1. 2\_combo\_proximity\_bridge\_debug.py

cat > 2\_combo\_proximity\_bridge\_debug.py << 'EOF'

#!/usr/bin/env python3

"""

Debug Version of Working Combo Proximity Bridge

Shows detailed sensor outputs in terminal while sending to Pixhawk

"""

import time

import sys

import numpy as np

import threading

from rplidar import RPLidar

from pymavlink import mavutil

try:

import pyrealsense2 as rs

REALSENSE\_AVAILABLE = True

except ImportError:

REALSENSE\_AVAILABLE = False

class DebugComboProximityBridge:

def \_\_init\_\_(self):

self.lidar\_port = '/dev/ttyUSB0'

self.pixhawk\_port = '/dev/serial/by-id/usb-Holybro\_Pixhawk6C\_1C003C000851333239393235-if00'

self.pixhawk\_baud = 57600

self.min\_distance\_cm = 20

self.max\_distance\_cm = 2500

self.quality\_threshold = 10

self.num\_sectors = 8

self.lidar = None

self.realsense\_pipeline = None

self.mavlink = None

# Data storage with thread safety

self.lidar\_sectors = [self.max\_distance\_cm] \* self.num\_sectors

self.realsense\_sectors = [self.max\_distance\_cm] \* self.num\_sectors

self.fused\_sectors = [self.max\_distance\_cm] \* self.num\_sectors

# Threading

self.lidar\_thread\_running = False

self.lidar\_data\_lock = threading.Lock()

# Statistics

self.lidar\_success\_count = 0

self.realsense\_success\_count = 0

self.total\_cycles = 0

# Sector names for readable output

self.sector\_names = [

"Forward", # 0

"Forward-Right", # 1

"Right", # 2

"Rear-Right", # 3

"Rear", # 4

"Rear-Left", # 5

"Left", # 6

"Forward-Left" # 7

]

def aggressive\_buffer\_clear(self):

"""Aggressively clear RPLidar buffers"""

try:

if self.lidar and hasattr(self.lidar, '\_serial') and self.lidar.\_serial:

serial\_conn = self.lidar.\_serial

for \_ in range(3):

serial\_conn.reset\_input\_buffer()

serial\_conn.reset\_output\_buffer()

time.sleep(0.05)

while serial\_conn.in\_waiting > 0:

try:

serial\_conn.read(serial\_conn.in\_waiting)

except:

break

time.sleep(0.01)

except Exception as e:

print(f"Buffer clear error: {e}")

def connect\_devices(self):

try:

print("Connecting to Pixhawk...")

self.mavlink = mavutil.mavlink\_connection(

self.pixhawk\_port,

baud=self.pixhawk\_baud,

source\_system=1,

source\_component=195

)

self.mavlink.wait\_heartbeat(timeout=10)

print("✓ Pixhawk connected")

success\_lidar = self.connect\_rplidar()

success\_realsense = self.connect\_realsense()

if not success\_lidar and not success\_realsense:

print("ERROR: No sensors connected")

return False

return True

except Exception as e:

print(f"Device connection failed: {e}")

return False

def connect\_rplidar(self):

try:

print("Connecting RPLidar S3...")

self.lidar = RPLidar(self.lidar\_port, baudrate=1000000, timeout=0.1)

self.aggressive\_buffer\_clear()

info = self.lidar.get\_info()

health = self.lidar.get\_health()

print(f"✓ RPLidar connected - Model: {info['model']}, Health: {health[0]}")

return True

except Exception as e:

print(f"✗ RPLidar connection failed: {e}")

self.lidar = None

return False

def connect\_realsense(self):

if not REALSENSE\_AVAILABLE:

print("✗ RealSense library not available")

return False

try:

print("Connecting RealSense camera...")

self.realsense\_pipeline = rs.pipeline()

config = rs.config()

config.enable\_stream(rs.stream.depth, 424, 240, rs.format.z16, 15)

self.realsense\_pipeline.start(config)

for \_ in range(5):

self.realsense\_pipeline.wait\_for\_frames()

print("✓ RealSense connected successfully")

return True

except Exception as e:

print(f"✗ RealSense connection failed: {e}")

self.realsense\_pipeline = None

return False

def lidar\_continuous\_thread(self):

