# TOSCA Laser Control System - Image Processing Pipeline



## Architecture Diagrams

**Document Version:** 1.0 **Date:** 2025-10-15

## Overview

The image processing pipeline handles camera feed processing for: - **Ring detection** - Locating laser ring position and size - **Focus measurement** - Quantifying image sharpness - **Video recording** - Capturing treatment sessions - **Live display** - Real-time feed with overlays

All processing is done using OpenCV (cv2) and NumPy.

## Camera System

### Hardware

- **SDK:** VmbPy (Allied Vision Vimba Python SDK)
- **Interface:** USB 3.0 / GigE
- **Resolution:** 1920x1080 (configurable)
- **Frame rate:** 30 FPS (configurable)
- **Color:** Monochrome or Color (prefer monochrome for laser applications)

### Camera Controller

```
from vmbpy import *
import numpy as np
import threading
import queue

class CameraController:
```

```
"""
Wrapper for VmbPy camera control
"""

def __init__(self, camera_id=None):
    self.vmb = VmbSystem.get_instance()
    self.camera = None
    self.is_streaming = False
    self.frame_queue = queue.Queue(maxsize=10)
    self.frame_callback = None

    # Camera settings
    self.exposure_us = 10000  # 10ms default
    self.gain_db = 0
    self.resolution = (1920, 1080)
    self.fps = 30

def connect(self, camera_id=None) -> tuple[bool, str]:
    """Connect to camera"""
    try:
        with self.vmb:
            cameras = self.vmb.get_all_cameras()
            if not cameras:
                return (False, "No cameras found")

            # Use specified camera or first available
            if camera_id:
                self.camera = self.vmb.get_camera_by_id(camera_id)
            else:
                self.camera = cameras[0]

            self.camera.open()

            # Configure camera
            self._configure_camera()

            return (True, f"Connected to {self.camera.get_id()}")

    except Exception as e:
        return (False, f"Camera connection error: {str(e)}")

def _configure_camera(self):
    """Set camera parameters"""
    # Set exposure
    self.camera.ExposureTime.set(self.exposure_us)

    # Set gain
    self.camera.Gain.set(self.gain_db)

    # Set resolution (if camera supports)
    try:
        self.camera.Width.set(self.resolution[0])
        self.camera.Height.set(self.resolution[1])
    except:
        pass  # Use camera default

    # Set frame rate
    try:
        self.camera.AcquisitionFrameRate.set(self.fps)
    except:
        pass

def start_streaming(self):
    """Start continuous frame acquisition"""
    if not self.camera:
        return

    self.is_streaming = True
    self.camera.start_streaming(self._frame_handler)

def stop_streaming(self):
    """Stop frame acquisition"""
    if self.camera and self.is_streaming:
        self.camera.stop_streaming()
```

```
            self.is_streaming = False

    def _frame_handler(self, cam, stream, frame):
        """
        Callback for each frame from camera

        Args:
            frame: VmbFrame object
        """
        try:
            # Convert to numpy array
            img_array = frame.as_numpy_ndarray()

            # Add to queue (non-blocking)
            if not self.frame_queue.full():
                self.frame_queue.put(img_array, block=False)

            # Call external callback if set
            if self.frame_callback:
                self.frame_callback(img_array)

        except queue.Full:
            pass  # Drop frame if queue full
        except Exception as e:
            logger.error(f"Frame handler error: {e}")

    def get_latest_frame(self) -> np.ndarray:
        """Get most recent frame from queue"""
        try:
            # Clear queue and get latest
            frame = None
            while not self.frame_queue.empty():
                frame = self.frame_queue.get_nowait()
            return frame
        except queue.Empty:
            return None

    def disconnect(self):
        """Disconnect camera"""
        self.stop_streaming()
        if self.camera:
            self.camera.close()
            self.camera = None
```

