this timestamp OR use interpolation (see below)

Handling Frame Rate Differences

Scenario: Cameras with Different FPS

- Camera A: 30fps (frames every 33.3ms)
- Camera B: 25fps (frames every 40ms)
- Camera C: 60fps (frames every 16.7ms)

Strategy: Normalize to Target Frame Rate

Option 1: Downsample to Lowest FPS
- Target FPS = min(camera_fps) = 25fps
- Sample frames at 40ms intervals
- Guarantees no interpolation needed
- Trade-off: Wastes high FPS camera data

Option 2: Upsample to Highest FPS (Phase 2+)
- Target FPS = max(camera_fps) = 60fps
- Interpolate missing frames for slower cameras
- Trade-off: Adds computational cost

Phase 1 Implementation: Option 1 (Downsample)

```
def compute_target_fps(cameras: List[VideoSource]) -> float:
    camera_fps = [cam.get_fps() for cam in cameras]
    return min(camera_fps)
```

Handling Missing Frames

Causes of Missing Frames

1. **Dropped frames**: Camera buffer overflow, encoding issues
2. **Occlusion**: Ball/player not visible in camera
3. **Timing gaps**: No frame within tolerance window
4. **Corruption**: Frame read error

Detection

```python
@dataclass
class Frame:
    camera_id: str
    timestamp: float
    frame_number: int
    image: Optional[np.ndarray]  # None if missing
    is_valid: bool
```

Handling Strategies

Strategy 1: Skip Timestamp (Conservative)
- If any required camera is missing, skip this timestamp entirely
- Pro: No false data
- Con: May skip too many frames

Strategy 2: Partial Sync (Flexible)
- Proceed with available cameras only
- 3D reconstruction may degrade but still possible with 2+ cameras
- Pro: Maximizes data usage
- Con: Lower accuracy when <3 cameras

Strategy 3: Interpolation (Advanced - Phase 2+)
- Linearly interpolate missing frames from neighbors
- Only for brief gaps (<3 frames)
- Pro: Smooth trajectory
- Con: Introduces synthetic data

Phase 1 Implementation: Strategy 2 (Partial Sync)

Require minimum 2 cameras with valid frames to proceed.

```python
def is_frame_batch_valid(frames: Dict[str, Frame], min_cameras: int = 2) -> bool:
    valid_count = sum(1 for f in frames.values() if f.is_valid)
    return valid_count >= min_cameras
```

Frame Jitter and Clock Drift

Jitter

Small random variations in frame timestamps (±5ms typical).

Mitigation:
- Use tolerance window (±33ms)
- Average timestamps over multiple frames if needed

Clock Drift

Gradual divergence of camera clocks over time.

Detection:

```
def detect_drift(camera: VideoSource, reference: VideoSource, duration: float) ->
float:
    expected_frames = duration * camera.get_fps()
    actual_frames = camera.get_frame_count()
    drift_frames = actual_frames - expected_frames
    drift_seconds = drift_frames / camera.get_fps()
    return drift_seconds
```

Correction (Phase 2+):
- Compute drift rate (seconds per second)
- Apply time-varying offset correction
- Re-synchronize periodically

Phase 1: Assume drift is negligible for short clips (<30 seconds).

Plugging In Different Video Sources

Recorded Files (Phase 1)

Supported Formats:
- MP4 (H.264, H.265)
- MOV (QuickTime)
- AVI

Interface:

```
@dataclass
class FileVideoSourceConfig:
    path: str
    camera_id: str
    offset_seconds: float = 0.0  # Manual time offset if needed
```

Implementation:
- Use `cv2.VideoCapture(path)` to open
- Read frames sequentially with `cap.read()`
- Compute timestamp from `cap.get(cv2.CAP_PROP_POS_MSEC)` or frame number

Live Streams (Phase 2+)

Supported Protocols:
- RTSP (IP cameras)
- RTMP (streaming platforms)
- WebRTC (browser-based)

Interface:

```
@dataclass
class StreamVideoSourceConfig:
    url: str
    camera_id: str
    buffer_size: int = 30  # Frames to buffer
```

Implementation:
- Use `cv2.VideoCapture(url)` for RTSP

- Or specialized libraries (e.g., `aiortc` for WebRTC)
- Handle buffering and latency
- Timestamp from system clock or stream metadata

Synthetic Test Data (Phase 1)

Purpose: Testing without real match footage.

Interface:

```
class SyntheticVideoSource(VideoSource):
    def __init__(self, camera_id: str, duration: float, fps: float, scene_config:
SceneConfig)
```

Implementation:
- Generate frames programmatically
- Render simple shapes (ball, players) moving in 3D space
- Project to 2D using known camera matrices
- Add noise, motion blur for realism

See `ml/mock_data_generator.py` for details.