"""RPLidar background processing with detailed output"""

print("Starting RPLidar background thread...")

buffer\_clear\_interval = 0

while self.lidar\_thread\_running:

try:

buffer\_clear\_interval += 1

if buffer\_clear\_interval >= 5:

self.aggressive\_buffer\_clear()

buffer\_clear\_interval = 0

self.lidar.start\_motor()

time.sleep(0.3)

scan\_data = []

measurement\_count = 0

start\_time = time.time()

for measurement in self.lidar.iter\_measurments():

if not self.lidar\_thread\_running:

break

if len(measurement) >= 4:

\_, quality, angle, distance = measurement[:4]

if quality >= self.quality\_threshold and distance > 0:

scan\_data.append((quality, angle, distance))

measurement\_count += 1

if len(scan\_data) > 20 and time.time() - start\_time > 0.5:

break

if measurement\_count > 200 or time.time() - start\_time > 1.0:

break

if len(scan\_data) > 10:

sectors = [self.max\_distance\_cm] \* self.num\_sectors

for quality, angle, distance\_mm in scan\_data:

distance\_cm = max(self.min\_distance\_cm,

min(int(distance\_mm / 10), self.max\_distance\_cm))

sector = int((angle + 22.5) / 45) % 8

if distance\_cm < sectors[sector]:

sectors[sector] = distance\_cm

with self.lidar\_data\_lock:

self.lidar\_sectors = sectors

try:

self.lidar.stop()

except:

pass

time.sleep(0.2)

except Exception as e:

print(f"Lidar thread error: {e}")

self.aggressive\_buffer\_clear()

time.sleep(0.5)

def get\_realsense\_data(self):

"""Get RealSense data with detailed output"""

if not self.realsense\_pipeline:

return False

try:

frames = self.realsense\_pipeline.wait\_for\_frames(timeout\_ms=500)

depth\_frame = frames.get\_depth\_frame()

if not depth\_frame:

return False

depth\_image = np.asanyarray(depth\_frame.get\_data())

height, width = depth\_image.shape

sectors = [self.max\_distance\_cm] \* self.num\_sectors

regions = [

(height//3, 2\*height//3, width//3, 2\*width//3), # Forward center

(height//3, 2\*height//3, 2\*width//3, width), # Forward right

(height//3, 2\*height//3, 0, width//3), # Forward left

]

forward\_sectors = [0, 1, 7]

for i, (y1, y2, x1, x2) in enumerate(regions):

region = depth\_image[y1:y2, x1:x2]

valid\_region = region[0:2\*(y2-y1)//3, :]

valid\_depths = valid\_region[(valid\_region > 100) & (valid\_region < 5000)]

if len(valid\_depths) > 10:

closest\_mm = np.percentile(valid\_depths, 5)

closest\_cm = max(self.min\_distance\_cm,

min(int(closest\_mm / 10), self.max\_distance\_cm))

sectors[forward\_sectors[i]] = closest\_cm

self.realsense\_sectors = sectors

return True

except Exception as e:

print(f"RealSense error: {e}")

return False

def print\_sensor\_data(self, lidar\_success, realsense\_success):

"""Print detailed sensor data to terminal"""

print("\n" + "="\*80)

print(f"CYCLE {self.total\_cycles} - Sensor Data Debug Output")

print("="\*80)

# Get current data thread-safely

with self.lidar\_data\_lock:

current\_lidar\_sectors = self.lidar\_sectors.copy()

# Print RPLidar data

print("RPLidar S3 Data:")

if lidar\_success:

print(" Status: ✓ ACTIVE")

for i, distance in enumerate(current\_lidar\_sectors):

if distance < self.max\_distance\_cm:

print(f" Sector {i} ({self.sector\_names[i]:12}): {distance:4d}cm")

else:

print(f" Sector {i} ({self.sector\_names[i]:12}): ----cm (no detection)")

else:

print(" Status: ✗ NO DATA")

for i in range(self.num\_sectors):

print(f" Sector {i} ({self.sector\_names[i]:12}): ----cm (no data)")

print()

