```python
from dataclasses import dataclass, field
from typing import List, Tuple, Optional
from src.ai import SkeletonPoint # Using SkeletonPoint defined in Phase 2
import numpy as np
import cv2
import math

# Reflecting C++ macros/constants from gRPCFileClientDlg.h
RESET_MOTION_STATUS = -1
READY_MOTION_STATUS = 0
SPEAK_MOTION_STATUS = 1
RAPID_MOTION_STATUS = 2

READY_LOCATION = 700 # Y-axis threshold for hand position in C++ (relative to 1080p)
RAPID_DISTANCE = 0.1 # Normalized distance threshold for rapid motion

# Warp corners from C++ (gRPCFileClientDlg.h)
# Used for normalizing the skeleton position before checking 'Ready' status.
# Target coordinates in 1920x1080 space.
WARP_CORNERS = np.float32([
    [720, 500],   # Target for Right Shoulder (Index 12)
    [1200, 500],  # Target for Left Shoulder (Index 11)
    [960, 400]    # Target for Nose (Index 0)
])

# Source indices for Affine Transform (C++: pixels[12], pixels[11], pixels[0])
# Note: MediaPipe 12=RightShoulder, 11=LeftShoulder, 0=Nose.
SRC_INDICES = [12, 11, 0]

@dataclass
class HandTurnDetector:
    """
    Port of C++ HandTurnDetector class.
    Detects hand turns based on angle change and deceleration.
    """
    def __init__(self, angle_deg_th: float = 30.0, speed_ratio_th: float = 0.8, min_speed: float = 1e-3):
        self.angle_rad_th = angle_deg_th * math.pi / 180.0
        self.speed_ratio_th = speed_ratio_th
        self.min_speed = min_speed
        self.reset()

    def reset(self):
        """Resets the detector's internal state."""
        self.prev_pos: Optional[np.ndarray] = None
        self.prev_vel: Optional[np.ndarray] = None
        self.has_prev_pos = False
        self.has_prev_vel = False
        self.frame_idx = 0

    def update(self, current_pos: Tuple[float, float], dt: float) -> bool:
        """
        Updates the detector with new hand position and time delta.
        Returns True if a "Turn + Slow" event is detected.
        """
        pos = np.array(current_pos, dtype=float)
        detected = False

        if not self.has_prev_pos:
            self.prev_pos = pos
            self.has_prev_pos = True
            self.frame_idx += 1
            return False

        # Calculate velocity
        if dt > 1e-6:
            vel = (pos - self.prev_pos) * (1.0 / dt)
        else:
            vel = pos - self.prev_pos

        speed = np.linalg.norm(vel)

        if self.has_prev_vel:
            prev_speed = np.linalg.norm(self.prev_vel)

            # Check logic only if moving fast enough
            if prev_speed > self.min_speed:
                # Unit vectors
                u1 = self.prev_vel / prev_speed
                u2 = vel / (speed + 1e-6)

                # Dot product for angle
                dot = np.dot(u1, u2)
                # Clamp for acos safety
                dot = max(-1.0, min(1.0, dot))
                dtheta = math.acos(dot)

                # Speed ratio
```

```
                ratio = speed / (prev_speed + 1e-6)

                # Condition: Angle > Threshold AND Deceleration (Ratio < Threshold)
                if dtheta > self.angle_rad_th and ratio < self.speed_ratio_th:
                    detected = True

        # Update state for next frame
        self.prev_pos = pos
        self.prev_vel = vel
        self.has_prev_vel = True
        self.frame_idx += 1

        return detected

def get_ref_hl_2d_points_mp(joints: List[SkeletonPoint]) -> List[Tuple[float, float]]:
    """
    Extracts 2D reference points for wrists from MediaPipe 3D skeleton coordinates.
    Matches C++ GetRefHL2DPointsMP function which applies Affine Transformation
    to normalize body position/scale before extracting wrist coordinates.

    Input: joints (List[SkeletonPoint], MediaPipe 3D skeleton coordinates)
    Output: List[Tuple[float, float]] (Normalized Left/Right Hand Wrist 2D Coordinates)
            Note: Returns normalized (0-1) coordinates based on 1080p height to match logic usage.
    """
    if not joints or len(joints) < 17: # Ensure enough points for wrists (idx 15 and 16)
        return []

    # 1. Convert relevant joints to Pixel Coordinates (assuming 1920x1080 as base like C++)
    # C++ uses hardcoded 1920.0 * x, 1080.0 * y
    BASE_W = 1920.0
    BASE_H = 1080.0

    def get_pt(idx):
        return [joints[idx].x * BASE_W, joints[idx].y * BASE_H]

    try:
        src_tri = np.float32([get_pt(i) for i in SRC_INDICES]) # [RightShoulder, LeftShoulder, Nose]

        # 2. Compute Affine Transform Matrix
        # Mapping src_tri (Current Person) -> WARP_CORNERS (Standard Template)
        M = cv2.getAffineTransform(src_tri, WARP_CORNERS)

        # 3. Transform Wrists
        # Left Wrist: 15, Right Wrist: 16
        wrists = np.array([get_pt(15), get_pt(16)], dtype=np.float32)
        # Reshape for cv2.transform: (N, 1, 2)
        wrists = wrists.reshape(-1, 1, 2)

        transformed_wrists = cv2.transform(wrists, M)