# Ring Detection

## Algorithm: Hough Circle Transform

The laser produces a circular ring that must be detected for alignment verification.

```
import cv2
import numpy as np
from dataclasses import dataclass

@dataclass
class DetectedRing:
    """Result of ring detection"""
    center_x: int
    center_y: int
    radius: int
    confidence: float  # 0.0 to 1.0
    detected: bool

class RingDetector:
    """
    Detect circular laser ring in camera images
    """

    def __init__(self):
        # Detection parameters (tunable)
        self.dp = 1  # Inverse ratio of accumulator resolution
        self.min_dist = 100  # Minimum distance between circle centers
```

```
        self.param1 = 50  # Canny edge detector high threshold
        self.param2 = 30  # Accumulator threshold for circle centers
        self.min_radius = 30  # Minimum circle radius (pixels)
        self.max_radius = 400  # Maximum circle radius (pixels)

        # Preprocessing parameters
        self.blur_kernel = (9, 9)
        self.blur_sigma = 2

        # Region of interest (optional - speeds up detection)
        self.roi = None  # (x, y, width, height) or None for full frame

    def detect(self, frame: np.ndarray) -> DetectedRing:
        """
        Detect laser ring in frame

        Args:
            frame: Input image (color or grayscale)

        Returns:
            DetectedRing object
        """
        # Convert to grayscale if needed
        if len(frame.shape) == 3:
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        else:
            gray = frame.copy()

        # Apply ROI if set
        if self.roi:
            x, y, w, h = self.roi
            gray = gray[y:y+h, x:x+w]
        else:
            x, y = 0, 0

        # Preprocessing
        preprocessed = self._preprocess(gray)

        # Detect circles using Hough Transform
        circles = cv2.HoughCircles(
            preprocessed,
            cv2.HOUGH_GRADIENT,
            dp=self.dp,
            minDist=self.min_dist,
            param1=self.param1,
            param2=self.param2,
            minRadius=self.min_radius,
            maxRadius=self.max_radius
        )

        if circles is None or len(circles) == 0:
            return DetectedRing(0, 0, 0, 0.0, False)

        # Select best circle
        best_circle = self._select_best_circle(circles[0], preprocessed)

        # Adjust coordinates for ROI offset
        return DetectedRing(
            center_x=int(best_circle[0]) + x,
            center_y=int(best_circle[1]) + y,
            radius=int(best_circle[2]),
            confidence=self._calculate_confidence(best_circle, preprocessed),
            detected=True
        )

    def _preprocess(self, gray: np.ndarray) -> np.ndarray:
        """Preprocess image for better circle detection"""

        # 1. Gaussian blur to reduce noise
        blurred = cv2.GaussianBlur(gray, self.blur_kernel, self.blur_sigma)

        # 2. Enhance contrast (optional - adjust if needed)
        # clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
        # enhanced = clahe.apply(blurred)
```

```
        return blurred

    def _select_best_circle(self, circles: np.ndarray, image: np.ndarray):
        """
        Select best circle from multiple detections

        Criteria:
        - Brightness of ring perimeter
        - Darkness of interior
        - Circularity
        """
        if len(circles) == 1:
            return circles[0]

        best_score = -1
        best_circle = circles[0]

        for circle in circles:
            score = self._score_circle(circle, image)
            if score > best_score:
                best_score = score
                best_circle = circle

        return best_circle

    def _score_circle(self, circle, image: np.ndarray) -> float:
        """
        Score circle quality

        Higher score = better match to laser ring
        """
        cx, cy, r = circle

        # Create mask for ring perimeter
        mask_outer = np.zeros(image.shape, dtype=np.uint8)
        cv2.circle(mask_outer, (int(cx), int(cy)), int(r), 255, thickness=3)

        # Create mask for interior
        mask_inner = np.zeros(image.shape, dtype=np.uint8)
        cv2.circle(mask_inner, (int(cx), int(cy)), int(r - 10), 255, thickness=-1)

        # Calculate mean intensity of ring vs interior
        ring_intensity = cv2.mean(image, mask=mask_outer)[0]
        inner_intensity = cv2.mean(image, mask=mask_inner)[0]

        # Good laser ring: bright perimeter, darker interior
        contrast = ring_intensity - inner_intensity

        # Normalize score
        score = contrast / 255.0

        return max(0.0, min(1.0, score))

    def _calculate_confidence(self, circle, image: np.ndarray) -> float:
        """Calculate detection confidence (0.0 to 1.0)"""
        return self._score_circle(circle, image)

    def draw_overlay(self, frame: np.ndarray, ring: DetectedRing) -> np.ndarray:
        """
        Draw detected ring on frame

        Returns:
            Frame with overlay
        """
        overlay = frame.copy()

        if not ring.detected:
            # No ring detected - draw warning
            cv2.putText(overlay, "RING NOT DETECTED", (50, 50),
                        cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
            return overlay

        # Draw circle
```

```
        color = (0, 255, 0) if ring.confidence > 0.5 else (0, 165, 255)  # Green if confident, orange if
        not
        cv2.circle(overlay, (ring.center_x, ring.center_y), ring.radius, color, 2)

        # Draw center crosshair
        cv2.drawMarker(overlay, (ring.center_x, ring.center_y), color,
                       cv2.MARKER_CROSS, 20, 2)

        # Draw info text
        info_text = f"Ring: {ring.radius}px ({ring.confidence:.2f})"
        cv2.putText(overlay, info_text, (ring.center_x - 50, ring.center_y - ring.radius - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, color, 2)

        return overlay

    def calibrate_ring_size(self, ring: DetectedRing, actuator_position_um: float) -> dict:
        """
        Create calibration point mapping ring pixel radius to physical size

        Returns:
            Calibration data point
        """
        return {
            'actuator_position_um': actuator_position_um,
            'ring_radius_pixels': ring.radius,
            'center_x': ring.center_x,
            'center_y': ring.center_y,
            'confidence': ring.confidence
        }
```