# Print RealSense data

print("RealSense Depth Data:")

if realsense\_success:

print(" Status: ✓ ACTIVE")

for i, distance in enumerate(self.realsense\_sectors):

if distance < self.max\_distance\_cm:

print(f" Sector {i} ({self.sector\_names[i]:12}): {distance:4d}cm")

else:

print(f" Sector {i} ({self.sector\_names[i]:12}): ----cm (no detection)")

else:

print(" Status: ✗ NO DATA")

for i in range(self.num\_sectors):

print(f" Sector {i} ({self.sector\_names[i]:12}): ----cm (no data)")

print()

# Print fused data (what gets sent to Pixhawk)

print("FUSED Data (Sent to Pixhawk):")

for i, distance in enumerate(self.fused\_sectors):

lidar\_dist = current\_lidar\_sectors[i]

realsense\_dist = self.realsense\_sectors[i]

# Determine source

if i in [0, 1, 7]: # Forward sectors

if realsense\_dist < self.max\_distance\_cm:

source = "RealSense"

elif lidar\_dist < self.max\_distance\_cm:

source = "RPLidar"

else:

source = "Default"

else: # Side/rear sectors

if lidar\_dist < self.max\_distance\_cm:

source = "RPLidar"

elif realsense\_dist < self.max\_distance\_cm:

source = "RealSense"

else:

source = "Default"

if distance < self.max\_distance\_cm:

print(f" Sector {i} ({self.sector\_names[i]:12}): {distance:4d}cm (from {source})")

else:

print(f" Sector {i} ({self.sector\_names[i]:12}): ----cm (default)")

# Print statistics

lidar\_rate = (self.lidar\_success\_count / self.total\_cycles) \* 100

realsense\_rate = (self.realsense\_success\_count / self.total\_cycles) \* 100

obstacles = sum(1 for d in self.fused\_sectors if d < self.max\_distance\_cm)

closest = min(self.fused\_sectors)

print()

print("Statistics:")

print(f" RPLidar Success Rate: {lidar\_rate:5.1f}% ({self.lidar\_success\_count}/{self.total\_cycles})")

print(f" RealSense Success Rate: {realsense\_rate:5.1f}% ({self.realsense\_success\_count}/{self.total\_cycles})")

print(f" Obstacles Detected: {obstacles}/8 sectors")

print(f" Closest Obstacle: {closest}cm")

print("="\*80)

def fuse\_sensor\_data(self):

"""Combine sensor data intelligently"""

with self.lidar\_data\_lock:

current\_lidar\_sectors = self.lidar\_sectors.copy()

fused = [self.max\_distance\_cm] \* self.num\_sectors

for i in range(self.num\_sectors):

lidar\_dist = current\_lidar\_sectors[i]

realsense\_dist = self.realsense\_sectors[i]

# Forward sectors: prefer RealSense

if i in [0, 1, 7]:

if realsense\_dist < self.max\_distance\_cm:

fused[i] = realsense\_dist

elif lidar\_dist < self.max\_distance\_cm:

fused[i] = lidar\_dist

# Other sectors: prefer RPLidar

else:

if lidar\_dist < self.max\_distance\_cm:

fused[i] = lidar\_dist

elif realsense\_dist < self.max\_distance\_cm:

fused[i] = realsense\_dist

self.fused\_sectors = fused

def send\_proximity\_data(self, sector\_distances):

"""Send data to Pixhawk"""

try:

timestamp = int(time.time() \* 1000) & 0xFFFFFFFF

orientations = [0, 2, 2, 4, 4, 6, 6, 0]

for sector\_id, distance in enumerate(sector\_distances):

self.mavlink.mav.distance\_sensor\_send(

time\_boot\_ms=timestamp,

min\_distance=self.min\_distance\_cm,

max\_distance=self.max\_distance\_cm,

current\_distance=distance,

type=1,

id=sector\_id,

orientation=orientations[sector\_id],

covariance=0

)

except Exception as e:

print(f"MAVLink send error: {e}")

def run(self):

if not self.connect\_devices():

return False

print("\nDebug Combo Proximity Bridge")

print("============================")

print("Shows detailed sensor outputs while sending to Pixhawk")

print("Press Ctrl+C to stop\n")