Data Structures

Frame

```
@dataclass
class Frame:
    camera_id: str
    timestamp: float  # Seconds, synchronized to master timeline
    frame_number: int  # Original frame index in source
    image: Optional[np.ndarray]  # HxWx3 BGR image, None if missing
    is_valid: bool
    metadata: Dict[str, Any]  # Resolution, codec, etc.
```

VideoMetadata

```
@dataclass
class VideoMetadata:
    camera_id: str
    width: int
    height: int
    fps: float
    total_frames: int
    duration: float  # Seconds
    codec: str
```

VideoSourceConfig

```python
@dataclass
class VideoSourceConfig:
    path: str  # File path or stream URL
    camera_id: str
    source_type: str  # "file", "stream", "synthetic"
    offset_seconds: float = 0.0
    calibration_path: Optional[str] = None  # Path to calibration file
```

SyncBuffer

Internal data structure for efficient frame lookup.

```python
@dataclass
class SyncBuffer:
    frames: Dict[str, List[Frame]]  # camera_id -> sorted list of frames
    start_time: float
    end_time: float
    target_fps: float
```

API Examples

Example 1: Load and Synchronize Recorded Clips

```python
from backend.video_ingest import VideoIngestor, VideoSourceConfig

# Configure sources
sources = [
    VideoSourceConfig(
        path="/data/match1_cam1.mp4",
        camera_id="main_cam",
        source_type="file",
    ),
    VideoSourceConfig(
        path="/data/match1_cam2.mp4",
        camera_id="side_cam",
        source_type="file",
        offset_seconds=0.5,  # Manual offset if needed
    ),
    VideoSourceConfig(
        path="/data/match1_cam3.mp4",
        camera_id="end_cam",
        source_type="file",
    ),
]

# Initialize ingestor
ingestor = VideoIngestor(sources)
ingestor.load_sources()

# Get single synchronized frame at t=10.5s
frames = ingestor.get_frame_at_time(10.5)
for camera_id, frame in frames.items():
    if frame.is_valid:
        print(f"{camera_id}: frame {frame.frame_number}, shape {frame.image.shape}")

# Get batch of frames from t=10s to t=15s
batch = ingestor.get_frame_batch(start_time=10.0, end_time=15.0)
print(f"Retrieved {len(batch)} synchronized frame sets")

# Clean up
ingestor.release()
```

Example 2: Using Synthetic Data

```
from ml.mock_data_generator import SyntheticVideoGenerator, SceneConfig

# Configure synthetic scene
scene = SceneConfig(
    field_length=100.0,  # meters
    field_width=70.0,
    ball_trajectory=[(0, 50, 1), (10, 55, 1.5), (20, 60, 1)],  # (x, y, z) over time
    player_positions=[(5, 52), (15, 58)],  # (x, y) for passer and receiver
)

# Generate synthetic videos
generator = SyntheticVideoGenerator()
generator.create_video(
    output_path="/tmp/synthetic_cam1.mp4",
    camera_id="cam1",
    scene=scene,
    duration=5.0,
    fps=30.0,
)
generator.create_video(
    output_path="/tmp/synthetic_cam2.mp4",
    camera_id="cam2",
    scene=scene,
    duration=5.0,
    fps=30.0,
)

# Now ingest as normal
sources = [
    VideoSourceConfig(path="/tmp/synthetic_cam1.mp4", camera_id="cam1", source_type="f
ile"),
    VideoSourceConfig(path="/tmp/synthetic_cam2.mp4", camera_id="cam2", source_type="f
ile"),
]
ingestor = VideoIngestor(sources)
ingestor.load_sources()
```

Performance Considerations

Memory Management

Problem: Loading all frames into memory is impractical.

Solution: Streaming with sliding window buffer.

```
class VideoIngestor:
    def __init__(self, sources: List[VideoSourceConfig], buffer_window: float = 5.0):
        self.buffer_window = buffer_window  # Keep only 5 seconds in memory
```

- As new frames are read, old frames are discarded
- Seek backwards if needed (less efficient)

I/O Optimization

- **Parallel Reading**: Use threading to read from multiple files concurrently
- **Prefetching**: Read ahead to minimize seek time

- **Decoding**: Decode frames in parallel with processing

Latency Budget

For Phase 1 target of 10s total latency:

- **Video loading**: 2 seconds
- Open files: 0.5s
- Read and sync frames: 1.5s
- **Overhead**: Minimal (<0.5s)