# Focus Measurement

## Algorithm: Laplacian Variance

Focus quality is measured using the variance of the Laplacian operator - sharper images have higher variance.

```
class FocusAnalyzer:
    """
    Measure image focus quality
    """

    def __init__(self):
        # Calibration - these values depend on your camera/optics
        self.min_score = 0  # Completely out of focus
        self.max_score = 1000  # Perfectly in focus
        self.good_focus_threshold = 300  # Minimum for "acceptable" focus

    def calculate_focus_score(self, frame: np.ndarray) -> float:
        """
        Calculate focus quality score

        Args:
            frame: Input image (color or grayscale)

        Returns:
            Focus score (higher = better focus)
        """
        # Convert to grayscale if needed
        if len(frame.shape) == 3:
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        else:
            gray = frame

        # Calculate Laplacian
        laplacian = cv2.Laplacian(gray, cv2.CV_64F)

        # Calculate variance
        variance = laplacian.var()

        return variance
```

```python
def calculate_focus_percent(self, frame: np.ndarray) -> float:
    """
    Calculate focus as percentage (0-100)

    Returns:
        Focus percentage (100 = excellent focus, 0 = very poor)
    """
    score = self.calculate_focus_score(frame)

    # Normalize to 0-100 scale
    percent = ((score - self.min_score) / (self.max_score - self.min_score)) * 100
    percent = max(0, min(100, percent))

    return percent

def is_in_focus(self, frame: np.ndarray) -> tuple[bool, float]:
    """
    Determine if image is acceptably in focus

    Returns:
        (is_focused, focus_score)
    """
    score = self.calculate_focus_score(frame)
    is_focused = score >= self.good_focus_threshold

    return (is_focused, score)

def draw_focus_indicator(self, frame: np.ndarray) -> np.ndarray:
    """
    Draw focus quality indicator on frame

    Returns:
        Frame with focus indicator overlay
    """
    overlay = frame.copy()

    # Calculate focus
    is_focused, score = self.is_in_focus(frame)
    focus_percent = self.calculate_focus_percent(frame)

    # Draw focus bar (top-left corner)
    bar_x, bar_y = 20, 20
    bar_width, bar_height = 200, 30

    # Background
    cv2.rectangle(overlay, (bar_x, bar_y), (bar_x + bar_width, bar_y + bar_height),
                  (100, 100, 100), -1)

    # Fill based on focus percentage
    fill_width = int(bar_width * (focus_percent / 100))
    color = (0, 255, 0) if is_focused else (0, 165, 255)  # Green if focused, orange if not
    cv2.rectangle(overlay, (bar_x, bar_y), (bar_x + fill_width, bar_y + bar_height),
                  color, -1)

    # Border
    cv2.rectangle(overlay, (bar_x, bar_y), (bar_x + bar_width, bar_y + bar_height),
                  (255, 255, 255), 2)

    # Text
    text = f"Focus: {focus_percent:.0f}%"
    cv2.putText(overlay, text, (bar_x + 5, bar_y + 20),
                cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 2)

    return overlay

def calibrate_focus_thresholds(self, focused_frames: list, unfocused_frames: list):
    """
    Calibrate focus thresholds based on sample images

    Args:
        focused_frames: List of frames that are in good focus
        unfocused_frames: List of frames that are out of focus
    """
    # Calculate scores for focused frames
```

```
        focused_scores = [self.calculate_focus_score(frame) for frame in focused_frames]
        min_focused = min(focused_scores)
        max_focused = max(focused_scores)

        # Calculate scores for unfocused frames
        unfocused_scores = [self.calculate_focus_score(frame) for frame in unfocused_frames]
        max_unfocused = max(unfocused_scores)

        # Set threshold between max unfocused and min focused
        self.good_focus_threshold = (max_unfocused + min_focused) / 2