# Start RPLidar background thread

if self.lidar:

self.lidar\_thread\_running = True

lidar\_thread = threading.Thread(target=self.lidar\_continuous\_thread, daemon=True)

lidar\_thread.start()

try:

while True:

self.total\_cycles += 1

# Process sensors

realsense\_success = self.get\_realsense\_data()

if realsense\_success:

self.realsense\_success\_count += 1

lidar\_success = False

with self.lidar\_data\_lock:

if any(d < self.max\_distance\_cm for d in self.lidar\_sectors):

lidar\_success = True

self.lidar\_success\_count += 1

# Fuse data and send to Pixhawk

self.fuse\_sensor\_data()

self.send\_proximity\_data(self.fused\_sectors)

# Print detailed debug output every cycle

self.print\_sensor\_data(lidar\_success, realsense\_success)

time.sleep(2) # Slower rate for readable output

except KeyboardInterrupt:

print(f"\nStopping debug bridge...")

finally:

self.lidar\_thread\_running = False

self.cleanup()

def cleanup(self):

self.lidar\_thread\_running = False

if self.lidar:

try:

self.lidar.stop()

self.lidar.disconnect()

print("RPLidar disconnected")

except:

pass

if self.realsense\_pipeline:

try:

self.realsense\_pipeline.stop()

print("RealSense disconnected")

except:

pass

if self.mavlink:

try:

self.mavlink.close()

print("Pixhawk disconnected")

except:

pass

if \_\_name\_\_ == "\_\_main\_\_":

bridge = DebugComboProximityBridge()

bridge.run()

EOF

chmod +x 2\_combo\_proximity\_bridge\_debug.py

1. 8\_row\_following\_system.py

cat > 8\_row\_following\_system.py << 'EOF'

#!/usr/bin/env python3

"""

Agricultural Row Following System

Runs alongside proximity bridge to provide row guidance

"""

import time

import sys

import numpy as np

import cv2

import threading

from pymavlink import mavutil

try:

import pyrealsense2 as rs

REALSENSE\_AVAILABLE = True

except ImportError:

REALSENSE\_AVAILABLE = False

class RowFollowingSystem:

def \_\_init\_\_(self):

# Use different Pixhawk connection (or UDP if proximity script uses USB)

self.mavlink\_connection = 'udpout:127.0.0.1:14552' # Different port

# Camera configuration

self.camera\_pipeline = None

self.mavlink = None

# Row detection parameters

self.image\_width = 640

self.image\_height = 480

self.roi\_top = 0.4 # Start ROI 40% down image

self.roi\_bottom = 0.9 # End ROI 90% down image

# Navigation state

self.row\_center\_pixel = None

self.heading\_error\_degrees = 0

self.lateral\_offset\_pixels = 0

self.rows\_detected = False

self.end\_of\_row = False

# Control parameters

self.steering\_sensitivity = 0.05 # Degrees per pixel

self.max\_steering\_angle = 15 # Maximum correction

# Statistics

self.frame\_count = 0

self.detection\_count = 0

self.running = False

def connect\_devices(self):

try:

# Connect to ArduPilot for sending guidance commands

print("Connecting to ArduPilot for row guidance...")

self.mavlink = mavutil.mavlink\_connection(

self.mavlink\_connection,

source\_system=1,

source\_component=220 # Different component ID

)

print("Row guidance MAVLink connected")

# Connect camera for row detection

if not self.connect\_camera():

return False

return True

except Exception as e:

print(f"Row system connection failed: {e}")

return False

def connect\_camera(self):

if not REALSENSE\_AVAILABLE:

print("RealSense required for row detection")

return False

try:

print("Connecting RealSense for row detection...")

self.camera\_pipeline = rs.pipeline()

config = rs.config()

# RGB stream for row detection

config.enable\_stream(rs.stream.color, self.image\_width, self.image\_height,

rs.format.bgr8, 30)

self.camera\_pipeline.start(config)