Error Handling

Guard Clauses for Validation

```python
def load_sources(self, sources: List[VideoSourceConfig]) -> None:
    # Guard: Check sources list
    if not sources:
        raise ValueError("Sources list cannot be empty")

    # Guard: Minimum cameras
    if len(sources) < 2:
        raise ValueError("At least 2 cameras required for 3D reconstruction")

    # Guard: Check file existence
    for source in sources:
        if source.source_type == "file" and not os.path.exists(source.path):
            raise FileNotFoundError(f"Video file not found: {source.path}")

    # Proceed with loading
    self._load_video_sources(sources)
```

Runtime Error Recovery

```python
def get_frame_at_time(self, timestamp: float) -> Dict[str, Frame]:
    frames = {}

    for camera_id, source in self.sources.items():
        try:
            frame = source.read_frame_at(timestamp)
            frames[camera_id] = frame
        except Exception as e:
            logger.warning(f"Failed to read frame from {camera_id}: {e}")
            # Insert invalid frame placeholder
            frames[camera_id] = Frame(
                camera_id=camera_id,
                timestamp=timestamp,
                frame_number=-1,
                image=None,
                is_valid=False,
                metadata={},
            )

    return frames
```

Testing Strategy

Unit Tests

Test File: `backend/tests/test_video_ingest.py`

```
def test_video_source_opens_valid_file():
    source = VideoSource(path="test_data/sample.mp4", camera_id="test")
    assert source.open() is True
    metadata = source.get_metadata()
    assert metadata.fps > 0
    assert metadata.total_frames > 0

def test_video_source_fails_on_invalid_file():
    source = VideoSource(path="nonexistent.mp4", camera_id="test")
    assert source.open() is False

def test_video_synchronizer_aligns_frames_within_tolerance():
    # Create mock frames with known timestamps
    frames_cam1 = [Frame(camera_id="cam1", timestamp=1.000, ...)]
    frames_cam2 = [Frame(camera_id="cam2", timestamp=1.033, ...)]

    sync = VideoSynchronizer(sync_tolerance=0.040)
    aligned = sync.synchronize_frames({"cam1": frames_cam1, "cam2": frames_cam2}, tar-
get_time=1.0)

    assert "cam1" in aligned
    assert "cam2" in aligned
    assert abs(aligned["cam1"].timestamp - 1.0) < 0.040
    assert abs(aligned["cam2"].timestamp - 1.0) < 0.040
```

Integration Tests

Test with synthetic videos:

```
def test_ingestor_with_synthetic_videos():
    # Generate synthetic test videos
    generator = SyntheticVideoGenerator()
    generator.create_test_videos(output_dir="/tmp/test_videos")

    # Ingest
    sources = [
        VideoSourceConfig(path="/tmp/test_videos/cam1.mp4", camera_id="cam1",
source_type="file"),
        VideoSourceConfig(path="/tmp/test_videos/cam2.mp4", camera_id="cam2",
source_type="file"),
    ]
    ingestor = VideoIngestor(sources)
    ingestor.load_sources()

    # Verify synchronization
    frames = ingestor.get_frame_at_time(2.5)
    assert len(frames) == 2
    assert all(f.is_valid for f in frames.values())
```

Future Enhancements (Phase 2+)

Auto-Synchronization from Audio

- Extract audio tracks from videos
- Use cross-correlation to find time offset
- More robust than manual offset entry

Auto-Calibration from Field Markings

- Detect field lines in first frame
- Estimate camera pose from homography
- Validate against known field dimensions

Adaptive Frame Rate

- Dynamically adjust target FPS based on scene motion
- Higher FPS during fast action (pass events)
- Lower FPS during static scenes (scrum, lineout)

GPU-Accelerated Decoding

- Use hardware video decoders (NVDEC, QuickSync)
- Offload decoding from CPU
- Reduce latency significantly

References

- OpenCV VideoCapture documentation: https://docs.opencv.org/4.x/d8/dfe/classcv_1_1VideoCapture.html
- Multi-view synchronization paper: Svoboda et al., “A Convenient Multi-Camera Self-Calibration for Virtual Environments”
- Frame interpolation: “Video Frame Interpolation via Adaptive Separable Convolution”

Summary

This design provides a robust, flexible foundation for multi-camera video ingestion and synchronization. Key features:

- **Modular**: Separate concerns (ingestion, synchronization, source management)
- **Robust**: Handles missing frames, jitter, frame rate differences
- **Extensible**: Easy to add new source types (streams, synthetic)
- **Testable**: Clear interfaces for unit and integration testing
- **Guard Clause Compliant**: Early validation, minimal nesting

The implementation focuses on Phase 1 requirements (recorded files, simple synchronization) while being architected for easy extension to Phase 2+ (streams, interpolation, auto-calibration).