        self.max_score = max_focused * 1.5  # Leave headroom

        print(f"Calibration results:")
        print(f"  Focused range: {min_focused:.1f} - {max_focused:.1f}")
        print(f"  Unfocused range: {min(unfocused_scores):.1f} - {max_unfocused:.1f}")
        print(f"  Threshold set to: {self.good_focus_threshold:.1f}")
```

# Video Recording

## VideoRecorder Class

```
class VideoRecorder:
    """
    Record video from camera feed
    """

    def __init__(self, session_folder: str):
        self.session_folder = session_folder
        self.video_writer = None
        self.is_recording = False
        self.frame_count = 0
        self.start_time = None

        # Video settings
        self.fourcc = cv2.VideoWriter_fourcc(*'XVID')  # or 'mp4v', 'H264'
        self.fps = 30
        self.resolution = (1920, 1080)

    def start_recording(self, filename: str = None) -> tuple[bool, str]:
        """
        Start video recording

        Args:
            filename: Optional custom filename (default: video.avi)

        Returns:
            (success, filepath)
        """
        if self.is_recording:
            return (False, "Already recording")

        # Generate filename
        if not filename:
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            filename = f"treatment_{timestamp}.avi"

        filepath = os.path.join(self.session_folder, filename)

        # Create VideoWriter
        try:
            self.video_writer = cv2.VideoWriter(
                filepath,
                self.fourcc,
                self.fps,
                self.resolution
            )

            if not self.video_writer.isOpened():
                return (False, "Failed to open video writer")

            self.is_recording = True
            self.frame_count = 0
```

```
            self.start_time = time.time()

            return (True, filepath)

        except Exception as e:
            return (False, f"Recording start error: {str(e)}")

    def write_frame(self, frame: np.ndarray):
        """
        Write frame to video file

        Args:
            frame: Image frame (must match resolution)
        """
        if not self.is_recording or self.video_writer is None:
            return

        # Resize if needed
        if frame.shape[:2] != (self.resolution[1], self.resolution[0]):
            frame = cv2.resize(frame, self.resolution)

        # Write frame
        self.video_writer.write(frame)
        self.frame_count += 1

    def stop_recording(self) -> dict:
        """
        Stop recording and finalize video

        Returns:
            Recording metadata dict
        """
        if not self.is_recording:
            return {}

        duration = time.time() - self.start_time

        # Release video writer
        if self.video_writer:
            self.video_writer.release()
            self.video_writer = None

        self.is_recording = False

        metadata = {
            'frame_count': self.frame_count,
            'duration_seconds': duration,
            'fps': self.fps,
            'resolution': f"{self.resolution[0]}x{self.resolution[1]}"
        }

        return metadata

    def capture_snapshot(self, frame: np.ndarray, label: str = "") -> str:
        """
        Save single frame as image

        Args:
            frame: Image to save
            label: Optional label for filename

        Returns:
            Filepath of saved image
        """
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S_%f")
        if label:
            filename = f"snapshot_{label}_{timestamp}.png"
        else:
            filename = f"snapshot_{timestamp}.png"

        filepath = os.path.join(self.session_folder, "snapshots", filename)

        # Create directory if needed
        os.makedirs(os.path.dirname(filepath), exist_ok=True)
```

```
        # Save image
        cv2.imwrite(filepath, frame)

        return filepath
```