# Warm up

for \_ in range(10):

self.camera\_pipeline.wait\_for\_frames()

print("Camera connected for row detection")

return True

except Exception as e:

print(f"Camera connection failed: {e}")

return False

def detect\_crop\_rows(self, image):

"""

Detect crop rows using computer vision

Returns: (row\_detected, lateral\_offset, heading\_error)

"""

height, width = image.shape[:2]

# Extract ROI (region of interest)

roi\_y\_start = int(height \* self.roi\_top)

roi\_y\_end = int(height \* self.roi\_bottom)

roi = image[roi\_y\_start:roi\_y\_end, :]

# Convert to HSV for better vegetation detection

hsv = cv2.cvtColor(roi, cv2.COLOR\_BGR2HSV)

# Create mask for green vegetation (crops)

lower\_green = np.array([35, 40, 40])

upper\_green = np.array([85, 255, 255])

vegetation\_mask = cv2.inRange(hsv, lower\_green, upper\_green)

# Create mask for soil/brown areas (between rows)

lower\_brown = np.array([10, 20, 20])

upper\_brown = np.array([30, 150, 150])

soil\_mask = cv2.inRange(hsv, lower\_brown, upper\_brown)

# Find contours in vegetation mask

contours, \_ = cv2.findContours(vegetation\_mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

if len(contours) < 2:

return False, 0, 0

# Sort contours by area and take largest ones (crop rows)

contours = sorted(contours, key=cv2.contourArea, reverse=True)

valid\_contours = [c for c in contours[:4] if cv2.contourArea(c) > 500]

if len(valid\_contours) < 2:

return False, 0, 0

# Calculate centers of crop rows

row\_centers = []

for contour in valid\_contours:

M = cv2.moments(contour)

if M["m00"] != 0:

cx = int(M["m10"] / M["m00"])

row\_centers.append(cx)

if len(row\_centers) < 2:

return False, 0, 0

# Sort row centers and find the gap to navigate through

row\_centers.sort()

# Take two main rows (leftmost and rightmost, or middle pair)

if len(row\_centers) == 2:

left\_row = row\_centers[0]

right\_row = row\_centers[1]

else:

# Multiple rows - take middle pair

mid\_index = len(row\_centers) // 2

left\_row = row\_centers[mid\_index - 1]

right\_row = row\_centers[mid\_index]

# Calculate desired path (center between rows)

desired\_center = (left\_row + right\_row) / 2

image\_center = width / 2

# Calculate lateral offset (pixels from center)

lateral\_offset = desired\_center - image\_center

# Convert to heading error (degrees)

heading\_error = lateral\_offset \* self.steering\_sensitivity

heading\_error = np.clip(heading\_error, -self.max\_steering\_angle, self.max\_steering\_angle)

return True, lateral\_offset, heading\_error

def detect\_end\_of\_row(self, image):

"""Detect if we've reached the end of crop rows"""

height, width = image.shape[:2]

# Look at far forward area

forward\_roi = image[int(height \* 0.1):int(height \* 0.4), :]

# Check for vegetation in forward view

hsv = cv2.cvtColor(forward\_roi, cv2.COLOR\_BGR2HSV)

lower\_green = np.array([35, 40, 40])

upper\_green = np.array([85, 255, 255])

vegetation\_mask = cv2.inRange(hsv, lower\_green, upper\_green)

vegetation\_percentage = np.sum(vegetation\_mask > 0) / vegetation\_mask.size

# If very little vegetation ahead, we're at end of row

return vegetation\_percentage < 0.15

def send\_guidance\_command(self, heading\_error):

"""

Send row following guidance to ArduPilot

This could be manual control or guided mode commands

"""

try:

# For now, just log the guidance commands

# In practice, you'd send actual control commands to ArduPilot

if abs(heading\_error) > 1.0: # Only significant corrections

direction = "RIGHT" if heading\_error > 0 else "LEFT"

print(f"ROW GUIDANCE: Steer {direction} {abs(heading\_error):.1f} degrees")