        # 4. Normalize back to 0-1 range (optional, but logic expects normalized threshold)
        # C++ logic checks y <= READY_LOCATION (700).
        # get_rough_hand_status_from_mp compares (y * 1080) vs 700 OR normalized y vs 700/1080.
        # Let's return Normalized Y based on BASE_H (1080) to keep existing logic consistent.
        # Transformed Y is in 1080p pixel space.

        left_wrist_trans = transformed_wrists[0][0]
        right_wrist_trans = transformed_wrists[1][0]

        # Clamp values (C++ does clamp)
        left_x = max(0.0, min(BASE_W, left_wrist_trans[0]))
        left_y = max(0.0, min(BASE_H, left_wrist_trans[1]))

        right_x = max(0.0, min(BASE_W, right_wrist_trans[0]))
        right_y = max(0.0, min(BASE_H, right_wrist_trans[1]))

        # Return normalized coordinates (x/W, y/H)
        return [
            (left_x / BASE_W, left_y / BASE_H),
            (right_x / BASE_W, right_y / BASE_H)
        ]

    except Exception as e:
        # Fallback if transform fails (e.g. points collinear)
        # Return raw coordinates
        return [
            (joints[15].x, joints[15].y),
            (joints[16].x, joints[16].y)
        ]

def get_rough_hand_status_from_mp(mp_pose: List[SkeletonPoint]) -> int:
    """
    Determines the rough hand status (Ready/Dominant/Non-dominant/Both).
    Uses Affine Transformed coordinates to check against READY_LOCATION.
    """
    if not mp_pose or len(mp_pose) < 17:
        return RESET_MOTION_STATUS # -1

    hl_points = get_ref_hl_2d_points_mp(mp_pose)
```

```python
    if not hl_points or len(hl_points) < 2:
        return RESET_MOTION_STATUS

    # C++ READY_LOCATION (700) is relative to 1080p.
    normalized_ready_location = READY_LOCATION / 1080.0

    left_hand_y = hl_points[0][1] # y-coordinate of left wrist
    right_hand_y = hl_points[1][1] # y-coordinate of right wrist

    left_hand_up = left_hand_y <= normalized_ready_location
    right_hand_up = right_hand_y <= normalized_ready_location

    if right_hand_up and left_hand_up:
        return 3 # Both hands up
    elif right_hand_up and not left_hand_up:
        return 1 # Right hand up
    elif not right_hand_up and left_hand_up:
        return 2 # Left hand up
    else:
        return 0 # Ready status (both down or below threshold)

def get_motion_status_from_mp(current_motion_status: int, cur_mp: List[SkeletonPoint], prev_mp: List[SkeletonPoint]) -> int:
    """
    Updates the motion status (READY/SPEAK/RAPID) by comparing current and previous skeleton coordinates.
    Similar to C++ GetMotionStatusFromMP function.
    Input: current_motion_status (int, current motion status)
           cur_mp (List[SkeletonPoint], current frame skeleton coordinates)
           prev_mp (List[SkeletonPoint], previous frame skeleton coordinates)
    Output: int (New motion status)
    """
    if current_motion_status == RESET_MOTION_STATUS:
        return RESET_MOTION_STATUS # Request to reset state

    motion_status = current_motion_status

    if not prev_mp or not cur_mp or len(prev_mp) < 17 or len(cur_mp) < 17:
        # If previous frame data is invalid, start in READY status
        return READY_MOTION_STATUS

    # C++ code uses 2D distance for Rdev/Ldev, so we will follow that.
    right_wrist_idx = 16
    left_wrist_idx = 15

    prev_R_wrist = np.array([prev_mp[right_wrist_idx].x, prev_mp[right_wrist_idx].y])
    cur_R_wrist = np.array([cur_mp[right_wrist_idx].x, cur_mp[right_wrist_idx].y])
    Rdev = np.linalg.norm(cur_R_wrist - prev_R_wrist)

    prev_L_wrist = np.array([prev_mp[left_wrist_idx].x, prev_mp[left_wrist_idx].y])
    cur_L_wrist = np.array([cur_mp[left_wrist_idx].x, cur_mp[left_wrist_idx].y])
    Ldev = np.linalg.norm(cur_L_wrist - prev_L_wrist)

    if Rdev > RAPID_DISTANCE or Ldev > RAPID_DISTANCE:
        return RAPID_MOTION_STATUS

    if current_motion_status == READY_MOTION_STATUS:
        hand_status = get_rough_hand_status_from_mp(cur_mp)
        if hand_status > 0: # If any hand is up
            motion_status = SPEAK_MOTION_STATUS
    elif current_motion_status == SPEAK_MOTION_STATUS:
        # C++ logic for SPEAK_MOTION_STATUS to READY_MOTION_STATUS transition is simplified:
        # if prev_mp and cur_mp exist, it moves to READY_MOTION_STATUS if no RAPID_MOTION.
        # This implies a detection of "end of speaking" or similar.
        # For a more robust state machine, additional heuristics would be needed here.
        motion_status = READY_MOTION_STATUS # Simplified transition based on C++ original

    return motion_status
```