# Image Processing Pipeline Integration

## Complete Frame Processor

```
class FrameProcessor:
    """
    Complete image processing pipeline
    """

    def __init__(self, session_folder: str):
        self.ring_detector = RingDetector()
        self.focus_analyzer = FocusAnalyzer()
        self.video_recorder = VideoRecorder(session_folder)

        self.enable_ring_detection = True
        self.enable_focus_display = True
        self.enable_recording = False

    def process_frame(self, frame: np.ndarray) -> tuple[np.ndarray, dict]:
        """
        Process camera frame

        Args:
            frame: Raw camera frame

        Returns:
            (processed_frame, metadata)
        """
        processed = frame.copy()
        metadata = {}

        # Ring detection
        if self.enable_ring_detection:
            ring = self.ring_detector.detect(frame)
            processed = self.ring_detector.draw_overlay(processed, ring)
            metadata['ring'] = {
                'detected': ring.detected,
                'center_x': ring.center_x,
                'center_y': ring.center_y,
                'radius': ring.radius,
                'confidence': ring.confidence
            }

        # Focus measurement
        if self.enable_focus_display:
            is_focused, focus_score = self.focus_analyzer.is_in_focus(frame)
            processed = self.focus_analyzer.draw_focus_indicator(processed)
            metadata['focus'] = {
                'is_focused': is_focused,
                'score': focus_score,
                'percent': self.focus_analyzer.calculate_focus_percent(frame)
            }

        # Record frame if recording active
        if self.enable_recording and self.video_recorder.is_recording:
            self.video_recorder.write_frame(processed)

        return (processed, metadata)

    def start_recording(self):
        """Start video recording"""
        self.enable_recording = True
        return self.video_recorder.start_recording()

    def stop_recording(self):
        """Stop video recording"""
```

```
        self.enable_recording = False
        return self.video_recorder.stop_recording()
```