# Example of sending a manual control command (uncomment to use):

# self.mavlink.mav.manual\_control\_send(

# target\_system=1,

# x=0, # Forward/back

# y=int(heading\_error \* 10), # Left/right steering

# z=500, # Throttle (middle position)

# r=0, # Rudder

# buttons=0

# )

except Exception as e:

print(f"Guidance command error: {e}")

def run(self):

"""Main row following loop"""

if not self.connect\_devices():

return False

print("\nAgricultural Row Following System")

print("=================================")

print("Detecting crop rows and providing steering guidance")

print("Run this alongside your proximity bridge script")

print("Press Ctrl+C to stop\n")

self.running = True

try:

while self.running:

self.frame\_count += 1

# Get camera frame

frames = self.camera\_pipeline.wait\_for\_frames(timeout\_ms=1000)

color\_frame = frames.get\_color\_frame()

if not color\_frame:

continue

# Convert to numpy array

image = np.asanyarray(color\_frame.get\_data())

# Detect crop rows

rows\_detected, lateral\_offset, heading\_error = self.detect\_crop\_rows(image)

# Check for end of row

end\_of\_row = self.detect\_end\_of\_row(image)

# Update state

self.rows\_detected = rows\_detected

self.lateral\_offset\_pixels = lateral\_offset

self.heading\_error\_degrees = heading\_error

self.end\_of\_row = end\_of\_row

if rows\_detected:

self.detection\_count += 1

self.send\_guidance\_command(heading\_error)

status = f"ROWS DETECTED - Offset: {lateral\_offset:.0f}px, "

status += f"Heading: {heading\_error:.2f}°"

elif end\_of\_row:

status = "END OF ROW - Stop and turn"

else:

status = "NO ROWS - Continue straight"

# Status every 30 frames (~1 second)

if self.frame\_count % 30 == 0:

detection\_rate = (self.detection\_count / self.frame\_count) \* 100

print(f"Frame {self.frame\_count}: {status} | Detection rate: {detection\_rate:.1f}%")

time.sleep(0.033) # ~30 FPS

except KeyboardInterrupt:

print(f"\nStopping row following system...")

detection\_rate = (self.detection\_count / self.frame\_count) \* 100 if self.frame\_count > 0 else 0

print(f"Final stats: {self.detection\_count}/{self.frame\_count} frames with rows ({detection\_rate:.1f}%)")

finally:

self.cleanup()

def cleanup(self):

self.running = False

if self.camera\_pipeline:

try:

self.camera\_pipeline.stop()

except:

pass

if self.mavlink:

try:

self.mavlink.close()

except:

pass

if \_\_name\_\_ == "\_\_main\_\_":

system = RowFollowingSystem()

system.run()

EOF

chmod +x 8\_row\_following\_system.py

1. 9\_crop\_monitoring\_system.py

cat > 9\_crop\_monitoring\_system.py << 'EOF'

#!/usr/bin/env python3

"""

Crop Monitoring and Analysis System

Takes pictures and analyzes crop readiness

"""

import time

import sys

import os

import cv2

import numpy as np

from datetime import datetime

try:

import pyrealsense2 as rs

REALSENSE\_AVAILABLE = True

except ImportError:

REALSENSE\_AVAILABLE = False

class CropMonitoringSystem:

def \_\_init\_\_(self):

self.camera\_pipeline = None

# Create output directories

self.timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")

self.output\_dir = f"crop\_monitoring\_{self.timestamp}"

self.images\_dir = f"{self.output\_dir}/images"

self.analysis\_dir = f"{self.output\_dir}/analysis"

os.makedirs(self.images\_dir, exist\_ok=True)

os.makedirs(self.analysis\_dir, exist\_ok=True)

# Monitoring parameters

self.capture\_interval = 10 # Seconds between captures

self.image\_counter = 0

# Analysis parameters

self.min\_green\_threshold = 30 # Minimum vegetation percentage

self.maturity\_color\_ranges = {

'young': ([35, 100, 100], [55, 255, 255]), # Bright green

'mature': ([25, 50, 50], [35, 200, 200]), # Yellow-green

'ready': ([15, 100, 100], [25, 255, 255]) # Yellow/brown

}

print(f"Crop monitoring data will be saved to: {self.output\_dir}")

def connect\_camera(self):

if not REALSENSE\_AVAILABLE:

print("RealSense required for crop monitoring")

return False

try:

print("Connecting RealSense for crop monitoring...")

self.camera\_pipeline = rs.pipeline()

config = rs.config()

# High resolution for detailed crop analysis

config.enable\_stream(rs.stream.color, 1280, 720, rs.format.bgr8, 30)

self.camera\_pipeline.start(config)

# Warm up

for \_ in range(10):

self.camera\_pipeline.wait\_for\_frames()

print("Camera connected for crop monitoring")

return True

except Exception as e:

print(f"Camera connection failed: {e}")

return False

def analyze\_crop\_health(self, image):

"""

Analyze crop health and readiness from image

"""

height, width = image.shape[:2]

# Convert to HSV for better color analysis

hsv = cv2.cvtColor(image, cv2.COLOR\_BGR2HSV)

# Calculate overall vegetation coverage

lower\_green = np.array([25, 30, 30])

upper\_green = np.array([85, 255, 255])

vegetation\_mask = cv2.inRange(hsv, lower\_green, upper\_green)

vegetation\_percentage = (np.sum(vegetation\_mask > 0) / vegetation\_mask.size) \* 100

# Analyze maturity stages

maturity\_analysis = {}

total\_vegetation\_pixels = np.sum(vegetation\_mask > 0)

for stage, (lower, upper) in self.maturity\_color\_ranges.items():

stage\_mask = cv2.inRange(hsv, np.array(lower), np.array(upper))

# Only count pixels that are also vegetation

stage\_vegetation = cv2.bitwise\_and(stage\_mask, vegetation\_mask)

stage\_pixels = np.sum(stage\_vegetation > 0)

if total\_vegetation\_pixels > 0:

stage\_percentage = (stage\_pixels / total\_vegetation\_pixels) \* 100

else:

stage\_percentage = 0

maturity\_analysis[stage] = stage\_percentage

# Determine dominant maturity stage

dominant\_stage = max(maturity\_analysis, key=maturity\_analysis.get)

# Calculate crop density (vegetation pixels per area)

crop\_density = vegetation\_percentage

# Simple readiness assessment

readiness\_score = 0

if maturity\_analysis['ready'] > 50:

readiness\_score = min(100, maturity\_analysis['ready'] + 20)

elif maturity\_analysis['mature'] > 60:

readiness\_score = min(80, maturity\_analysis['mature'])

elif maturity\_analysis['young'] > 70:

readiness\_score = min(40, maturity\_analysis['young'] \* 0.5)

return {

'vegetation\_percentage': vegetation\_percentage,

'crop\_density': crop\_density,

'maturity\_analysis': maturity\_analysis,

'dominant\_stage': dominant\_stage,

'readiness\_score': readiness\_score,

'assessment': self.get\_readiness\_assessment(readiness\_score)

}

def get\_readiness\_assessment(self, score):

"""Convert readiness score to human-readable assessment"""

if score >= 80:

return "READY FOR HARVEST"

elif score >= 60:

return "NEARLY READY - Monitor closely"

elif score >= 40:

return "DEVELOPING - Several weeks remaining"

elif score >= 20:

return "YOUNG GROWTH - Months to harvest"

else:

return "EARLY STAGE - Season just beginning"

def create\_analysis\_overlay(self, image, analysis):

"""Create annotated image with analysis results"""

annotated = image.copy()

height, width = annotated.shape[:2]

# Create semi-transparent overlay for text background

overlay = annotated.copy()

cv2.rectangle(overlay, (10, 10), (width-10, 200), (0, 0, 0), -1)

annotated = cv2.addWeighted(annotated, 0.7, overlay, 0.3, 0)

# Add analysis text

y\_offset = 40

font = cv2.FONT\_HERSHEY\_SIMPLEX

# Timestamp

timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

cv2.putText(annotated, f"Captured: {timestamp}", (20, y\_offset), font, 0.6, (255, 255, 255), 2)

y\_offset += 25

# Vegetation coverage

cv2.putText(annotated, f"Vegetation Coverage: {analysis['vegetation\_percentage']:.1f}%",