# UI Integration

## Video Display Widget (PyQt6)

```
from PyQt6.QtWidgets import QLabel, QWidget, QVBoxLayout
from PyQt6.QtGui import QImage, QPixmap
from PyQt6.QtCore import QTimer, pyqtSignal
import cv2

class VideoDisplayWidget(QWidget):
    """
    PyQt6 widget for displaying live camera feed
    """

    frame_processed = pyqtSignal(dict)  # Emit metadata

    def __init__(self, camera_controller, frame_processor):
        super().__init__()

        self.camera = camera_controller
        self.processor = frame_processor

        # UI setup
        self.image_label = QLabel()
        layout = QVBoxLayout()
        layout.addWidget(self.image_label)
        self.setLayout(layout)

        # Update timer
        self.timer = QTimer()
        self.timer.timeout.connect(self.update_frame)
        self.timer.start(33)  # ~30 FPS

    def update_frame(self):
        """Update displayed frame"""
        # Get latest frame from camera
        frame = self.camera.get_latest_frame()
        if frame is None:
            return

        # Process frame
        processed, metadata = self.processor.process_frame(frame)

        # Emit metadata
        self.frame_processed.emit(metadata)

        # Convert to Qt format and display
        self.display_frame(processed)

    def display_frame(self, frame):
        """Convert OpenCV frame to Qt and display"""
        # Convert BGR to RGB
        rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        h, w, ch = rgb_frame.shape
        bytes_per_line = ch * w

        # Convert to QImage
        qt_image = QImage(rgb_frame.data, w, h, bytes_per_line, QImage.Format.Format_RGB888)

        # Display
        self.image_label.setPixmap(QPixmap.fromImage(qt_image))
```

# Calibration Procedures

## 1. Ring Size Calibration

**Goal:** Map actuator position to physical ring diameter

```python
def calibrate_ring_size():
    """Interactive ring size calibration"""

    calibration_points = []

    print("Ring Size Calibration")
    print("---------------------")

    # Move actuator through range
    positions_um = [0, 500, 1000, 1500, 2000, 2500]

    for position in positions_um:
        # Move actuator
        actuator.move_to_position(position)
        time.sleep(1)  # Allow settling

        # Capture frame
        frame = camera.get_latest_frame()

        # Detect ring
        ring = ring_detector.detect(frame)

        if not ring.detected:
            print(f"  Position {position}µm: RING NOT DETECTED")
            continue

        # Prompt user for actual physical measurement
        print(f"  Position {position}µm: Detected {ring.radius}px")
        physical_size_mm = float(input("    Enter actual ring diameter (mm): "))

        calibration_points.append({
            'actuator_position_um': position,
            'ring_size_mm': physical_size_mm,
            'ring_radius_pixels': ring.radius
        })

    # Save calibration to database
    save_calibration('actuator_position_to_ring_size', calibration_points)

    print("Calibration complete!")
    return calibration_points
```

## 2. Focus Threshold Calibration

**Goal:** Determine focus score threshold for “in focus” determination

```python
def calibrate_focus():
    """Interactive focus calibration"""

    print("Focus Calibration")
    print("-----------------")
    print("Adjust focus to BEST focus, then press Enter...")
    input()

    # Capture focused frames
    focused_frames = []
    for i in range(10):
        frame = camera.get_latest_frame()
        focused_frames.append(frame)
        time.sleep(0.1)

    print("Now adjust to OUT OF FOCUS, then press Enter...")
    input()

    # Capture unfocused frames
    unfocused_frames = []
    for i in range(10):
        frame = camera.get_latest_frame()
        unfocused_frames.append(frame)
        time.sleep(0.1)
```

```
# Calibrate thresholds
focus_analyzer.calibrate_focus_thresholds(focused_frames, unfocused_frames)

# Save calibration
calibration_data = {
    'good_focus_threshold': focus_analyzer.good_focus_threshold,
    'max_score': focus_analyzer.max_score
}
save_calibration('camera_focus_calibration', calibration_data)

print("Focus calibration complete!")
```

---

# Performance Optimization

## Tips for Real-Time Processing

1. **Use ROI:** Restrict ring detection to region of interest
2. **Downscale:** Process smaller image for speed (e.g., 960x540 instead of 1920x1080)
3. **Threading:** Process frames in separate thread from UI
4. **Skip frames:** Don't process every frame if 30 FPS is too fast
5. **GPU acceleration:** Use OpenCV with CUDA if available

---

**Document Owner:** Image Processing Engineer **Last Updated:** 2025-10-15