(20, y\_offset), font, 0.6, (0, 255, 0), 2)

y\_offset += 25

# Dominant maturity stage

cv2.putText(annotated, f"Dominant Stage: {analysis['dominant\_stage'].upper()}",

(20, y\_offset), font, 0.6, (0, 255, 255), 2)

y\_offset += 25

# Readiness assessment

color = (0, 255, 0) if analysis['readiness\_score'] >= 80 else (0, 165, 255) if analysis['readiness\_score'] >= 40 else (0, 0, 255)

cv2.putText(annotated, f"Assessment: {analysis['assessment']}",

(20, y\_offset), font, 0.6, color, 2)

y\_offset += 25

# Readiness score

cv2.putText(annotated, f"Readiness Score: {analysis['readiness\_score']:.0f}/100",

(20, y\_offset), font, 0.6, (255, 255, 255), 2)

return annotated

def save\_monitoring\_data(self, image, analysis):

"""Save image and analysis data"""

self.image\_counter += 1

timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")

# Save original image

original\_filename = f"{self.images\_dir}/crop\_{self.image\_counter:04d}\_{timestamp}.jpg"

cv2.imwrite(original\_filename, image)

# Create and save annotated image

annotated\_image = self.create\_analysis\_overlay(image, analysis)

annotated\_filename = f"{self.analysis\_dir}/analysis\_{self.image\_counter:04d}\_{timestamp}.jpg"

cv2.imwrite(annotated\_filename, annotated\_image)

# Save analysis data as text

analysis\_filename = f"{self.analysis\_dir}/data\_{self.image\_counter:04d}\_{timestamp}.txt"

with open(analysis\_filename, 'w') as f:

f.write(f"Crop Analysis Report\n")

f.write(f"Timestamp: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")

f.write(f"Image: {original\_filename}\n\n")

f.write(f"Vegetation Coverage: {analysis['vegetation\_percentage']:.1f}%\n")

f.write(f"Crop Density: {analysis['crop\_density']:.1f}%\n")

f.write(f"Dominant Maturity Stage: {analysis['dominant\_stage']}\n")

f.write(f"Readiness Score: {analysis['readiness\_score']:.0f}/100\n")

f.write(f"Assessment: {analysis['assessment']}\n\n")

f.write("Maturity Breakdown:\n")

for stage, percentage in analysis['maturity\_analysis'].items():

f.write(f" {stage.capitalize()}: {percentage:.1f}%\n")

print(f"Saved crop analysis {self.image\_counter}: {analysis['assessment']} ({analysis['readiness\_score']:.0f}/100)")

return original\_filename, annotated\_filename, analysis\_filename

def run(self):

"""Main crop monitoring loop"""

if not self.connect\_camera():

return False

print("\nCrop Monitoring and Analysis System")

print("===================================")

print(f"Capturing images every {self.capture\_interval} seconds")

print(f"Data saved to: {self.output\_dir}")

print("Press Ctrl+C to stop\n")

last\_capture\_time = 0

try:

while True:

current\_time = time.time()

# Check if it's time to capture

if current\_time - last\_capture\_time >= self.capture\_interval:

# Get camera frame

frames = self.camera\_pipeline.wait\_for\_frames(timeout\_ms=2000)

color\_frame = frames.get\_color\_frame()

if color\_frame:

# Convert to numpy array

image = np.asanyarray(color\_frame.get\_data())

# Analyze crop health

analysis = self.analyze\_crop\_health(image)

# Save data

self.save\_monitoring\_data(image, analysis)

last\_capture\_time = current\_time

time.sleep(1) # Check every second

except KeyboardInterrupt:

print(f"\nStopping crop monitoring...")

print(f"Captured {self.image\_counter} images with analysis")

print(f"Data saved in: {self.output\_dir}")

finally:

self.cleanup()

def cleanup(self):

if self.camera\_pipeline:

try:

self.camera\_pipeline.stop()

except:

pass

if \_\_name\_\_ == "\_\_main\_\_":

system = CropMonitoringSystem()

system.run()

EOF

chmod +x 9\_crop\_monitoring\_system.py